IBM

Series/1

SC34-0312-2

File No. S1-34

LICENSED
PROGRAM

IBM Series/1
Event Driven Executive
System Guide

Program Numbers: 5719-XS1 5719-XS2 5719-MS1
5719-XX2 5719-XX3 5719-AM3
5719-UT3 5719-UT4
5719-LM5 5719-LM6
5740-LM2 5740-LM3

SC34-0312-2

File No. S1-34

LICENSED
PROGRAM

IBM Series/1
Event Driven Executive
System Guide

Program Numbers:

5719-XS1	5719-XS2	5719-MS1
5719-XX2	5719-XX3	5719-AM3
5719-UT3	5719-UT4	
5719-LM5	5719-LM6	
5740-LM2	5740-LM3	

Second Edition (April 1980)

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services which are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Use this publication only for the purpose stated.

Changes are periodically made to the information herein; any such changes will be reported in subsequent revisions or Technical Newsletters.

Publications are not stocked at the address given below. Requests for copies of IBM publications should be made to your IBM representative or the IBM branch office serving your locality.

This publication could contain technical inaccuracies or typographical errors. A form for reader's comments is provided at the back of this publication. If the form has been removed, address your comments to IBM Corporation, Systems Publications, Department 27T, P.O. Box 1328, Boca Raton, Florida 33432. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

SUMMARY OF AMENDMENTS

4969 Magnetic Tape Support (Version 2 only)

The following chapters have been modified to include information for the IBM Series/1 4969 Magnetic Tape

- Chapter 3 Data Management - Tape functions and storage capacities
- Chapter 4 Operator Commands and Utilities - $VARYOFF and $VARYON operator commands, $TAPEUT1 utility
- Chapter 6 System Configuration - The TAPE configuration statement
- Chapter 7 System Generation - Sample configuration to illustrate including tape support in a system
- Chapter 10 The Session Manager - Examples of menus and options for tape utility
- Chapter 11 Tape organization - A new chapter explaining the use of, and support provided for, magnetic tapes

Remote Management Utility

The following chapters have been modified to include Remote Management Utility

- Chapter 6 System Configuration - The BSCLINE and TERMINAL statements
- Chapter 7 System Generation - Sample configuration to illustrate BSCLINE and TERMINAL statements

Bibliography

The bibliography lists the books in the Event Driven Executive library and a recommended reading sequence. Other publications related to the Event Driven Executive are also listed.

Storage Estimates

Storage estimates for V2.0 supervisor and utility programs have been added in Appendix A.

Supervisor Module Names

Supervisor Module Names for V2.0 have been updated in Appendix B.

Program Preparation Example

Appendix D shows a detailed example of how to code and prepare an interactive terminal program.

Miscellaneous Changes

This manual has been modified to include new function and to improve technical accuracy and clarity. Additional material and technical changes are indicated by vertical bars in the left margin.

HOW TO USE THIS BOOK

The material in this section is a guide to the use of this book. It defines the purpose, audience, and content of the book, as well as listing aids for using the book and background materials.

PURPOSE

The IBM Series/1 Event Driven Executive System Guide, SC34-0312 discusses system concepts and facilities. Examples of system concepts presented in the book are the generation of a static system, cross-partition communication between programs, and address translation. Examples of system facilities discussed are management of system resources, access methods, device support, and error logging. The System Guide also presents the details required for coding a tailored supervisor and advanced application topics.

AUDIENCE

This book is written primarily for system and application programmers. It does not include information for remote communications and advanced terminal applications.

The System Guide is intended for use by:

- Programmers who need a general understanding of the Event Driven Executive system
- Programmers concerned with coding applications or extending the system capabilities
- System programmers responsible for generating a customized system
- Programmers who will use the Indexed Access Method

HOW THIS BOOK IS ORGANIZED

This publication is organized in four parts, consisting of an introductory overview, system generation information, a description of the Indexed Access Method, and material explaining how to extend system capabilities.

Part I introduces you to the Event Driven Executive system and its capabilities.

Part II contains system generation and configuration information.

Part III contains both an overview of the Indexed Access Method and the detailed information necessary to write application programs using the Indexed Access Method.

Part IV describes severals ways to extend the capabilities of your system, such as modifying the session manager, using tape labels, and using diagnostic aids and facilities.

Appendix A discusses storage estimating.

Appendix B lists the supervisor module names (Version 1.1).

Appendix C lists the supervisor module names (Version 2).

Appendix D contains an example of how to code and prepare an interactive terminal program.

The bibliography discusses the Event Driven Executive library and lists related publications.

The Glossary defines terms.

The Common Index includes entries from all books in the Event Driven Executive library.

In general, the System Guide is organized according to the difficulty and depth of the information offered. Early material is overview information, followed by more detailed information for specialized use.

EXAMPLES AND OTHER AIDS

Throughout this book, both conceptual and coding examples are used to clarify system concepts and coding techniques. Coding examples are fully executable portions of complete programs that can be entered as shown. Coding illustrations are non-executable portions of incomplete programs that show the correct format of all required parameters on a statement. Missing code or code you must provide is indicated by a series of three vertical or horizontal dots.

Several other aids are provided to assist you in using this book:

- A Summary of Amendments lists the significant changes made to this publication since the last edition

- A Bibliography:

  - Lists the books in the Event Driven Executive library along with a brief description of each book and a recommended reading sequence

  - Lists related publications and materials

- A Glossary defines terms

- A Common Index which includes entries from each book in the Event Driven Executive library

References to other manuals are made throughout this manual using shortened titles. For the full title and order number of manuals mentioned in the text, refer to the bibliography at the back of this book.

RELATED PUBLICATIONS

Related publications are listed in the bibliography.

SUBMITTING AN APAR

If you have a problem with the Series/1 Event Driven Executive services, you are encouraged to fill out an authorized program analysis report (APAR) form as described in the IBM Series/1 Authorized Program Analysis Report (APAR) User's Guide, GC34-0099.

LIST OF FIGURES

PART I - INTRODUCTION

Part I is organized into five chapters which introduce you to the Event Driven Executive system and its capabilities.

CHAPTER 1. OVERVIEW

The Event Driven Executive system simplifies the implementation of application programs on the Series/1. Event driven implies that the system is activated by interrupts. An interrupt can be, for example, an operator pressing the ENTER key on a terminal or an external process interrupt.

The Event Driven Executive consists of the following IBM Series/1 licensed programs:

- Event Driven Executive Basic Supervisor and Emulator -- 5719-XS1 or 5719-XS2
- Event Driven Executive Utilities -- 5719-UT3 or 5719-UT4
- Event Driven Executive Macro Library/Host -- 5740-LM2 or 5740-LM3
- System/370 Program Preparation Facilities for Series/1 -- 5798-NNQ
- Event Driven Executive Host Communication Facility -- 5796-PGH
- Event Driven Executive Program Preparation Facility -- 5719-XX2 or 5719-XX3
- Event Driven Executive Macro Library -- 5719-LM5 or 5719-LM6
- FORTRAN IV Compiler and Object Support Library -- 5719-F02
- Event Driven Executive Mathematical and Functional Subroutine Library (MFSL) --5719-LM3
- Event Driven Executive COBOL Compiler and Resident Library -- 5719-CB3, and Event Driven Executive COBOL Transient Library -- 5719-CB4
- Event Driven Executive PL/I Compiler and Resident Library -- 5719-PL5, and Event Driven Executive PL/I Transient Library -- 5719-PL6
- Event Driven Executive Sort/Merge -- 5719-SM2
- Event Driven Executive Macro Assembler -- 5719-ASA
- Event Driven Executive Multiple Terminal Manager - 5719-MS1
- Event Driven Executive Indexed Access Method - 5719-AM3

The Basic Supervisor and Emulator is required for all Series/1 processors (4952, 4953, or 4955) that execute Event Driven Executive application programs or utilities. You must also have the licensed programs required for program development. Program development includes building the Basic Supervisor and Emulator and developing application programs using either the Event Driven Language compiler, the Series/1 Macro Assembler and Macro Library, the COBOL Compiler/Resident Library and Transient Library, the PL/I Compiler/Resident Library and Transient Library, or the FORTRAN IV Compiler and Object Support Library. You can assemble and execute application programs while other programs or utilities are running on the same Series/1.

The Basic Supervisor and Emulator allows you to process multiple, independent, concurrent application programs with limited concern for, or knowledge of, either the supervisor program or other application programs that share the same system.

The Event Driven Executive is equally appropriate for:

- A small unattended Series/1, without disk storage, dedicated to a single application
- Multiple large Series/1's, each serving several terminals and several realtime applications, which can be connected to a System/370
- Commercial transaction processing and event driven applications

LICENSED PROGRAM DESCRIPTIONS

Basic Supervisor and Emulator

The control program, or supervisor, manages the resources of the Series/1 and your application programs that execute on the Series/1. This support includes:

- An emulator and instruction set for coding application programs
- The ability to initiate an application program either from a terminal or from another application program that can pass parameters to the new program
- Multitasking within each application program, with preemptive task switching

- Interval timing, with timing precision based on requirements of the applications
- Multiple terminal support allowing terminals to be dynamically assigned to the application requiring them
- A relocating loader allowing application program to execute in any available main storage area
- The ability to operate independently of a host computer
- Support for a wide range of Series/1 devices
- Support for 4969 tape device - Version 2 only (5719-XS2)

Utilities

The system utilities provide interactive support for tailored supervisor generation, source module preparation, disk initialization, data set/volume maintenance, program preparation, and other system functions.

Remote Management Utility support is provided in Version 2 (5719-UT4) only.

Macro Library/Host

The Macro Library/Host is a set of libraries and procedures to be installed on a System/370 to allow Event Driven Language programs to be compiled and Series/1 assembler programs to be assembled on a host machine. The macros support all Event Driven Executive functions provided by the Program Preparation Facilities for Series/1. This licensed program operates in conjunction with the System/370 Program Preparation Facility.

Program Preparation Facility

The Program Preparation Facility consists of programs that allow you to compile and link edit Event Driven Language programs concurrently with the execution of other programs (including other Program Preparation Facility programs). You can also reconfigure, assemble, and link edit tailored supervisors.

If you code only Event Driven Language instructions, all application program preparation can be performed using this product.

Macro Library

This macro library, in conjunction with the Series/1 Macro Assembler, allows you to assemble application programs having a mix of Event Driven Language instructions and Series/1 assembler instructions. This library can also be used to create customized supervisors.

FORTRAN IV Compiler and Object Support Library

FORTRAN IV is a high level, mathematically oriented language for manipulating numerical data and formatting input/output operations. You can write FORTRAN IV source programs for scientific and engineering applications and general problem solving. The IBM Series/1 Event Driven Executive FORTRAN IV language is based on the American National Standard FORTRAN, X3.9-1966, with the exception of object time FORMATS, adjustable dimensions, COMPLEX data type, G-format specifications, and two-level FORMAT parenthesis.

When the FORTRAN IV compiler is installed on your Series/1,it transforms FORTRAN IV source programs into machine instructions. The compiler executes under the Basic Supervisor and Emulator to produce an object module for the Program Preparation Facility linkage editor that can then be processed by the $UPDATE utility. After $UPDATE processing, your program executes under the Basic Supervisor and Emulator.

Publications:

- IBM Series/1 FORTRAN IV Language Reference, GC34-0133-1
- IBM Series/1 Event Driven Executive FORTRAN IV User's Guide, SC34-0315

Mathematical and Functional Subroutine Library

The Mathematical and Functional Subroutine Library (MFSL) is a set of subroutines for IBM Series/1 Event Driven Executive application programs written in FORTRAN IV, or Event Driven Language, or Assembler Language. MFSL is a requirement for Event Driven Executive FORTRAN IV and support is provided for

functions such as:

- Mathematical functions to aid the application programmer, such as sine, cosine, logarithms and exponentiation functions, maximum and minimum functions, and modular arithmetic.

- Conversion routines to convert numeric data from EBCDIC to a Series/1 internal format suitable for mathematical operations.

- Error checking.

- Commercial subroutines to provide output formatting, data conversion, variable-length decimal arithmetic and utilities.

Publications:

- IBM Series/1 Mathematical and Functional Subroutine Library User's Guide SC34-0139

COBOL Compiler and Resident Library, and Transient Library

COBOL offers a wide range of commercial functions, plus extensive facilities for handling input and output, sorting and merging data files, accessing indexed data files, structuring source and object programs, and debugging. Also supported is local communication with Series/1 devices.

The COBOL compiler produces an object module which, along with the required COBOL support routines, is input to the Program Preparation Facility linkage editor. The linkage editor output is then processed by the $UPDATE utility to produce an executable relocatable load module. After $UPDATE processing your program executes under the Basic Supervisor and Emulator.

IBM Series/1 Event Driven Executive COBOL is designed according to specifications for American National Standard COBOL X3.23-1974, as understood and interpreted by IBM as of March 1979, with the exception of the RERUN Clause. IBM Series/1 Event Driven Executive COBOL exceeds the Low Intermediate Level COBOL as defined by FIPS 21-1.

Publications:

- IBM Series/1 COBOL Language Reference, GC34-0234

- IBM Series/1 Event Driven Executive COBOL Programmers Guide, SL23-0014

PL/I Compiler and Resident Library, and Transient Library

PL/I allows applications to use the full function capabilities of the hardware and operating system. The PL/I language is extensive in function, permitting development of applications that can be easily modified and maintained. Highlights of the PL/I offering include:

- Communications support
- Indexed Access Method support
- Full-screen support
- Sort/Merge support
- Commercial programming functions
- Dynamic allocation and freeing of storage
- Optimized object code
- Magnetic tape support

PL/I's fixed decimal facility allows you to process large and fractional numbers on the IBM Series/1 4952 and 4953 processors which do not have the floating point feature.

The PL/I compiler produces an object module which, along with the required PL/I support routines, is input to the Program Preparation Facility linkage editor. The linkage editor output is then processed by the $UPDATE utility to produce an executable relocatable load module. After $UPDATE processing your program will execute under the Basic Supervisor and Emulator.

IBM Series/1 Event Driven Executive PL/I is a subset of the American National Standard Programming Language PL/I (ANSI X3.53-1976), as understood and interpreted by IBM as of July 1979, plus multitasking language extensions.

Publications:

- IBM Series/1 Event Driven Executive PL/I Language Reference, GC34-0147
- IBM Series/1 Event Driven Executive PL/I User's Guide, SC34-0148

Sort/Merge

The Sort/Merge licensed program sorts and merges records from up to eight input data sets into one output data set in either ascending or descending order. You can specify one or more control fields in the records to be sorted. The Sort/Merge program compares the control fields to determine the relative sequence of the records.

The Event Driven Executive Sort/Merge program executes under the Basic Supervisor and Emulator.

Publications:

- IBM Series/1 Event Driven Executive Sort/Merge: Programmer's Guide, SL23-0016
- IBM Series/1 Event Driven Executive Sort/Merge: Specifications Sheet Form, GX23-0009

Series/1 Macro Assembler

The Macro Assembler converts text data sets containing machine, assembler, and macro instructions that have been coded in the Series/1 instruction set into object modules. The object modules can then be processed by the linkage editor.

When the assembler is used in conjunction with the Macro Library, applications coded in the Event Driven Language can also be processed by the Macro Assembler, including customizing the supervisor. You can also include in the macro library your own macros for commonly used routines. The Macro Assembler and the Macro Library can be used in place of the Program Preparation Facility ($EDXASM).

With the Macro Assembler you can assemble device support modules or modules that modify supervisor functions. You can also assemble exit routines written in Series/1 Macro Assembler language. The resulting object module is input to the Program Preparation Facility linkage editor, together with your applications generated in Event Driven Language instructions, PL/I, FORTRAN IV, and/or COBOL. Your program will execute under the Basic Supervisor and Emulator after it has been processed by the library update utility ($UPDATE).

Publications:

- IBM Series/1 Event Driven Executive Macro Assembler, GC34-0317
- IBM Series/1 Macro Assembler Reference Summary, SX34-0076

Multiple Terminal Manager

The IBM Series/1 Event Driven Executive Multiple Terminal Manager provides a set of high level functions that simplify the design, implementation, and maintenance of transaction-oriented applications. High level language programs (COBOL, PL/I, FORTRAN IV, or Event Driven Language) can execute in an interactive environment, where one or more applications can run concurrently using one or more display devices. Additional interfaces are provided for indexed or direct files (access to indexed files requires the Indexed Access Method) and an operator interface for functions such as sign on, connect or disconnect, terminal status reports, and printing the contents of the transaction program library.

Publications: Refer to the Multiple Terminal Manager topics in the master index of this publication.

Indexed Access Method

The Indexed Access Method provides data management facilities that support indexed file operations. It allows you to build, access, and maintain records in indexed data sets via a predetermined field called a key. An index of keys provides fast access to records in an indexed data set. The access method supports a high degree of insert/delete activity, providing both direct and sequential access to the data from multiple, concurrently executing programs. Applications that use the Indexed Access Method can be programmed in the Event Driven Language, PL/I, or in COBOL. It is supported by the Sort/Merge licensed program, which will accept Indexed Access Method data sets as input files. Also provided is a utility to define indexed data sets. This utility can be invoked from a terminal or from a program.

The Indexed Access Method provides keyed access to data to support a variety of applications, ranging from batch processing to interactive applications.

The data file organization provides direct and sequential processing of files. This is accomplished by using cascading index techniques for direct processing and by sequence chaining of the data blocks for sequential processing.

The access method supports files which have high add/delete activity (such as open order files) with nominal performance degradation. This is accomplished by distributing free space for additions throughout the file, by updating and inserting additions in place, and by dynamically reclaiming space after deletions.

Indexed Access Method supports multiple programs and tasks sharing the same files. In a shared environment, data integrity is maintained by record and block level locking to prevent access to a record while it is being modified.

Publications: Refer to the Indexed Access Method topics in the master index of this publication.

Series/1 Data Collection Interactive PRPQ Support

The Series/1 Data Collection Interactive PRPQ (P82600) provides Series/1 programming support for attachment of the 5234, 5235, 5236 Data Entry Stations. Value Read (5239) is supported; also the output of up to 8 bytes of data from the Series/1 to the 5235/5236 Data Entry Station displays. These bytes can consist of any digit and some alpha characters.

Support is provided for:

1. Personalization functions:
   - via console prompting
   - transportable and modifiable configuration definitions
   - Auto IPL, using the last executing configuration
2. IOCS functions:
   - Write 4 characters - Time-of-Day
   - Write 8 characters - display of any numeric and some alpha characters
   - Initiate online test
   - Read 180 byte buffer
   - Set audible alarm/contact closure
   - Error handling
3. Data Routing/Formatting functions:
   - Route to storage
   - Route to Disk/Diskette
     - Optionally, common data with buffers

- Optionally, sequenced buffers
- Single output format per controller
- Incomplete transactions written with regular records
- Diskette output 5231-002 compatible

This product has potential uses in data collection time and attendance, limited data base inquiry and interaction, and plant and process control type applications.

Publications

- IBM Series/1 Data Collection Interactive Programming RPQ P82600 Users Guide, SC34-1654

PROGRAM FEATURES

Multiprogramming, Multitasking Supervisor

The Event Driven Executive Supervisor and Emulator controls the execution of your application programs. It is a multiprogramming supervisor that is capable of controlling concurrent program execution. Some of the design features are:

- Up to 510 task priorities consisting of 255 priorities within each of two hardware levels
- A storage efficient instruction emulator that is table driven
- Provision for interprogram communication
- Capability for automatic restart following a power outage
- Storage management support for storage sizes greater than 64K bytes
- The capability for concurrent execution of
  - Multiple applications
  - Utilities
  - Program preparation

Event Driven Language

The Event Driven Language provides dynamic control of operation sequencing, calculations, and decision making. It has the following features:

- Integer and floating-point calculations
- Logical and shifting functions
- Structured programming functions: IF, THEN, ELSE, DO UNTIL, DO WHILE
- Interval timing and time of day functions
- Task control and synchronization

- Generalized binary synchronous communication capability modeled after OS/BTAM, for basic read/write to other systems
- Assembler language subroutines
- Data definition
- Program control
- Graphics
- Queue processing

Multiple Terminal Support

The Event Driven Executive terminal support is as device independent as possible. The following devices are supported:

- IBM 4978 Display Station
- IBM 4979 Display Station
- IBM 3101 Display Terminal
- IBM 4973 Line Printer
- IBM 4974 Matrix Printer
- IBM 2741 Communications Terminal
- Graphics terminal (Tektronix[1] or equivalent)
- Teletype[2] ASR 33/35 (TTY) or equivalent

Timer Support

The Event Driven Executive supports the following timers:

- IBM Feature #7840 Timer Attachment
- IBM 4952 processor native timer

[1] Registered trademark of the Tektronix Corporation.
[2] Registered trademark of the Teletype Corporation.

Sensor Input/Output Support

The Event Driven Executive supports the Series/1 sensor I/O devices. The following functions are available:

- Analog input/output, digital input/output, process interrupt
- Sequential and random addressing of devices
- External synchronization
- Sharing device groups and subgroups between programs
- Relay or solid state multiplexing
- Multi-range analog input
- A program for testing your sensor I/O devices

Storage Requirements

The supervisor occupies at least 12K bytes of storage. The actual amount depends upon the support you require for your application. Typical supervisor storage requirements are 24K to 32K bytes. The remaining storage is available for your application programs. Each Event Driven Language instruction requires an average of six to eight bytes of storage.

Disk and Diskette Support

The following Series/1 disk and diskette units are supported:

- 4964 Diskette Unit
- 4966 Diskette Magazine Units
- 4962 Disk Storage Unit (Models 1, 1F, 2, 2F, 3, and 4)
- 4963 Disk Subsystem (Models 23A, 23B, 29A, 29B, 58A, 58B, 64A, and 64B)

The following disk and diskette functions are available:

- Fixed head support for system and for application programs (disk only)

- Multiple logical volumes on a physical disk or in a diskette magazine unit
- Sequential or random access
- Initial Program Load text

Tape Support (Version 2 only)

The Series/1 Event Driven Executive 4969 Magnetic Tape Subsystem supports the following:

- Up to four IBM 4969 tape drives attached to each tape controller
- 9 track tape drives with either vacuum or mechanical take-up arm
- Tape drives of 800 BPI NRZI mode, 1600 PE recording mode, or dual mode

The 4969 supports the following:

- Standard label tapes (SL) (DOS/VS compatible)
- Unlabeled tapes (NL)
- Bypass label processing (BLP)

Binary Synchronous Communications Support

The Event Driven Executive has the following binary synchronous communications capabilities:

- Multiple BSC medium speed, single line feature cards -- Feature 2074
- Multiple BSC high speed, single line feature cards -- Feature 2075
- Multiple BSC 8-line control feature cards -- Feature 2093 (each with one or two BSC 4-line feature cards -- Feature 2094)
- Point-to-point, leased or switched lines (switched lines provide auto answer and manual dialing)
- Multipoint operation as either master or tributary

- Transparent mode of operation
- Limited conversational mode of operation
- Automatic retry
- Remote Management Utility support (Version 2 only)

I/O Level Control (EXIO)

The I/O level control functions (EXIO instructions) allow you to control, at the device level, any device attached to the system that meets the standard I/O interface. The EXIO instructions provide the capability to control devices not otherwise accessible using the Event Driven Language instructions. You also may use the EXIO interface to support standard devices in a non-standard manner.

Communications Support

Communications support enables you to communicate with other processors. The following functions are available:

- Generalized binary synchronous support for processor-to-processor communications
- Multiple lines in point-to-point, switched, multi-point master, or multipoint tributary
- In conjunction with the Host Communications Facility IUP, direct read/write access to host files and direct job submission to a host batch processor over a leased line, using the single-line BSC adapter
- Communications capability with other IBM systems:
  - Series/1
  - System/3
  - System/32
  - System/34
  - 5100
  - 5110

  - System/360
  - System/370

- Remote job entry capability to Remote Job Entry facilities on System/370

Program Preparation

Event Driven Executive program preparation support allows you to assemble and link edit programs while other tasks are executing. You are not limited to application development. You can also configure, assemble, and link edit tailored supervisors. Program preparation support includes:

- The Event Driven Language compiler
- The Series/1 Macro Assembler
- The Host Preparation Facility
- The linkage editor
- The text editors
- The object reformatting programs

The PL/I, FORTRAN IV, COBOL compilers and resident libraries, the PL/I and COBOL transient libraries, and the Mathematical and Functional Subroutine Library are also available for program preparation.

Diagnostic Aids

The diagnostic aids are a set of programs and utilities that can improve the process of error detection and correction for both hardware and software errors. Some of these programs are independently executable utilities; others are resident in the supervisor.

The hardware-oriented aids provide I/O error logging, sensor I/O testing, tracing of BSC line activity, and utility programs to format and print the trace output.

The software-oriented aids provide an interactive application debugging tool; operator commands to display and patch storage; a program exception trace; and utilities for monitoring exception events, dumping storage to a data set, and printing the data set.

Depending upon the type of errors your system is encountering and your application requirements, you can select the appropriate diagnostic aids.

The following aids are provided:

- $BSCTRCE - traces the I/O activities on a given BSC line
- $BSCUT1 - formats the $BSCTRCE file
- $BSCUT2 - verifies the system's BSC configuration
- $D - displays storage in hexadecimal format
- $DEBUG - provides interactive program debugging capability
- $DISKUT2 - formats the data recorded by $LOG
- $DUMP - prints storage dumps taken by $TRAP
- $IOTEST - tests the operation of sensor I/O devices
- $LOG - records I/O errors
- $P - modifies storage
- $TAPEUT1 - general tape utility functions (Version 2 only)
- $TRAP - detects exception events and dumps storage
- Trace Table - contains data concerning program exceptions

For further information concerning the diagnostic aids, refer to the common index of this publication.

APPLICATION SUPPORT

The Event Driven Executive contains a system supervisor that controls the execution of your applications. The supervisor controls the execution of multiple tasks so they can operate concurrently. The Event Driven Language is provided for writing application programs. A key design feature is the support of multiple independent (time or event driven) applications with minimum interaction imposed by the system.

The following categories of applications are supported:

- Realtime data acquisition and control
- Data reduction and report generation
- Program preparation and testing
- Commercial applications
- Communications applications

The Event Driven Executive via multiprogramming allows multiple application programs within a single computer. The total number of simultaneously operating applications depends on the data rates, program complexities, and the hardware configuration.

Data Interchange

The data interchange or exchange function provides:

- The ability to transmit data to and from a host with BSC support, and to initiate host program execution using remote job entry support
- The ability to transmit data to and from a host; to initiate program execution at the host from a Series/1 terminal or from your program, and to synchronize program events on the host and Series/1 using the Host Communications Facility IUP (installed user program)
- Support of multiple Series/1 Event Driven Executive systems by a single host
- Interchange of data through the use of basic data exchange formatted type diskettes. This function provides data transfer capability to and from other systems which read and write the basic data exchange diskette.

- The 4969 Magnetic Tape Subsystem support can be used as a data exchange method between systems. The following modes of exchange are possible for 9-track 800/1600 BPI tapes:

  - Non-labeled
  - Standard labeled
  - Other logical layouts may be processed by Bypass Label Processing

Minimum Execution System Configuration

For Series/1 program execution with full multiprogramming capability, the minimum hardware requirements are:

- One of the following:
  - 4952 Processor with 32K bytes of storage
  - 4953 or 4955 Processor with 16K bytes of storage
- 4964 Diskette Unit or 4966 Diskette Magazine Unit

The floating-point computational and conversion capabilities of the Event Driven Executive require the hardware floating-point feature of the 4955 processor (#3920).

The use of the timer functions requires the timer feature (#7840), for processors other than the 4952.

The three-dimensional capability of the Graphics Display Processor Utility requires a 4955 processor with the floating-point feature (#3920).

A limited function (multitasking only) system is also feasible with a 16K-byte processor and no disks, diskettes, or terminals.

Minimum Program Preparation Requirements

The minimum Series/1 configuration for preparation of Event Driven Executive programs is:

- 4953 or 4955 processor with 48K bytes of storage or 4952 processor with 64K bytes of storage
- 4964 Diskette Unit or 4966 Diskette Magazine Unit
- 4962 Disk Storage Unit or a 4963 Disk Subsystem
- 4973 Line Printer or 4974 Matrix Printer
  - One of the following:
  - 4978 Display Station

- 4979 Display Station
- 3101 Display Terminal or equivalent teletypewriter device

Minimum Licensed Program Requirements

The programs you require depend upon your application and which language you will use to code your applications. The choices are COBOL, FORTRAN IV, PL/I, Event Driven Language, or Macro Assembler Language.

The first requirement is the Basic Supervisor and Emulator. Then, based upon your choice of languages and your type of work, the following can be used as guidelines:

- COBOL

  Program preparation requires the COBOL Compiler and Resident Library, the Utilities, and the link editor of either the Program Preparation Facility or the Series/1 Macro Assembler. It allows you to:

  - Install the COBOL Compiler and Resident Library and the COBOL Transient Library
  - Allocate data sets
  - Enter source programs
  - Compile
  - Link edit
  - Format screens

  Execution and test require the COBOL Transient Library and the Utilities. During execution and test, you may:

  - Use diagnostic aids
  - Load programs
  - Back up and copy data sets

- PL/I

  Program preparation requires the PL/I Compiler and Resident Library, the Utilities, and the link editor of either the Program Preparation Facility or the Series/1 Macro Assembler; it allows you to:

- Install the PL/I Compiler and Resident Library and the PL/I Transient Library
- Allocate data sets
- Enter source programs
- Compile
- Link edit
- Format screens

Execution and test require the PL/I Transient Library and the Utilities. During execution and test, you may:

- Use diagnostic aids
- Load programs
- Back up and copy data sets

- FORTRAN IV

  Program preparation requires FORTRAN IV, the Utilities, the Mathematical and Functional Subroutine Library, and the link editor of either the Program Preparation Facility or the Series/1 Macro Assembler; it allows you to:

  - Install FORTRAN IV and the Mathematical and Functional Subroutine Library
  - Allocate data sets
  - Enter source programs
  - Compile
  - Link edit
  - Format screens

  Execution and test require the the Utilities. During execution and test, you may:

  - Use diagnostic aids
  - Load programs
  - Back up and copy data sets

- Event Driven Language

Program preparation requires Utilities and the Program Preparation Facility; it allows you to:

- Install the Program Preparation Facility
- Allocate data sets
- Enter source programs
- Assemble
- Link edit
- Format screens

Execution and test require the Utilities. During execution and test, you may:

- Use diagnostic aids
- Load programs
- Back up and copy data sets

- Macro Assembler Language

  Program preparation requires the Utilities, the Series/1 Macro Assembler, and the Macro Library; it allows you to:

  - Install the Series/1 Macro Assembler and the Macro Library
  - Allocate data sets
  - Enter source programs
  - Add macros that you have written
  - Assemble
  - Link edit
  - Format screens

  Execution and test require the Utilities. During execution and test, you may:

  - Use diagnostic aids
  - Load programs
  - Back up and copy data sets

Any application can use the Indexed Access Method, Sort/Merge, or the Mathematical and Functional Subroutine Library. An indexed data set can be accessed by using instructions provided by COBOL and PL/I. FORTRAN IV requires the Mathematical and Functional Subroutine Library. FORTRAN, EDL or assembly language must be used to interface to MFSL.

If your application calls for transaction processing, the Multiple Terminal Manager can be used.

For link edits you can use the Program Preparation Facility or the Series/1 Macro Assembler since both contain the linkage editor. The Series/1 Macro Assembler allows you to intermix macro assembler language and Event Driven Language instructions, although with a loss of assembly time performance when compared to the Program Preparation facility.

INSTALLING THE SYSTEM

The IBM licensed programs that comprise the Event Driven Executive system are shipped on one or more diskettes along with a document called a Program Directory. The diskettes contain the programs and source material. The Program Directory provides additional reference material, including the installation procedure. This procedure identifies and describes the contents of each diskette volume and contains the step by step prompt/reply sequences that are used to install the product.

The installation process is simply a sequence of:

1. initialization

2. copy data sets to disk

3. change diskette($VARYON)

4. repeat steps 2 and 3, until all diskettes are processed

An optional installation verification procedure allows you to verify that the installation is complete. The Program Directory contains the step by step instructions for executing the verification procedure.

CHAPTER 2. SUPERVISOR AND EMULATOR

The supervisor and emulator provides the system services required to assign processor time to your applications; data management and device management services; error handling and recording; and serialization logic. The emulator executes Event Driven Language instructions.

PROGRAM/TASK CONCEPTS AND STRUCTURE

In an Event Driven Executive system, system resources are allocated to tasks, according to the priority of the tasks. A **task** is a unit of work, defined by the application programmer. A **program** is a disk or diskette resident collection of **one or more tasks** that can be loaded into storage for execution. Although **program** and **task** are sometimes used synonymously, the basic **executable** unit for the supervisor is the **task**; a **program** is the unit that is loaded into storage.

The Program

A program is defined by the application programmer. In its simplest form, a program consists of a single task and contains a PROGRAM statement, instructions, an ENDPROG statement, and an END statement. In its more complex form, a program contains more than one task.

One of the operands on the PROGRAM statement defines the initial entry point. When the program is brought into storage, the initial entry point is the place in the program at which execution will begin. The programmer is responsible for initiating other tasks that are contained in the program.

The name of a program is the name of the data set in which a program resides. A program may be brought into storage either by a terminal operator or by another program.

The Task

Tasks are formed by combining instructions within an application program. Each task is assigned a priority which the supervisor uses to allocate time for execution. An application program can consist of more than one task. Each task within the system runs independently, with its own priority, and each com-

petes equally for the resources it requests from the system.

Task priority is assigned by the application programmer when the task is coded. Valid priorities range between 1 and 510, with 1 being the highest priority, and 510 the lowest. Tasks with priorities ranging from 1 to 255 execute on hardware level 2, and tasks with priorities ranging from 256 to 510 execute on hardware level 3. Levels zero and one are reserved for the Basic Supervisor and Emulator.

Task Switching and Supervisor Control Routines

The supervisor always allocates the processor resource to the highest priority task which is ready to execute.

When a higher priority task becomes ready by task action or via an external event, the supervisor dispatches the higher priority task.

In addition, routines under supervisor control support device and resource queuing, sensor I/O, interval timing, and process interrupt functions. Services are also provided to manage storage, host communications, disks, printers, tapes, and terminals.

Task Definition and Control Functions

You can use the following Event Driven Language instructions to define tasks and to control which tasks are executing at a given time:

ATTACH Makes a new task ready for execution

ATTNLIST Provides entry to one or more of your asynchronous attention interrupt handling routines. ATTNLIST produces a list of the command names that you have defined and their associated entry points

DETACH Removes a task from the ready state

ECB Generates an event control block

ENDATTN Terminates ATTNLIST processing

ENDPROG Defines the end of a program

ENDTASK Defines the logical end of a task

LOAD Loads a main program or program overlay from the currently executing program

POST Signals the occurrence of an event

PROGRAM Defines the basic parameters and primary task of a program

PROGSTOP Terminates execution of all of the tasks in a program and releases the storage allocated to the program

RESET Places an event in the "not occurred" state

TASK Defines and names a task

WAIT Halts task execution pending the occurrence of an event

WHERES Returns the address and address key of a named program

Subroutines

A function may be required at several points in a program's execution. Rather than code the sequence of instructions that performs that function each time the program needs it, the function can be coded once and defined as a subroutine. The subroutine can then be executed from as many points in the application program as required.

The subroutine capability is provided by the following instructions:

CALL Transfers control to a subroutine

RETURN Returns control from the subroutine to the calling program

SUBROUT Defines the entry point and parameters of a subroutine

USER Allows inline or subroutine use of instructions written in Series/1 assembly language

CALLFORT Transfers control to a FORTRAN program or subroutine from an Event Driven Language program

Queue Processing

You can use the Event Driven Language queuing instructions to define queues and to access entries in queues. You must define the size of a queue by specifying the number of entries it can hold. The following queuing instructions are provided:

DEFINEQ Establishes a queue

LASTQ Retrieves entries on a last-in-first-out (LIFO) basis

FIRSTQ Retrieves entries on a first-in-first-out (FIFO) basis

NEXTQ Adds an entry to a queue

Timers

If you have the hardware timer feature installed on your Series/1 4953 or 4955 processor, or if you have a 4952 processor, you can include support in your Event Driven Executive supervisor providing several timing functions that can be used by application programs. In addition to maintaining a time of day clock, the system also provides a time interval (elapsed time) clock, and has the capability of suspending task execution (entering the wait state) for specified lengths of time. The system also provides interrupts at the end of a time interval.

The time-of-day (TOD) clock is maintained in hours, minutes, and seconds. At initial program load (IPL), the clock is all zeros and begins running. You can set the actual clock time using the $T operator command function or with instructions in an initialization routine that you write, and the system will maintain clock time from that point on. The timer-related instructions are listed below:

GETTIME Moves the time of day values into an application program

INTIME Reads the relative clock value (elapsed time since IPL) into an application program and computes elapsed time (since a previous INTIME)

STIMER Starts the timer running for the specified time interval for a specific application task. When the interval expires, an ECB is posted.

TIMER Defines the timer feature (#7840) address during system generation

PRINTDATE Prints the date on the terminal

PRINTIME Prints the time of day on the terminal

Serial Resource Control

A resource, a physical or logical entity within the system, can be a subroutine, a data area within a particular program, or a data set or I/O device known across the system. A resource can be shared (used) by more than one task at the same time. For example, a table of constants might be referenced from two or more asynchronously executing tasks within a program. Since, by definition, the values in the table are constant (that is, read only), access to the table (resource) is unrestricted.

Unrestricted access to some shared resources can be undesirable. For instance, if a task were updating a disk data set, and other tasks had free access to the data set, the state of the data set is unpredictable. In this case, the data set is a shared serially reusable resource -- one that is shared but should be used by only one task at a time.

With Event Driven Language instructions, you can gain exclusive use of a serially reusable shared resource, and retain control over that resource until explicitly releasing it for use by other tasks. These instructions are:

DEQ Frees the resource and gives control of the resource to the longest waiting task, regardless of priority

ENQ Acquires exclusive control of a shared serially reusable resource -- one that is shared but should be used by only one task at a time.

QCB Generates a resource control block

Task Error Exit Facility

During the execution of a task, exception conditions may occur either in the task itself or in the hardware. The Series/1 signals the Event Driven Executive supervisor when these conditions arise. Then, the supervisor, for most programs, performs a system default action to clear the condition. While the system response to exceptions is usually desired, it may be inappropriate for some programs. For these programs, the supervisor provides a method, the Task Error Exit facility, for

tasks to gain control at a point specified by the task when an exception occurs. Pertinent status information is provided to the error exit routine so that it may take action to correct the exception and, if possible, continue.

Single Task Program

For most applications, a complex program structure is not required, and programs will consist of a single task in a single program, as shown in Figure 1

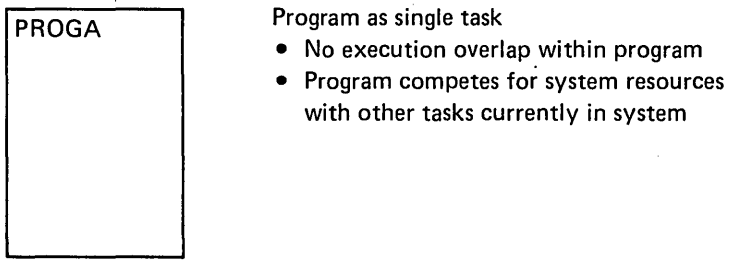


Figure 1. Single task program structure

Figure 2 on page 35 is an example of the type of application that lends itself to the single task program structure. The job is basically sequential and will be waiting for operator input most of the time. Since there is no requirement for asynchronous execution of functions or I/O overlap with processing, nothing can be gained by a more complex structure.

Multiple Task Programs

Figure 3 on page 36 illustrates multitasking in a single program. PROGB, the first task in the program, is started by the system when the program is loaded, and is called the initial task. The other tasks shown in PROGB will not start until a command is executed that tells the tasks to begin. The initial task within a program commences execution when the program is loaded into storage. Initiation of additional tasks is performed by any other active task; the means of initiation are

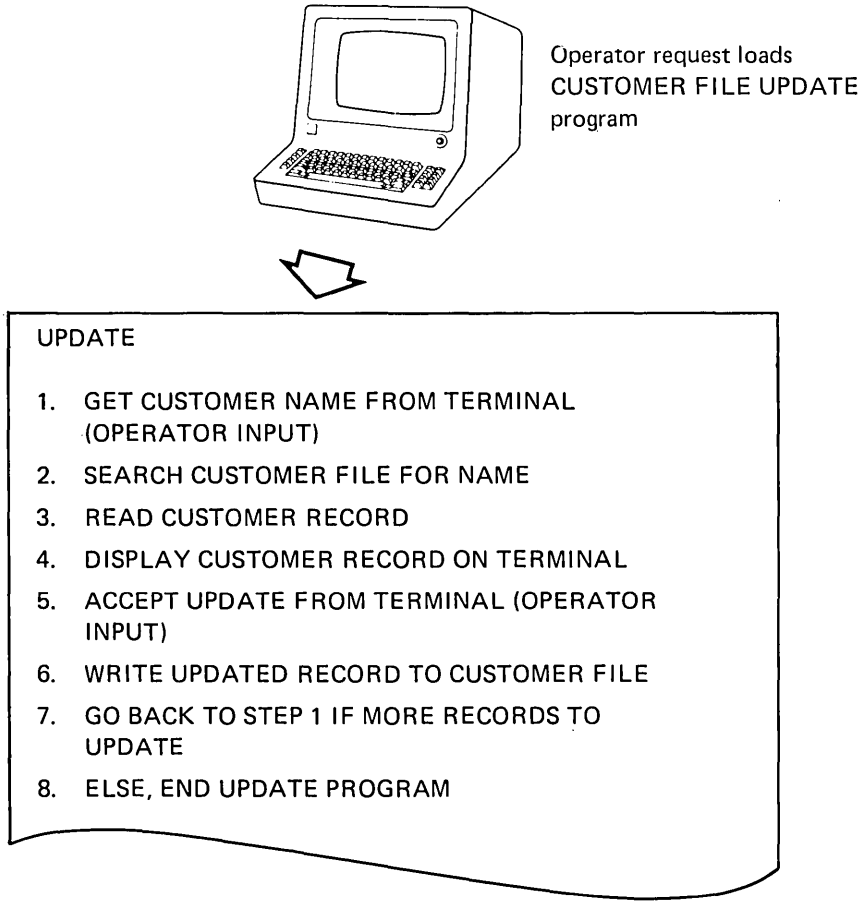


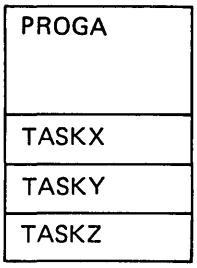
Figure 2. Single task application example

discussed in the next section.

Once in execution, all tasks within a program compete with one another for system resources, and with all other tasks active in the system. The supervisor considers each task as a discrete unit of work, and assigns processor time based on task priority, regardless of whether a task is the initial task of a program. All tasks compete equally for resources.

An Example of Multiple Programs and Multiple Tasks

Figure 4 on page 37 explains how the supervisor controls the execution of multiple tasks.



Program made up of multiple tasks
- Concurrent (asynchronous) execution of tasks within program
- Tasks compete for system resources with all other tasks currently in system

Figure 3. Multitasking program structure

The figure has the following components:

A The resident supervisor consists of:

B The instruction routine library consisting of the routines invoked by the emulator for the various application program instructions.

C Routines to service interrupts generated by I/O devices, process interrupts, timers, etc.

D I/O operation routines, general supervisor function routines, etc.

E The emulator and dispatcher examine each instruction definition in the application programs then pass control to the appropriate instruction execution routine in B to perform the specified function. This section also performs the supervisor functions WAIT/POST, ENQ/DEQ, and ATTACH/DETACH to ensure that the highest priority ready task is being executed.

F PROGRAM1, an application program, has been loaded into main storage from disk or diskette by the multiprogramming feature (not shown) of the supervisor, A. PROGRAM1 is composed of three tasks, each represented by a vertical column of rectangles. Each rectangle in a column is the string of constants generated by the assembly of an instruction. The instructions shown are for illustration purposes only.

G PROGRAM2, another application program, has also been loaded for execution. PROGRAM2 is composed of two tasks shown in a manner similar to PROGRAM1.

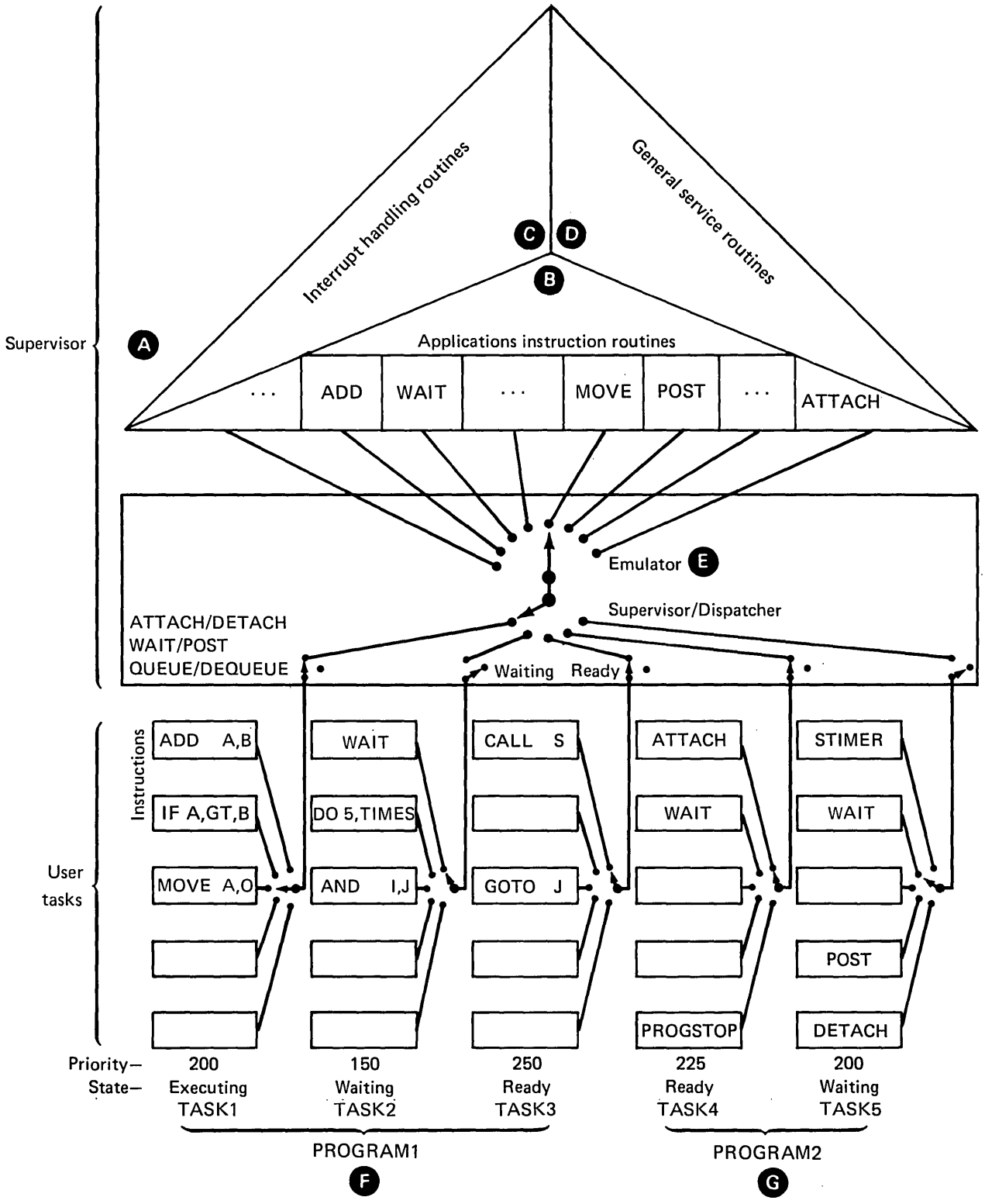


Figure 4. Executing multiple programs and multiple tasks

Figure 4 shows the following information concerning program execution:

- Task states
- Names
- Assigned task priorities, each shown at the bottom of each task.

TASK1 is currently executing (active); only one task at a time can execute. TASK1 has a priority of 200, which is higher than the priorities of the other two tasks READY for execution --TASK3 with a priority of 250 and TASK4 with a priority of 225. Priorities allowed are 1 through 510, with 1 being the highest priority.

TASK1 is executing a MOVE instruction. The emulator has decoded the instruction and passed control to the MOVE routine within the supervisor. The dispatcher will allow TASK1 to execute until one of the following occurs:

- A task of higher priority becomes ready due to the occurrence of some event such as an external process interrupt or expiration of a time interval.
- The task executing relinquishes control by issuing an instruction such as WAIT or DETACH, or by beginning an I/O operation.
- The task executing is canceled or a program check occurs.

TASK2 is in wait state and not available for dispatching as the active task until the event for which it is waiting occurs.

TASK3 and TASK4 are ready for execution but have not been dispatched since they are of a lower priority than the active task, TASK1.

TASK5 is currently in wait state, waiting for the expiration of a time interval. When the time interval expires, TASK5 will be placed in the ready state. However, it will not become the active task if TASK1 is still executing.

When two tasks of equal priority are ready for execution, a first-in-first-out situation exists and the first task to become ready will execute until it relinquishes control. Then the second task will gain control and execute.

The possible task states are:

Inactive	Task is detached or is not yet attached
Waiting	Task is waiting for the occurrence of an event
Ready	Task is ready, but is not the highest priority task
Active	Task is attached and is the highest priority task on its level
Executing	Task is using the processor

A program can consist of one or more tasks. Normally, a program will consist of only one task unless its operation requires simultaneously active, independent functions (tasks).

Multiple Program Structure

An application program consists of a collection of one or more tasks. After an application program is prepared for execution, it is stored under a unique name on disk or diskette. A terminal operator can request that a program be loaded into storage and placed in execution by entering a request for the supervisor load utility ($L) and specifying the program name.

Programs can also be loaded by executing a LOAD instruction in another program that is already in execution. When the supervisor receives a request to load a program, either from a terminal or a task already in execution, the supervisor:

1. Finds the program on disk or diskette
2. Finds a section of unused storage large enough to accommodate the program
3. Loads the program from disk or diskette
4. Opens any data sets or program overlays
5. Relocates the program into the unused area
6. Starts the program's primary task

Programs are dynamically relocated into storage as load requests are received, so the size and structure of your programs can have an effect on system throughput.

Any program can be loaded by the operator, another program, or an overlay.

Overlays

A program can have several overlay programs that utilize the same overlay area at different times during execution. An application that needs to be loaded quickly when requested can benefit by being implemented as an overlay.

You can specify a program as an overlay in a main program with the Event Driven Language PROGRAM statement. At primary program load time, sufficient storage is reserved within the primary program for the largest overlay. Overlay loads can thus be performed quickly by the system because the storage has already been preallocated.

In Figure 5, the application is split into separate programs. PHASE1, the primary program, loads the overlay programs (PHASE2, PHASE3, and PHASE4) as requested. When PHASE1 is loaded, the loader recognizes that overlay programs are referenced. The loader looks at each program that is designated as an overlay and reserves enough storage to hold PHASE1 plus the largest overlay program (PHASE3) as shown in Figure 6 on page 41.

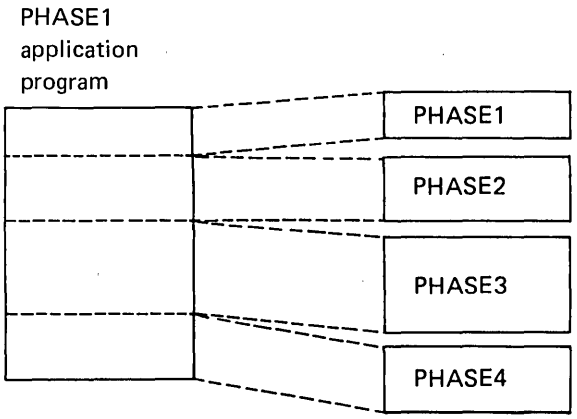


Figure 5. Program overlays

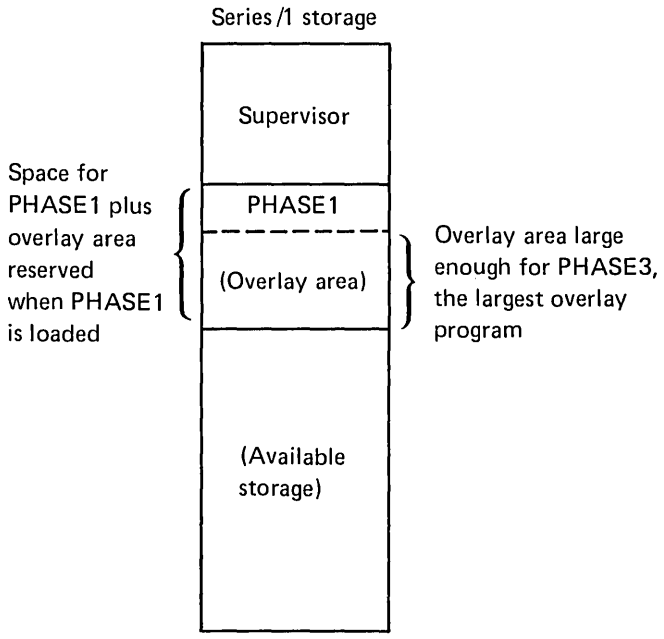


Figure 6. Program overlays in Series/1 storage

PHASE2 is loaded when PHASE1 issues a LOAD statement referencing PHASE2. The system loads PHASE2 into the overlay area already reserved and PHASE2 starts executing. There is no contention for the storage with other applications that are waiting to be loaded in the overlay area because the overlay area is reserved for the exclusive use of PHASE1 overlay programs.

As each overlay program completes execution, PHASE1 loads the next overlay, until all required programs have run. When PHASE1 terminates execution, the storage reserved for PHASE1 and its overlays is released.

Program Loading and Task Execution

Programs are ready for execution when they are loaded into storage from disk or diskette. A program will not execute immediately unless its primary task has a higher priority than the currently executing task. Programs are loaded when you issue the $L operator command or when a LOAD instruction is executed in a task that is in execution. In either case, the program to

be loaded is referenced by the name under which it is stored on disk or diskette.

Multiple copies of the same program can be in storage and active at the same time. A program can be loaded from one or more terminals, from one or more programs already executing, or as an overlay by an executing program.

STORAGE MANAGEMENT

With the address relocation translator feature on your Series/1, the supervisor can provide storage management support for main storage sizes up to 256K bytes (1K = 1024 bytes).

Storage and Partitions

You can divide storage not used by the supervisor into partitions. A partition is a contiguous fixed length area of storage which can be used for the execution of disk or diskette resident programs. You can define up to eight partitions for the 4955, two partitions for the 4952, and one partition for the 4953.

Each partition can contain more than one program simultaneously, within the limits of the storage assigned to each partition. Each partition must be defined as an address space in some multiple of 2K bytes.

You can specify the storage size of the processor, the number of partitions, the maximum number of programs allowed in each partition concurrently, and the storage to be assigned to each partition.

The supervisor is always located in partition one. The storage you use in partition one is limited to 64K bytes minus the number of bytes occupied by the supervisor. All other partitions have a maximum size of 64K bytes, within the limits imposed by the amount of storage available. It is also possible to logically prefix part of the supervisor onto each partition using the mapping capabilities of the address relocation feature. However, this option limits the size of each of your partitions to 64K bytes minus the size of the prefixed portion of the supervisor.

SYSTEM CONTROL BLOCKS

System control blocks are used by the Event Driven Executive and are not to be altered in your application programs. Exceptions to this rule can be found in the Language Reference and Communications and Terminal Applications Guide.

CHAPTER 3. DATA MANAGEMENT

This chapter discusses data management concepts and procedures. Among the topics covered are:

- I/O functions
- Direct access storage devices
- Disk and diskette functions
- Tape functions
- Data set naming conventions
- Storage capacities
- Volume and library definitions

The chapter first presents specifics of data management, then discusses the utilities with which you can modify data, and concludes with system concepts and application requirements.

I/O FUNCTIONS

Terminal Support

Terminal support is designed to be device independent. With few exceptions, you need not be concerned with the type of device. The same sequence of terminal output instructions, for instance, can be used to print data on a matrix or line printer, on a locally attached Teletype device, on a remote terminal, or to display the data on an electronic display screen device.

Terminals are defined to the system during system generation.

The high degree of device independence is achieved in part by treating all terminals as though they were line printers that differ only in their page sizes (forms length) and margin settings. The multi-terminal support provides instructions allowing interactive communications between you and your application programs. Terminals supported are the 4978 and 4979 display stations, the IBM 3101 Display Terminal, 4973 and 4974 printers, 2741 Communications Terminal, other Series/1 computers, the 5100 and 5110, the Tektronix #4013[3] DI/DO Parallel Interface terminals, and teletypewriters and equivalent devices.

[3] Registered trademark of the Tektronix Corporation.

The terminal used by a program is the same terminal that was used to invoke the program. Therefore, the terminal assigned can vary from one program invocation to the next, with no change to the application program.

Terminals are referenced by symbolic name and accessed through various instructions. Forms and screen format control can be dynamically changed within your program and the 4978/4979 display screen can be copied to a hard copy terminal at any point in the program.

Terminals with Special Control Characters

Terminals that have special control characters and/or hardware capabilities, such as graphics functions, are easily controlled by the terminal instructions.

Graphic terminals which perform point-to-point vector drawing and comply with the screen coordinate algorithm are supported by the terminal instructions and a set of graphic control instructions.

Terminal I/O

When a program is loaded from a terminal, that terminal is dynamically designated by the system as the terminal to be used by terminal I/O instructions in the program. Each terminal I/O instruction has exclusive use of the terminal while executing, and extended control can be requested for multiple I/O operations.

If more than one task is using the terminal, terminal operations from different tasks could become interspersed. When this is not desirable, you can reserve the terminal for the exclusive use of a task, thereby preventing other tasks from using the terminal until the task releases it. You can gain exclusive control of any named terminal in the system.

Three symbolic terminal names are used by the supervisor for system utility programs:

$SYSLOG Names the system logging device or operator station, and must be defined in every system. In the starter supervisor, $SYSLOG defines a 4978 display station.

$SYSLOGA Names the alternate system logging device. If unrecoverable errors prevent use of $SYSLOG, the system will use the $SYSLOGA terminal as the system logging device/operator station. If defined, this device

should be a terminal with keyboard capability, not just a printer. The starter supervisor defines the $SYSLOGA terminal as a teletypewriter device.

$SYSPRTR Names the system printer. If defined, the hard copy output from some system programs will be directed to this device. The starter supervisor defines a 4974 printer as the $SYSPRTR device.

Terminal Definition and Control Functions

ATTNLIST Provides entry to one or more of your asynchronous attention interrupt handling routines. Produces a list of command names that you have defined and their associated entry points

DEQT Releases a terminal from exclusive use

ENQT Acquires exclusive access to a terminal

ERASE Clears designated portions of STATIC type screens

FORMAT Describes the type of conversion to be performed on data

GETEDIT Moves data from a terminal, converting it according to a FORMAT specification

GETVALUE Reads one or more integer values that are entered by the terminal operator

IOCB Describes the attributes of a terminal

PRINDATE Prints the date on the terminal

PRINTNUM Converts a floating-point variable or integer variable to printable form, and writes it on the terminal with an optional format specification

PRINTEXT Writes an alphameric text string to a terminal

PRINTIME Prints the time of day on the terminal

PUTEDIT Moves data from storage to a terminal, converting it according to a FORMAT specification

QUESTION Prints a message and queries the operator for a Y (yes) or N (no)

RDCURSOR Reads the current cursor position

READTEXT Reads an alphameric text string from the terminal

TERMCTRL Provides support for special terminal control features, some of which are device dependent

TERMINAL Defines each input or output terminal attached to the system, including printers. Use this statement only during system generation.

WAIT Causes the issuing task to wait until the operator depresses an ENTER key or a PF key. Specified in association with the KEY option

Sensor I/O

Sensor I/O is used in a variety of application areas, including process control, laboratory automation, and plant automation. Sensor I/O devices available on the Series/1 are as follows:

Digital Input/Output

A unit of digital sensor I/O is a physical group of sixteen contiguous points. The entire group of sixteen points is accessed as a unit on the I/O instruction level: programming support allows logical access down to the single point level.

Digital input (DI) is usually used to acquire information from instruments which present binary encoded output, or to monitor contact/switch status (open/closed). Digital output (DO) is used to control electrically operated devices through closing relay contacts, such as pulsing stepping motors.

Process interrupt (PI) is a special form of digital input. If a point of digital input changes state, and then changes state again, without an intervening READ operation from the program, the status change will be undetected. With process interrupt, a point changing from the off state to on generates a hardware interrupt, which is then routed through software support to an interrupt servicing application program that can respond to the external event which caused the interrupt. Process interrupt is often used for monitoring critical or alarm conditions, which must be serviced quickly, the occurrence of which must not go undetected.

Analog Input/Output

A physical unit of analog input (AI) can be a group of eight points or sixteen points, depending on the type. Analog output (AO) is installed in groups of two points. Each point of analog input or analog output is accessed separately.

Analog input is used to monitor devices that produce output voltages proportional to the physical variable or process being measured. Examples include laboratory instruments, strain gauges, temperature sensors, or other non-digitizing instruments. Digital input was described as monitoring an on/off status; only two conditions were possible. With analog input, the information is carried in the amplitude of the voltage sensed rather than in its presence or absence.

The starter supervisor contains no support for sensor I/O. You must do a tailored system generation to include the required support modules in your own supervisor.

Figure 7 on page 50 shows how sensor devices are connected to a Series/1 through the 4982 sensor I/O unit. The devices (DI, DO, PI, AO, and AI) attach to a controller, which in turn attaches to the Series/1. The sensor I/O attachment (controller), and each of the devices attaching to it, have unique hardware addresses. In this figure, the physical connections are there, and the hardware addresses are assigned (wired in), but the starter supervisor in storage lacks the support necessary to operate the devices.

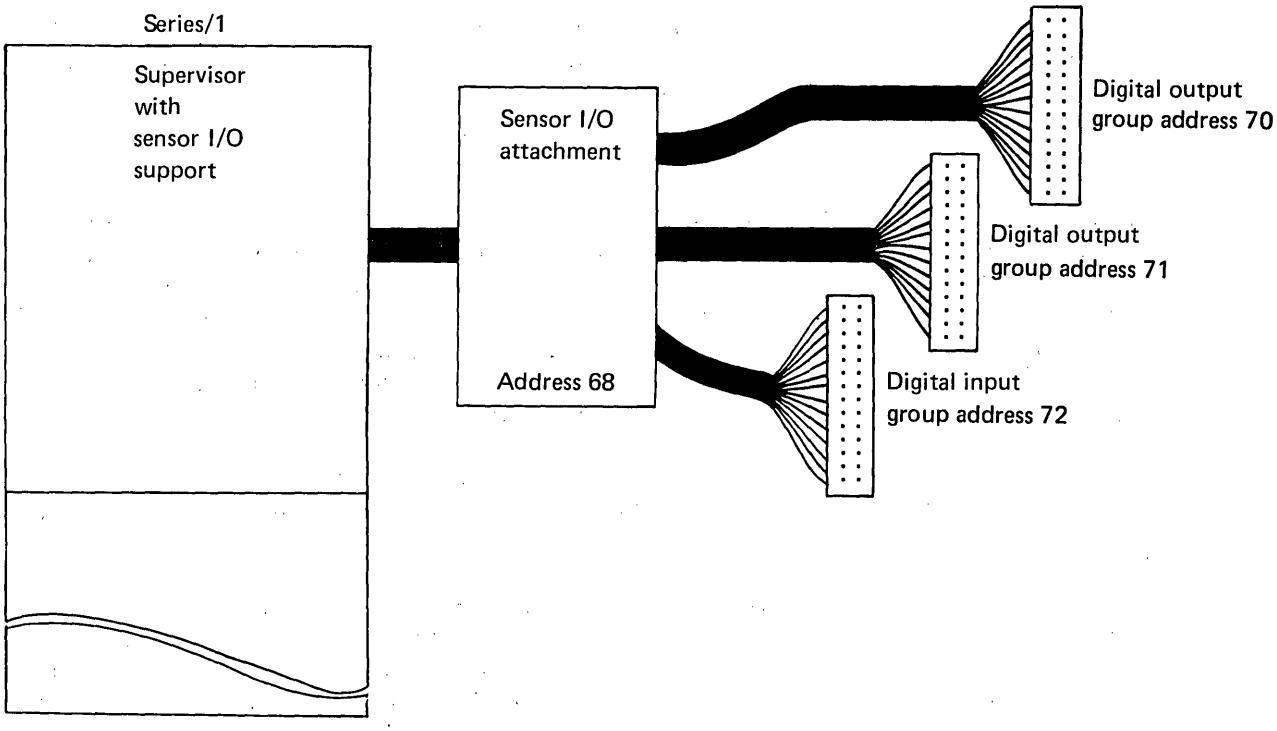


Figure 7. Sensor device connections

Building a tailored supervisor involves the assembly of a series of system configuration statements that reflect the I/O configuration you wish to support. For more information on system configuration statements, refer to "Chapter 6. System Configuration" on page 75. When programs reference these devices, they use symbolic references, rather than actual addresses. The I/O definition statement (IODEF) establishes the logical link between the addresses defined in the supervisor, and the symbols used to read from and write to the devices at those addresses from an application program.

All sensor-based input/output operations are performed by executing a Sensor Based I/O (SBIO) instruction. The type of operation is determined by the type of device referenced in the instruction. For more information on the SBIO statement, refer to Language Reference. The symbolic reference to a logical device in the SBIO statement is linked to the definition in the IODEF statement, which relates that device to the hardware address specified by the system configuration statement at system generation time.

Sensor Based I/O Definition and Control Statements

IODEF An Event Driven Language instruction that establishes the logical link and definition of subgroups of sensor based I/O devices defined in the supervisor and the symbols used to read from and write to the subgroups.

SBIO An Event Driven Language instruction that performs analog and digital input/output operations.

SENSORIO A system configuration statement that defines the hardware device addresses for the supervisor.

SPECPIRT An Event Driven Language instruction that provides return linkage from the special process interrupt routines specified in the IODEF instruction.

THE EXIO INTERFACE

The EXIO interface permits you to directly control the operation of Series/1 devices. You supply the immediate device control blocks (IDCBs) and Device Control Blocks (DCBs) that are required for I/O operations to be performed. This allows close control of performance and response time. Any device meeting the standard I/O interface, attached to the Series/1 can be controlled through the EXIO interface. To use the EXIO interface, the programmer should be familiar with assembler language coding, I/O programming in general, and the devices involved in the I/O operations.

Definition and Control Statements

The instructions and statements necessary to use EXIO are:

DCB An Event Driven Language instruction that creates a Device Control Block

EXIO An Event Driven Language instruction that requests execution of an I/O command

EXOPEN An Event Driven Language instruction that specifies the device addresses to which EXIO commands will be directed

EXIODEV A system configuration statement that defines the devices to be supported via the EXIO interface

IDCB An Event Driven Language instruction that creates an Immediate Device Control Block

DIRECT ACCESS STORAGE ORGANIZATION

The following definitions are used by the Event Driven Executive.

Sector

The smallest addressable unit of storage on a disk or diskette is known as a sector (or a record on the 4963). Sectors on a 4962 disk and records on a 4963 contain 256 bytes of data. Therefore, a 4962 sector and a 4963 record are equivalent to a record. Diskette sectors can be either 128, 256, 512, or 1024 bytes long. However, in the Event Driven Executive system the IBM standard for information interchange, 128 bytes, has been adopted. Therefore, two diskette sectors equal one record. This is handled within the system and you refer only to 256 byte records.

Volume

A volume is a physical direct access storage device, or a subset of a physical direct access storage device. You can assign a name, or volume label, to each volume. The volume label must be 1-6 alphameric characters. A volume begins on a cylinder boundary and contains an integral number of cylinders. The maximum volume size in records is 32,767. A fixed head area, if it exists, is defined as another volume.

Volumes containing programs or macros are usually called libraries. A library is the collection of data and programs and the directory used to access them.

Directory

A directory is a series of contiguous records at the beginning of a library. The directory describes the library contents in terms of allocated data sets, programs, and free space.

Data Set

A data set is a group of contiguous records which have been allocated -- reserved and assigned collectively. The data set name consists of 1 to 8 characters. No special restrictions exist within the system for valid names but the system utility programs require a name consisting of alphameric characters for access and allocation.

A data set, or member of a library, can contain either data or an executable program. These data sets may also be partitioned data sets when allocated with the $PDS utility. $PDS defines members as a group of contiguous records within the partitioned data set which have been allocated and assigned a name.

Record

A record is the basic unit of direct access storage available to an application program. The records are fixed, unblocked, and 256 bytes long. Data set records are numbered beginning with one.

Access

Data set access routines are available within the supervisor for multiple diskettes and disks, with or without fixed head features. File access is either sequential or direct. Multiple logical volumes can be created on any physical disk drive.

Note: A diskette is always a single volume.

When a program is first loaded for execution, all of the defined data sets are opened for access (reading or writing), beginning with record number 1. Sequential and random access operations can be mixed. For instance, if five sequential READ instructions of one record each have been issued to a data set, then the next sequential operation will involve record number six. A random access READ could be issued for some other record, say record 23, and the next sequential operation would still take place with record 6.

Volumes on disk devices are defined during system generation, using the DISK configuration statement. For further information on the DISK configuration statement, refer to "Chapter 6. System Configuration" on page 75. Diskette volumes are defined with the utility program $INITDSK. Refer to the Utilities, Operator Commands, Program Preparation, Messages and Codes for a discussion of $INITDSK.

DISK AND DISKETTE FUNCTIONS

The following instructions are provided for disk and diskette functions:

DISK Defines each direct access storage device and the volumes it contains. Use the DISK statement only during system generation

NOTE Saves the next record pointer in a program location that you define

POINT Sets the next record pointer from a program variable that you define

READ Transfers one or more 256-byte records from disk or diskette to the requester's storage

WRITE Transfers one or more 256-byte records from the requester's storage to disk or diskette

The DSCB statement generates a data set control block (DSCB) which provides information required to access a data set.

The $DISKUT3 data management utility provides execution time support that allows you to allocate, delete, open, and rename data sets. It also allows you to release space from a data set.

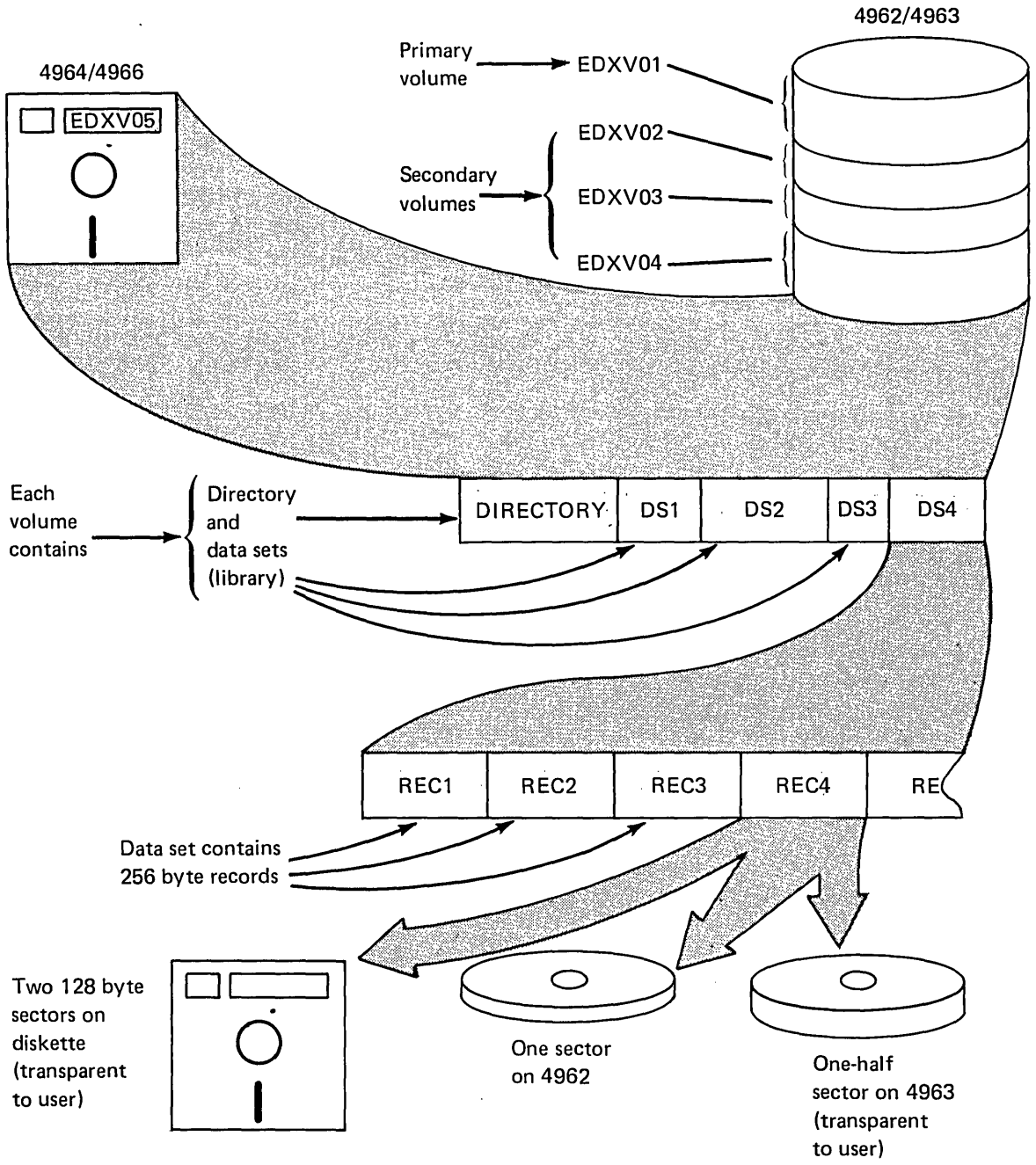


Figure 8. DASD logical organization

In addition to the single primary volume required for each disk storage unit, as many secondary volumes as required can be defined (within the physical limits of the device). As with primary volumes, secondary volumes are defined at system generation using DISK configuration statements and are initialized by the $INITDSK utility.

Volumes can also exist on diskette. Each diskette has a separate volume occupying the entire diskette. Diskette volumes are also initialized using the $INITDSK utility.

TAPE FUNCTIONS (VERSION 2 ONLY)

The following instructions are provided for tape functions:

TAPE Defines each tape device to be used on the system. Use the TAPE statement only during system generation

NOTE Saves the next record pointer in a program location that you define

POINT Sets the next record pointer from a program variable that you define

READ Transfers records of 18 to 32767 bytes in length from tape to the requester's storage

WRITE Transfers records 18 to 32766 bytes in length from the requester's storage to tape

CONTROL Allows physical manipulation, such as; forward or backward spacing of records or files and the writing of tapemarks.

The DSCB statement generates a data set control block (DSCB) which provides information required to access a data set.

The $TAPEUT1 utility allows you to allocate tape data sets and copy data sets or volumes from disk or diskette to tape, tape to disk or diskette, or tape to tape. The utility also allows you to change tape attributes.

For information on tape organization see "Chapter 11. Tape Organization" on page 233.

DATA SET NAMING CONVENTIONS

Data sets are specified for system use at one of four times:

1. When coding a PROGRAM instruction and completing the DS= operand
2. When coding a LOAD instruction and completing the DS= operand

3. When a program is loaded by the $L operator command

4. During execution of some system utility programs

A general data set specification consists of two parts:

1. The data set name (dsname)

2. An optional volume label (volume) which specifies the volume on which the data set resides

The format for a data set specification is:

```
dsname,volume
```

The volume specification is optional and if not specified, the system assumes that the target data set resides on the primary volume on the direct access device from which the system was IPLed.

dsname An alphameric character string of eight characters. When fewer than eight characters are specified, blanks are added to the string.

volume An alphameric character string of six characters. To locate the volume on a disk, it must have been defined in the VOLSER= parameter of a DISK configuration statement in the system I/O definition. To locate the volume on a diskette or tape, the TAPE or DISK statement must be in the system I/O definition and the volume name loaded into the system by issuing the operator command $VARYON, specifying the diskette or tape device address. The diskette must have been initialized by $INITDSK. Tapes must be initialized by the $TAPEUT1 utility. When fewer than six characters are specified, blanks are added to the right to complete the string.

Two special data set names are known to the system and must be used with care:

$$EDXVOL Used to obtain absolute record reference to an entire volume on disk or diskette.

$$EDXLIB Used to obtain absolute record reference to the beginning of the volume directory on disk or diskette within a volume.

STORAGE CAPACITIES

Disk/Diskette

The following table summarizes storage capacities of the various Series/1 direct access storage devices.

Device	Storage capacity (records)	Cyl/dev	Logical rcds/trk	Trk/cyl	Volume max (cyls)
Single-sided (type 1) diskette	949	77*	13	1	73
Double-sided (type 2) diskette	1924	77*	13	2	74
4962 disk		303**			
-1	36120		60	2	273
-1F	36600		60	2	273
-2	36120		60	2	273
-2F	36600		60	2	273
-3	54180		60	3	182
-4	54180		60	3	182
4963 disk		360***			
-23	92160		64	4	128
-29	114560		64	5	102
-58	229632		64	10	51
-64	252032		64	11	46

* 73 cylinders are available for data (001-073) on type 1 diskettes. 74 cylinders are available for data (001-074) on type 2 diskettes. On both types, 2 cylinders are reserved for alternate tracks and 1 cylinder is reserved for IPL and volume identification.

** 301 cylinders are available for data (000, 002-301); cylinder 001 is reserved for alternate sector assignments; 302 is reserved for CE use.

*** 358 cylinders are available for data (0-357), while cylinder 358 is reserved for alternate sectors and cylinder 359 is reserved for CE use.

Tape

The following figure summarizes approximate storage capacities for 800, 1200, and 2400 foot tape volumes. The Event Driven Executive 4969 Magnetic Tape Subsystem supports 18 to 32,767 byte records. The record lengths depicted are used for illustrative purposes only. These estimates are approximate and are based on the hardware specifications for the 4969 tape drive. (Refer to IBM Series/1 4969 Magnetic Tape Subsystem Description, (GA34-0087) for more detailed information.) Use these estimates to calculate the size of the reel or volume needed to fulfill your requirements.

TAPE CAPACITY		
	NUMBER OF RECORDS	
	800 BPI	1600 BPI
Tape length		
800 feet		
256 byte records	10,078	11,411
1024 byte records	5,019	7,265
8192 byte records	882	1,654
1200 feet		
256 byte records	15,118	17,117
1024 byte records	7,529	10,898
8192 byte records	1,324	2,482
2400 feet		
256 byte records	30,236	34,234
1024 byte records	15,058	21,797
8192 byte records	2,648	4,964

DEFINING VOLUMES

Volumes and libraries are defined at system configuration time. The system handles disks, diskettes and tapes differently, as described below.

Diskette

One, and only one, volume is defined for each 4964 diskette drive to be known to the supervisor. However, for the 4966 Diskette Magazine Unit, up to 23 diskette volumes can be mounted. The diskette mounted in slot one is considered the primary volume; the rest of the diskettes are secondary volumes. Because diskettes are mountable storage media, the actual volume label, library origin, and library size must be determined by the system each time a new diskette is mounted. This is accomplished through the operator command $VARYON. Volume labels are recorded on each diskette in accordance with IBM Standards for Information Interchange.

Disk

At least one volume is defined for each 4962 or 4963 disk drive to be known to the supervisor. Because volume size is limited to 32,767 records, several volumes must be defined per disk to be able to use the entire storage capacity.

The first defined volume, the primary volume, has its origin at cylinder zero. Because certain records and cylinders are reserved for system use, the library associated with a primary volume cannot begin with the first record.

The library origin of additional volumes, called secondary volumes, can be the first record in the volume.

For example, addressability of an entire 4962 disk could be established with the following definitions:

	Volume origin (cylinder #)	Volume size (cylinders)	Library origin (record #)
Primary	0	153	241 (cylinder 2)
Secondary	153	150	1

Volume labels for all disk volumes are maintained within the supervisor and are not physically recorded on the device.

The following table summarizes the library origin for primary and secondary volumes.

	Library origin	
	Primary volume	Secondary volume
Single-sided Diskette	14	N/A
Double-sided Diskette	27	N/A
4962 Disk Models		
1, 2	241	1
1F, 2F	241	1
3, 4	361	1
4963 Disk Models		
29	129	1
23	129	1
64	129	1
58	129	1
N/A means not applicable.		

Figure 9. Library origins

The fixed-head area of a 4962-1F, 4962-2F, 4963-23, or 4963-58 is automatically defined as a secondary volume by the DISK configuration statement; you are required to specify an associated volume label. Use the FHVOL parameter of the DISK configuration statement to assign the volume label. The fixed head volumes contain 480 records on the 4962 and 512 records on the 4963. The library origin on both devices is record one.

A fixed head volume is treated in a special manner:

- During the disk initialization part of IPL, each data record is read and rewritten to reduce the probability of errors.
- If the IPL device is a disk with fixed heads, the system, during the loader initialization part of IPL, searches the fixed-head volume for the transient loader routine $LOADER. If it is found, it is used for program loading, thereby providing the fastest and most constant loader performance. If $LOADER is not present, the system attempts to

locate it in the IPL device's primary volume.

Tape (Version 2 only)

One, and only one, volume is defined for each tape drive known to the supervisor. Tape volumes are not defined at system configuration time. Because tapes are a mountable storage medium, the actual volume label is determined by the system each time a tape is mounted. The operator command $VARYON causes a tape to be mounted. For more information on tape labels and volumes, refer to "Chapter 11. Tape Organization" on page 233 .

OPERATOR COMMANDS

When the ATTN key on a terminal is pressed, the system responds with the prompt character, >.

The operator commands that can be entered are:

$A	Displays the names and load points of all programs that are active within the partition to which the requesting terminal is currently assigned
$B	Completely erases (blanks) all protected and unprotected areas on the screen of the requesting terminal
$C	Cancels a program and frees the storage that it occupied
$CP	Changes a terminal's partition assignment
$D	Displays the contents of storage in hexadecimal
$E	Advances the system printer to the top of form (performs a page eject)
$L	Loads a program
$P	Patches storage locations
$T	Sets date and time for the 24-hour system clock/calendar. It can be used only from terminals named $SYSLOG or $SYSLOGA. Operator input is not validated by the supervisor.
$VARYOFF	Places a disk, diskette, or tape in offline status
$VARYON	Places a disk, diskette, or tape in online status
$W	Displays the 24-hour clock and the date

You may add attention interrupt handling routines by using the ATTNLIST statement. When you code the statement, you provide your command name and its address. This command name may then be entered whenever the system issues the > prompt.

UTILITIES

The utilities provide productivity aids for Series/1 application program development, program maintenance, and distributed processing functions with a host System/370. These utilities are independent program load modules capable of running concurrently with other application programs or utilities. Types of utilities are:

- Data Management utilities
- Communication utilities.
- Text editing utilities
- Diagnostic utilities
- Graphics utilities
- Terminal utilities
- Program preparation utilities

Data Management utilities

Data Management utilities can define, patch, dump, delete, rename, compress, copy, and initialize data sets. The following Data Management utilities are available:

$COMPRES Compresses libraries

$COPY Copies disk or diskette data sets or volumes

$COPYUT1 Copies disk or diskette data sets, dynamically allocating the receiving data sets

$DASDI Initializes, formats, and verifies disks or diskettes

$DISKUT1 Allocates and deletes disk or diskette data sets; lists directory data

$DISKUT2 Patches and dumps disk or diskette data sets; lists the hardware error log

$IAMUT1 Builds and maintains Indexed Access Method data sets

$INITDSK Initializes and verifies a direct access storage volume

$MOVEVOL Transfers volumes of data between systems and creates backup copies of an online data base

$PDS Manages partitioned data sets

$TAPEUT1 (Version 2 only) Prints tape records, copies data sets to or from tape, copies tape to tape, initializes tapes, dumps and restores disk devices on tape, and runs a tape exerciser as a hardware/software test

Communication Utilities

Communication utilities provide options for communicating with another processor and diagnostic aids for troubleshooting teleprocessing lines. Two facilities are available to communicate with a System/360 or System/370:

- The Host Communications Facility which requires the Host Communications Facility IUP on the System/360 or System/370 and provides direct two-directional transfer between host direct access data sets and Series/1 storage. Also, a job submission capability allows you, through a terminal on the Series/1, to invoke batch program execution on the System/360 or System/370 host system. A point-to-point leased line and the BSC Single Line feature #2074 is required for Host Communications Facility operation.
- Host communications similar to IBM 2780/3780 remote job entry (RJE) capabilities to host RJE systems. (Refer to the Communications and Terminal Applications Guide for more information.) Data streams including either transparent or non-transparent data can be submitted to the host, as can single card image commands. Printed and/or punched output from the host can be stored in disk or diskette data sets or printed on any supported terminal attached to the Series/1.

These utilities are:

$BSCTRCE Traces I/O activities on a binary synchronous communication line.

$BSCUT1 Formats binary synchronous trace files for printing.

$BSCUT2 Checks out the binary synchronous communications access method.

$HCFUT1 Uses the Host Communications Facility on the Series/1 to interact with the Host Communication Facility on the System/370.

$PRT2780 Prints spool records produced by $RJE2780.

$PRT3780 Prints spool records produced by $RJE3780.

$RJE2780 Allows communication between a System/360 or System/370 and a Series/1 by simulating an IBM 2780.

$RJE3780 Allows communication between a System/360 or System/370 and a Series/1 by simulating an IBM 3780.

$RMU Allows a user written HOST program to communicate with a remote Series/1 over a binary synchronous communications line. (Version 2 only)

Text Editing Utilities

The text editing utilities provide facilities for entering and editing source programs.

$EDIT1 A line editor which allows you to enter and edit source programs while other programs are executing. $EDIT1 provides commands for data communication using the Host Communications Facility IUP on the System/370 so program preparation can be controlled from a Series/1 terminal.

$EDIT1N A line editor which allows you to enter and edit source programs. It is the same as $EDIT1 except that it edits data that resides on Series/1 direct access volumes.

$FSEDIT A full screen editor for entering and editing source programs using a 4978 or 4979 display terminal. The source may be located either on the Series/1 or on a host processor.

The text editing utilities provide you with a line of the System/370 OS/TSO text editing facility in the editors. The full screen editor provides a subset of functions similar to the System/370 Structured Programming Facility (SPF) full screen editor.

In the full screen editor, functions such as browse, edit, and merge are provided. Additional commands are offered in both editors for read/write from or to source data sets on either the local Series/1 or a remote host System/370 with the Host Communications Facility IUP. This allows full control of program development from a Series/1 terminal. Full-screen edit-

ing is limited to the 4978 and 4979 display terminals.

Diagnostic Utilities

The following diagnostic utilities are available:

$DEBUG An interactive program debugging aid

$DUMP Formats and displays the data saved by $TRAP on an error condition

$IOTEST Performs the following functions:

- Tests the operation of sensor based I/O features
- Lists the hardware configuration of the Series/1
- Lists the devices supported by the supervisor
- Lists volume information

$LOG Logs I/O errors into a data set

$TRAP Intercepts certain class interrupts and records the environment on a disk or diskette data set

$DEBUG allows you to stop, modify, trace, and restart an application program with no impact on the execution of other programs.

The sensor I/O test utility ($IOTEST) allows you to exercise the sensor I/O (AI, AO, DI, DO, PI) devices on a Series/1. You can perform functions such as read/write digital input/output, write digital output with selected time intervals, and read/write analog. During any exercising function, which can be selected via a terminal command, trace printing is done to the terminal for each exercising option.

Graphics Utilities

The following graphics utilities are available:

$DICOMP Generates and modifies displays using an online composer

$DIINTR Uses an interpreter to display and process the data base

$DIUTIL Maintains the resulting data base

Graphics utilities enable you to generate, maintain, and display two- and three-dimensional fixed graphic backgrounds, and to store them in files. Access to these background files is available from your application programs. Realtime data can also be superimposed over the displayed fixed graphic backgrounds.

Terminal Utilities

The following terminal utilities are available:

$FONT Creates and modifies character image tables for 4978 display terminals

$IMAGE Defines formatted screen images for 4978/4979 display terminals

$PFMAP Displays program function key assignments

$TERMUT1 Alters logical device names, address assignments, or terminal configurations

$TERMUT2 Loads control and image stores for a 4978 display

$TERMUT3 Sends messages from one terminal to another

The screen format builder utility ($IMAGE) enables you to design formatted screen images for static screens on the 4978/4979 Display Stations. These images are generated interactively on a terminal and can be saved in disk or diskette data sets for later retrieval and use by application programs. Images previously stored on the disk or diskette can be retrieved and modified.

Program Preparation Utilities

The following program preparation utilities are available:

$COBOL Compiles COBOL Language programs

$EDXASM Assembles Event Driven Language programs

$EDXLIST Prints $EDXASM listings

$LINK Link edits object modules

$PL/I Compiles PL/I Language programs (Version 2 only)

$PREFIND A prefind capability for data sets and overlay programs to shorten program loading time

$S1ASM Assembles Series/1 assembler language and Event Driven Language programs

$UPDATE Converts an object module into an executable program

$UPDATEH Converts a host object module into an executable program

The Job Stream Processor Utility

The job stream processor utility can be used to invoke a predefined sequence of program preparation utilities and pass parameters to them. $JOBUTIL can be invoked by the Session Manager.

CHAPTER 5. PROGRAM PREPARATION FACILITY

The Program Preparation Facility consists of an Event Driven Language assembler, a compilation listing program, and a linkage editor. The Program Preparation Facility has the following features:

- Program Preparation Facility programs can run concurrently with other programs.
- Multiple copies of the assembler, listing program, and the linkage editor can run concurrently.
- Source programs can be stored on disk or diskette.
- All references to programs and files are by symbolic names.

The Program Preparation utilities can be invoked from any terminal and loaded into any available storage. Although any of the Program Preparation Facility programs can execute from a diskette based system, the limitations of file space and access speed severely restrict the program preparation capability. A disk-based system is recommended for an efficient, full capability development system.

EVENT DRIVEN LANGUAGE COMPILER

The Event Driven Language assembler assembles programs written exclusively in the Event Driven Language instruction set. This includes application programs as well as supervisor system generation (definition and configuration) statements. If your program consists of Series/1 assembler language instructions or contains Event Driven Language USER exits, you must assemble the program with the Series/1 Macro Assembler.

The assembler uses a set of overlay programs which define and describe each instruction in the Event Driven Language instruction set. You can add new instructions to the assembler by writing additional overlay programs.

LINKAGE EDITOR

The output from the Series/1 Macro Assembler, the Event Driven Language assembler, the PL/I compiler, the FORTRAN compiler, or the COBOL compiler is input to the linkage editor After processing by the linkage editor, the relocatable object module must be converted to an executable program by $UPDATE. The

advantages of linkage editing are:

- Large programs can be broken into smaller segments, improving development productivity and maintainability
- Series/1 macro assembler routines can be included in the program
- Library modules, such as the Mathematical and Functional Subroutine Library or other object library routines, can be link edited with an Event Driven Language program.

It is possible to bypass the link edit step. A single program module can be assembled with the Event Driven Language compiler if all the coding is done with Event Driven Language instructions. The resulting output must be converted to an executable program by the utility $UPDATE, even if the assembled object module contains no external references. However, when using $S1ASM to assemble Event Driven Language and/or assembler programs, the resulting output must be converted by $LINK to an acceptable format for input to $UPDATE.

PART II - SYSTEM GENERATION AND CONFIGURATION

The creation of a customized supervisor is a two step process. Step 1 is a definition phase. Step 2 is the generation phase.

In step 1, you define the configuration of the system by preparing configuration statements which describe the attributes of the devices (such as disks, diskettes, and terminals) you want your system to support. You also define the number and size of the partitions that will be available in your system. Configuration statements are described in "Chapter 6. System Configuration" on page 75.

In step 2, you enter your configuration statements and assemble them. Then you modify the system-supplied INCLUDE file, $LNKCNTL, ensuring that all the support you require is built into the supervisor. The linkage editor combines the supervisor definition with the supervisor functions you selected to create a customized supervisor.

The system generation process is described in "Chapter 7. System Generation" on page 115.

SYSTEM CONFIGURATION STATEMENTS

The characteristics of your Series/1 installation are defined by configuration statements. They are used in the system generation process only.

- BSCLINE - Define a binary synchronous line
- DISK - Define direct access storage devices
- EXIODEV - Define EXIO interface devices
- HOSTCOMM - Define host communication support
- SENSORIO - Define sensor I/O devices
- SYSTEM - Define processor characteristics
- TAPE - Define tape device (Version 2 only)
- TERMINAL - Define terminals
- TIMER - Define system timer feature

BSCLINE

BSCLINE - Define a Binary Synchronous Line

BSCLINE defines the binary synchronous lines to be supported in the generated system. One BSCLINE statement is required for each line to be referenced by programs using the Binary Synchronous Communications Access Method. All BSCLINE statements must be grouped together with the last BSCLINE statement including an END=YES specification. (Refer to the Communications and Terminal Applications Guide for a description of the Binary Synchronous Communications Access Method.)

If Remote Management Utility is to be used, a BSCLINE statement is necessary.

Syntax

```
blank      BSCLINE ADDRESS=,TYPE=,RETRIES=,MC=,END=

Required: None
Defaults: ADDRESS=9,TYPE=PT,RETRIES=6,MC=NO,END=NO
Indexable: Not Applicable
```

Operands	Description
ADDRESS=	The hardware address (in hexadecimal) of the line.
TYPE=	PT - The line is a point-to-point (non-switched) line with a single remote station. The adapter should be jumpered with DTR permanently enabled. SM - The line is on a switched network and connection will be established manually by the operator. The adapter should be jumpered for switched line operation and DTR should not be permanently enabled. SA - The line is on a switched network and calls should be answered automatically by the BSC Access Method (during BSCOPEN). The adapter should be jumpered for switched line operation and DTR should not be permanently enabled.

MC - The Series/1 is the controlling station on a multipoint line. The adapter should be jumpered with DTR permanently enabled and multipoint line should not be jumpered.

MT - The Series/1 is a tributary station on a multipoint line. The adapter should be jumpered for multipoint tributary operation with DTR permanently enabled.

RETRIES= The number of attempts which should be made to recover from common error conditions before posting a permanent error.

MC= NO - The binary synchronous adapter located at the address specified in the ADDRESS= operand is either a medium speed, single line feature card or a high speed, single line feature card.

YES - The binary synchronous adapter located at the address specified in the ADDRESS= operand is part of a multi-line controller feature configuration. When generating supervisors using multi-line controller attachments, note the following:

- The character string YES must be specified. Any other character string will be equivalent to NO.
- All multi-line feature cards must start at a base address ending with either X'0' or X'8'. A BSCLINE statement must exist for the line at this base address if any of the other lines of the multi-line attachment are to be used.

END= YES, for the last BSCLINE statement in the system definition module.

Examples:

```
BSCLINE ADDRESS=28,TYPE=PT,RETRIES=10,MC=NO
BSCLINE ADDRESS=30,TYPE=SM,RETRIES=2,MC=YES,END=YES
```

DISK

DISK - Define Direct Access Storage

DISK defines the direct access storage devices and logical volumes to be supported in the generated system. All DISK statements must be grouped together. The last DISK statement must include an END=YES specification.

DISK is only needed in the system generation process. Refer to "Chapter 3. Data Management" on page 45 for a general discussion of direct access storage organization, functions, and naming conventions.

Syntax

```
blank      DISK     DEVICE=,ADDRESS=,VOLSER=,VOLORG=,
                    VOLSIZE=,VERIFY=,BASEVOL=,FHVOL=,
                    LIBORG=,END=,TASK=

Required:
    For 4964, 4966:  DEVICE=,ADDRESS=
    For 4962, 4963:  DEVICE=,ADDRESS=,VOLSER=,VOLSIZE=
    For 4962, 4963 (with fixed head): DEVICE=,ADDRESS=
                       VOLSER=,VOLSIZE,FHVOL=
Defaults: LIBORG=241 for 4962-1 or 4962-2 primary volume
          LIBORG=1 for secondary volume
          LIBORG=361 for 4962-1F or 4962-2F primary vol
          LIBORG=129 for 4963-64 or 4963-58 primary vol
          LIBORG=129 for 4963-29 or 4963-23 primary volum
          END=NO,TASK=NO,VERIFY=YES
```

Operands	Description
DEVICE=	4964, to define a 4964 Diskette Drive, or one of the following for the six models of the 4962 disk:

```
4962-1   for a 9.3  megabyte unit
4962-1F  for a 9.3  megabyte unit
                    with fixed heads
4962-2   for a 9.3  megabyte unit
                    with a diskette unit
4962-2F  for a 9.3  megabyte unit
                    with fixed heads
                    and a diskette unit
4962-3   for a 13.9 megabyte unit
4962-4   for a 13.9 megabyte unit
                    with a diskette unit
```

or

one of the following for the four models of the 4963 disk:

```
4963-29 for a 29 megabyte unit
4963-23 for a 23 megabyte unit with fixed heads
4963-64 for a 64 megabyte unit
4963-58 for a 58 megabyte unit with fixed heads
```

or

4966, to define a 4966 Diskette Magazine Unit.

Note: If 4962 or 4963 is specified, VOLSER= must be specified; LIBORG= may be specified.

ADDRESS= The hexadecimal address of the unit. This parameter is required for primary volumes only.

VOLSER= Volume label (1-6 characters) to be assigned to the unit. This operand is required if the DEVICE=4962- or DEVICE=4963- is specified. Otherwise, it is ignored.

VOLORG= The physical cylinder number of the first cylinder of the volume. Cylinder numbering begins with zero. A primary volume must begin at cylinder zero. (Refer to Figure 9 on page 58.)

VOLSIZE= The size of the volume in physical cylinders. (Refer to Figure 9 on page 58.)

VERIFY= NO, to omit the WRITE-with-verify option. YES, to cause each WRITE to be verified. YES is the default. This parameter is required for primary volumes only.

Note: You should choose the VERIFY=YES option for volumes containing critical data. This causes a slight performance degradation but improves reliability. With the YES option, each WRITE is immediately followed by a READ, thus lengthening the operation by the time it takes the unit to make one revolution.

BASEVOL= The volume label of the primary volume if a secondary volume is being defined.

FHVOL= The volume label to be assigned to the automatically generated secondary volume if the DISK statement is defining a primary volume on any 4962 or 4963 having fixed heads.

LIBORG= The origin, by number of records, of the directory on the volume. Defaults are described under 'Syntax'. This operand is only applicable when DEVICE=4962 or 4963 and is intended for special use when the initial portion of the volume is reserved for other storage.

END= YES, for the last DISK statement in the system definition module.

TASK= YES, to cause a new I/O task to be generated. This task will be used to service I/O requests for this and subsequent primary volumes until a new DISK statement with TASK=YES is encountered. NO, or omit, if a new task is not required. This operand is valid only for primary volumes and is optional.

Specifying TASK=YES on a primary volume allocates a Task Control Block that is used in servicing READ and WRITE requests for the group of devices being defined. The effect is to allow READ and WRITE requests to proceed in parallel with requests to other groups of devices. The resulting overlap may significantly improve performance when concurrent requests to different groups of devices occur. To achieve maximum flexibility and performance, you should specify TASK=YES on each primary volume. Additional storage required for each TASK=YES is 128 bytes.

Example 1: One I/O task for all direct access drives.

```
DISK  DEVICE=4964,ADDRESS=02
DISK  DEVICE=4962-1F,ADDRESS=03,VOLSER=EDX002,VOLSIZE=153, C
      FHVOL=EDX004
DISK  DEVICE=4962-1,VOLSER=EDX003,VOLORG=153,VOLSIZE=150,  C
      BASEVOL=EDX002
DISK  DEVICE=4963-23,ADDRESS=30,VOLSER=EDX005,VOLSIZE=128, C
      FHVOL=FH005
DISK  DEVICE=4963-23,VOLSER=EDX006,VOLSIZE=128,            C
      END=YES,VOLORG=128,BASEVOL=EDX005
```

Example 2: One I/O task for the two 4964s and a second I/O task for the 4962.

```
DISK  DEVICE=4964,ADDRESS=02
DISK  DEVICE=4964,ADDRESS=12
DISK  DEVICE=4962-1F,ADDRESS=03,VOLSER=EDX002,VOLSIZE=153,  C
      FHVOL=EDX004,TASK=YES
DISK  DEVICE=4962-1F,VOLSER=EDX003,VOLORG=153,VOLSIZE=150,  C
      BASEVOL=EDX002,END=YES
```

EXIODEV

EXIODEV - Define EXIO Interface Device

EXIODEV defines the devices to be supported via the EXIO interface in the generated system. All EXIODEV statements must be grouped together. The last EXIODEV statement must include an END=YES specification.

Syntax

```
blank      EXIODEV  ADDRESS=,END=,MAXDCB=,RSB=

Required: ADDRESS=
Defaults: MAXDCB=0,RSB=0,END=NO
Indexable: Not applicable
```

Operands	Description
ADDRESS=	The device address (two hexadecimal digits).
END=	Specify YES for the last EXIODEV statement in the system definition module.
MAXDCB=	The maximum number of chained DCBs which will be used for this device. Must be three or less.
RSB=	The number of residual status bytes the device will transfer. Enter zero or an even decimal number between 4 and 16 inclusive.

Examples

```
EXIODEV  ADDRESS=00

EXIODEV  ADDRESS=E0,RSB=12,MAXDCB=2,END=YES
```

Note: Any device defined via EXIODEV should not be defined on any other statement such as DISK or TERMINAL. Doubly defined devices will cause unpredictable results when accessed by, for example, a combination of READ/WRITE and EXIO. You must load any control store that is required by the device.

HOSTCOMM - Define Host Communications Support

HOSTCOMM defines the type and address in the generated system of the device to be used for host communication support. This support operates in conjunction with Host Communications Facility IUP.

Syntax

```
blank      HOSTCOMM  DEVICE=,ADDRESS=

Required: DEVICE=, ADDRESS=
Defaults: None
```

Operands	Description
DEVICE=	The type of communication to be used. BSCA, for Binary Synchronous Communications Adapter support. This is the only device supported and must be a single line BSC adapter (feature 2074 or 2075). Only one is allowed.
ADDRESS=	The hexadecimal address of the device.

Example

```
  HOSTCOMM  DEVICE=BSCA,ADDRESS=09
```

SENSORIO

SENSORIO - Define Sensor I/O Devices

SENSORIO defines the sensor I/O devices to be supported in the generated system. All SENSORIO statements must be grouped together with the last one including an END=YES specification.

Syntax

```
blank    SENSORIO  ADDRESS=,DEVICE=,PI=,DI=,DO=,AI=,AO=,
                   AITYPE=,LEVEL=,END=

Required: DEVICE,ADDRESS
Defaults: AITYPE=RELAY,LEVEL=1,END=NO
```

Operands	Description
ADDRESS=	The base address of the device (in hexadecimal). This is the only required address if DEVICE=IDIO unless PI is needed on this unit.
DEVICE=	IDIO - The integrated digital I/O non-isolated feature (feature #1560). 4982 - The sensor I/O unit.

Note: For the PI, DI, DO, AI, and AO parameters, multiple addresses must be included in parentheses.

Operands	Description
DI=	The address or list of addresses of the digital input group(s) on this device. PI can be read as DI.
PI=	The address or list of addresses of the digital input group(s) to be used as process interrupt.
DO=	The address or list of addresses of the digital output group(s) on this device.
AO=	The address or list of addresses of the analog output point(s) on this device.

AI= The address or list of addresses of the analog input multiplexor feature(s) on this device.

AITYPE= The type of AI multiplexer(s). Valid entries are:

- RR or RELAY - for relay
- SS or SOLID - for solid state

(The names have a one-to-one relationship with addresses on the AI operand.)

LEVEL= A number (from 0-3) to assign the hardware interrupt level to the device.

Note: This assignment is for all features on that device.

END= YES, for the last SENSORIO statement in the system definition module.

Examples

```
SENSORIO  DEVICE=IDIO,ADDRESS=68

SENSORIO  DEVICE=4982,ADDRESS=60,AO=65,DO=62,DI=64,      C
          PI=63,AI=61,AITYPE=SS

SENSORIO  DEVICE=4982,ADDRESS=70,DI=(70,71)

SENSORIO  DEVICE=4982,ADDRESS=60,AI=(62,63),             C
          AITYPE=(RELAY,SOLID),AO=64,DI=(65,66),DO=67

SENSORIO  DEVICE=IDIO,ADDRESS=68,PI=68,END=YES
```

SYSTEM

SYSTEM - Define Processor

SYSTEM defines the characteristics of the processor and the system generation options. This statement must be specified once.

Syntax

```
blank     SYSTEM   STORAGE=,MAXPROG=,PARTS=,DATEFMT=,
                   IABUF=,COMMON=

Required:  STORAGE=
Defaults:  MAXPROG=10,PARTS=32,DATEFMT=MMDDYY
           IABUF=20,COMMON=EDXSYS
```

Operands	Description
STORAGE=	The size in K bytes (K=1024) of the Series/1 processor storage. Enter one of the following numbers: 16, 32, 48, 64, 80, 96, 112, 128, 160, 192, 224, or 256.
MAXPROG=	The maximum number of concurrently executing programs to be allowed in the partition. Add one to your calculated number for each occurrence of $JOBUTIL in that partition. Add two for each occurrence of the session manager in that partition. Four words of storage are required in the nucleus for each program specified. If a storage size larger than 64K bytes is specified, multiple partitions must be defined. You must specify a list of the maximum number of concurrently executing programs allowed in each partition. The number of programs which can run concurrently in a system is a function of several variables, such as: • Processor storage • Program size

- Processor time requirements

These items vary with each installation.

PARTS= The number of 2K (1K=1024 bytes) blocks of storage to be assigned to each partition. Use only if STORAGE= is specified as greater than 64. Enter a list showing the maximum size of each partition. Up to eight partitions can be defined for the 4955, up to two for the 4952, and one for the 4953. The list must contain the same number of entries as the list coded for MAXPROG=.

The method for calculating the maximum size for partition one is as follows:

Determine the available storage in the first 64K by subtracting the size of the supervisor from 64K. See Appendix A to estimate the supervisor size.

The size of partition one is determined when you IPL, by using the smaller of:

- The size you define in the PARTS= parameter
- 64K minus the size of the supervisor

The maximum value that can be specified is 32; the minimum is 2. When specifying the size to be assigned to partition one, you may code 32 rather than calculating the value, if you wish partition one to have all storage not used by the supervisor. Otherwise, you must calculate the size of partition one.

The Multiple Terminal Manager partition size can be calculated by using the information in the Communications and Terminal Applications Guide.

DATEFMT= The format to be used when the date is displayed (PRINDATE or $W) or when entering the date via $T. A return code is set in response to a GETTIME request with the DATE option.

Specify MMDDYY for a date format of month.day.year. Specify DDMMYY for a date format of day.month.year. MMDDYY is the default.

Note: Timer support must be included in your supervisor in order to have date support.

SYSTEM

IABUF= The maximum number of interrupts that may be buffered by the task supervisor. The default value is adequate for most systems. The value should be increased if the system could be overloaded by a large number of interrupts. (The system will stop or enter a continuous run loop.) Each increment increases the supervisor storage requirements by four bytes.

COMMON= The label of the last supervisor address to be mapped in every partition. The value will be automatically rounded upward to a 2K byte boundary. To map the entire supervisor, specify COMMON=START. To map only the supervisor data areas, specify COMMON=EDXSVCX. The default, COMMON=EDXSYS, implies no mapping. Refer to "$SYSCOM - Define Optional Common Data Area" on page 113 for additional information.

Example 1

```
SYSTEM   STORAGE=96,MAXPROG=(3,2,3),                C
      PARTS=(32,6,10)
```

This three partition system is possible on a 96KB 4955 and maps as follows:

PARTITION 1	28KB SUPERVISOR	36KB USER SPACE
PARTITION 2	12KB USER SPACE	
PARTITION 3	20KB USER SPACE	

1. Partition 1 is 36KB and can execute up to three programs concurrently.
2. Partition 2 is 12KB and can execute up to two programs concurrently.
3. Partition 3 is 20KB and can execute up to three programs concurrently.

Note: The 28KB supervisor size is used for illustrative purposes only.

Example 2

```
SYSTEM   STORAGE=64,MAXPROG=5
```

A map of this single partition system is as follows:

PARTITION 1	28KB SUPERVISOR	36KB USER SPACE

Up to five programs can execute concurrently.

Note: The 28KB supervisor size is used for illustrative purposes only.

Example 3

```
SYSTEM   STORAGE=196,MAXPROG=(1,2,1,3,4,1),    C
      PARTS=(9,12,7,4,20,32)
```

This six partition system is possible on a 196KB 4955 and maps as follows:

PARTITION 1	28KB SUPERVISOR	18KB USER SPACE
PARTITION 2	24KB USER SPACE	
PARTITION 3	14KB USER SPACE	
PARTITION 4	8KB USER SPACE	
PARTITION 5	40KB USER SPACE	
PARTITION 6	64KB USER SPACE	

1. Partition 1 is 18KB and can execute one program at a time.
2. Partition 2 is 24KB and can execute up to two programs concurrently.
3. Partition 3 is 14KB and can execute one program at a time.
4. Partition 4 is 8KB and can execute up to three programs concurrently.
5. Partition 5 is 40KB and can execute up to two programs concurrently.

6. Partition 6 is 64KB and can execute one program at a time

Note: The 28KB supervisor size is used for illustrative purposes only.

Example 4

```
SYSTEM   STORAGE=128,MAXPROG=(10,10,10),      C
           PARTS=(27,9,23)
```

This three partition system is possible on a 128KB 4955 and maps as follows:

PARTITION 1	28KB SUPERVISOR	36KB USER SPACE
PARTITION 2	18KB USER SPACE	
PARTITION 3	46KB USER SPACE	

1. Partition 1 is 36KB and can execute up to ten programs concurrently.
2. Partition 2 is 18KB and can execute up to ten programs concurrently.
3. Partition 3 is 46KB and can execute up to ten programs concurrently.

Note: The 28KB supervisor size is used for illustrative purposes only.

Example 5

```
SYSTEM   STORAGE=128,MAXPROG=(3,6),           C
           PARTS=(32,32),COMMON=EDXSVCX
```

This two partition system is possible on a 128KB 4952 and maps as follows:

PARTITION 1	28KB SUPERVISOR	36KB USER SPACE
PARTITION 2	4KB CONTROL BLOCKS	60KB USER SPACE

1. Partition 1 is 36KB and can execute up to three programs concurrently.

2. Partition 2 is 60KB and can execute up to six programs concurrently. The programs all have direct addressability to supervisor control blocks (for example, the CVT and DVT) because of the COMMON=EDXSVCX parameter.

3. When the date is displayed, it will be in month, day, and year format.

Note: The 30KB supervisor size and the 4KB control block size are used for illustrative purposes only.

Example 6

```
SYSTEM   STORAGE=128,MAXPROG=(4,4),                C
      PARTS=(32,32),DATEFMT=MMDDYY
```

This two partition system is possible on a 128KB 4952 and maps as follows:

PARTITION 1	30KB SUPERVISOR	34KB USER SPACE
PARTITION 2	64KB USER SPACE	

1. Partition 1 is 34KB and can execute up to four programs concurrently.

2. Partition 2 is 64KB and can execute up to four programs concurrently.

3. When the date is displayed, it will be in month, day, and year format.

Note: The 30KB supervisor size is used for illustrative purposes only.

Example 7

```
SYSTEM   STORAGE=256,                              C
      MAXPROG=(3,1,5,2,2,1,1,4),                   C
      PARTS=(15,4,21,13,17,11,8,23)
```

This eight partition system is possible on a 256KB 4955 and maps as follows:

PARTITION 1	32KB SUPERVISOR	30KB USER SPACE
PARTITION 2	8KB USER SPACE	
PARTITION 3	42KB USER SPACE	
PARTITION 4	26KB USER SPACE	
PARTITION 5	34KB USER SPACE	
PARTITION 6	22KB USER SPACE	
PARTITION 7	16KB USER SPACE	
PARTITION 8	46KB USER SPACE	

1. Partition 1 is 30KB and can execute up to three programs concurrently.

2. Partition 2 is 8KB and can execute one program at a time.

3. Partition 3 is 42KB and can execute up to five programs concurrently.

4. Partition 4 is 26KB and can execute up to two programs concurrently.

5. Partition 5 is 34KB and can execute up to two programs concurrently.

6. Partition 6 is 22KB and can execute one program at a time.

7. Partition 7 is 16KB and can execute one program at a time

8. Partition 8 is 46KB and can execute up to four programs concurrently.

Note: The 32KB supervisor size is used for illustrative purposes only.

Example 8

```
SYSTEM   STORAGE=96,MAXPROG=(3,4),                C
      PARTS=(16,18),COMMON=START
```

This two partition system is possible on a 96KB 4952 and maps as follows:

PARTITION 1	28KB SUPERVISOR	32KB USER SPACE
PARTITION 2	36KB USER SPACE	

1. Because COMMON=START was specified, the supervisor is mapped in both partition 1 and partition 2, providing direct addressability to the supervisor for all programs that execute on this system.

2. Partition 1 is 32KB and can execute up to three programs concurrently.

3. Partition 2 is 36KB and can execute up to four programs concurrently.

Note: The 28KB number for the supervisor is used for illustrative purposes only.

TAPE

TAPE - Define Tape Device (Version 2 only)

TAPE defines the tape devices on a system. One TAPE statement is required for each tape device on the system. It is recommended that you group all DISK statements together, followed by all the TAPE statements. The last TAPE or DISK statement must include an END=YES specification.

Syntax

```
blank  TAPE DEVICE=,ADDRESS=,DENSITY=,LABEL=,ID=,
            TASK=,END=
    Required: DEVICE=,ADDRESS=,ID=
    Defaults: DENSITY=1600,LABEL=SL,TASK=NO,END=NO
```

Operands	Description
DEVICE=	Device type (4969 to define IBM 4969 tape unit)
ADDRESS=	A two digit hexadecimal number specifying the address assigned to the unit
DENSITY=	Tape density to be used for this device (800,1600,DUAL). When DUAL is coded, density defaults to 1600 BPI.
LABEL=	Type of processing to be done on this device. Standard label (SL), non-label (NL), and bypass label processing (BLP) are the only types supported.
ID=	A one-to six-character name that is associated with the device. This operand is used primarily for specifying the drive when NL or BLP is used.
TASK=	YES, causes a new I/O task to be generated. This task is used to service I/O request for this and subsequent tapes until a new TAPE statement with TASK=YES is encountered. For best performance, specify TASK=YES for each tape unit that has a controller.
END=	YES, for the last statement in the DISK/TAPE sequence.

Example

```
TAPE  DEVICE=4969,ADDRESS=4C,DENSITY=1600,          X
      LABEL=SL,ID=$TAPE1,                           X
      TASK=YES,END=YES
```

Note: END=YES is specified only once for the DISK/TAPE definition statements.

TERMINAL - Define Input/Output Terminals

TERMINAL defines each input/output terminal to be supported in the generated system. Output only devices, such as line printers, are also specified with TERMINAL statements. All TERMINAL statements must be grouped together with the last statement including an END=YES specification.

A TERMINAL statement specifying DEVICE=VIRT can be entered in an application program provided exactly the same statement is entered in the system configuration program. All TERMINAL statements within the application program are automatically converted to an IOCB statement. The label on the TERMINAL statement is used for the label and the operand of the IOCB statement.

Before preparing your TERMINAL statements, you need to know the characteristics of your terminals, the way they will be attached to your Series/1, and how you plan to use them in your application. Review the appropriate hardware manuals, the topic entitled "Terminal I/O" in the Language Reference, and the appropriate topics in Communications and Terminal Applications Guide.

If you use the Remote Management Utility and need the PASSTHRU function, two virtual terminals are required. For a detailed description of the PASSTHRU function see the Remote Management Utility chapter in Communications and Terminal Applications Guide. See Figure 10 on page 107 for a sample configuration.

Syntax

```
label TERMINAL DEVICE=,ADDRESS=,PAGSIZE=,LINSIZE=,
               CODTYPE=,TOPM=,BOTM=,NHIST=,LEFTM=,RIGHTM=
               OVFLINE=,LINEDEL=,CHARDEL=,CRDELAY=,ECHO=,
               BITRATE=,RANGE=,LMODE=,ADAPTER=,COD=,CR=,
               LF=,HDCOPY=,ATTN=,PF1=,SYNC=,SCREEN=,PART=
               DI=,DO=,PI=,END=,TYPE=

Required: DEVICE= ,and one of the following:

•   ADDRESS= except for DI/DO terminals

•   DI=,DO=,PI= for DI/DO terminals

Defaults: PART=1,END=NO
```

Operands	Description
DEVICE=	One of the following codes for the indicated device:
TTY	A 3101 Display Terminal or other ASCII Terminal attached via Teletypewriter Adapter (7850)
4979	4979 display station attached via 3585 Adapter
4978	4978 display station attached via RPQ D02038
4974	4974 matrix printer attached via 5620 Adapter
4973	4973 line printer attached via 5630 Adapter
2741	2741 communications terminal attached via 1610 controller
4013	Graphics terminal attached via 1560 adapter (Refer to Communications and Terminal Applications Guide for hardware considerations.)

TERMINAL

ACCA A 3101 Display Terminal or other ASCII terminal attached via 1610 controller or 2091 controller with 2092 adapter or 2095 controller with 2096 adapter (Refer to Communications and Terminal Applications Guide for hardware considerations.)

PROC Processor-to-processor communication

VIRT Inter-program communication. (Refer to "Chapter 14. Inter-Program Communications" on page 279.)

ADDRESS= The address (in hexadecimal) of the device. (Refer to "Chapter 14. Inter-Program Communications" on page 279 for the use of this parameter in connection with virtual terminal communications.)

PAGSIZE= The physical page size (form length) of the I/O medium. Specify a decimal number between 1 and the maximum value which is meaningful for the device. For printers, specify the number of lines per page, or for screen devices the size of the screen in lines. This operand is not required for the 4978/4979 display; its value is forced to 24.

CODTYPE= The transmission code used by the terminal. Specify either ASCII, EBCDIC, EBCD (PTTC/EBCD), CRSP (PTTC/correspondence), or EBASC (8 bit data interchange code) as in the following table:

DEVICE=	TYPE OF ADAPTER		
	7850	1610 or 2091 w/2092	2095 w/2096
TTY	ASCII (default)	N/A	N/A
2741	N/A	Specify: EBCD or CRSP	N/A
ACCA	N/A	EBASC (default)	Specify: ASCII

LINSIZE= The maximum length of an input or output line for the device. The value of this operand can be less than the maximum which the device can accommodate

(for example, 80 for the 4978/4979 display station or 132 for the 4974 printer), but the value is then fixed and cannot be altered dynamically.

TOPM= The top margin (a decimal number between zero and PAGSIZE-1) to indicate the top of the logical page within the physical page for the device.

NHIST= The number of history lines to be retained when a page eject is performed on the 4978/4979 display. The line at TOPM+NHIST corresponds to logical line zero for purposes of the terminal I/O instructions. When a page eject (LINE=0) is performed, the screen area from TOPM to TOPM+NHIST-1 will contain lines from the previous page. This operand is meaningful for roll screens only. (See the discussion of the SCREEN operand which follows.)

BOTM= The bottom margin, the last usable line on a page. Its value must be between TOPM+NHIST and PAGSIZE-1. If an output instruction would cause the line number to increase beyond this value, then a page eject, or wrap to line zero, is performed before the operation is continued.

LEFTM= The left margin, the character position at which input or output will begin. Specify a decimal value between zero and LINSIZE-1.

RIGHTM= A value (between LEFTM and LINSIZE-1) that determines the last usable character position within a line. Position numbering begins at zero.

OVFLINE= YES, if output lines which exceed the right margin are to be continued on the next line.

LINEDEL= A two-digit hexadecimal character that defines the character the operator will enter when he wishes to restart an input line. In some cases, input of this character causes a repeat of the previous output message. Usually, this operand is not meaningful for devices such as the 4979 display station, whose input is formatted locally before entry. (For the ACCA terminals attached via the 1610 or 2091 controllers and the 2092 adapter, code in mirror image. Refer below for a description of mirror images.)

CHARDEL= A two-digit hexadecimal character which indicates deletion of the previous input character. It is meaningful only for devices whose mode of transmission is one character at a time, as described in

the LINEDEL operand. For the ACCA terminals attached via the 1610 or 2091 controllers and the 2092 adapter, enter in mirror image.

CRDELAY= The number of idle times required for a carriage return to complete for teletypewriter devices. If printing occurs during the carriage return, CRDELAY is too small. For interprocessor communications (DEVICE=PROC), refer to the Communications and Terminal Applications Guide.

ECHO= NO, for devices that do not require input characters to be written back (echoed) by the processor for printing.

YES (the default) is appropriate for most devices connected through the teletypewriter adapter. NO is required for ACCA. See the LF parameter description regarding suppression of the echo of the CR character.

BITRATE= The rate (in bits per second) that this terminal will be operating. (Used with ACCA, 2741 and PROC support only.)

RANGE= Enter HIGH or LOW to match hardware jumper that is installed on the adapter card. (Used with ACCA, 2741 and PROC support only.)

LMODE= SWITCHED or PTTOPT. If this line is used with a switched connection, then enter SWITCHED. Otherwise, enter PTTOPT. (Used with ACCA support only.)

ADAPTER= One of the following to indicate the ACCA type:

SINGLE For the single line controller

TWO For the eight line controller with up to two lines active

FOUR For the eight line controller with up to four lines active

SIX For the eight line controller with up to six lines active

EIGHT For the eight line controller with up to eight lines active

All multiple line feature cards must start at a base address ending with with X'0' or X'8'. A terminal statement with DEVICE=ACCA must exist for the line at the base address. Furthermore, the terminal defined as the base address must be specified as the first terminal for the multiline controller. The remaining terminals defined on the multiline controller (if any) must immediately follow the base address terminal and should be in ascending order by address.

Note: For DEVICE=2741, only SINGLE is allowed.

This should match the jumpers on the controller cards. (Refer to the Communications and Terminal Applications Guide for hardware considerations.)

COD= Additional characters, other than the CR=, ATTN=, and LINEDEL= values, that will terminate a READ operation. (COD means change of direction, for example, READ to WRITE.) (Used with ACCA only.) Code in mirror image as follows:

```
COD=11
or
COD=(12,B6,42...)
```

From one to four COD characters may be entered.

CR= The character to be tested to determine if a new line function is to be performed. (Code in mirror image for ACCA terminals attached via the 1610 or 2091 controllers with the 2092 adapter.)

LF= The character to be sent to the terminal when a new line function is to be performed. Code in mirror image for ACCA terminals attached via the 1610 or 2091 controllers with the 2092 adapter. If the same value is coded for LF= as was coded (or defaulted) for CR= then the CR character which terminates an input operation will not be echoed to the terminal; the terminal is assumed to be an auto-line feed device.

HDCOPY= Support for the 4978/4979 display station includes a means of printing the contents of the display screen on a hardcopy device for permanent record. (For an explanation of the hardcopy feature, refer to Utilities, Operator Commands, Program Preparation, Messages and Codes). The hardcopy function is defined by coding:

HDCOPY=(terminal name, key),

terminal name — The symbolic name of the terminal to which the hardcopy contents will be directed

key — The code of the program function key which is to invoke the function. For example, HDCOPY=($SYSPRTR,4) designates $SYSPRTR as the hardcopy device and PF4 as the activating key. If the hardcopy terminal name alone is specified, as for example in HDCOPY=$SYSPRTR, then the default is PF6. Note: The terminal specified (Terminal name) must not be defined with ATTN=NO.

ATTN= NO, if the attention key and the 4978/4979 PF keys are to be disabled for the terminal. Such disabling is then permanent for the generated system. If you do not specify ATTN=, the default is the ATTN key.

LOCAL, to limit the attention functions to those defined by ATTNLISTs within programs loaded from the terminal.

NOSYS, to exclude only the system functions ($L, $C, etc.).

NOGLOB, to exclude only the global ATTNLIST functions. (GLOBAL is the ATTNLIST of all programs in the same partition at one time.)

Note: This operand can also be entered with a two-digit hexadecimal character for the attention key if the system default is not desired.

The attention key can be redefined with a two-digit hexadecimal character for the 4978/4979 displays or ASCII terminals.

For terminals attached via the 1610 or 2091 controllers and the 2092 adapter, use mirror image. (Refer to "Mirror Image" on page 109 for a discussion of mirror image.)

For the 3101 display terminal, enter X'D9' if the terminal is attached via the 1610 or 2091 controllers and X'9B' if it is attached via the 2095 controller. You may have the Mark Parity Switch set on (refer to the IBM 3101 Display Terminal Description

GA18-2034, for information on switch settings).

The default for ATTN for ASCII terminals is ASCII X'1B', the ESC key. The mirror image of X'1B' is X'D8'. Note: If the terminal being defined is specified in the HDCOPY= parameter of an other terminal, do not code ATTN=NO.

PF1= For the 4978 display, code the two-digit hexadecimal character which is to be interpreted as Program Function key 1. Successive values are then interpreted as PF2 and PF3.

The default for this operand is 2.

SYNC= This keyword applies to virtual terminal communications. Code SYNC=YES if synchronization events will be posted to this virtual terminal.

This means that attempted actions over the virtual channel will be indicated in the task control word. This allows the two terminals to synchronize their actions so that when one terminal is writing, the other is reading.

SYNC=NO is the default.

SCREEN= One of the following to indicate whether the terminal is a hardcopy or screen device:

YES or ROLL for screens which are to be operated like a typewriter.

For screen devices which are attached through the teletypewriter adapter, this indicates that a pause will be performed when a screen-full condition occurs during continuous output.

NO for hardcopy devices. For 4978 or 4979 devices, NO results in inhibiting the pause when the screen fills up (the screen acts as a roll screen).

STATIC for a full-screen mode of operation, if this mode is supported for the device.

Note: The initial terminal configuration should be STATIC only if the terminal is reserved for data display and data entry operations. Normal system operations, such as those directed to $SYSLOG or those involving the utility programs, assume a roll screen configuration. The application program can define the static screen configuration by means of

the ENQT and IOCB instructions described in the Language Reference.

PART= A number (1-8) to indicate the partition with which the terminal is normally associated.

This is valid only if the STORAGE= operand of the SYSTEM statement was specified to be greater than 64. You can change the partition assignment at execution time with the $CP Command described in Utilities, Operator Commands, Program Preparation, Messages and Codes.

END= YES, for the last TERMINAL statement in a system definition module.

TYPE= Specify DSECT to generate a CCB DSECT in your program. for programs processed by $S1ASM. Do not specify DSECT in programs processed by $EDXASM; use COPY CCBEQU elsewhere in your program.

The following three operands are for terminals connected via digital I/O only:

Operands	Description
DI=(address,termaddr)	
address	The digital input group address.
termaddr	The hardware subaddress (0-7) of the terminal defining the value used to select the terminal for digital input.
DO=(address,termaddr)	
address	The digital output group address
termaddr	The hardware subaddress (0-7) to define the digital output subaddress of the terminal
PI=(address,bit)	
address	The process interrupt group address.
bit	The bit (0-15) to define the particular interrupting point assigned to the terminal.

Terminal support is provided for digital I/O devices such as the Tektronix 4010 Series of Display Terminals equipped with the General Purpose Parallel Interface (Tektronix Custom Feature Number CM021-0109-03) or terminals having equivalent hardware interfaces. (Refer to the Communications and Terminal Applications Guide.)

Examples and Defaults

Default values for optional parameters on the TERMINAL statement vary with the device type. In the following examples, the default assignments for each device support are shown as if they were explicitly coded in the TERMINAL statement. If a parameter is not shown, then it is not relevant for the device. Address assignments are for illustration only.

4978/4979 Display TERMINAL Statement

```
TERMINAL  DEVICE=4978 (or 4979),ADDRESS=04,PAGSIZE=24,   C
          LINSIZE=80,TOPM=0,NHIST=12,BOTM=23,LEFTM=0,    C
          RIGHTM=79,SCREEN=ROLL,OVFLINE=NO,ATTN=YES
```

4974 Matrix Printer or 4973 Line Printer TERMINAL Statement

```
TERMINAL  DEVICE=4974 (or 4973),ADDRESS=01,PAGSIZE=66,   C
          LINSIZE=132,TOPM=3,BOTM=63,LEFTM=0,            C
          RIGHTM=131,OVFLINE=NO
```

TERMINAL

ASCII Terminal via 7850 Adapter TERMINAL Statement

```
TERMINAL  DEVICE=TTY,ADDRESS=00,PAGSIZE=35,LINSIZE=80,   C
          CODTYPE=ASCII,TOPM=0,BOTM=34,LEFTM=0,          C
          RIGHTM=79,SCREEN=NO,OVFLINE=NO,LINEDEL=7F,     C
          CHARDEL=08,CRDELAY=0,ECHO=YES,ATTN=YES,        C
          CR=0D,LF=0A
```

IBM 2741 Terminal TERMINAL Statement

```
TERMINAL  DEVICE=2741,ADDRESS=08,PAGSIZE=66,             C
          LINSIZE=130,CODTYPE=EBCD,TOPM=0,BOTM=65,       C
          LEFTM=0,RIGHTM=129,SCREEN=NO,OVFLINE=NO,       C
          LINEDEL=A0,CHARDEL=5D,CRDELAY=0,ECHO=NO,       C
          CR=5B,LF=5B,BITRATE=134,ADAPTER=SINGLE
```

ASCII Terminal via 1610 Controller TERMINAL Statement

```
TERMINAL  DEVICE=ACCA,ADDRESS=70,PAGSIZE=35,             C
          LINSIZE=80,CODTYPE=EBASC,TOPM=0,               C
          BOTM=34,LEFTM=0,RIGHTM=79,SCREEN=NO,           C
          OVFLINE=NO,CRDELAY=0,ECHO=NO,                  C
          BITRATE=300,RANGE=HIGH,LMODE=PTTOPT,           C
          ATTN=YES,ADAPTER=SINGLE,LF=5B,CHARDEL=11
```

PROC (via 1610 Controller) TERMINAL Statement

```
TERMINAL  DEVICE=PROC,ADDRESS=7F,CODTYPE=EBCDIC,         C
          LINSIZE=130,CRDELAY=(PROMPT,30000),            C
          BITRATE=9600,RANGE=HIGH,CR=5B,LF=5B
```

4013[4] (DI/DO Parallel Interface) TERMINAL Statement

```
TERMINAL  DEVICE=4013,DI=(80,01),DO=(87,01),                    C
          PI=(84,04),PAGSIZE=35,LINSIZE=72,                     C
          CODTYPE=ASCII,TOPM=0,BOTM=34,LEFTM=0,                 C
          RIGHTM=71,SCREEN=NO,OVFLINE=NO,                       C
          LINEDEL=7F,CHARDEL=08,CRDELAY=0,ECHO=YES,             C
          CR=0D,LF=0A
```

Remote Management Utility using the PASSTHRU function - TERMINAL Statements

```
CDRVTA    TERMINAL  DEVICE=VIRT,ADDRESS=CDRVTB,                 C
                    SYNC=YES,LINSIZE=132

CDRVTB    TERMINAL  DEVICE=VIRT,ADDRESS=CDRVTA,                 C
                    SYNC=NO,LINSIZE=132
```

Note: This example shows a line size of 132. The maximum line size value is 254.
The names CDRVTA and CDRVTB are required.

The following statements are coded with values that are not defaults for parameters PAGSIZE, ATTN, CR, CHARDEL, LINEDEL, ADAPTER, BOTM, SCREEN, BITRATE, RANGE, and MODE. Use these values if the IBM 3101 Display Terminal is attached to your system. For DEVICE=ACCA, you must set the mark parity switch on (refer to the IBM 3101 Display Terminal Description, GA18-2033, for information on switch settings).

[4] Registered trademark of the Tektronix Corporation.

IBM 3101 TERMINAL Statement (via 7850 adapter)

```
TERMINAL  DEVICE=TTY,ADDRESS=00,CRDELAY=4,                    C
          PAGSIZE=24,SCREEN=YES
```

IBM 3101 TERMINAL Statement (via 2095 controller)

```
TERMINAL  DEVICE=ACCA,ADDRESS=60,BITRATE=110,                 C
          PAGSIZE=24,LINSIZE=80,                              C
          CODTYPE=ASCII,TOPM=0,BOTM=23,LEFTM=0,               C
          RIGHTM=79,SCREEN=YES,OVFLINE=NO,                    C
          LINEDEL=FF,CHARDEL=88,CRDELAY=0,ECHO=NO,            C
          RANGE=LOW,LMODE=PTTOPT,                             C
          CR=8D,LF=0A,ATTN=9B,ADAPTER=FOUR
```

IBM 3101 TERMINAL Statement
(via 1610 or 2091 controller)

```
TERMINAL  DEVICE=ACCA,ADDRESS=6B,BITRATE=110,                 C
          PAGSIZE=24,LINSIZE=80,                              C
          CODTYPE=EBASC,TOPM=0,BOTM=23,LEFTM=0,               C
          RIGHTM=79,SCREEN=YES,OVFLINE=NO,                    C
          LINEDEL=FF,CHARDEL=88,CRDELAY=0,ECHO=NO,            C
          RANGE=LOW,LMODE=SWITCHED,                           C
          CR=B1,LF=50,ATTN=D9,ADAPTER=EIGHT
```

IBM 3101 Model 2 (block mode) under Multiple Terminal Manager TERMINAL Statement (via 1610 or 2091 controller)

```
TERMINAL  DEVICE=ACCA,ADDRESS=08,BITRATE=2400,          C
          PAGSIZE=24,LINSIZE=80,                        C
          CODTYPE=EBASC,TOPM=0,BOTM=23,LEFTM=0,         C
          RIGHTM=79,SCREEN=YES,OVFLINE=NO,              C
          LINEDEL=FF,CHARDEL=11,CRDELAY=0,ECHO=NO,      C
          RANGE=HIGH,LMODE=PTTOPT,                      C
          CR=B1,LF=50,ATTN=FB,ADAPTER=SINGLE
```

IBM 3101 Model 2 (block mode) under Multiple Terminal Manager TERMINAL Statement (via 2095 controller)

```
TERMINAL  DEVICE=ACCA,ADDRESS=61,BITRATE=2400,          C
          PAGSIZE=24,LINSIZE=80,                        C
          CODTYPE=ASCII,TOPM=0,BOTM=23,LEFTM=0,         C
          RIGHTM=79,SCREEN=YES,OVFLINE=NO,              C
          LINEDEL=FF,CHARDEL=88,CRDELAY=0,ECHO=NO,      C
          RANGE=HIGH,LMODE=PTTOPT,                      C
          CR=8D,LF=0A,ATTN=DF,ADAPTER=FOUR
```

Mirror Image

Mirror image is used by ASCII terminals attached via the 1610 or 2091 controllers and the 2092 adapter. Mirror image reverses the bit pattern for data. For example, the EBCDIC character 1 would look as follows:

X'F1' EBCDIC

X'31' ASCII

X'8F' Mirror Image EBCDIC

X'8C' Mirror Image ASCII

When using XLATE=NO on Event Driven language instructions PRINTEXT and READTEXT, the data sent must be in mirror image. Data received is in mirror image.

ASCII Terminal Codes

Terminals and other devices equivalent to the Teletype ASR 33/35 are referred to in this document as "ASCII terminals." These terminals may be attached to the Series/1 in a variety of ways. Note that while the bit representation of a character appearing at the terminal is the same for all the attachments, two different representations for a given character are used internally.

One representation is ASCII, in which the characters appear in main storage in ASCII code. This code is used by features #7850, #2095, and #2096.

The other representation is the Eight Bit Data Interchange Code. It is used by the 1610 and 2091 controllers and the 2092 adapter. This representation is the mirror image within a **byte** of the ASCII representation. The bits appear swapped end-for-end within each byte.

Note also that ASCII terminals may use even, odd, or no parity. The parity bit appears as the high order bit in ASCII code and as the low order bit in Eight Bit Data Interchange Code. You must incorporate the proper parity, if any, within the data characters. You must also incorporate the proper parity, if any, within the control characters specified by the LINEDEL, CHARDEL, COD, CR, and LR parameters of the TERMINAL statement.

Symbolic Reference to Terminals

The optional label on the TERMINAL statement is used to assign a name to the device for purposes of reference by the application program. Three such names have special meaning to the supervisor and should be assigned to the appropriate device:

$SYSLOG Names the system logging device or operator station, and must be defined in every system. In the starter supervisor, $SYSLOG defines a 4978 display station.

$SYSLOGA Names the alternate system logging device. If unrecoverable errors prevent use of $SYSLOG, the system will use the $SYSLOGA terminal as the system logging device/operator station. If defined, this device should be a terminal with keyboard capability, not just a printer. The starter supervisor defines the $SYSLOGA terminal as a teletypewriter device.

$SYSPRTR Names the system printer. If defined, the hard copy output from some system programs will be directed to this device. The starter supervisor defines a 4974 matrix printer as the $SYSPRTR device.

$SYSLOG is referred to by the supervisor and the utility programs and must be defined on every system. $SYSLOGA is an optional assignment which will be used, if defined, as the output device for $SYSLOG messages if $SYSLOG develops an uncorrectable error condition.

$SYSPRTR is an optional specification which, if defined, will be used as the output device by some of the utility programs. If $SYSPRTR is not defined, the output will be directed to the terminal from which the program was invoked.

Assignment of a name to a terminal designates that terminal as a global resource which can be accessed by any application program through use of the ENQT and DEQT instructions described in the Language Reference.

TIMER

TIMER - Define System Timer Features

TIMER is used to define the address of the #7840 Timer Feature to be used as the system timers in the generated system. One of the two timers on the card will be used for time of day recording and the other will be used for interval timing.

This statement is used only for defining the #7840 timer. If the system has a native timer (4952 processor only) that is used instead of the #7840 timer feature card, it is not necessary to use this or any other statement. The native timer and the #7840 timer are mutually exclusive.

Syntax

```
blank      TIMER      ADDRESS=

Required: ADDRESS=
Defaults: None
```

Operands	Description
ADDRESS=	The hexadecimal address of the #7840 Timer Feature.

Example

```
  TIMER     ADDRESS=40
```

$SYSCOM - Define Optional Common Data Area

$SYSCOM is an optional data area in the supervisor which can be accessed from application programs. If you select this option, you must map the portion of the supervisor containing $SYSCOM into each address space.

The common area is referenced indirectly in application programs through a storage location with the label $SYSCOM. This storage location contains the address of the first word of the common area. Therefore, in order to reference data in the common area, the contents of $SYSCOM should be loaded into a register, such as #1. Data elements can then be referenced by a displacement from this register.

The common area can contain Event Control Blocks (ECB), Queue Control Blocks (QCB), or any data blocks that must be accessed by more than one program. For example, if several programs perform a file update that must be performed serially, a QCB is defined in the common area which is related to this file update process. Programs that perform the file update should ENQ on this QCB before reading the file and DEQ the QCB after writing the file. Many of the functions available through the use of $SYSCOM are also provided by the cross partition capabilities of the Event Driven Language instruction set. (Refer to the Language Reference and "Chapter 14. Inter-Program Communications" on page 279 for details.)

You define the size and contents of $SYSCOM. However, since storage is mapped in 2K byte increments, the minimum common data area is 2K bytes. For example, a 12K byte partition requires at least 14K bytes (the PARTS= operand must specify 7 (14KB)).

The actual size of the mapped area is rounded up to a 2K boundary.

If you require a common data area and wish to minimize the storage it occupies in each partition, use the following process:

1. Prepare a module that contains the items you wish to include in the common data area. The last statements should be:

```
         ENTRY UCOMM
UCOMM    EQU   *
         END
```

2. Name this module USERCOM and store it in ASMLIB.

3. Insert INCLUDE USERCOMM,ASMLIB after INCLUDE EDXSYS in $LNKCNTL. This makes your common data area the second module in the nucleus.

4. Enter COMMON=UCOMM on your SYSTEM statement.

$SYSCOM then defines the beginning of your common area. The address of UCOMM, the end of your common area, rounded up to a 2KB boundary, is the size of the mapped area.

Example

Common area, containing two QCBs and two ECBs.

```
$SYSCOM   CSECT
          QCB
          QCB
          ECB
          ECB
```

To reference the first QCB from your application, the following instructions can be coded:

```
          MOVE    #1,$SYSCOM
          ENQ     (0,#1)
           .
           .      perform serial operation
           .
          DEQ     (0,#1)
```

Since a QCB is ten bytes in length, the second QCB is referenced as follows:

```
          ENQ    (10,#1)
```

It may be convenient to define an equate table which identifies each element of the common area by a symbolic name. The PL/I User's Guide shows how to use and access $SYSCOM as a GLOBAL communication area.

To generate an Event Driven Executive system, you must have access to a Series/1 capable of preparing the supervisor program and application programs. System generation requires that the following licensed programs be installed:

- Basic Supervisor and Emulator
- Event Driven Executive Utilities
- Event Driven Executive Program Preparation Facility

  or

  Series/1 Macro Assembler and Macro Library

The Program Preparation Facility enables you to prepare programs to be executed on any Series/1 that has the required hardware configuration and licenses.

GENERATING THE SUPERVISOR

Creating a supervisor program tailored to your Series/1 hardware configuration requires the use of several of the utilities and program preparation programs; these include:

- Disk data set management ($DISKUT1)
- Text editor ($EDIT1N)

  or

  Full-screen editor ($FSEDIT)
- Batch job stream processor ($JOBUTIL)
- Event Driven Language compiler ($EDXASM)
- Linkage editor ($LINK)
- Object module conversion ($UPDATE)

You should become familiar with these utilities, especially the text editors, before attempting to generate the supervisor. These utilities are described in Utilities, Operator Commands, Program Preparation, Messages and Codes.

The following major steps are required:

- Step A. Allocate required data sets.
- Step B. Edit $EDXDEF, the system configuration file, to match your hardware configuration..
- Step C. Edit $LNKCNTL, the system-supplied INCLUDE file, to specify which supervisor program object modules are to be included in your supervisor.
- Step D. Edit $SUPPREP, the system-supplied job stream processor file, to use your allocated data sets.
- Step E. Use $JOBUTIL and the procedure file created in Step D to:
  - Assemble the supervisor definition module created in Step B
  - Link edit the resulting object module with the other necessary supervisor object modules using the link edit control data set created in Step C.
  - Using $UPDATE, convert the output of the link edit process into an executable supervisor, and store it in a data set named $EDXNUCT.
- Step F. Test the created supervisor on a disk based system.
- Step G. Verify the system generation process (optional).

Step A - Allocate Required Data Sets

1. IPL the system from disk volume EDX002.
2. Load utility program $DISKUT1 and use the AL command to allocate the following data sets on volume EDX002. All data sets must be specified as TYPE=DATA.

Data Set Name	Number of Records
EDITWORK	200
ASMOBJ	250
ASMWORK	250
SUPVLINK	450
LEWORK1	400
LEWORK2	150

Step B - Edit $EDXDEF to Match Hardware Configuration

Edit $EDXDEF to match your hardware configuration:

1. Load utility program $EDIT1N or $FSEDIT and specify EDITWORK as the reply to WORKFILE=.

2. Read the supplied data set $EDXDEF from volume ASMLIB. Figure 11 on page 133 shows a sample configuration of $EDXDEF. The supplied configuration can be seen in the Program Directory.

   The first time you use EDITWORK as a work file for the text editor, you will be asked if you can use the EDITWORK data set as a work data set; respond YES and continue.

3. Add to or delete from the contents of EDITWORK as necessary to create a set of system configuration statements. (System configuration statements are described in "Chapter 6. System Configuration" on page 75.) Some printer on the Series/1 should be designated as $SYSPRTR. When editing ensure that:

   - Continuation indicators in column 72 are not removed.
   - If required, a continuation character is placed in column 72 and the statement is continued in column 16 of the next line
   - A field does not extend beyond column 71

   The editing process consists of the following procedure:

   a. Calculate the total amount of storage available, the number of partitions desired, and the number of 2K blocks of storage desired for each partition. This information is inserted into the SYSTEM statement to define the characteristics of the processor. Refer to "Chapter 6. System Configuration" on page 75 for a description of the SYSTEM statement.

   b. Define the hardware features to be supported, using the appropriate system configuration statements (TIMER, SENSORIO, HOSTCOMM, BSCLINE, EXIODEV, DISK, TERMINAL, TAPE).

   c. Define the direct access storage devices and logical volumes to be supported in the generated system, using the DISK system configuration statement. Sample DISK configuration statements are supplied for each device in the $EDXDEF data set on ASMLIB. Refer to "Chapter 3. Data Management" on page 45 for storage capacities of the supported direct access storage devices. With this information, you can define your disk volumes.

The only restrictions are (1) that you define the required Event Driven Executive volumes (EDX002, EDX003, ASMLIB) in addition to your volumes and (2) that you follow the rules pertaining to library origins and maximum volume sizes.

Note: Optional software products may require additional volumes. Volume requirements are supplied with the product documentation.

d. Define the characteristics of all printers, displays, and teletypewriters, using the TERMINAL statement. Examples of various types of TERMINAL statements are included in the $EDXDEF data set.

4. Save the final version of the definition statements in the data set $EDXDEFS on volume EDX002.

Step C - Specify Object Modules

Edit $LNKCNTL to specify which supervisor program object modules are to be included.

1. Read data set $LNKCNTL from volume ASMLIB. The supplied contents of $LNKCNTL are shown in the following tables; footnotes are provided on required usage. The $LNKCNTL data set supplied with Version 1 does not include TAPE support.

Sample Contents of $LNKCNTL (Version 1.1)

```
*************************************************************
*  COMMENTS MAY BE INCLUDED BY PUTTING AN '*' IN COLUMN 1.  *
*  USE THIS TECHNIQUE TO OMIT UNNEEDED MODULES              *
*************************************************************
 OUTPUT  SUPVLINK,EDX002     ENTRY=$START
*************************************************************
* SUPERVISOR SUPPORT
*************************************************************
 INCLUDE EDXSYS,XS1002   *0* SYSTEM TABLES AND WORK AREAS
 INCLUDE ASMOBJ,EDX002   *0* OUTPUT FROM USER SYSTEM GENERATION
*INCLUDE EDXSVCX,XS1002  *0,K* TASK SUPERVISOR (XL)
 INCLUDE EDXSVCXU,XS1002 *0,L* TASK SUPERVISOR (UN-XL)
 INCLUDE EDXSTART,XS1002 *0* INITIALIZATION & ERROR HANDLER
*INCLUDE $DBUGNUC,XS1002 *0,G* RESIDENT $DEBUG SUPPORT
 INCLUDE EDXALU,XS1002   *0* EDL INSTRUCTION EMULATOR
*************************************************************
* DEVICE SUPPORT -- DISK(ETTE)S
*************************************************************
 INCLUDE DISKIO,XS1002   *M* BASIC DISK(ETTE) SUPPORT
 INCLUDE D49624,XS1002   *M* 4962/4964 DISK(ETTE) SUPPORT
 INCLUDE D4963A,XS1002   *M* 4963 SUBSYSTEM SUPPORT
 INCLUDE D4966A,XS1002   *M* 4966 MAGAZINE SUPPORT
*************************************************************
* DEVICE SUPPORT -- TERMINALS
*************************************************************
*INCLUDE EDXTIO,XS1002   *1,K* BASIC TERMINAL SUPPORT (XL)
 INCLUDE EDXTIOU,XS1002  *1,L* BASIC TERMINAL SUPPORT (UN-XL)
*INCLUDE EDXTERMQ,XS1002 *1,K* ENQT/DEQT & TERMINAL QING (XL)
 INCLUDE EDXTRMQU,XS1002 *1,L* ENQT/DEQT & TERMINAL QING (UN-XL)
*INCLUDE IOS4979,XS1002  *M,K* 4978/4979 DISPLAY SUPPORT
 INCLUDE IOS4979U,XS1002 *M,L* 4978/4979 DISPLAY SUPPORT
*INCLUDE IOS4974,XS1002  *M,K* 4973/4974 PRINTER SUPPORT
 INCLUDE IOS4974U,XS1002 *M,L* 4973/4974 PRINTER SUPPORT
 INCLUDE IOSTERM,XS1002  *2*   REQD FOR TTY, ACCA, 4013 & 2741
 INCLUDE IOSTTY,XS1002   *M*   ASR 33/35 TELETYPEWRITER SUPPORT
*INCLUDE IOSACCA,XS1002  *3*   ASCII ACCA TERMINAL SUPPORT
*INCLUDE IOS4013,XS1002  *M*   DIGITAL I/O TERMINAL SUPPORT
*INCLUDE IOS2741,XS1002  *M*   2741 TERMINAL SUPPORT
 INCLUDE IOSVIRT,XS1002  *M*   VIRTUAL TERMINAL SUPPORT
*************************************************************
* DEVICE SUPPORT -- TRANSLATION TABLES
*************************************************************
 INCLUDE TRASCII,XS1002  *4*   TELETYPEWRITER TRANSLATION
*INCLUDE TREBASC,XS1002  *3*   MIRROR IMAGE ASCII TRANSLATION
*INCLUDE TREBCD,XS1002   *5*   2741 EBDC TRANSLATION
*INCLUDE TRCRSP,XS1002   *5*   2741 CORRESPONDENCE TRANSLATION
*************************************************************
* DEVICE SUPPORT -- TIMERS
*************************************************************
*INCLUDE EDXTIMER,XS1002 *6*   4953/4955 TIMER (7840) SUPPORT
```

```
*INCLUDE EDXTIMR2,XS1002 *6*   4952 TIMER SUPPORT
*****************************************************************
* DEVICE SUPPORT -- BINARY SYNCHRONOUS COMMUNICATIONS
*****************************************************************
*INCLUDE BSCAM,XS1002    *7,K* BSC COMM. ACCESS SUPPORT (XL)
*INCLUDE BSCAMU,XS1002   *7,L* BSC COMM. ACCESS SUPPORT (UN-XL)
*INCLUDE TPCOM,XS1002    *8*   HOST COMMUNICATION SUPPORT
*****************************************************************
* DEVICE SUPPORT -- SENSOR INPUT/OUTPUT
*****************************************************************
*INCLUDE SBCOM,XS1002    *9*   BASIC SENSOR I/O SUPPORT
*INCLUDE IOLOADER,XS1002 *9,K* SENSOR I/O DEVICE OPEN (XL)
*INCLUDE IOLOADRU,XS1002 *9,L* SENSOR I/O DEVICE OPEN (UN-XL)
*INCLUDE SBAI,XS1002     *M*   ANALOG INPUT SUPPORT
*INCLUDE SBAO,XS1002     *M*   ANALOG OUTPUT SUPPORT
*INCLUDE SBDIDO,XS1002   *M*   DIGITAL INPUT/OUTPUT SUPPORT
*INCLUDE SBPI,XS1002     *M*   PROCESS INTERRUPT SUPPORT
*****************************************************************
* DEVICE SUPPORT -- EXIO CONTROL
*****************************************************************
*INCLUDE IOSEXIO,XS1002  *M*   EXIO DEVICE CONTROL SUPPORT
*****************************************************************
* SYSTEM SUPPORT -- ERROR LOGGING
*****************************************************************
 INCLUDE SYSLOG,XS1002   *A*   I/O ERROR LOGGING
*INCLUDE NOSYSLOG,XS1002 *A*   NO I/O ERROR LOGGING
 INCLUDE CIRCBUFF,XS1002 *B*   PROGRAM/MACHINE CHECK LOGGING
*****************************************************************
* OPTIONAL FUNCTION SUPPORT
*****************************************************************
*INCLUDE RLOADER,XS1002  *C,K* RELOCATING PROGRAM LOADER (XL)
 INCLUDE RLOADERU,XS1002 *C,L* RELOCATING PROGRAM LOADER (UN-XL)
*INCLUDE EDXFLOAT,XS1002 *D*   FLOATING POINT ARITHMETIC
 INCLUDE NOFLOAT,XS1002  *D*   FOR SYSTEMS WITHOUT FLOATING POINT
*INCLUDE EBFLCVT,XS1002  *E*   EBCDIC/FLOATING PT CONV.
 INCLUDE QUEUEIO,XS1002  *F*   QUEUE PROCESSING SUPPORT
*****************************************************************
* SYSTEM SUPPORT -- INITIALIZATION
*****************************************************************
 INCLUDE EDXINIT,XS1002  *H*   SUPERVISOR INITIALIZATION
 INCLUDE DISKINIT,XS1002 *M*   DISK(ETTE) INITIALIZATION
 INCLUDE LOADINIT,XS1002 *C*   PROGRAM LOADER INITIALIZATION
 INCLUDE RW4963ID,XS1002 *M*   4963 FIXED HEAD REFRESH SUPPORT
 INCLUDE TERMINIT,XS1002 *1*   TERMINAL INITIALIZATION
 INCLUDE INIT4978,XS1002 *M*   4978 DISPLAY INITIALIZATION
*INCLUDE INIT4013,XS1002 *M*   DIGITAL I/O TERMINAL INIT
*INCLUDE $ACCARAM,XS1002 *3*   ACCA MULTI-LINE ADAPTER RAM LOAD
*INCLUDE BSCINIT,XS1002  *7*   BISYNC (BSCAM) INITIALIZATION
*INCLUDE $BSCARAM,XS1002 *7*   BISYNC MULT-LINE ADAPTER RAM LOAD
*INCLUDE TPINIT,XS1002   *8*   HCF (TPCOM) INITIALIZATION
*INCLUDE TIMRINIT,XS1002 *6*   4953/4955 TIMER INITIALIZATION
*INCLUDE CLOKINIT,XS1002 *6*   4952 TIMER INITIALIZATION
*INCLUDE SBIOINIT,XS1002 *M*   SENSOR I/O INITIALIZATION
*INCLUDE EXIOINIT,XS1002 *M*   EXIO INITIALIZATION
```

Sample Contents of $LNKCNTL (Version 2.0)

```
*************************************************************
*  COMMENTS MAY BE INCLUDED BY PUTTING AN '*' IN COLUMN 1. *
*  USE THIS TECHNIQUE TO OMIT UNNEEDED MODULES              *
*************************************************************
 OUTPUT  SUPVLINK,EDX002     ENTRY=$START
*************************************************************
* SUPERVISOR SUPPORT
*************************************************************
 INCLUDE EDXSYS,XS2002    *0* SYSTEM TABLES AND WORK AREAS
 INCLUDE ASMOBJ,EDX002    *0* OUTPUT FROM USER SYSTEM GENERATION
*INCLUDE EDXSVCX,XS2002   *0,K* TASK SUPERVISOR (XL)
 INCLUDE EDXSVCXU,XS2002 *0,L* TASK SUPERVISOR (UN-XL)
 INCLUDE EDXALU,XS2002    *0* EDL INSTRUCTION EMULATOR
 INCLUDE EDXSTART,XS2002 *0* INITIALIZATION & ERROR HANDLER
*************************************************************
* DEVICE SUPPORT -- DISK(ETTE)S
*************************************************************
 INCLUDE DISKIO,XS2002    *M* BASIC DISK(ETTE) SUPPORT
 INCLUDE D49624,XS2002    *M* 4962/4964 DISK(ETTE) SUPPORT
 INCLUDE D4963A,XS2002    *M* 4963 SUBSYSTEM SUPPORT
 INCLUDE D4966A,XS2002    *M* 4966 MAGAZINE SUPPORT
*************************************************************
* DEVICE SUPPORT -- TAPES
*************************************************************
*INCLUDE D4969A,XS2002    *M* 4969 TAPE SUPPORT
*************************************************************
* DEVICE SUPPORT -- TERMINALS
*************************************************************
*INCLUDE EDXTIO,XS2002    *1,K* BASIC TERMINAL SUPPORT (XL)
 INCLUDE EDXTIOU,XS2002   *1,L* BASIC TERMINAL SUPPORT (UN-XL)
*INCLUDE EDXTERMQ,XS2002 *1,K* ENQT/DEQT & TERMINAL QING (XL)
 INCLUDE EDXTRMQU,XS2002 *1,L* ENQT/DEQT & TERMINAL QING (UN-XL)
*INCLUDE IOS4979,XS2002   *M,K* 4978/4979 DISPLAY SUPPORT (XL)
 INCLUDE IOS4979U,XS2002 *M,L* 4978/4979 DISPLAY SUPPORT (UN-XL)
*INCLUDE IOS4974,XS2002   *M,K* 4973/4974 PRINTER SUPPORT (XL)
 INCLUDE IOS4974U,XS2002 *M,L* 4973/4974 PRINTER SUPPORT (UN-XL)
 INCLUDE IOSTERM,XS2002   *2*    REQD FOR TTY, ACCA, 4013 & 2741
 INCLUDE IOSTTY,XS2002    *M*    ASR 33/35 TELETYPEWRITER SUPPORT
*INCLUDE IOSACCA,XS2002   *3*    ASCII ACCA TERMINAL SUPPORT
*INCLUDE IOS4013,XS2002   *M*    DIGITAL I/O TERMINAL SUPPORT
*INCLUDE IOS2741,XS2002   *M*    2741 TERMINAL SUPPORT
*INCLUDE IOSVIRT,XS2002   *M,N* VIRTUAL TERMINAL SUPPORT
```

```
*************************************************************
* DEVICE SUPPORT -- TRANSLATION TABLES
*************************************************************
 INCLUDE TRASCII,XS2002  *4*   TELETYPEWRITER TRANSLATION
*INCLUDE TREBASC,XS2002  *3*   MIRROR IMAGE ASCII TRANSLATION
*INCLUDE TREBCD,XS2002   *5*   2741 EBDC TRANSLATION
*INCLUDE TRCRSP,XS2002   *5*   2741 CORRESPONDENCE TRANSLATION
*************************************************************
* DEVICE SUPPORT -- TIMERS
*************************************************************
*INCLUDE EDXTIMER,XS2002 *6*   4953/4955 TIMER (7840) SUPPORT
*INCLUDE EDXTIMR2,XS2002 *6*   4952 TIMER SUPPORT
*************************************************************
* DEVICE SUPPORT -- BINARY SYNCHRONOUS COMMUNICATIONS
*************************************************************
*INCLUDE BSCAM,XS2002    *7,K* BSC COMM. ACCESS SUPPORT (XL)
*INCLUDE BSCAMU,XS2002   *7,L* BSC COMM. ACCESS SUPPORT (UN-XL)
*INCLUDE TPCOM,XS2002    *8*   HOST COMMUNICATION SUPPORT
*************************************************************
* DEVICE SUPPORT -- SENSOR INPUT/OUTPUT
*************************************************************
*INCLUDE SBCOM,XS2002    *9*   BASIC SENSOR I/O SUPPORT
*INCLUDE IOLOADER,XS2002 *9,K* SENSOR I/O DEVICE OPEN (XL)
*INCLUDE IOLOADRU,XS2002 *9,L* SENSOR I/O DEVICE OPEN (UN-XL)
*INCLUDE SBAI,XS2002     *M*   ANALOG INPUT SUPPORT
*INCLUDE SBAO,XS2002     *M*   ANALOG OUTPUT SUPPORT
*INCLUDE SBDIDO,XS2002   *M*   DIGITAL INPUT/OUTPUT SUPPORT
*INCLUDE SBPI,XS2002     *M*   PROCESS INTERRUPT SUPPORT
*************************************************************
* DEVICE SUPPORT -- EXIO CONTROL
*************************************************************
*INCLUDE IOSEXIO,XS2002  *M*   EXIO DEVICE CONTROL SUPPORT
*************************************************************
* SYSTEM SUPPORT -- ERROR LOGGING
*************************************************************
 INCLUDE SYSLOG,XS2002   *A*   I/O ERROR LOGGING
*INCLUDE NOSYSLOG,XS2002 *A*   NO I/O ERROR LOGGING
 INCLUDE CIRCBUFF,XS2002 *B*   PROGRAM/MACHINE CHECK LOGGING
*************************************************************
* OPTIONAL FUNCTION SUPPORT
*************************************************************
*INCLUDE RLOADER,XS2002  *C,K* RELOCATING PROGRAM LOADER (XL)
 INCLUDE RLOADERU,XS2002 *C,L* RELOCATING PROGRAM LOADER (UN-XL)
*INCLUDE EDXFLOAT,XS2002 *D*   FLOATING POINT ARITHMETIC
 INCLUDE NOFLOAT,XS2002  *D*   FOR SYSTEMS WITHOUT FLOATING POINT
*INCLUDE EBFLCVT,XS2002  *E*   EBCDIC/FLOATING PT CONV.
 INCLUDE QUEUEIO,XS2002  *F*   QUEUE PROCESSING SUPPORT
*INCLUDE $DBUGNUC,XS2002 *G*   RESIDENT $DEBUG SUPPORT
```

```
*************************************************************
* SYSTEM SUPPORT -- INITIALIZATION
*************************************************************
 INCLUDE EDXINIT,XS2002  *H*   SUPERVISOR INITIALIZATION
 INCLUDE DISKINIT,XS2002 *M*   DISK(ETTE) INITIALIZATION
*INCLUDE TAPEINIT,XS2002 *M*   TAPE INITIALIZATION
 INCLUDE LOADINIT,XS2002 *C*   PROGRAM LOADER INITIALIZATION
 INCLUDE RW4963ID,XS2002 *M*   4963 FIXED HEAD REFRESH SUPPORT
 INCLUDE TERMINIT,XS2002 *1*   TERMINAL INITIALIZATION
 INCLUDE INIT4978,XS2002 *M*   4978 DISPLAY INITIALIZATION
*INCLUDE INIT4013,XS2002 *M*   DIGITAL I/O TERMINAL INIT
*INCLUDE $ACCARAM,XS2002 *3*   ACCA MULTI-LINE ADAPTER RAM LOAD
*INCLUDE BSCINIT,XS2002  *7*   BISYNC (BSCAM) INITIALIZATION
*INCLUDE $BSCARAM,XS2002 *7*   BISYNC MULT-LINE ADAPTER RAM LOAD
*INCLUDE TPINIT,XS2002   *8*   HCF (TPCOM) INITIALIZATION
*INCLUDE TIMRINIT,XS2002 *6*   4953/4955 TIMER INITIALIZATION
*INCLUDE CLOKINIT,XS2002 *6*   4952 TIMER INITIALIZATION
*INCLUDE SBIOINIT,XS2002 *M*   SENSOR I/O INITIALIZATION
*INCLUDE EXIOINIT,XS2002 *M*   EXIO INITIALIZATION
```

NOTES

```
*0*   Must be included first and in this order
*1*   Required if any terminals are installed, including 4973
*     or 4974
*2*   Required if IOSTTY, IOS2741, or IOSACCA is included
*3*   Required if non-2741 terminals are on ACCA
*4*   Required if IOSTTY is included
*5*   Either TREBCD or TRCRSP or both are required if IOS2741
*     is included, depending on the code used by the 2741
*     terminals - correspondence or ASCII
*6*   Attached TIMERS (feature 7840) and the 4952 native TIMER
*     are mutually exclusive.  Select the TIMER support
*     required for your configuration or none if no TIMER
*     support is required.
*7*   Required for binary synchronous communication using
*     BSCREAD/BSCWRITE or Remote Management Utility support.
*8*   Required for communication to a S/370 with the EDX Host
*     Communication Facility
*9*   Required if any Sensor I/O support is to be used
*     (AI,AO,DI,DO, or PI)
*A*   One, but not both, of these modules is required
*B*   Required if the in storage program check/machine check
*     log is to be kept
*C*   Required if programs are to be loaded from disk(ette).
*     If not included, an application program must be link
*     edited with the supervisor.
*D*   One, but not both, of these modules is required
*E*   Required for data formatting operations (GETEDIT,
*     PUTEDIT, FORMAT)
*F*   Required for queueing operations (FIRSTQ, NEXTQ, LASTQ,
*     DEFINEQ)
*G*   Required for program debugging ($DEBUG)
```

```
*H*   Required and must follow all of the previously listed
*     modules.
*     All other initialization modules must follow EDXINIT.
*J*   For starter supervisor use only
*K*   There are two versions of this module.  This one is
*     for systems that support the address translator
*     feature of the 4952 and 4955 processors.  Include this
*     version if your system is to support both the function
*     the module implements and the address translator
*     feature. (XL)
*L*   There are two versions of this module.  This one is
*     for systems that do not support the address translator
*     feature of the 4952 and 4955 processors.  Include this
*     version if your system is to support the function
*     the module implements, but not the address translator
*     feature. (UN-XL)
*M*   Optional module; required if device or feature is to be
*     supported.
*N*   Required if using Remote Management Utility with PASSTHRU
*     function.
  END
```

Note: You should include DDBFIX and CCBFIX with the other system intialization modules if you wixh to regenerate the starter system.

2. Enter an asterisk (*) in column one (1) of each INCLUDE statement not required to create your supervisor. The asterisk makes the statement a comment and the module with the asterisk is not included in your supervisor. Be sure that the system definition statements created in Step B agree with the modules you include in this step.

   The modules with note L can be used if your generated system is to execute either on a Series/1 without the address translator feature or on a 64KB 4952 processor. These modules do not support the address translator. The SYSTEM configuration statement must specify STORAGE as 64 or less and PARTS may not be specified.

3. Save the edited version of $LNKCNTL in a data set named LINKCNTL on EDX002.

Step D - Assemble and Link Edit the Supervisor

Edit $SUPPREP to use your allocated data sets.

1. Read the data set $SUPPREP from volume ASMLIB. Figure 10 on page 125 shows $SUPPREP.

```
LOG         $SYSPRTR
JOB         $SUPPREP
REMARK   **ENTER GO AFTER XS2002 HAS BEEN VARIED ONLINE**
PAUSE
PROGRAM     $EDXASM,ASMLIB
NOMSG
PARM
DS          $EDXDEFS,EDX002
DS          ASMWORK,EDX002
DS          ASMOBJ,EDX002
EXEC
JUMP        ENDJOB,GT,4
PROGRAM     $LINK,ASMLIB
NOMSG
PARM        $SYSPRTR
DS          LINKCNTL,EDX002
DS          LEWORK1,EDX002
DS          LEWORK2,EDX002
EXEC
JUMP        ENDJOB,GT,4
PROGRAM     $UPDATE,EDX002
NOMSG
PARM        $SYSPRTR SUPVLINK,EDX002 $EDXNUCT,EDX002 YES
EXEC
LABEL       ENDJOB
EOJ
```

Figure 10. Example of V2.0 Procedure $SUPPREP on ASMLIB

2. Modify any of the procedure statements, particularly the DS data set names, and volumes to satisfy your conventions. No changes are necessary for your first supervisor generation if you allocated all the required data sets as instructed in Step A. $EDXNUCT is automatically allocated by the $UPDATE Step and you may wish to change this name to $EDXNUCx (x = any alphameric character) to save different supervisor versions in individual data sets. The supervisor name must start with the seven characters $EDXNUC.

3. Save the edited version of $SUPPREP in a data set named SUPPREPS on EDX002.

Step E - Format the Supervisor

When you invoke the procedure SUPPREPS, the job stream assembles and link edits $EDXDEF and formats the supervisor.

1. Vary on diskette XS1002 (Version 1.1) or XS2002 (Version 2).

2. Load utility program $JOBUTIL. When prompted for the procedure name, reply SUPPREPS,EDX002.

3. When $JOBUTIL completes execution, examine the output printed on $SYSPRTR for errors. Errors are usually caused by incorrect editing of $EDXDEF, $LNKCNTL, or $SUPPREP. If errors are found, examine your supervisor specification and link edit statements and then edit $EDXDEFS, LINKCNTL, or SUPPREPS as necessary.

   When you have corrected the errors, reload $JOBUTIL to repeat the procedure.

   Unresolved WXTRN messages resulting from the execution of $LINK can occur, and you should examine the messages to determine whether the referenced names refer to modules that you require in your supervisor.

   An unresolved WXTRN of $PROG1 will normally occur unless you link edit an application program with the supervisor, as is described in "Other Considerations" on page 128.

   Unresolved EXTRN messages should not occur if a valid supervisor has been created. A complete listing of all supervisor module section names and entry point labels is included in Appendix B.

Step F - Test the Generated Supervisor

Test the generated supervisor for a disk based system.

1. Load the utility program $COPY or $COPYUT1 to copy $EDXNUCT into $EDXNUC on EDX002.

   Note: Procedure SUPPREPS stores the created supervisor as member $EDXNUCT on EDX002.

2. IPL the system from volume EDX002 to load the new supervisor.

   Wait until the system is initialized before loading a program. If your system has timers, the system is initialized when the SET TIME AND DATE USING $T message appears (or when the time and date are printed). If your system does not have timers, the system is initialized when it enters the wait state after the storage map has been displayed.

3. Test the supervisor by executing utility programs that exercise the various supervisor components (such as disk I/O, sensor I/O, etc.)

Notes:

1. If the new supervisor fails to operate correctly, you must restore the original contents of $EDXNUC by IPLing from a diskette. Use $COPY or $COPYUT1 to copy the starter supervisor from diskette UT3001 or UT4001 to $EDXNUC on EDX002.

2. If any errors are encountered, repeat Steps B through E of this procedure.

3. If you relocated any volumes in a tailored system generation (particularly EDX002), copy the new supervisor into the $EDXNUC data set on a copy of the utility diskette (UT3001 or UT4001) and perform a complete system installation.

4. The actual addresses of CSECT and ENTRY point labels in the $EDXNUCT or $EDXNUC modules stored on disk will be X'100' greater than those shown on the link edit map. This is because $UPDATE adds a 256 byte header to all $EDXNUCx modules.

Step G - Verify the System Generation Process

To verify that the system generation has been performed successfully:

1. Assemble and execute the sample program CALCSRC.

   Note: CALCDEMO source instructions are located in the data set CALCSRC on the disk volume EDX002. To assemble CALCDEMO, refer to the procedure for program preparation described in Utilities, Operator Commands, Program Preparation, Messages and Codes.

2. When the assembly is complete, load the test program into storage for execution by using the $L operator command.

3. When you receive the prompts A= and B=, enter any decimal integer values less than 2 billion, followed by a carriage return or ENTER after each entry.

   A sample of the entries and resulting output follows:

```
> $L CALCDEMO

CALCDEMO      3P,10:59:55, LP= 7F00
Press ATTENTION and enter CALC or STOP
> CALC

A = 12
B = 52

A + B =            64
A - B =           -40
A * B =           624
A / B =             0 REMAINDER =           12
Press ATTENTION and enter CALC or STOP
> CALC
```

OTHER CONSIDERATIONS

System Generation without the Program Preparation Facility

For Series/1 systems that do not include the Program Preparation Facility, installation requires the following general steps:

1. Assemble and link edit the supervisor for the target Series/1 on a system that supports program preparation.

2. Assemble application programs for the target Series/1 on a system that supports program preparation.

3. Use utility program $INITDSK to initialize one or more diskettes with IPL text, space for the supervisor program, and a library to contain your application programs.

4. Transfer your supervisor to $EDXNUC on diskette(s) with either $COPY or $COPYUT1.

5. Copy $LOADER, any of the utilities, and the application programs that will be required on the target Series/1, onto the diskette(s) with $COPYUT1.

6. Install the diskette(s) on the target machine for execution.

Program Loading from Diskettes

If multiple diskettes are processed on a single diskette unit, each diskette must contain the program $LOADER in the same location. To load a program into storage from diskette, the diskette containing the program must be online ($VARYON) when the LOAD instruction or the $L command is executed.

Automatic Application Initialization and Restart

You can design your system so that your application program(s) are automatically started following a manual IPL of the system or an automatic IPL invoked by the restoration of power after a power outage.

There is no system requirement for operator involvement in the IPL procedure, other than to insure the IPL mode switch is in the "AUTO IPL" position and to turn on the power for the initial Series/1 IPL. Any other requirement for operator involvement (such as for entry of time and date) is a function of your application.

The automatic application initialization facility allows you to start an application immediately after the system initialization process has been completed.

Consideration must be given to the type of program control the Event Driven Executive will be supporting. In a multiprogramming, multitasking system, the relocatable loader loads programs from disk or diskette to storage. In a single program, multitasking system, a single application program is link edited with the Event Driven Executive supervisor and loaded at IPL time. In either system the program may consist of a primary task or a primary task and secondary task(s).

Multiprogramming, Multitasking System

In a multiprogramming, multitasking system, the automatic application initialization facility requires a system with the Event Driven Executive program load facility and is loaded (via IPL) from disk or diskette. Further, if your system contains both disk and diskette devices, then the automatic IPL must be performed with a disk as the IPL source.

The facility works in the following manner. At the end of the regular system initialization process (when all I/O devices have been prepared and the system is ready for normal operation), a LOAD instruction will be issued for your program

named $INITIAL which must be located on the IPL volume. If no such program exists, no further action is taken and programs must be initiated via $L commands entered at terminals. If $INITIAL does exist, it is loaded for execution. The functions which can be performed by $INITIAL, such as data base initialization, data logging, outboard device initialization, and loading of other application programs, are entirely under your control.

$INITIAL is loaded in partition one immediately after the supervisor. The system attempts to pass to it a one word parameter indicating the IPL mode. Zero in this word indicates a manual IPL. A one in this word indicates "Auto IPL". In order to receive this word, PARM=1 must have been coded in the PROGRAM statement of $INITIAL. If PARM=1 is not coded, the IPL mode cannot be determined.

One function that $INITIAL can perform differently for manual versus automatic IPL situations is the setting of the supervisor time and date. In a manual IPL situation the time and date are normally entered by an operator via the $T command. In an unattended auto-IPL situation it may be required that $INITIAL obtain the time and date information from such sources as an external battery operated clock (connected to Series/1 Digital Input features), a checkpoint file maintained on disk or diskette by the application program during normal operation, etc.

Regardless of the source of the time and date information, the following instructions will move the information from $INITIAL to the supervisor time and date table. If $INITIAL is to be assembled by $EDXASM, then the statement COPY PROGEQU must be included after the PROGRAM statement to define the label $TIMRTBL. In the following example TIMRDATA is a six word table within $INITIAL containing the time and date as hexadecimal values in the sequence hours, minutes, seconds, month, day, year.

For example, the following code sets the clock to 13:24:05 and the date to December 25, 1979.

```
          MOVE   #1,$TIMRTBL
          MOVE   (8,#1),TIMRDATA,6
            .
            .
            .
TIMRDATA  DC     X'000D'
          DC     X'0018'
          DC     X'0005'
          DC     X'000C'
          DC     X'0019'
          DC     X'004F'
```

$INITIAL can also load additional programs. For example, if you wish to have automatic initialization of the Multiple Terminal Manager in partition two, the Indexed Access Method in partition three, and the Session Manager in partition four, your $INITIAL program would have the following statements:

```
          LOAD   $MTM,PART=2,ERROR=NOMTM
          LOAD   $IAM,PART=3,ERROR=NOIAM
          LOAD   $SMMAIN,PART=4,ERROR=NOSESS
            .
            .
            .
NOIAM     (Routine to handle the error)
NOSESS    (Routine to handle the error)
NOMTM     (Routine to handle the error)
```

$INITIAL can have data sets and overlay programs defined in its PROGRAM statement but cannot use the '??' option which implies data set definition at load time. Note that no program load logging message is printed out with $INITIAL and if any errors occur during the load process such as unresolved data set names, no logging message will be printed. If at any time it is desired to disable the automatic initialization feature for a period of time, the program $INITIAL should be renamed.

Single Program, Multitasking System

In a single program, multitasking system, the relocatable loader is excluded from your supervisor and the disk or diskette is used for data only.

A single application program must be link edited with the supervisor to form a single load module that can be loaded at IPL time. This application program must contain a CSECT named

$PROG1. In addition, the PROGSTOP instruction is not permitted in this program. Therefore, the program source module should contain statements as follows:

```
$PROG1   CSECT
MAIN     PROGRAM  START
           .
           .
           .
         ENDPROG
         END
```

When the supervisor is loaded at IPL time and the multiprogramming feature is not included, the above program is automatically started.

To remove the multiprogramming feature from the Event Driven Executive supervisor, do not include the module "RLOADER" in the custom system generation and delete the transient loader ($LOADER) from the system resident disk volume (normally EDX002), if there is a disk on the system.

Initializing Secondary Volumes

If you create a supervisor that defines secondary disk volumes in addition to those defined as primary volumes, the additional secondary volumes must be initialized before they can be used for data or program storage. To initialize a secondary volume, execute the utility program $INITDSK and create the directory for each additional volume defined. You must initialize a fixed head volume before doing an IPL from it. This allows the system to search for the program $LOADER during initialization.

Creating a Supervisor for Another Series/1

You can create a supervisor for another Series/1 on a Series/1 with the Program Preparation Facility. Follow the procedure for "Generating the Supervisor" on page 115, but save the edited copies of $EDXDEF and $LNKCNTL under different names for each different Series/1 hardware configuration. SUPPREPS must be modified to support the different data set names.

The resulting supervisor programs generated would be stored under different $EDXNUCx names. These can then be copied to diskette from $EDXNUCx for transfer to the target Series/1. The diskette must have been initialized previously by the utility $INITDSK to include IPL text and space for a supervisor.

Sample Configurations

The following figures show sample configurations for $EDXDEF. For actual definitions refer to the Program Directory.

```
          SYSTEM   STORAGE=96,MAXPROG=(3,2,3),          C
                PARTS=(32,6,10)
          DISK     DEVICE=4962-1,ADDRESS=03,            C
                VOLSER=EDX002,VOLORG=0,VOLSIZE=100,     C
                LIBORG=241
          DISK     DEVICE=4962-1,VOLSER=EDX003,         C
                BASEVOL=EDX002,VOLORG=100,             C
                VOLSIZE=101,LIBORG=1
          DISK     DEVICE=4962-1,VOLSER=ASMLIB,         C
                BASEVOL=EDX002,VOLORG=201,             C
                VOLSIZE=100,LIBORG=1
          DISK     DEVICE=4964,ADDRESS=02
          DISK     DEVICE=4966,ADDRESS=22,END=YES
$SYSLOG   TERMINAL DEVICE=4978,ADDRESS=04,              C
                HDCOPY=$SYSPRTR
$SYSLOGA  TERMINAL DEVICE=TTY,ADDRESS=00,CRDELAY=4,     C
                PAGSIZE=24,BOTM=23,SCREEN=YES
$SYSPRTR  TERMINAL DEVICE=4974,ADDRESS=01,END=YES
          ENTRY    $EDXPTCH
$EDXPTCH  DATA     128F'0'          SYSTEM PATCH AREA
          END
```

Figure 11. Example of $EDXDEF: Configuration for 4962-1F (9.3MB disk)

```
         SYSTEM   STORAGE=64,MAXPROG=5
         DISK     DEVICE=4962-1F,ADDRESS=03,                C
               VOLSER=EDX002,VOLORG=0,VOLSIZE=100,         C
               LIBORG=241,FHVOL=FHVOL
         DISK     DEVICE=4962-1F,VOLSER=EDX003,             C
               BASEVOL=EDX002,VOLORG=100,                  C
               VOLSIZE=101,LIBORG=1
         DISK     DEVICE=4962-1F,VOLSER=ASMLIB,             C
               BASEVOL=EDX002,VOLORG=201,                  C
               VOLSIZE=100,LIBORG=1
         DISK     DEVICE=4964,ADDRESS=02
         DISK     DEVICE=4966,ADDRESS=22,END=YES
$SYSLOG  TERMINAL DEVICE=4979,ADDRESS=04,                   C
               HDCOPY=$SYSPRTR
$SYSLOGA TERMINAL DEVICE=TTY,ADDRESS=00,CRDELAY=4,          C
               PAGSIZE=24,BOTM=23,SCREEN=YES
$SYSPRTR TERMINAL DEVICE=4974,ADDRESS=01,END=YES
         ENTRY    $EDXPTCH
$EDXPTCH DATA     128F'0'            SYSTEM PATCH AREA
         END
```

Figure 12. Example of $EDXDEF: Configuration for 4962-1 (9.3MB fixed-head disk)

```
         SYSTEM   STORAGE=196,MAXPROG=(1,2,1,3,4,1),    C
               PARTS=(9,12,7,4,20,32)
         DISK     DEVICE=4962-3,ADDRESS=03,             C
               VOLSER=EDX002,VOLORG=0,VOLSIZE=100,      C
               LIBORG=361
         DISK     DEVICE=4962-3,VOLSER=EDX003,          C
               BASEVOL=EDX002,VOLORG=100,               C
               VOLSIZE=101,LIBORG=1
         DISK     DEVICE=4962-3,VOLSER=ASMLIB,          C
               BASEVOL=EDX002,VOLORG=201,               C
               VOLSIZE=100,LIBORG=1
         DISK     DEVICE=4964,ADDRESS=02
         DISK     DEVICE=4966,ADDRESS=22,END=YES
$SYSLOG  TERMINAL DEVICE=4979,ADDRESS=04,               C
               HDCOPY=$SYSPRTR
$SYSLOGA TERMINAL DEVICE=TTY,ADDRESS=00,CRDELAY=4,      C
               PAGSIZE=24,BOTM=23,SCREEN=YES
$SYSPRTR TERMINAL DEVICE=4974,ADDRESS=01,END=YES
         ENTRY    $EDXPTCH
$EDXPTCH DATA     128F'0'          SYSTEM PATCH AREA
         END
```

Figure 13. Example of $EDXDEF: Configuration for 4962-3 or 4962-4 (13.9MB disk)

```
          SYSTEM   STORAGE=128,MAXPROG=(10,10,10),       C
                PARTS=(32,9,23)
          DISK     DEVICE=4964,ADDRESS=02,TASK=YES
          DISK     DEVICE=4964,ADDRESS=12,TASK=YES
          DISK     DEVICE=4963-23,ADDRESS=48,            C
                VOLSER=EDX002,VOLORG=0,VOLSIZE=100,      C
                LIBORG=129,FHVOL=FHVOL,TASK=YES
          DISK     LIBORG=1,VOLSIZE=80,BASEVOL=EDX002,   C
                VOLSER=ASMLIB,DEVICE=4963-23,VOLORG=100
          DISK     DEVICE=4963-23,VOLSER=PRGRMS,         C
                BASEVOL=EDX002,VOLORG=180,               C
                VOLSIZE=33,LIBORG=1
          DISK     DEVICE=4963-23,VOLSER=MTMSTR,         C
                BASEVOL=EDX002,VOLORG=213,               C
                VOLSIZE=22,LIBORG=1
          DISK     DEVICE=4963-23,VOLSER=SCRNS,          C
                BASEVOL=EDX002,VOLORG=235,               C
                VOLSIZE=12,LIBORG=1
          DISK     DEVICE=4963-23,VOLSER=EDX003,         C
                BASEVOL=EDX002,VOLORG=100,               C
                VOLSIZE=111,LIBORG=1,END=YES
$SYSLOG   TERMINAL DEVICE=4978,ADDRESS=2A,               C
                HDCOPY=$SYSPRTR,PART=1
$SYSLOGA TERMINAL DEVICE=TTY,ADDRESS=00,CRDELAY=4,       C
                PAGSIZE=24,BOTM=23,SCREEN=YES
D497800   TERMINAL DEVICE=4978,ADDRESS=24,               C
                HDCOPY=LPRINTER,PART=2
D497801   TERMINAL DEVICE=4978,ADDRESS=25,               C
                HDCOPY=$SYSPRTR,PART=3
D497802   TERMINAL DEVICE=4978,ADDRESS=26,               C
                HDCOPY=$SYSPRTR,PART=3
D497803   TERMINAL DEVICE=4978,ADDRESS=27,               C
                HDCOPY=$SYSPRTR,PART=3
D497804   TERMINAL DEVICE=4978,ADDRESS=28,               C
                HDCOPY=$SYSPRTR,PART=3
D497805   TERMINAL DEVICE=4978,ADDRESS=29,               C
                HDCOPY=$SYSPRTR,PART=3
$TERMA    TERMINAL DEVICE=VIRT,ADDRESS=TERMB,SYNC=YES
$TERMB    TERMINAL DEVICE=VIRT,ADDRESS=TERMA
$SYSPRTR TERMINAL DEVICE=4974,ADDRESS=01
LPRINTER TERMINAL DEVICE=4973,ADDRESS=21,END=YES
          BSCLINE  ADDRESS=19,TYPE=PT,RETRIES=5,         C
                MC=NO,END=YES
          TIMER    ADDRESS=40
          ENTRY    $EDXPTCH
$EDXPTCH DATA     128F'0'          SYSTEM PATCH AREA
          END
```

Figure 14. $EDXDEF and Multiple Terminal Manager Volume Definition: Configuration for 4963-23 (23MB disk)

```
         SYSTEM   STORAGE=128,MAXPROG=(3,6),                C
               PARTS=(32,32),COMMON=EDXSVC
         DISK     DEVICE=4963-29,ADDRESS=48,                C
               VOLSER=EDX002,VOLORG=0,VOLSIZE=100,         C
               LIBORG=129
         DISK     DEVICE=4963-29,VOLSER=EDX003,             C
               BASEVOL=EDX002,VOLORG=100,                  C
               VOLSIZE=100,LIBORG=1
         DISK     DEVICE=4963-29,VOLSER=ASMLIB,             C
               BASEVOL=EDX002,VOLORG=200,                  C
               VOLSIZE=100,LIBORG=1
         DISK     DEVICE=4964,ADDRESS=02
         DISK     DEVICE=4966,ADDRESS=22,END=YES
$SYSLOG  TERMINAL DEVICE=4979,ADDRESS=04,                   C
               HDCOPY=$SYSPRTR
$SYSLOGA TERMINAL DEVICE=TTY,ADDRESS=00,CRDELAY=4,          C
               PAGSIZE=24,BOTM=23,SCREEN=YES
$SYSPRTR TERMINAL DEVICE=4974,ADDRESS=01,END=YES
         ENTRY    $EDXPTCH
$EDXPTCH DATA     128F'0'          SYSTEM PATCH AREA
         END
```

Figure 15. Example of $EDXDEF: Configuration for 4963-29 (29MB disk)

```
         SYSTEM   STORAGE=128,MAXPROG=(4,4),                C
               PARTS=(32,32),DATEFMT=MMDDYY
         DISK     DEVICE=4963-23,ADDRESS=48,                C
               VOLSER=EDX002,VOLORG=0,VOLSIZE=100,          C
               LIBORG=129,FHVOL=FHVOL
         DISK     DEVICE=4963-23,VOLSER=EDX003,             C
               BASEVOL=EDX002,VOLORG=100,                   C
               VOLSIZE=100,LIBORG=1
         DISK     DEVICE=4963-23,VOLSER=ASMLIB,             C
               BASEVOL=EDX002,VOLORG=200,                   C
               VOLSIZE=100,LIBORG=1
         DISK     DEVICE=4964,ADDRESS=02
         DISK     DEVICE=4966,ADDRESS=22,END=YES
$SYSLOG  TERMINAL DEVICE=4979,ADDRESS=04,                   C
               HDCOPY=$SYSPRTR
$SYSLOGA TERMINAL DEVICE=TTY,ADDRESS=00,CRDELAY=4,          C
               PAGSIZE=24,BOTM=23,SCREEN=YES
$SYSPRTR TERMINAL DEVICE=4974,ADDRESS=01,END=YES
         ENTRY    $EDXPTCH
$EDXPTCH DATA     128F'0'          SYSTEM PATCH AREA
         END
```

Figure 16. Example of $EDXDEF with date format specified: Configuration for 4963-23 (23MB fixed-head disk)

```
         SYSTEM   STORAGE=256,                              C
               MAXPROG=(3,1,5,2,2,1,1,4),                   C
               PARTS=(15,4,21,13,17,11,8,23)
         DISK     DEVICE=4963-64,ADDRESS=48,                 C
               VOLSER=EDX002,VOLORG=0,VOLSIZE=46,           C
               LIBORG=129
         DISK     DEVICE=4963-64,VOLSER=EDX003,              C
               BASEVOL=EDX002,VOLORG=46,                    C
               VOLSIZE=46,LIBORG=1
         DISK     DEVICE=4963-64,VOLSER=ASMLIB,              C
               BASEVOL=EDX002,VOLORG=92,                    C
               VOLSIZE=46,LIBORG=1
         DISK     DEVICE=4963-64,VOLSER=EDX004,              C
               BASEVOL=EDX002,VOLORG=138,                   C
               VOLSIZE=46,LIBORG=1
         DISK     DEVICE=4963-64,VOLSER=EDX005,              C
               BASEVOL=EDX002,VOLORG=184,                   C
               VOLSIZE=46,LIBORG=1
         DISK     DEVICE=4963-64,VOLSER=EDX006,              C
               BASEVOL=EDX002,VOLORG=230,                   C
               VOLSIZE=46,LIBORG=1
         DISK     DEVICE=4963-64,VOLSER=EDX007,              C
               BASEVOL=EDX002,VOLORG=276,                   C
               VOLSIZE=46,LIBORG=1
         DISK     DEVICE=4963-64,VOLSER=EDX008,              C
               BASEVOL=EDX002,VOLORG=322,                   C
               VOLSIZE=46,LIBORG=1
         DISK     DEVICE=4964,ADDRESS=02,VERIFY=NO
         DISK     DEVICE=4966,ADDRESS=22,                    C
               VERIFY=NO,END=YES
$SYSLOG  TERMINAL DEVICE=4979,ADDRESS=04,                    C
               HDCOPY=$SYSPRTR
$SYSLOGA TERMINAL DEVICE=TTY,ADDRESS=00,CRDELAY=4,           C
               PAGSIZE=24,BOTM=23,SCREEN=YES
$SYSPRTR TERMINAL DEVICE=4974,ADDRESS=01,END=YES
         ENTRY    $EDXPTCH
$EDXPTCH DATA     128F'0'          SYSTEM PATCH AREA
         END
```

Figure 17. Example of $EDXDEF: Configuration for 4963-64 (64MB disk) with a mapping of all (358) available cylinders

```
         SYSTEM   STORAGE=96,MAXPROG=(3,4),                C
               PARTS=(16,18),COMMON=START
         DISK     DEVICE=4963-58,ADDRESS=48,               C
               VOLSER=EDX002,VOLORG=0,VOLSIZE=46,          C
               LIBORG=129,FHVOL=FHVOL
         DISK     DEVICE=4963-58,VOLSER=EDX003,            C
               BASEVOL=EDX002,VOLORG=46,                   C
               VOLSIZE=46,LIBORG=1
         DISK     DEVICE=4963-58,VOLSER=ASMLIB,            C
               BASEVOL=EDX002,VOLORG=92,                   C
               VOLSIZE=46,LIBORG=1
         DISK     DEVICE=4964,ADDRESS=02
         DISK     DEVICE=4966,ADDRESS=22,END=YES
$SYSLOG  TERMINAL DEVICE=4979,ADDRESS=04,                  C
               HDCOPY=$SYSPRTR
$SYSLOGA TERMINAL DEVICE=TTY,ADDRESS=00,CRDELAY=4,         C
               PAGSIZE=24,BOTM=23,SCREEN=YES
$SYSPRTR TERMINAL DEVICE=4974,ADDRESS=01,END=YES
         ENTRY    $EDXPTCH
$EDXPTCH DATA     128F'0'          SYSTEM PATCH AREA
         END
```

Figure 18. Example of $EDXDEF: Configuration for 4963-58 (58MB fixed-head disk)

```
          SYSTEM   STORAGE=256,MAXPROG=(5,5,5,5),          C
                PARTS=(16,32,32,32),
          DISK     DEVICE=4962-3,ADDRESS=03,               C
                VOLSER=EDX002,VOLORG=0,VOLSIZE=100,       C
                LIBORG=361,VERIFY=NO,TASK=YES
          DISK     DEVICE=4962-3,VOLSER=EDX003,            C
                BASEVOL=EDX002,VOLORG=100,                C
                VOLSIZE=51,LIBORG=1
          DISK     DEVICE=4962-3,VOLSER=CDRSRC,            C
                BASEVOL=EDX002,VOLORG=151,                C
                VOLSIZE=50,LIBORG=1
          DISK     DEVICE=4962-3,VOLSER=ASMLIB,            C
                BASEVOL=EDX002,VOLORG=201,                C
                VOLSIZE=100,LIBORG=1
          DISK     DEVICE=4964,ADDRESS=02,                 C
                VERIFY=NO,TASK=YES
          TAPE     DEVICE=4969,ADDRESS=4C,ID=TAPE01,       C
                 DENSITY=DUAL,LABEL=BLP,TASK=YES
          DISK     DEVICE=4966,ADDRESS=22,                 C
           VERIFY=NO,TASK=YES,END=YES
CDRVTA    TERMINAL DEVICE=VIRT,ADDRESS=CDRVTB,             C
                LINSIZE=132,SYNC=YES
CDRVTB    TERMINAL DEVICE=VIRT,ADDRESS=CDRVTA,             C
                LINSIZE=132
$SYSLOG   TERMINAL DEVICE=4979,ADDRESS=04,                 C
                HDCOPY=$SYSPRTR,PART=2
$SYSLOGA  TERMINAL DEVICE=4978,ADDRESS=24,                 C
                HDCOPY=$SYSPRTR,PART=3
$SYSPRTR  TERMINAL DEVICE=4974,ADDRESS=01,END=YES
          TIMER ADDRESS=40
          BSCLINE ADDRESS=68,TYPE=PT,RETRIES=6,MC=YES
          BSCLINE ADDRESS=69,TYPE=PT,RETRIES=6,MC=YES
          BSCLINE ADDRESS=6A,TYPE=SA,RETRIES=6,MC=YES
          BSCLINE ADDRESS=6B,TYPE=SA,RETRIES=6,MC=YES,     C
                END=YES
          ENTRY    $EDXPTCH
$EDXPTCH  DC       128F'0'
          END
```

Figure 19. Example of $EDXDEF: Configuration for 4969 tape and Remote Management Utility using the PASSTHRU function

PART III - THE INDEXED ACCESS METHOD

Part III is organized into two chapters. "Chapter 8. Overview of the Indexed Access Method" on page 145 describes the features, components, and functions that comprise the Indexed Access Method. It also provides an overview of the indexed data set, how the Indexed Access Method processes the data set, how to prepare your applications, and how to get your data into an indexed data set.

"Chapter 9. Planning and Designing Indexed Applications" on page 159 contains detailed information on how to define, create, access, and maintain an indexed data set and how to handle errors. It also explains how indexed data sets are structured and managed.

CHAPTER 8. OVERVIEW OF THE INDEXED ACCESS METHOD

The Indexed Access Method licensed program is a data management facility that operates under the Event Driven Executive. It allows you to build, maintain, and access indexed data sets. In an indexed data set, each of your records is identified by the contents of a predefined field called a **key**. The Indexed Access Method builds into the data set an index of keys that provides access to your records.

The Indexed Access Method offers the following features:

- Direct and sequential processing. Multiple levels of indexing are used for direct access, and sequence chaining of data blocks is used for sequential access.

- Support for high insert and delete activity without significant performance degradation. Free space can be distributed throughout the data set and in a free pool at the end of the data set so that new records can be inserted. The space occupied by a deleted record is immediately available for new records.

- Concurrent access to a single data set by several requests. These requests can be from one or more programs. Data integrity is maintained by a file, block, and record level locking system that prevents other programs from accessing the portion of the file being modified.

- Implementation as a separate task. A single copy of the Indexed Access Method executes and coordinates all requests. A buffer pool supports all requests and optimizes the space required for physical I/O; the only buffer required in an application program is the one for the record being processed.

- A utility program ($IAMUT1) which allows you to create, format, load, unload, and reorganize an indexed data set.

- File compatibility. Data files created by the Event Driven Executive Indexed Access Method are compatible with those created by the IBM Series/1 Realtime Programming System Indexed Access Method licensed program, 5719-AM1, provided that the block size is a multiple of 256.

- Data Protection. All input/output operations are performed by system functions. Therefore, all data protection facilities offered by the system also apply to indexed files. The following additional data protection is provided:

- The exclusive option - specifies that the file is for the exclusive use of a requester.

- Record locking - automatically prevents two requests from accessing the same data record at the same time.

- Immediate write back - causes all updated records to be written back to the file immediately.

- Accidental key modification is prevented - this helps ensure that your index matches the corresponding data.

DEVICES SUPPORTED

The Indexed Access Method supports indexed data sets on the following direct access devices:

- 4962 Disk Storage Unit
- 4963 Disk Subsystem
- 4964 Diskette Unit
- 4966 Diskette Magazine Unit

FUNCTIONS

Functions available include those that can be called from an application program and a utility to define and maintain an indexed data set.

I/O Requests

I/O requests allow you to build an indexed data set and to perform direct or sequential processing on that data set. Routines using these functions are written in Event Driven Language and can be included in programs written in any language that supports the calling of Event Driven Executive Language routines.

You request the services of the Indexed Access Method through the Event Driven Language CALL instruction in the following general form:

```
CALL IAM,(func),iacb,(parm3),(parm4),(parm5)
```

For information on coding the parameters and functions, refer to the Language Reference.

The following requests can be invoked:

Operands	Description
PROCESS	Builds an Indexed Access Control Block (IACB) and connects it to an indexed data set. You can then use the IACB to issue requests to that data set to read, update, insert, and delete records. A program can issue multiple PROCESS functions to obtain multiple IACBs for the same data set, enabling the data set to be accessed by several requests concurrently within the same program.
LOAD	Similar to PROCESS but used to load or extend the initial collection of records.
GET	Directly retrieves a single record from the data set. If you specify the update mode, the record is locked (made unavailable to other requests) and held for possible modification or deletion. Use GET to retrieve a single record from the data set.
GETSEQ	Sequentially retrieves a single record from the data set. If you specify the update mode, the record is locked (made unavailable to other requests) and held for possible modification or deletion. Use GETSEQ when you are performing sequential operations.
PUT	Loads or inserts a new record depending on whether the data set was opened with the LOAD or PROCESS request. Use PUT when you are adding records to a data set.
PUTUP	Replaces a record that is being held for update. Use PUTUP to modify a record.
PUTDE	Deletes a record that is being held for update. Use PUTDE to delete a record.
RELEASE	Releases a record that is being held for update. Use RELEASE when a record that was retrieved for update is not changed.
DELETE	Deletes a single record, identified by its key, from the data set. Use DELETE to delete a record; unlike PUTDE, the record cannot have been retrieved for update.
ENDSEQ	Terminates sequential processing.

EXTRACT Provides information about the file (from the File Control Block).

DISCONN Disconnects an IACB from an indexed data set, thereby releasing any locks held by that IACB; writes out all buffers associated with the data set; and releases the storage used by the IACB.

The $IAMUT1 Utility

The $IAMUT1 utility formats, defines, creates, and writes control information to the indexed data set. Indexed Access Method requests can be used only on data sets defined either by this utility or by the Realtime Programming System Indexed Access Method. $IAMUT1 is described in the Utilities, Operator Commands, Program Preparation, Messages and Codes manual.)

OPERATION OF THE INDEXED ACCESS METHOD

The Indexed Access Method performs I/O operations by using standard data management requests.

A single copy of the Indexed Access Method load module $IAM serves the entire system. It can be loaded automatically at IPL time through the automatic initialization capability (refer to "Automatic Application Initialization and Restart" on page 129), or it can be loaded manually by using the $L operator command. However, since the link module loads $IAM automatically, $IAM does not need to be loaded before it is used by any program. Once loaded, the Indexed Access Method remains in storage until cancelled with the $C operator command.

$IAM can be loaded into any partition, including partition one. It can be invoked (through the link module) from any partition, including the partition it is in. Figure 20 on page 149 shows an example of a system containing the Indexed Access Method.

INDEXED DATA SETS - OVERVIEW

You can organize a collection of data into an indexed data set if the data consists of fixed-length records and if each record can be uniquely identified by the contents of a single predefined field called the key. In an indexed data set, the records are arranged in ascending order by key. Reserved space, called **free space**, can be distributed throughout the data set so that records can be inserted.

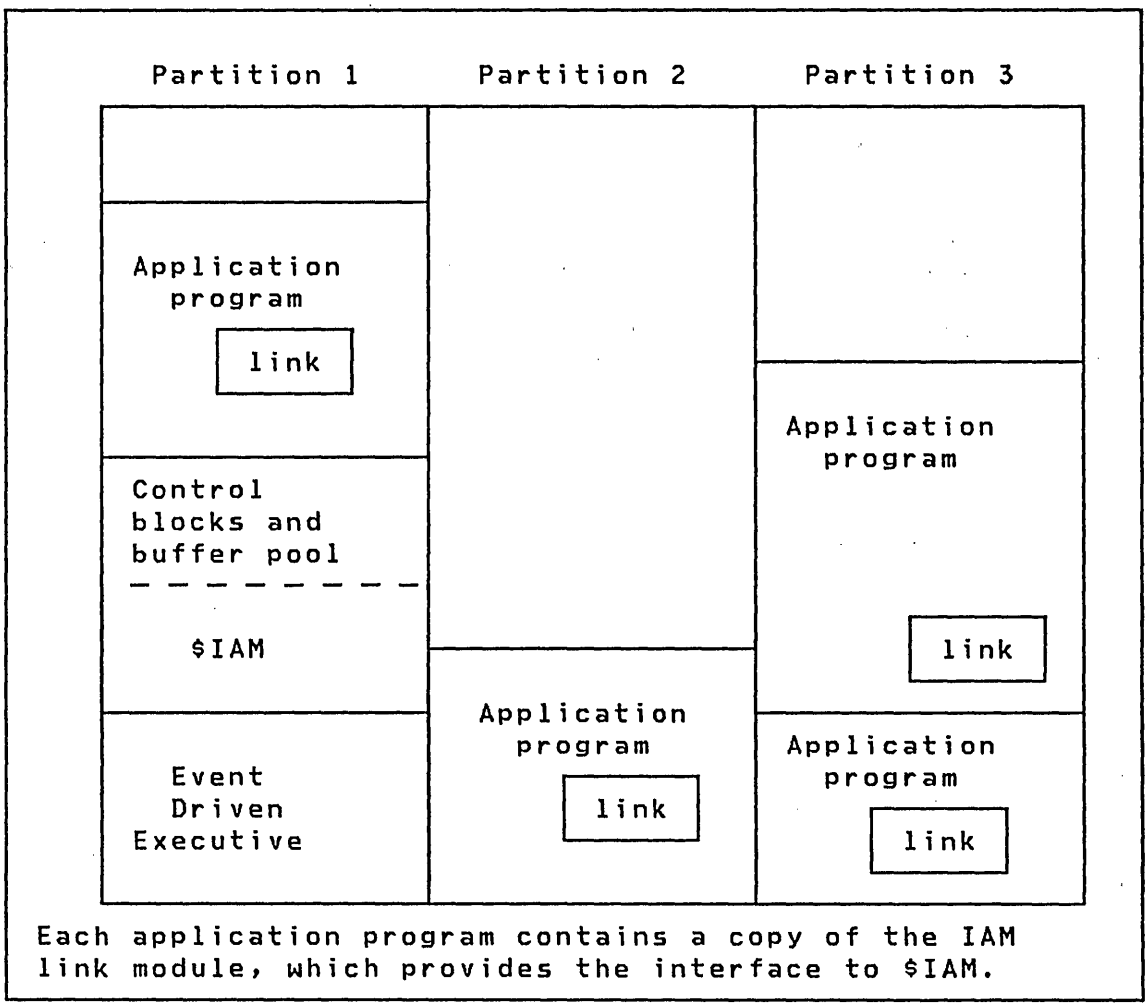


Figure 20. Example System Environment

An indexed data set contains base records, inserted records, a multilevel index, and the control information required to use the index and free space.

Indexed access applications are generally of two types: those which both read and modify files and those which only read files. The former are called **update** applications while the latter are called **inquiry** applications.

The Indexed Access Method uses two modes to place records into an indexed data set:

1. LOAD mode: records are loaded sequentially in ascending order by key, skipping any free space. The records loaded are called **base records**. Each record loaded must have a key higher than any key already in the data set.

2. PROCESS mode: records are inserted in their proper key position relative to records already in the data set. Records are inserted using the free space that was skipped during loading or, if a record has a new high key, in the unused space after the last loaded record.

The total number of base records that can be loaded is established when the indexed data set is built by the $IAMUT1 utility. It is not necessary, however, to load all the base records before processing can begin. The data set can be opened for loading some of the base records, closed and then reopened for processing (including inserts), and later opened for loading more base records. Figure 21 illustrates this sequence.

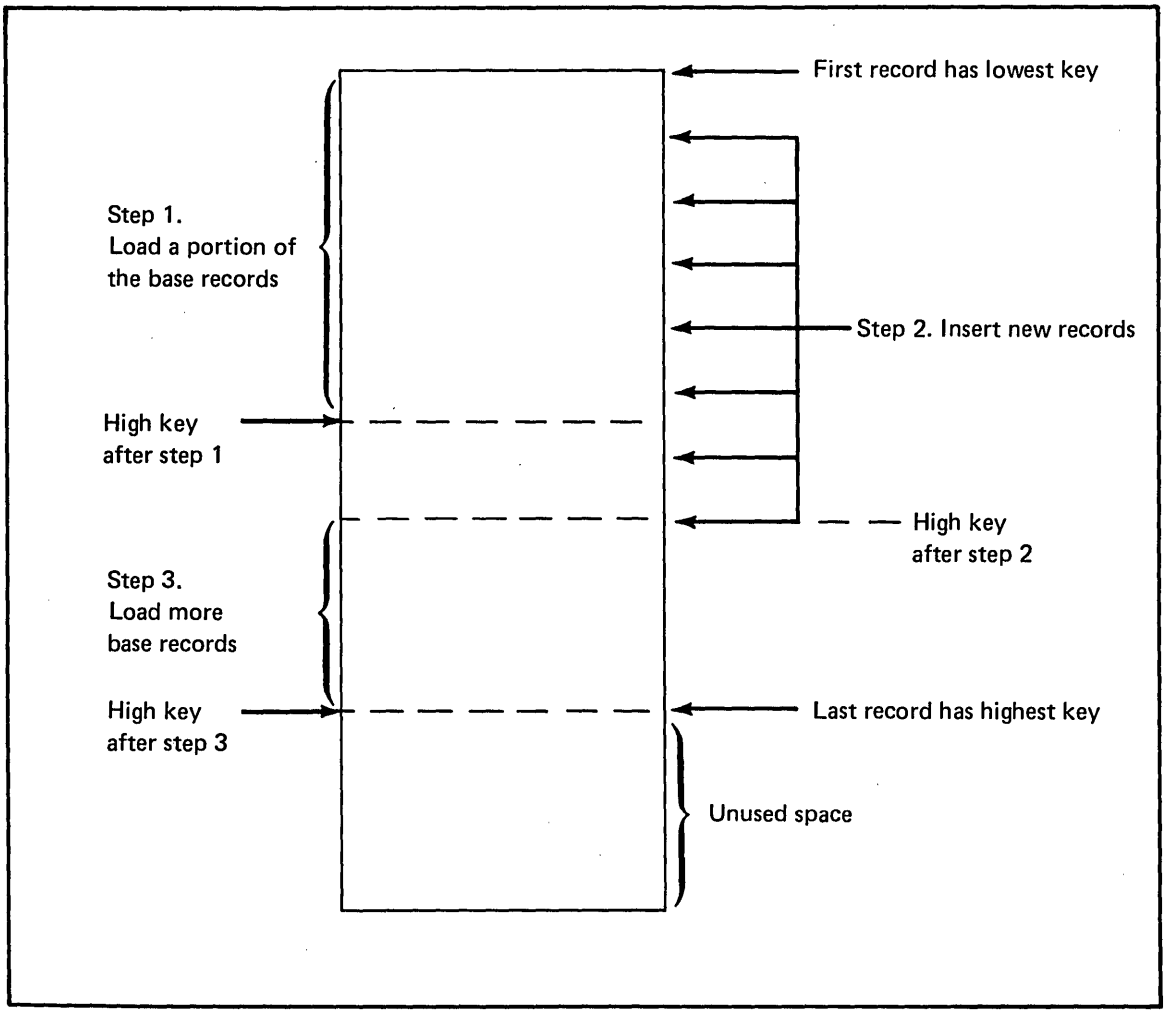


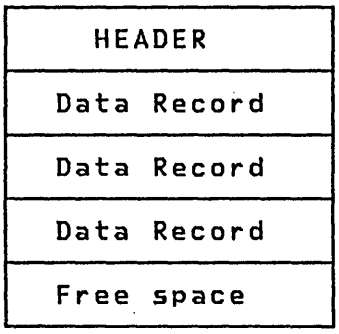
Figure 21. Loading and Inserting Records in an Indexed Data Set

The total amount of free space for inserts is specified to the $IAMUT1 utility when the indexed data set is built. This free space is distributed throughout the data set in the form of free records within each data block, free blocks within each block grouping, and/or in a free pool at the end of the data set.

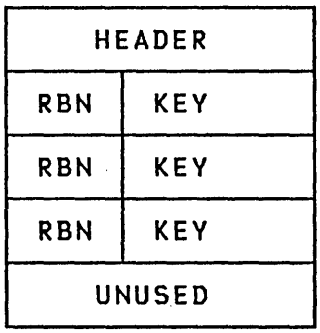
Data Set Format

Indexed data sets consist of data blocks which contain records, indexes (pointers) to the data blocks, and indexes to the index blocks. This technique is called a cascading index structure. The first block in the indexed data set, the file control block (FCB), describes the attributes of the data set.

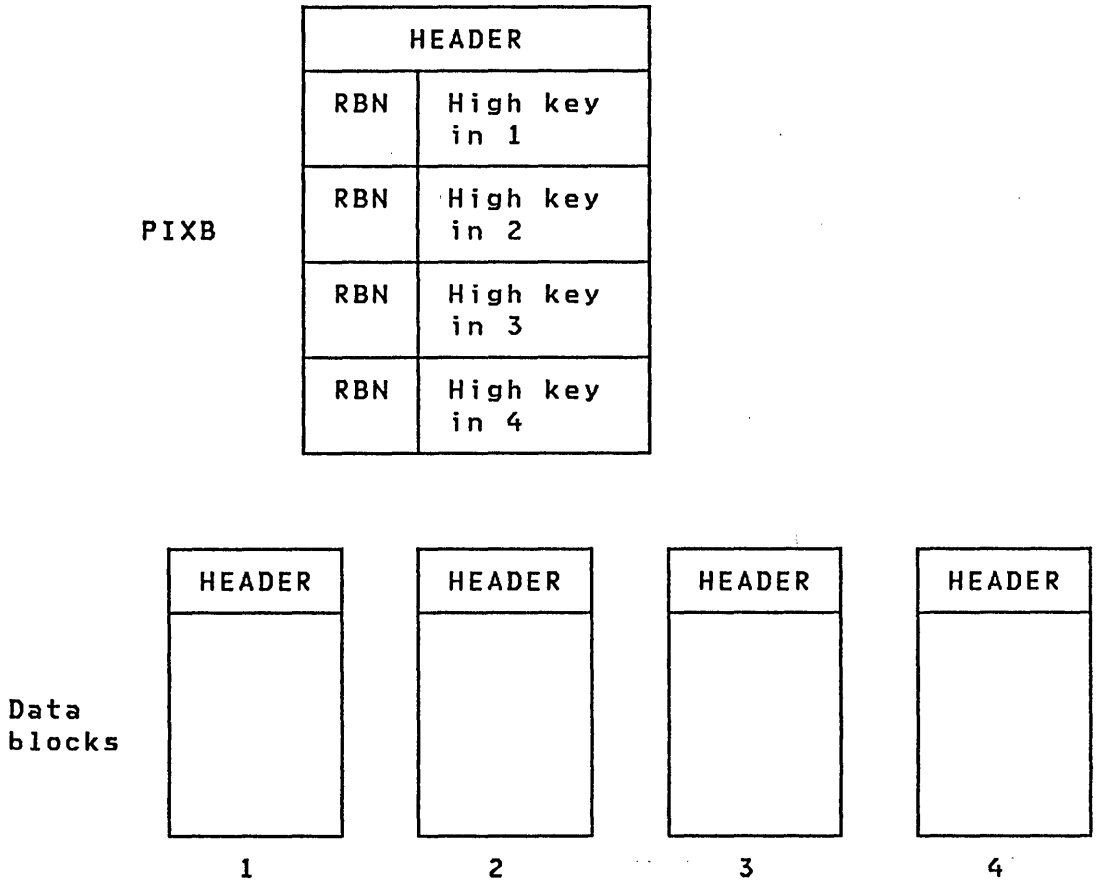
Each data block has the following format:



Each index block has the following format:



A set of data blocks is addressed (described) by a single index block. Each key in the index block is the highest key in the data block that its accompanying relative block number (RBN) addresses. A block is addressed by its RBN. The primary-level index block (PIXB) and the data blocks it describes are called a **cluster**.



A Sample Cluster

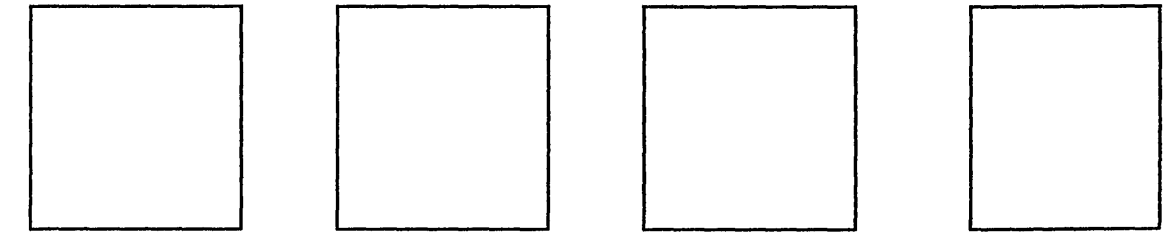
The records in each data block are in ascending order, according to the key field in each record.

Each data block header contains the address of the next sequential data block, allowing sequential processing.

Each PIXB (or cluster) has an entry in a second-level index block (SIXB) that contains the address of the PIXB and the highest key in the cluster. The SIXB has the following structure:

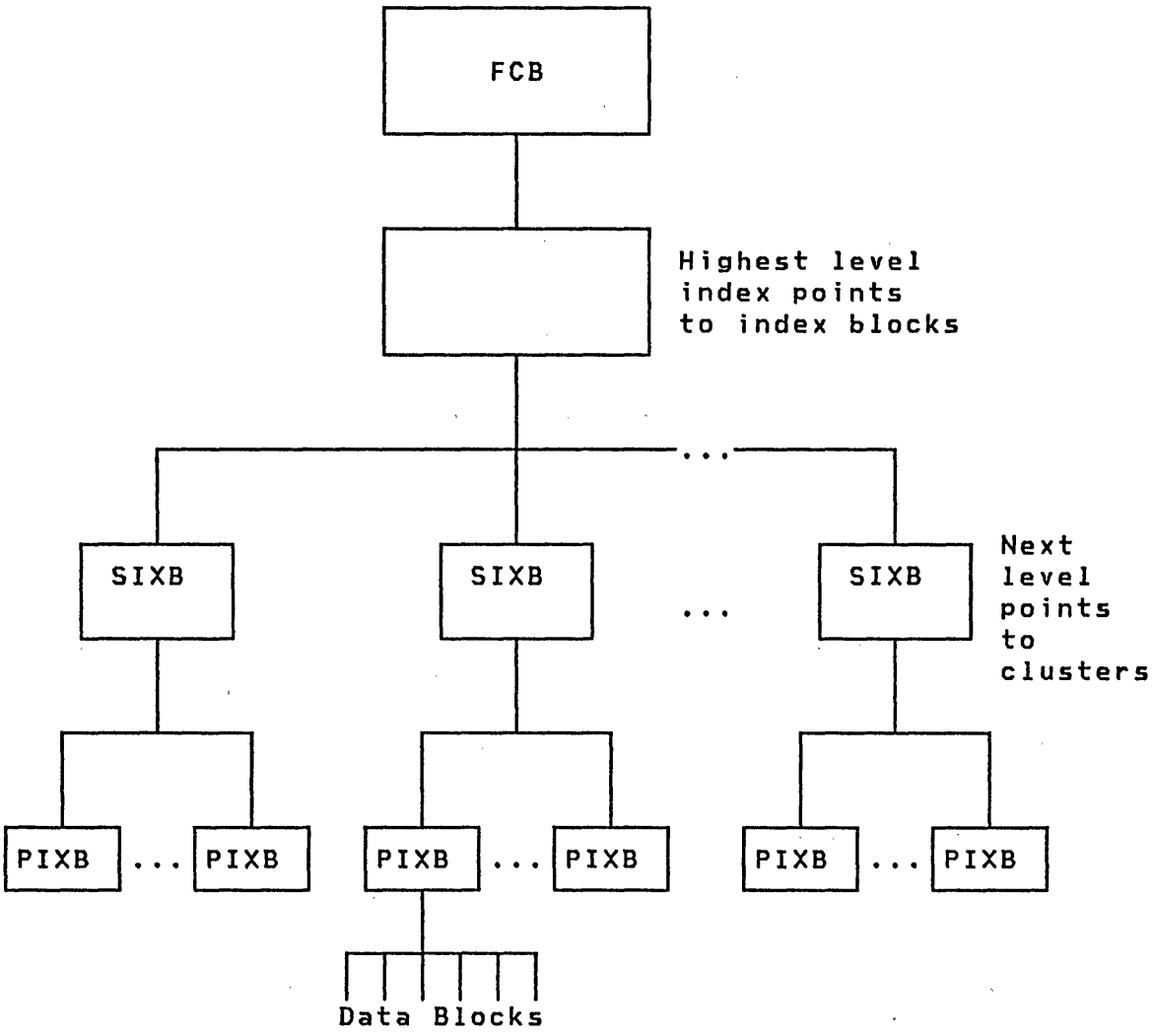
SIXB

HEADER	
RBN	High key in PIXB1
RBN	High key in PIXB2
RBN	High key in PIXB3
RBN	High key in PIXB4



PIXB1 PIXB2 PIXB3 PIXB4

The SIXBs in the data set are described by an index block in the same manner as the PIXB describes each cluster. There is, of course, an index block that describes the entire data set. The structure of the file is as follows:



Note that only the highest key in any data block is found in a PIXB entry, a SIXB entry contains only the highest key found in a PIXB, and so on, to the highest index block. This index technique is called **sparse indexing**.

REQUESTING RECORDS

When you request a record from your data set, the access method uses the index to retrieve the data block that contains the record. The index blocks and data blocks are read, using EDL READ instructions, into the central buffer. When the requested record is found, it is moved to the address you specified and control is returned to your program.

To minimize accesses to the disk, the buffer management algorithm tends to keep in the buffer the most frequently referenced blocks (index or data).

PREPARING TO EXECUTE INDEXED APPLICATIONS

The Indexed Access Method consists of the following components:

- A load module, $IAM, that supports the execution of the programs that contain your Indexed Access Method requests.
- A set of object modules that you may use to generate a customized load module. If you use the supplied load module, $IAM, you do not need the object modules.

  The object module, IAM, is called a **link module**. You include IAM with your program to provide the interface to the Indexed Access Method. This link module is sometimes called a **stub**.
- Two copy code modules, IAMEQU and FCBEQU. IAMEQU provides symbolic parameter values for constructing CALL parameter lists. FCBEQU provides a map of the file control block (FCB).
- A load module for the Indexed Access Method utility $IAMUT1.

Preparing Programs

To prepare an application programs that issues Indexed Access Method requests, perform the following steps:

1. Enter the source program, using one of the text editors ($FSEDIT, $EDIT1, or $EDIT1N).
2. Create the $LINK control statements required to combine your program with IAM (the link module) and any other object modules you may need in your application. These statements consist of a single OUTPUT statement, at least two INCLUDE statements - one for your program and one for IAM (the link module), and a single END statement. Use one of the text editors to perform this operation.
3. Assemble the source program using:

   The EDL compiler, $EDXASM, of the Program Preparation

Facility

or

The Series/1 macro assembler, $S1ASM, in conjunction with the Macro Library

or

The Series/1 macro assembler supplied by the System/370 Program Preparation Facility in conjunction with the Macro Library/Host

4. Use the linkage editor, $LINK, to combine the object modules into a single module, using the control statements prepared in Step 2.

5. Use the object program converter, $UPDATE or $UPDATEH, to convert your module to a loadable program.

When the preceding steps are completed, the program is ready to be executed.

Establishing the Data Set

Use the following steps to prepare the input for an indexed data set:

1. If your data records are 72 bytes or less use one of the text editors to enter your data or one of the communications utilities to get the data to your system. In either case, you must know the record format used by the utility. The utilities put two 80-byte records in each 256-byte EDX record. The first record begins at location 1, and the second record begins at location 129. The $IAMUT1 utility assumes unblocked input. $IAMUT1 takes only one logical record, the size of which was specified on the RECSIZE prompt, from each EDX record. Any record after the first logical record in each 256-byte EDX record is ignored. If you use the text editors, you must enter data on every other line starting with the first line.

2. If your records have more than 72 bytes of data, you must create a program that accepts the data records and writes them to a disk or diskette data set.

The data must be in ascending order, based upon the field you use as the key.

The process of creating an indexed data set from a sequential data set is:

1. Invoke $IAMUT1.

2. Enter an EC command. Respond to the prompt with a Y. This will put all further input and output of $IAMUT1 to the $SYSPRTR device and your terminal.

3. Enter an SE command. You will be prompted for the attributes of your data set. After the prompt/reply sequence ends, the utility will display your file attributes in numeric form. When you are satisfied with the file's structure (you can repeat SE commands, changing selected values), performing steps 4 through 8.

4. Enter a CR command to invoke $DISKUT1.

5. Enter a CV command to specify the volume. Then enter an AL

6. Enter an AL command followed by the data set name, specify the space in EDX records, and enter a Y in response to the data type prompt.

7. Enter an EN command to end $DISKUT1 and return to $IAMUT1.

8. Enter a DF command to map the file. The DF command also prompts for the immediate write back option and the data set and volume names.

9. Enter an LO command. Respond to the prompt for input by specifying your input data set name and volume. Respond to the output prompt by entering the data set name and volume specified on the DF command. Your data is then loaded to the indexed file.

10. Enter an EN command to end $IAMUT1. Your program can then be loaded and may begin to process the data.

A SAMPLE $JOBUTIL PROCEDURE AND LINK EDIT CONTROL

$JOBUTIL Procedure

```
*************************************************************
*
* THESE STATEMENTS WILL ASSEMBLE, LINK, AND UPDATE THE
* APPLICATION.
*
*************************************************************
JOB       ASSEMBLE
***   ASSEMBLE USERPROG SOURCE  ***
LOG       $SYSPRTR
PROGRAM   $EDXASM,ASMLIB
DS        USERPROG,EDX002          SOURCE MODULE
DS        ASMWORK,EDX002           ASSEMBLER WORK DATA SET
DS        USEROBJ,EDX002           ASSEMBLER OUTPUT
PARM      LIST     $SYSPRTR
EXEC
JOB       LINKAIAM
LOG       $SYSPRTR
PROGRAM   $LINK,ASMLIB
DS        LINKCTL,EDX002           LINKCTL IS NAME OF
*                                  LINK-CONTROL DATA SET
DS        LEWORK1,EDX002           LINK WORK DATA SET
DS        LEWORK2,EDX002           LINK WORK DATA SET
PARM      $SYSPRTR
EXEC
PROGRAM   $UPDATE
*      PUT EXECUTABLE LOAD MODULE INTO DATA SET 'ANYNAME'
PARM      $SYSPRTR  LINKOUT,EDX002   ANYNAME   YES
EXEC
LABEL     END
EOJ
```

Link Edit Control

```
*************************************************************
*
* LINK EDIT CONTROL DATA SET (LINKCTL)
*
*************************************************************
  OUTPUT    LINKOUT,EDX002  PUT LINK OUTPUT INTO LINKOUT
  INCLUDE   USEROBJ,EDX002  INCLUDE APPLICATION PGM OBJECT
  INCLUDE   IAM,ASMLIB      INCLUDE INDEXED ACCESS METHOD
  END
```

CHAPTER 9. PLANNING AND DESIGNING INDEXED APPLICATIONS

This chapter provides information for designing applications that use the Indexed Access Method. It contains information about:

- Defining programs
  - Interfacing to $IAM
  - Maintaining indexed data sets
  - Recovery, backup, and reorganization techniques
  - Concatenating indexed data sets
- Error handling
  - How to handle errors
  - Error exit facilities
  - Resource contention
- The indexed data set
  - How to define records
  - How to define the key
  - How the data set is structured
  - How the data set is formatted

Note: The Language Reference contains a detailed description of the coding syntax of each Indexed Access Method request. You may wish to refer to it while reading the next several pages.

CONNECTING AND DISCONNECTING THE INDEXED DATA SET

Prior to using an indexed file, you must issue either a LOAD or PROCESS request to connect it to your program. The file must be defined in your PROGRAM statement or by a DSCB statement. In the latter case use $DISKUT3 or DSOPEN to open the data set prior to issuing the LOAD or PROCESS.

A LOAD or PROCESS request builds an indexed access control block (IACB) that is associated with an indexed data set. The IACB connects a request to the data set.

When in load mode, records are placed in the file sequentially. Free space is skipped. When in process mode, records are placed in the first available slot in the file and free space is used.

Only one LOAD request can be active for a given data set. However, processing can take place concurrently with loading. No LOAD or PROCESS can be successful until the file has been formatted by the $IAMUT1 utility.

Multiple IACBs can be associated with the same data set. Data integrity is maintained by a locking system that allocates file, record, or block locks to the requesting IACB. This prevents concurrent modification of index or data records by other requests.

An IACB can hold only one lock at a time; if your application requires concurrent execution of functions that obtain locks (direct update or sequential update - see "Processing" on page 161 for a description of these functions), you must issue multiple PROCESSes to build multiple IACBs.

A DISCONN disconnects an IACB from the data set, releases the storage for that IACB, releases locked blocks or records being held by that IACB, and writes any blocks that are being held in the buffer. The DISCONN request can be issued at any time during loading or processing.

There is no automatic DISCONN on task termination. Failure to disconnect your indexed data sets prior to task termination may prevent resources that were allocated to your task from being allocated to other tasks and updated records from being written to your data set.

LOADING BASE RECORDS

Base records must be loaded in ascending order by key. If you are writing your own program to load the file, use a LOAD request to load base records. Then issue a PUT for each record. When the desired records have been loaded, issue a DISCONN request to terminate the load procedure. The only requests that can follow a LOAD request are: PUT, EXTRACT, and DISCONN.

You need not load all base records at one time. A data set that already contains records can be reconnected to load more records, but the key of each new record must be higher than any key already in the data set.

Also, the limit on base records as specified on the DEFINE command of the Indexed Access Method utility program ($IAMUT1) cannot be exceeded. If you attempt to load a record after the last allocated record area has been filled, an end-of-file condition occurs.

Only one LOAD request can be issued to a data set at any time. Other processing requests can be made to a data set that is being loaded, but an attempt to retrieve a record from the data block being loaded can result in a no-record-found condition.

PROCESSING

Initiate general purpose access to an indexed data set with a PROCESS request. After the PROCESS request has been issued, any of the following functions can be requested:

- Direct reading - Retrieving a single record independently of any previous request.
- Sequential reading - Retrieving the next logical record relative to the previous request.
- Direct updating - Retrieving a single record for update; complete the update by either replacing or deleting the record.
- Sequential updating - Retrieving the next logical record for update; complete the update by either replacing or deleting the record.
- Inserting - Placing a single record, in its logical key sequence, into the indexed data set.
- Deleting - Removing a single record from the indexed data set.
- Extracting - Extracting data that describes the data set.

Note that the update functions require more than one request.

When a function is complete, another function may be requested, except that a sequential function may be followed only by another sequential function. You may terminate processing at any time by issuing a DISCONN or ENDSEQ request. An end-of-data condition also terminates sequential processing.

Direct Reading

Use the GET request to read a record using direct access. The key parameter is required and must be the address of a field of full key length regardless of the key length specification.

The record retrieved is the first record in the data set that satisfies the search argument defined by the key and key relation (krel) parameters. The key field is updated to reflect the key contained in the record that satisfied the search.

If the key length is specified as less than the full key length, only part of the key field is used for comparison when searching the data set. For example, the keys in a data set are AAA, AAB, ABA, and ABB, the key field contains AB0, and key relation is EQ. If key length is zero, the search argument is the full key AB0 (the default) and a record-not-found code is returned. If the key length specification is 2 and the search argument is AB, the third record is read. If the key length specification is 1 and the search argument is A, the first record is read.

Direct Updating

To update a record using direct access:

1. Retrieve the record with a GET request, specifying the key and key relation (krel) parameters.

2. Modify the record in your buffer. Do not change the key field in the record. Return the updated record to the data set with a PUTUP request.

You can delete the record with a PUTDE request or leave it unchanged by issuing a RELEASE request.

The key parameter must be specified as the address of a field of full key length. The key cannot be modified during the update.

The only valid requests, other than DISCONN and EXTRACT, that can follow GET for direct update are PUTUP, PUTDE, and RELEASE.

During the update, the subject record is locked (made unavailable) to any other request until the update is complete. Even if no action is taken after the GET request is issued, the RELEASE request is required to release the lock on the record.

Sequential Reading

Use the GETSEQ request to read a record sequentially. After a sequential processing request has been initiated, only sequential functions can be requested until an end-of-data condition occurs or an ENDSEQ request is issued. Processing is terminated when a DISCONN request is issued or an error or warning is returned.

To begin sequential access with the first record in a data set, set the key address to zero. To start with any other record, specify a search argument by specifying the key and key relation (krel) parameters.

If you specify a search argument, the key field is modified to reflect the key of the first record found.

After the first retrieval, a GETSEQ retrieves the next sequential record regardless of any key or key relation specification. Therefore, you can use the same GETSEQ statement to read all records. A search argument on intermediate retrievals is ignored and the key field is not modified.

Specify ENDSEQ to stop reading before the end of data is reached. Reading ends automatically at the end of data. The end-of-data condition occurs when an attempt is made to retrieve a record after the last record in the data set.

If you specify the EODEXIT parameter on the PROCESS request, control is transferred to the address specified by the EODEXIT parameter when the End-of-Data condition occurs.

Sequential Updating

To update a record using sequential access, retrieve the record with a GETSEQ request, specifying the key and one of the update key relation parameters. The key is used only on the first retrieval and is not specified if processing is to begin with the first record in the data set. Processing is terminated with an ENDSEQ or an end-of-data condition.

The key in the record cannot be modified. The record can be returned to the data set with a PUTUP, deleted with a PUTDE, or left unchanged by specifying RELEASE. When the update is complete, the next record can be requested.

During sequential updating, the block that contains the record is locked, making all records in the block unavailable to other requesters until the last record of the block is processed or an ENDSEQ request is issued.

Terminate processing with an ENDSEQ request or a DISCONN request either before or after completing the update. Figure 22 on page 164 summarizes the protocol for sequential processing.

REQUEST/CONDITION	CAN BE FOLLOWED BY:
GET	DISCONN END-OF-DATA CONDITION ENDSEQ PUTUP PUTDE RELEASE
END-OF-DATA CONDITION	DISCONN GET PUT DELETE
PUTUP	DISCONN ENDSEQ GETSEQ
PUTDE	DISCONN ENDSEQ GETSEQ
RELEASE	DISCONN ENDSEQ GETSEQ

Figure 22. Protocol for Sequential Updating

Inserting

To insert a new record in a data set, issue a PUT request. The Indexed Access Method uses the key of the record to insert the record into the data set.

The key of the inserted record must be different from any key in the data set; otherwise, a duplicate key error occurs. The key can be higher than any key in the data set.

If no free space exists in the area associated with the insert or no blocks exist in the free pool, a no-more-space condition occurs. The no-more-space condition does not necessarily mean the data set is full but it does indicate the need for data set reorganization (refer to "Reorganization" on page 166).

Deleting

Use DELETE to delete a record from the data set. The full key of the record must be specified. If no record exists with the specified key, an error is indicated.

Deletion can also be performed as part of updating by following a GET for update with a PUTDE request.

Extracting

The EXTRACT request provides information about a data set from the file control block (FCB). This includes information such as key length, key displacement, block size, record size, and other data regarding the data set structure.

Execution of the EXTRACT request causes the file control block to be copied to an area that you provide. The data set must have been connected by a LOAD or PROCESS request.

The contents of the FCB are described by FCBEQU, a unit of copy code that is supplied by the access method. Use COPY FCBEQU to include these equates in your program.

MAINTAINING THE INDEXED DATA SET

The Indexed Access Method does not provide specific programs to perform indexed data set backup and recovery, nor does it include services to delete the data set or dump it to the printer. These procedures are provided by a combination of Event Driven Executive and Indexed Access Method services as suggested below. The Indexed Access Method utility $IAMUT1 does provide services to help you reorganize your data set as described below.

Backup and Recovery

To protect against the destruction of data, at regular intervals you should make a copy of the indexed data set (or the logical volume in which the data set exists) using the system $COPY utility. During the interval between making copies, you should keep a journal file of all transactions made against the indexed data set.

The journal file can be a consecutive data set containing records that describe the type of transaction and the pertinent data. A damaged indexed data set can be recovered by updating the backup copy from the journal file.

For example, suppose an indexed data set named REPORT is lost because of physical damage to the disk. The condition that caused the error has been repaired and the data set must be recovered. Delete REPORT, copy the backup version of REPORT to the desired volume, and process the journal file to recreate the data set.

If a data-set-shut-down condition exists, IPL again. Then issue a PROCESS to the REPORT data set and, using the journal file, reprocess the transactions that occurred after the back-up copy was made.

Recovery Without Backup

If you do not use the backup procedures outlined above and you encounter a problem with your data set, you still may be able to recreate your file. However, the status of requests that were in process at the time of the problem is uncertain.

To recreate your data set, follow the steps in "Reorganization" to reorganize your data set. After recreating the data set, verify the status of the requests that were in process at the time the problem occurred.

Reorganization

An indexed data set must be reorganized when a record cannot be inserted because of lack of space. The lack-of-space condition does not necessarily mean that there is no more space in the data set; it means that there is no space in the area where the record would have been placed. Therefore, you may be able to reorganize without increasing the size of the data set. Perform the following steps to reorganize a data set:

1. Ensure that all outstanding requests against the data set have been completed; issue a DISCONN for every current IACB.

2. Use the define command (DF) of the $IAMUT1 utility to define a new indexed data set. Estimate the number of base records and the amount and mix of free space in order to minimize the need for future reorganizations. Refer to "The Indexed Data Set" on page 182 for guidelines for making these estimates.

3. Use the reorganize command (RO) of the $IAMUT1 utility to load the new indexed data set from the indexed data set to be reorganized.

   Alternatively, you can use the unload command (UN) of the $IAMUT1 utility to transfer the data from an indexed data set to a sequential data set, then use the load command (LO) to load it back into the indexed data set.

4. Use system utilities to delete the old data set and rename the new data set.

Dumping

To print records, use the DP command of the $DISKUT2 utility. $DISKUT2 produces a hexadecimal dump of the entire data set including control information, index blocks, and data blocks. Information on the $DISKUT2 utility can be found in the Utilities, Operator Commands, Program Preparation, Messages and Codes.

Deleting

Delete an indexed data set the same way you delete any other data set. From a terminal, use the DE command of the $DISKUT1 utility (refer to Utilities, Operator Commands, Program Preparation, Messages and Codes), or from a program use the $DISKUT3 data management utility (refer to "Chapter 16. Advanced Topics" on page 309).

CONCATENATING DATA SETS

The ALTIAM subroutine allows you to concatenate multiple IAM data sets and to issue normal IAM commands to the concatenated file. This allows you to have more than 32,767 sectors in an IAM file or to put parts of a file on different devices to improve performance. The data sets may reside on the same or different volumes or devices. The keys of all data sets must have the same location and length. Each file must be loaded individually and have a unique range of keys, with no overlap of key ranges between the data sets.

To incorporate this function in your application, transcribe the ALTIAM subroutine using one of the text editors and modify it to meet your requirements. Compile it with $EDXASM or the Series/1 Macro Assembler and add the object program to your

object library. Include the object program when you link edit your application programs with the IAM link module.

Note: The ALTIAM subroutine is not compatible with the Multiple Terminal Manager.

The ALTIAM subroutine accepts all Indexed Access Method requests for single files. A special request, CONCAT, is issued to concatenate files. Only one set of files may be concatenated per copy of ALTIAM; when the file is disconnected, another set may be concatenated. The parameters to CONCAT are as follows:

```
CALL  ALTIAM,(CONCAT),IACB,(DSCBTAB),(OPENTAB),(MODE)
```

- Equate CONCAT to 14.
- IACB, OPENTAB, and MODE are the same as in the PROCESS request.
- DSCBTAB is the address of a list of opened data set control blocks (DSCBs) with the following format:

```
DSCBTAB     DATA  A(DS1)
            DATA  A(DS2)
            DATA  A(DS3)
            DATA  A(BUFFER)
```

The DSCBs must be in order of increasing key ranges of of the corresponding files. Three DSCBs is the default but you may increase or decrease the number. If only two data sets are needed, word three must be zero. The buffer must be large enough to hold the largest record in the concatenated file.

The CONCAT function issues PROCESS requests and reads the low key of each file. The default maximum key size (50 bytes) may be changed. The address of the IACB that is returned is used by ALTIAM to issue processing requests against the concatenated file.

The following requests may be made to a concatenated file:

```
GET
GETSEQ
PUT
PUTUP
PUTDE
DELETE
EXTRACT
ENDSEQ
RELEASE
DISCONN
```

The parameters for each function are identical to the parameters for requests to non-concatenated files.

You may want to modify the following items when using the ALTIAM subroutine:

- maximum number of concatenated data sets
- maximum key length
- error checking

To change the maximum number of data sets, change line 2740 so that DSCB# is equated to the number of files to be concatenated (N). Lines 2630 and 2640 allocate space for IACBs and key save areas. Line 2630 allocates N-1 words for IACBs. Line 2640 allocates KS*(N-1) bytes for key save areas, where KS is the maximum key size.

To change the maximum key size, change line 2600 to allocate the desired number of bytes (KS) for a key save area.

The ALTIAM subroutine does not perform the same error checking that occurs for non-concatenated data sets. You may want to check for the following errors:

- GETSEQ requests in one file that are followed by non-sequential requests to another file in the concatenated data set.
- PROCESS or LOAD requests being issued against concatenated datasets (unpredictable results may occur).
- GET or GETSEQ requests for update in one file followed by non-update requests (e.g.PUT) to another file in the concatenated data set.

The first error may be checked at line 300. If the sequential flag (ASEQ) is set and the request is a GET, DELETE, PUT, or EXTRACT, set an error code (10).

The second error may also be checked at line 300. If the request is PROCESS or LOAD, set an error code (10).

The third error may be checked by adding an update flag. The flag should be set at lines 1470 and 1530 if an update request is made. The flag should be reset at line 1180 for ENDSEQ, RELEASE, PUTUP, and PUTDE requests. The flag should also be reset at line 1000 in the DISC routine and at line 2440 in the ALTERR routine. At line 300 the flag should be checked. If the flag indicates a GET, GETSEQ, PUT, DELETE, or EXTRACT request, set error (10).

```
00010 ***********************************************************************
00020 ***
00030 *  ALTIAM IS A SUBROUTINE WHICH ALLOWS THE USER TO CONCATENATE
00040 *  IAM DATA SETS.  ALL PARMS AND CALLS ARE THE SAME AS IN IAM.
00050 *  REQUESTS FOR NON-CONCATENATED IAM FILES ARE SIMPLY PASSED THRU
00060 *  TO IAM.  TO OPEN A SET OF DATA SETS ISSUE THE CONCAT REQUEST AND
00070 *  PASS A TABLE OF DSCB'S.  THE KEYS IN ALL FILES MUST BE IN THE
00080 *  SAME POSITION AND BE THE SAME LENGTH.  KEY RANGES CAN NOT OVERLAP.
00090 *  VALID COMMANDS ARE:  GET, GETSEQ, DELETE, PUT, PUTUP, PUTDEL,
00100 *  RELEASE, ENDSEQ, EXTRACT, DISCONN, AND CONCAT FOR
00110 *  CONCATENATED FILES.
00120 ***
00130 ***********************************************************************
00140          SPACE 2
00150          SUBROUT ALTIAM,FUNCTION,IACB,PARM3,PARM4,PARM5
00160          ENTRY  ALTIAM
00170          MOVE   SAVEREGS,#1,2          SAVE USERS REGS
00180          MOVE   IFUNC,FUNCTION,5       COPY USERS PARMS TO CALL TO IAM
00190          IF     (FUNCTION,NE,+CONCAT),AND,    NOT THE SPECIAL FUNCTION X
00200               (IACB,NE,+ALTIACB)       OR A USE OF CONCAT. IAM DS
00210            MOVE   REGA,IACB            SAVE CURRENT IACB VALUE
00220            MOVEA  IIACB,REGB           POINT TO SAVE AREA
00230            CALL   CALLIAM              JUST PASS THE REQUEST THRU
00240            MOVE   IACB,REGB            COPY THE IACB BACK TO THE USER
00250            GOTO   EXIT                 RETURN TO USER
00260          ENDIF
00270          SPACE 5
00280 ***********************************************************************
00290 *  PROCESS THE SPECIAL CONCATENATED IAM FILE REQUESTS.
00300 ***********************************************************************
00310           GOTO (LAST,LAST,LAST,LAST,INS,LAST,DIR,SEQ,DEL,DISC, X
00320                LAST,LAST,LAST,LAST,CON),FUNCTION
00330           EJECT
00340 CON       EQU   *                     PROCESS THE CON. OPEN REQUEST
```

```
00350 ************************************************************************
00360 *  LOOP THRU USERS TABLE ISSUING IAM PROCESS REQUEST, EXTRACT
00370 *  FCB INFO, SAVE USERS EXIT INFO, AND FIND LOW KEY IN EACH DATA SET.
00380 ************************************************************************
00390          MOVE   ALTIACB,0,(+ALTTSIZE,BYTES) ZERO OUT THE ALT IACB @
00400          MOVE   #1,PARM3                GET USERS DSCB TABLE POINTER
00410          MOVE   BUFF,(BUFFADR,#1)       GET POINTER TO USERS BUFFER
00420          MOVEA  #2,ALTIACB              POINT AT ALTERNATE IACB
00430          MOVEA  IKEY,OPENTAB            POINT AT OUR OPEN TABLE
00440          MOVE   IFUNC,+PROCESS          SET UP TO DO IAM PROCESS
00450          SPACE 2
00460          DO     +DSCB#,TIMES            LOOP THRU THE USERS DSCB TABLE
00470            IF     ((0,#1),EQ,0),GOTO,EXIT1
00480            MOVE   IBUFF,(0,#1)           COPY A DSCB ADDR TO IAM CALL
00490            MOVE   IIACB,#2               POINT AT IACB SAVE ADDRESS
00500            CALL   CALLIAM                ISSUE IAM CALL
00510            ADD    #1,2                   POINT AT NEXT DSCB
00520            ADD    #2,+AENTSIZE           ADD ALT IACB ENTRY SIZE
00530          ENDDO
00540 *
00550 EXIT1    EQU   *                        EXIT FROM DO LOOP
00560 **  EXTRACT THE FCB INFORMATION
00570          MOVE    IFUNC,+EXTRACT         SET UP TO DO EXTRACT
00580          MOVE    IBUFF,BUFF             POINT TO OUT BUFFER
00590          MOVE    IKEY,+10               TRANSFER 10 BYTES OF FCB
00600          CALL    CALLIAM                ISSUE IAM CALL
00610          MOVE    #1,BUFF                SET UP FCB DSECT
00620          MOVE    AKPOS,(FCBKEYDP,#1)    SAVE KEY POSITION
00630          MOVE    AKSIZE,(FCBKEYLN,#1),BYTE   GET THE KEY LENGTH
00640          SHIFTR AKSIZE,8                SHIFT IT INTO POSITION
00650 *
00660          MOVE    #1,PARM4               PICK UP USERS OPEN TABLE
00670          MOVE    ASYSRC,#1              SAVE SYSRC CELL
00680          MOVE    AERR,(2,#1)            SAVE USERS ERROR EXIT ADDR
00690          MOVE    AEOD,(4,#1)            SAVE USERS END OF DATA EXIT
00700 *
00710 ** GET THE LOW KEY IN EACH DATA SET
00720          MOVE    IKEY,0                 SET UP DEFAULT 1ST KEY
00730          MOVE    #2,AKPOS               POINT AT KEY POSITION
00740          MOVE    MOVEKEY,AKSIZE         SET UP LENGTH OF MOVE
00750          MOVEA   #1,ALTIACB+AENTSIZE    POINT AT SECOND DATA SET
00760          DO      +DSCB#M1,TIMES         LOOP THRU DATA SETS
00770            IF      ((0,#1),EQ,+0),GOTO,EXIT2
00780            MOVE    IFUNC,+GETSEQ        SET UP FUNCTION
00790            MOVE    IIACB,#1             POINT AT IACB
00800            MOVE    IOPT,+GE             SET UP RELATION
00810            CALL    CALLIAM              ISSUE IAM GET
00820            MOVE    IFUNC,+ENDSEQ        SET UP END SEQ REQUEST
00830            CALL    CALLIAM              ISSUE IAM RELEASE
00840            MOVE    (-AMAXKEY,#1),(0,#2),(1,BYTE),P2=BUFF,P3=MOVEKEY
00850            ADD     #1,+AENTSIZE         POINT AT NEXT SLOT
00860          ENDDO
```

```
00870 *
00880 EXIT2    EQU   *                          DO LOOP EXIT
00890          MOVE   (-AMAXKEY,#1),X'FFFF',(+AMAXKEY,BYTES)  HIGH FILL
00900          MOVEA  IACB,ALTIACB             RETURN ALT IACB POINTER TO USER
00910 *
00920 EXIT     EQU   *                          RETURN TO USER
00930          MOVE  PARM3,0,3                 ZERO OUT LAST THREE PARMS
00940          MOVE  #1,SAVEREGS,2             RESTORE USERS REGISTERS
00950          RETURN
00960          EJECT
00970 DISC     EQU   *                          PROCESS ALTERNATE DISCONNECT
00980 ************************************************************************
00990 ** DISCONNECT ALL IAM FILES
01000 ************************************************************************
01010          MOVE   ASEQ,0                   RESET SEQENTIAL SWITCH
01020          MOVE   IACB,0                   ZERO OUT USERS IACB POINTER
01030          MOVEA  #1,ALTIACB               POINT AT IAM IACB TABLE
01040          DO     +DSCB#,TIMES             DO WHILE THERE ARE IACBS
01050            IF     ((0,#1),EQ,0),GOTO,EXIT3  IF EMPTY EXIT
01060            MOVE   IIACB,#1               POINT AT AN IACB
01070            CALL   CALLIAM                ISSUE IAM REQUEST
01080            ADD   #1,+AENTSIZE            POINT AT NEXT IACB
01090          ENDDO
01100 *
01110 EXIT3    EQU    *
01120          GOTO   EXIT                     RETURN TO USER
01130          SPACE 5
01140 LAST     EQU   *
01150 ************************************************************************
01160 ** THESE REQUESTS USE THE LAST IACB USED.  THEY ARE:  ENDSEQ, RELEASE,
01170 ** EXTRACT, PUTUP, AND PUTDEL.
01180 ************************************************************************
01190          MOVE  IIACB,ALSTIACB+2          USER THE LAST IACB
01200          IF    (FUNCTION,EQ,+ENDSEQ)     IF ENDING A SEQUENCE
01210            MOVE  ASEQ,0                  RESET THE SEQUENTIAL SWITCH
01220          ENDIF
01230          CALL  CALLIAM                   ISSUE IAM REQUEST
01240          GOTO  EXIT                      RETURN TO USER
01250          EJECT
01260 ************************************************************************
01270 ** THE NEXT SET OF FUNCTIONS USE THE CHECK ROUTINE TO DETERMINE
01280 ** WHICH IAM FILE TO ISSUE THE REQUEST TO.  THESE FUNCTIONS ARE:
01290 ** PUT, DELETE, GET, AND THE FIRST GETSEQ.  THE USER SUPPLIED KEY
01300 ** IS CHECKED AGAINST THE VALUE STORED DURING CONCAT.
01310 ************************************************************************
01320          SPACE 2
01330 INS      EQU   *
01340 **  PROCESS INSERT REQUESTS
01350          MOVE    #1,PARM3                POINT AT USERS KEY
01360          ADD     #1,AKPOS                ADD IN KEY OFFSET
01370          MOVE    COMPLEN,AKSIZE          FULL KEY SUPPLIED
01380          GOTO    CHECK
```

```
01390 *
01400 DEL       EQU   *
01410 **  PROCESS DELETE REQUESTS
01420           MOVE    #1,PARM3              POINT AT USERS KEY
01430           MOVE    COMPLEN,AKSIZE        FULL KEY SUPPLIED
01440           GOTO    CHECK
01450 *
01460 SEQ       EQU   *
01470 **  PROCESS GET SEQ REQUESTS
01480           IF      (ASEQ,EQ,1),GOTO,LAST IF NOT FIRST IN SEQUENCE
01490           MOVE    ASEQ,1                SIGNAL SEQUENTIAL MODE
01500 **  PROCESS FIRST SEQUENTIAL AS DIRECT
01510 *
01520 DIR       EQU   *
01530 **  PROCESS GET REQUESTS
01540           IF     (PARM4,EQ,0)           IF KEY IS NOT SET
01550             MOVEA  IIACB,ALTIACB        POINT AT FIRST FILE
01560             GOTO   INRANGE              SKIP CHECKING
01570           ENDIF
01580           MOVE   #1,PARM4               GET KEY POINTER
01590           MOVE   COMPLEN,(-1,#1),BYTE   GET KEY LENGTH
01600           SHIFTR COMPLEN,8              GET INTO POSITION
01610 *
01620 CHECK     EQU  *
01630 ************************************************************************
01640 ** LOOP THRU IACB TABLE COMPRING USERS KEY (#1) TO SAVED KEY IN
01650 ** THE TABLE.  THE SAVED KEY IS THE LOWEST KEY IN THE NEXT FILE.
01660 ************************************************************************
01670           MOVEA  #2,ALTIACB             POINT AT IACB TABLE
01680           MOVE   REGA,#1                SAVE USERS KEY ADDRESS
01690           DO     +DSCB#,TIMES           LOOP THRU IACBS
01700             IF     ((0,#2),EQ,0),GOTO,INRANGE      EXIT IF NO MORE
01710             MOVE   IIACB,#2             SAVE CURRENT IACB
01720             ADD    #2,2                 POINT AT SAVED KEY
01730             MOVE   COUNT,0              INITIALIZE STRING COUNTER
01740 *
01750             DO     WHILE,(COUNT,LE,COMPLEN)   LOOP THRU STRING
01760               IF    ((0,#1),LT,(0,#2),BYTE),GOTO,INRANGE CORRECT IACB
01770               IF    ((0,#1),GT,(0,#2),BYTE),GOTO,OUTRANGE WRONG IACB
01780               ADD    #1,1                 INCREMENT POINTERS
01790               ADD    #2,1                 * IF STRINGS ARE EQUAL
01800               ADD    COUNT,1
01810             ENDDO
01820 *
01830 **  IF STRINGS ARE EQUAL THEN THE KEY IS IN THE NEXT FILE.  UNLESS
01840 **  WE ARE USING THE LAST FILE ALREADY.
01850             ADD    IIACB,+AENTSIZE,RESULT=#2  POINT AT NEXT
01851             MOVE   DOUBLE1,0
01852             MOVE   DOUBLE2,#2
01860             IF     (DOUBLE1,LT,+ALSTIACB,DWORD) IF NOT THE LAST IACB
01870               MOVE   IIACB,#2           STORE NEW POINTER
01880             ENDIF
01890             GOTO   INRANGE              FOUND THE CORRECT IACB
```

```
01900 *
01910 OUTRANGE  EQU   *
01920 **  KEY IS NOT IN THIS RANGE. CHECK THE NEXT.
01930             ADD    IIACB,+AENTSIZE,RESULT=#2  BUMP THE IACB POINTER
01940             MOVE   #1,REGA             RESTORE THE USER KEY POINTER
01950           ENDDO
01960 *
01970 INRANGE   EQU   *
01980 **  KEY IS IN THIS RANGE.  ISSUE THE IAM CALL.
01990           CALL   CALLIAM
02000 *
02010           IF     (REGA,EQ,-58),AND,(PARM5,GT,+UPEQ)  NO RECORD FOUND
02020             ADD    IIACB,+AENTSIZE    POINT AT NEXT IACB
02021             MOVE   DOUBLE1,0
02030             MOVE   DOUBLE2,IIACB       IN A REGISTER
02031             MOVE   #1,DOUBLE2
02040             IF     (DOUBLE1,LT,+ALSTIACB,DWORD),AND, IN RANGE         X
02050                ((0,#1),NE,0),GOTO,INRANGE    * TRY NEXT FILE
02060           ENDIF
02070           GOTO   EXIT
02080           EJECT
02090 ************************************************************************
02100 **  INVOKE IAM AND SAVE RETURN CODE.
02110 ************************************************************************
02120           SUBROUT CALLIAM
02130           MOVE   ALSTIACB+2,IIACB      UPDATE LAST IACB CELL
02140           CALL   IAM,+PROCESS,IACB,(IACB),(IACB),+EQ,P2=IFUNC,        X
02150                P3=IIACB,P4=IBUFF,P5=IKEY,P6=IOPT
02160           MOVEA  TCW,$TCBCO-$TCB#1    OFFSET TO TASK CONTROL WORD
02170           MOVE   REGA,#1,P2=TCW       PICK UP TASK CONTROL WORD
02180           RETURN
02190           SPACE 5
02200 ALTEOD    EQU   *
02210 ************************************************************************
02220 **  END OF DATA EXIT.  IF NOT THE LAST FILE SWITCH TO THE NEXT ONE.
02230 **  IF THE LAST FILE PASS CONTROL TO USERS EOD EXIT.
02240 ************************************************************************
02250           ADD    IIACB,+AENTSIZE         POINT TO THE NEXT IACB
02251           MOVE   DOUBLE1,0
02252           MOVE   DOUBLE2,IIACB       IN A REGISTER
02260           MOVE   #1,IIACB
02270           IF     (DOUBLE1,LT,+ALSTIACB,DWORD),AND, IN RANGE           X
02280                ((0,#1),NE,0)
02290             MOVE   IKEY,0               GET FIRST KEY IN NEXT FILE
02300             GOTO   INRANGE              ISSUE IAM REQUEST
02310           ENDIF
02320 *
02330          MOVE    ASEQ,0                 RESET SEQUENTIAL SWITCH
02340          IF      (AEOD,NE,0)            IF END OF DATA EXIT EXISTS
02350            GOTO    (AEOD)               GO TO IT
02360          ELSE
02370            GOTO    EXIT
02380          ENDIF
02390           SPACE 5
```

```
02400 ALTERR    EQU  *
02410 ***************************************************************
02420 **  ERROR EXIT. RESET SEQUENTIAL FLAG AND PASS CONTROL TO USERS
02430 **  ERROR EXIT.
02440 ***************************************************************
02450           MOVE   ASEQ,0
02460           MOVE   #1,ASYSRC              GET USERS RETURN CODE LOCATION
02470           MOVE   (0,#1),OPENTAB         COPY SYSTEM RETURN CODE
02480           IF     (AERR,NE,0)            IF ERROR EXIT EXISTS
02490             GOTO    (AERR)              GO TO IT
02500           ELSE
02510             GOTO    EXIT
02520           ENDIF
02540 ***************************************************************
02550 **  DATA AREAS
02560 ***************************************************************
02570 ALTIACB   EQU   *                       START OF ALTERNATE IACB
02580           DATA  F'0'                    IACB POINTER
02590 A1KEY     EQU   *
02600           DATA  50X'0'                  KEY SAVE AREA (MAX LEN=50 BYTES)
02610 AENTSIZE  EQU   *-ALTIACB               SIZE OF ONE ENTRY
02620 AMAXKEY   EQU   *-A1KEY                 MAXIMUM KEY SIZE
02630           DATA  2F'0'                   EXTRA IACBS
02640           DATA  100X'0'                 EXTRA KEY AREAS
02650 ALSTIACB  DATA  D'0'                    LAST IACB SAVE AREA
02651 DOUBLE1   DATA  F'0'                    FIRST HALF OF DOUBLE WORD
02652 DOUBLE2   DATA  F'0'
02660 AKPOS     DATA  F'0'                    KEY POSITION
02670 AKSIZE    DATA  F'0'                    KEY LENGTH
02680 ASYSRC    DATA  F'0'                    USERS SYSRC CELL
02690 AERR      DATA  F'0'                    USERS ERROR EXIT
02700 AEOD      DATA  F'0'                    USERS END OF DATA EXIT
02710 ASEQ      DATA  F'0'                    SEQUENTIAL MODE SWITCH
02720 ALTTSIZE  EQU   *-ALTIACB
02740 DSCB#     EQU   3                       NUMBER OF ENTRIES IN DSCB TABLE
02750 *                                       * PASSED DURING CONCAT FUNCTION
02760 DSCB#M1   EQU   DSCB#-1
02770 BUFFADR   EQU   DSCB#*2
02780 CONCAT    EQU   14
02790 FCBKEYLN  EQU   1                       FCB KEY LENGTH OFFSET
02800 FCBKEYDP  EQU   2                       FCB KEY POSITION OFFSET
02810 COMPLEN   DATA  F'0'
02820 COUNT     DATA  F'0'
02830 OPENTAB   DATA  F'0'
02840           DATA  A(ALTERR)
02850           DATA  A(ALTEOD)
02860 REGA      DATA  F'0'
02870 REGB      DATA  F'0'
02880 SAVEREGS  DATA  2F'0'
02890           EJECT
02900           COPY   IAMEQU
02910           EJECT
02920           COPY   TCBEQU
02930           END
```

Sample Program Using ALTIAM

```
*****************************************************************
*                                                               *
*   SAMPLE PROGRAM USING ALTIAM SUBROUTINE FOR PROCESSING       *
*   CONCATENATED DATA SETS                                      *
*                                                               *
*****************************************************************
*
         EXTRN  ALTIAM
ALTSAMPL PROGRAM START,                                         X
               DS=((IAMDS1,??),(IAMDS2,??),(IAMDS3,???))
START    EQU    *
*****************************************************************
*   OPEN THE INDEXED ACCESS METHOD DATA SETS FOR                *
*   REQUEST PROCESSING VIA ALTIAM.                              *
*****************************************************************
         CALL ALTIAM,(CONCAT),IACB,(DSCBTAB),                   X
               (OPENTAB),(SHARE)
*****************************************************************
*   PERFORM A DIRECT RETRIEVAL OF THE FIRST RECORD              *
*   WHOSE KEY IS GREATER THAN '332-0000'.  THE KEY              *
*   FIELD WILL BE MODIFIED TO REFLECT THE KEY OF                *
*   THE RECORD RETRIEVED.  THIS RECORD IS LOCATED IN            *
*   THE FIRST DATA SET.                                         *
*****************************************************************
         CALL  ALTIAM,(GET),IACB,(BUFF),(KEY1),(GT)
*****************************************************************
*   PERFORM A SEQUENTIAL RETRIEVAL OF THE FIRST TWO             *
*   RECORDS WHOSE KEYS ARE GREATER THAN OR EQUAL TO             *
*   '587-1134'.  THESE RECORDS WILL BE FOUND IN THE             *
*   SECOND DATA SET.                                            *
*****************************************************************
         CALL  ALTIAM,(GETSEQ),IACB,(BUFF),(KEY2),(GE)
         CALL  ALTIAM,(GETSEQ),IACB,(BUFF)
*****************************************************************
*   DELETE THE RECORD WHOSE KEY IS '701-4320' BY A              *
*   SEQUENTIAL UPDATE.  THIS KEY IS IN THE THIRD INDEXED        *
*   FILE.                                                       *
*****************************************************************
         CALL  IAM,(GETSEQ),IACB,(BUFF),(KEY3),(UPEQ)
         CALL  IAM,(PUTDE),IACB,(BUFF)
         CALL  IAM,(ENDSEQ),IACB,(BUFF)   END SEQ PROCESSING
```

```
*************************************************************
*  INSERT A NEW RECORD WITH A KEY '370-6543' INTO           *
*  FIRST DATA SET.                                          *
*************************************************************
          CALL  IAM,(PUT),IACB,(NEWREC)
          GOTO  FINISH
ERROR     EQU   *
          MOVE  RTCODE,ALTSAMPL
          ENQT
          PRINTEXT 'ALTIAM ERROR RT CODE = ',LINE=0
          PRINTNUM RTCODE,TYPE=S,FORMAT=(3,0,I)
          DEQT
FINISH    EQU   *
          CALL  IAM,(DISCONN),IACB
          PROGSTOP
          EJECT
*************************************************************
*   DATA DEFINITION AND STORAGE AREAS                       *
*************************************************************
RTCODE    DATA  F'0'            INDEXED ACCESS METHOD RET CODE
OPENTAB   DATA  F'0'            SYSTEM RETURN CODE ADDRESS
          DATA  A(ERROR)        ADDRESS OF ERROR EXIT ROUTINE
          DATA  F'0'            ADDRESS OF END OF DATA ROUTINE
DSCBTAB   DATA  A(DS1)
          DATA  A(DS2)
          DATA  A(DS3)
BUFF      DATA  CL80' '
NEWREC    DATA  CL80'370-6543  RECORD FILLER'   RECORD TO BE
*                                                 INSERTED
KEY1      TEXT  '332-0000',LENGTH=28        KEY FROM DS1
KEY2      TEXT  '587-1134',LENGTH=28        KEY FROM DS2
KEY3      TEXT  '701-4320',LENGTH=28        KEY FROM DS3
IACB      DATA  F'0'                    ADDR OF IACB PUT HERE
CONCAT    EQU   14
          DSCB  DS#=DICDSCB,DSNAME=$$EDXVOL
          COPY  IAMEQU
          ENDPROG
          END
```

HANDLING ERRORS

All Indexed Access Method requests return a code in the task code word of the Task Control Block (TCB). The task code word is the same name as the task name. The return code reflects the condition of the requested function. Return codes are grouped in the following categories:

- -1 - Successful completion
- Positive - Error
- Negative - Warning

Error Exit Facilities

There are three types of error exits for your application:

- Task error exit, provided by the supervisor
- Error exit, provided by the Indexed Access Method
- The task error exit of the the Indexed Access Method itself

Task Error Exit

You can specify a task error exit routine that will receive control if your application program causes a soft exception or if a machine check occurs during the execution of your application.

Since your application may have outstanding pending requests (for example, a record is being held for update or a data set is being processed sequentially), you should notify the Indexed Access Method if you choose to terminate your application. Task error exit allows you to release records, disconnect from any data set you are connected to, and make your resources available to other applications. Use of the task error exit facility helps to ensure data integrity and allows proper termination or continuation of your application.

Implementing the task error exit facility is described in "Chapter 13. Diagnostic Aids and Facilities" on page 265.

Error Exit

In PROCESS and LOAD requests, the address of an error exit routine can be specified by the ERREXIT parameter. If specified, this routine is executed whenever an Indexed Access Method request terminates with a positive return code.

If the exit routine is not specified, the next sequential instruction after the request is executed regardless of the value of the return code.

$IAM Task Error Exit

The Indexed Access Method itself has a task error exit. If this error exit is given control by the supervisor, it writes two

messages to the $SYSLOG device: "$IAM HAS INCURRED A SEVERE ERROR" and "$IAM CENTRAL BUFFER ADDRESS IS n/xxxx" where n is the partition number and xxxx is the address. $IAM then goes into a non-recoverable wait and will not process any access requests. Use the dump facility to dump the central buffer and take appropriate action to quiesce your application. You may use the recovery and backup procedures to restore the data set, or you can resume execution of your application. To restart your application, you can either IPL again or cancel $IAM and reload it.

If you wish to extend the logic of the error exit, code your own exit to replace the $IAM task error exit. Then rename CDIERR (the $IAM task error exit), name your error exit CDIERR, and rebuild $IAM.

System Function Return Codes

If a system function called by an Indexed Access Method request terminates with a positive return code, the return code is placed in a location named by the SYSRTCD parameter in the PROCESS or LOAD request. This location is used until a DISCONN is issued.

For example, the GET request uses the supervisor read function. If the read terminates with a positive return code, that return code is saved in the location named by the SYSRTCD parameter in the PROCESS request associated with the GET request. The GET request also terminates with a positive return code in the task control word. The positive return code indicates that a read error has occurred. The cause of the read error can be determined from examining the location named by the SYSRTCD parameter.

The Data-Set-Shut-Down Condition

Sometimes an I/O error occurs that is not associated with a specific request. For example, task A issues a GET on data set X. To secure buffer space to satisfy the request, the Indexed Access Method attempts to write a block to data set Y and, in writing the record, an error occurs. Data set Y is damaged but there is no requesting program to accept an error return code.

The error is indicated by setting the data-set-shut-down condition for data set Y. After this condition occurs, no requests except a DISCONN are accepted for data set Y.

Later, if task B issues a GET on data set Y, the request is terminated with a data-set-shut-down return code. Task B should issue a DISCONN and use recovery and backup procedures to reconstruct the data set. An initial program load (IPL) cancels the data-set-shut-down condition.

Deadlocks and the Long-Lock-Time Condition

Since the Indexed Access Method uses record and block locks to preserve file integrity, deadlock and long-lock-time conditions may occur.

The deadlock condition occurs when two or more tasks interact in such a way that one or more resources becomes permanently locked, making further progress impossible. A deadlock can also occur when two requests from the same task require a lock on the same record or a lock on the same block in sequential mode.

A long-lock-time condition occurs when your program acquires a record for update and does not return the record to $IAM for a long time.

Application tasks should avoid using the Indexed Access Method in such a way that a record or block remains locked for a long period of time, since other tasks may attempt to use the same record or block. In a terminal oriented system, make every effort to ensure that a record or block is not locked during operator "think" time. Specifically, you should attempt to follow these rules:

- Do not retrieve a record for update, display the record at the terminal, and wait for the operator to modify it.
- Do not retrieve a record in sequential mode, display the record at the terminal, and wait for an operator response.

In both of these cases, a record or block is locked during operator "think" time and could be locked indefinitely.

A deadlock cannot be broken except by freeing the locks (records) that are being waited on.

If your application uses more than one IACB, deadlocks are possible. For example, one task has read record A and attempts to read record B, while another task has read record B and attempts to read record A. If you are using more than one IACB per task, use ENQ/DEQ and inter-program communications to avoid the deadlocks.

You can avoid the long-lock-time condition by using the following sequence of operations:

1. Retrieve the desired record without specifying update.

2. Perform processing in a work area.

3. Retrieve the record, specifying update.

4. Compare the record read in Step 1 with the record read in Step 3.

5. If the records are identical, issue a PUTUP request, specifying the address of the copy in the work area. If they are not identical, issue a RELEASE request for the record read in Step 3, and repeat Steps 1 through 5.

To retrieve records in sequential mode, use the technique described in "Resource Contention."

RESOURCE CONTENTION

Application programs that use the Indexed Access Method are executed the same as other application programs. Because the Indexed Access Method and the indexed data sets are resources available to all tasks, delays can occur under heavy system usage. When more than one task uses the Indexed Access Method, contention can occur between tasks for any of the following resources:

- An entire indexed file
- An index block in the data set
- A data block in the data set
- A data record in the data set
- Buffer space from the system buffer pool

For example, during the execution of a request from task A, some buffer space is required and an index block, data block, or record is locked (made unavailable to other requests). A request from task B requires more buffer space than is available or attempts to retrieve a block or record that was locked by task A. Task B must wait until the required resource becomes available.

Resources required by the Indexed Access Method are allocated only for the duration of a request except under the following circumstances:

- During an update, when control returns to the task after a GET or GETSEQ for update, the subject record is locked. The lock is released when the update is completed with a PUTUP, PUTDE, RELEASE, or DISCONN.

- During sequential processing, when control returns to the task after a GETSEQ, the block containing the subject record is locked and held in the buffer.

  Subsequent GETSEQ requests pick up records directly from the buffer. When a GET requires a record from the next block, the current block and buffer are released. Pending requests for a buffer area are satisfied and the next block is locked and held in the buffer. Except for momentary release of the buffer area between blocks, a block is locked while it is being processed. Processing is terminated by an end-of-data condition, an ENDSEQ request, or a DISCONN request.

The update should be completed promptly. Use the following guidelines to avoid resource contention:

- Disconnect all indexed data sets before task termination. The DISCONN request releases locked records or blocks and writes records that have not already been written.

- With multiple Indexed Access Method applications, use direct access to retrieve a group of records. A suggested method is the following:

  1. Retrieve the first record by key.

  2. Extract the key from the record and save it for the next retrieval.

  3. Retrieve the next record using the saved key and a greater than key relational operator (GT or UPGT).

  4. Repeat the second and third steps until processing is complete.

THE INDEXED DATA SET

Preparing the Data

The following sections describe how you can design an indexed data set that uses space efficiently and provides optimum performance.

Defining the Key

Define a single key field by specifying its size and position in the record when the data set is built by the define command of the $IAMUT1 utility. The longer the key, the larger the index. The key should not be longer than necessary but long enough to ensure uniqueness.

Ensuring Uniqueness of the Key. To identify each record in an indexed data set, each key must be unique. If key duplication is possible, the key field must be expanded.

For example, customer name is a key which may involve duplicates. To avoid duplication, lengthen the key field to include other characters such as part of the customer address or the account number. Since the characters in the key must be contiguous, you may need to rearrange the fields in the record.

Another way to eliminate duplication is to modify new records dynamically whenever a duplication occurs during loading or processing. One or more characters at the end of the key field can be reserved for a suffix code. Whenever a duplicate occurs, add a value to the suffix and make another attempt to add the record to the data set. The result is a data set that can contain a sequence of keys such as Smith, Smith1, and Smith2. If you add a suffix, you must use the entire unique key to access a record.

Providing Access by More Than One Key. To provide good performance with both direct and sequential access, each indexed data set is indexed by a single key. At times, however, it may be useful to locate records by a secondary key. For example, in a customer file indexed by account number, you might want to locate a record by customer name.

One way of providing access by a secondary key is to build a second indexed data set composed of short records that contain only the secondary and primary keys. Using the secondary key to access this data set, the associated primary key can be determined. The primary key can then be used to locate the desired record in the first data set.

Where there are multiple keys to a data set, ensure high performance by selecting as the primary key the one that is used most often or the one with which you plan to do sequential processing.

Selecting the Block Size

Records can be blocked in an indexed data set. The block size must be a multiple of 256. Blocking reduces I/O activity and

allows for free space to be interspersed among base records to provide for inserts. The three kinds of free space are: free record(s) in a data block, free block(s) at the end of each block grouping, and free cluster(s) at the end of the data set.

Specify record size and block size when building the data set by the setparms (SE) command of the $IAMUT1 utility. Each block has a 16-byte header. Therefore, the number of records per block is:

$$\frac{\text{(block size - 16)}}{\text{record size}}$$

The result is truncated; that is, any remainder is dropped. A remainder represents the number of unused bytes in the block. Selection of a block size is largely dependent on record size, but the block size must be a multiple of 256. Other factors to consider are insert activity and buffer space.

Insert Activity. Each block contains allocated record areas into which base records are loaded and free record areas into which records can be inserted. The ratio of allocated records to free records in a block should be the ratio of estimated base records to estimated inserts in the data set. Ideally, block size should be large enough to accommodate enough records to approximate this ratio.

Buffer Space. A large block size minimizes read/write activity but requires more buffer space. Some processing requires a buffer large enough for two blocks.

Examples. A data set consists of 1000 base records with an estimate of 500 records to be inserted and a record size of 70 bytes. Select a block size and a number of free records per block to build an indexed data set.

1. Selecting a block size of 256 with 1 free record per block implies (256-16)/70 = 3 records per block, with a remainder of 30 bytes. The ratio of 2 allocated records and 1 free record accurately reflects the insert activity. Buffer size is minimized. Some space is wasted on the disk (30 bytes per block). Designing 80-byte records and 256-byte blocks for this data set effectively uses these 30 bytes.

2. Selecting a block size of 512 with 2 free records per block implies (512-16)/70 = 7 records per block, with a remainder of 6 bytes. The ratio of 5 allocated records to 2 free records underestimates the insert activity. The larger block size requires a larger buffer but increases I/O efficiency. Fewer bytes are wasted on the disk (6 bytes in 2 sectors).

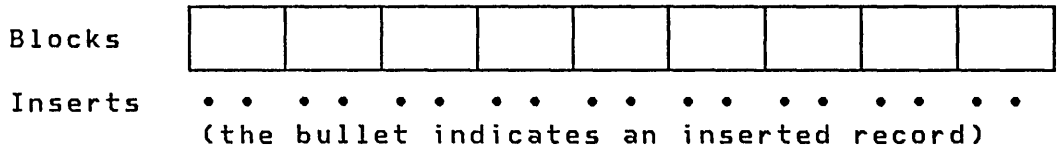
Estimating Free Space

Specify free space for inserts using the setparms (SE) command of the $IAMUT1 utility.

Estimating free space exactly is not necessary. Experience can be your best guide; if the need for file reorganization is signalled (no space for an insert) before a major portion of the free space is utilized, you know you must adjust the mix of free records and free blocks, reserve blocks, and reserve index blocks.

As a general approach, estimate not only the number of inserts but also their distribution throughout the data set. For example, consider a data set with 5 records per block, and 10 data blocks per cluster. Suppose that the data set consists of 300 base records and 200 inserts.

If the inserts are distributed evenly throughout the data set, the pattern of inserts is:

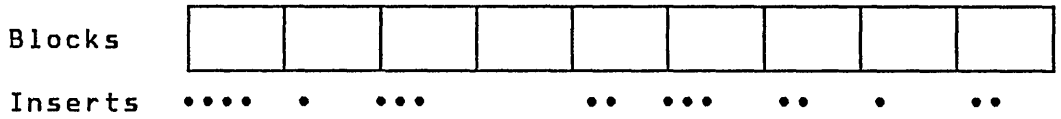


(the bullet indicates an inserted record)

With this kind of distribution you can specify 2 free records per block to absorb the inserts; no free blocks are needed.

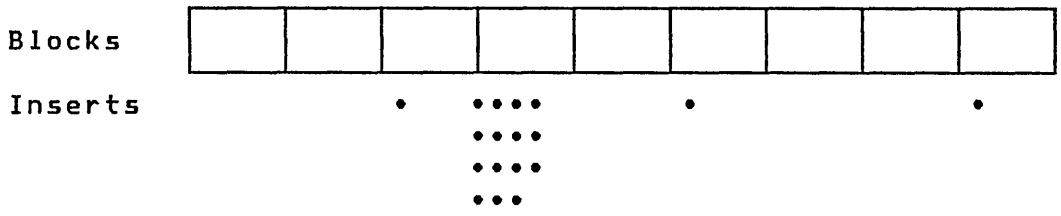
Of course inserts do not usually occur in such an even pattern. Free blocks help to absorb a concentration of inserts. The more uneven the expected distribution, the greater the free block specification should be.

Suppose the same number of inserts is distributed in this pattern:



With this distribution you must specify either 3 free records per block, or 20% free blocks with 2 free records per block.

Now suppose the distribution were more uneven:



In this case a satisfactory mix of free space is 1 free record per block and 40% free blocks.

Note: The next several paragraphs will be clearer if you refer to the definitions in "Data Set Format" on page 192.

If the anticipated insert activity is confined to a few clusters only, use a **free pool**. A free pool is a group of blocks at the end of an indexed data set that are available whenever they are needed. However, in order to use blocks from the free pool, the data set must be structured so that they can be logically connected where they are needed. This structure is specified with the RSVBLK and RSVIX parameters of the define (DF) command of the $IAMUT1 utility.

Use the RSVBLK parameter to indicate the percentage by which a cluster can grow. If you code the RSVBLK parameter, $IAMUT1 leaves reserve entries in the primary-level index blocks (PIXBs). These reserve entries can be used to point to the data blocks taken from the free pool.

Use the RSVIX parameter to indicate the percentage by which a cluster grouping can grow. If you code the RSVIX parameter, $IAMUT1 leaves reserve entries in the second-level index blocks (SIXBs). The reserve entries can be used to point to a PIXB taken from the free pool. The new PIXB can grow into a full-sized cluster as data blocks are taken from the free pool and added to this new cluster.

To illustrate the advantage of a free pool, assume that a data set contains 50 clusters of 10 data blocks each and that 40% of the blocks in the cluster are free blocks. There are 200 free blocks in the data set. If most of the inserts into the data set will fall into a relatively small key range and do not normally require more than 50 blocks, 150 blocks are saved by specifying no free blocks and a 40% RSVBLK.

A 25% FPOOL parameter provides the 50 blocks in the free pool to be used when inserts are required. The result is that the data set still accepts all the anticipated inserts and 150 blocks are saved.

If insert activity into the data set is anticipated to be relatively even, the best response time is achieved by reserving free records and free blocks for inserts.

If insert activity is to be primarily into one or more areas or key ranges, however, the space for inserts should be reserved as reserve blocks and/or reserve indexes. This results in the most efficient use of space in the data set.

The space for inserts can be divided between free records, free blocks, reserve blocks, and reserve indexes to suit your requirements.

To determine how many blocks are required for an indexed data set with a given combination of free records, free blocks, reserve blocks, reserve index blocks, and free pool size, use the SE command of the $IAMUT1 utility.

Building The Indexed Data Set

The SE and DF commands of the $IAMUT1 utility allow you to specify the size and format of your indexed data set and to format the data set. Use the SE command to enter those values that determine the size of the indexed data set and to receive a display of the size calculation information. Use the DF command to format the data set, using the values previously specified on the SE command.

Determining Size and Format

The structure of the data set is determined by the following parameters of the SE command:

- BASEREC - Estimated number of base records
- BLKSIZE - Block size
- RECSIZE - Record size
- KEYSIZE - Key size
- KEYPOS - Key position
- FREEREC - Number of free records per block
- FREEBLK - Percentage of free blocks
- RSVBLK - Percentage of reserved data blocks
- RSVIX - Percentage of reserved primary index blocks
- FPOOL - Percentage of free pool

- DELTHR - Percentage delete threshold

The define (DF) command fixes the size of the data set. Therefore, BASEREC, FREEREC, FREEBLK, RSVBLK, RSVIX, and FPOOL should be large enough to accommodate the maximum number of records planned for the data set. To calculate the size of the data set for a given combination of the define parameters, use the SE command.

The DF command allows you to select the immediate write-back option. If you select this option, modified records are written to the file immediately; this contributes to the integrity of the file; however, response time increases.

Defining and Creating the Indexed Data Set

The setparms (SE) command allows you to review the size calculation information without actually formatting the data set. $IAMUT1 returns to your terminal the size of the data set and other information. The calculations performed by the SE function are described in "Data Set Format" on page 192.

Use the DF command to format the data set. You are prompted for the volume and data set names and the immediate write-back option. (Note: the data set must have been previously created using the CR command of the $IAMUT1 utility or the AL command of the $DISKUT1 utility.) The data set is connected and then formatted by the define function. If the data set does not contain sufficient space to support the specified format, $IAMUT1 returns the amount of space required. Knowing the available space and using the SE command, you can vary the define parameters to design a data set that fits.

If the specified data set does not exist, a connect error occurs and $IAMUT1 gives the option to retry. If you retry, the utility prompts for the volume and data set names, and the function is attempted again.

Using the $IAMUT1 Utility - An Example

A data set is to accommodate 10,000 base records with a record size of 70 bytes. An estimated 5,000 records are to be inserted.

Selecting a block size of 256 allows three records per block ((256-16)/70)) with a remainder of 30 bytes. If the data set were created with one free record per block, the ratio of two base records to one free record would accurately reflect the insert activity. Buffer size is minimized. Some space (30

bytes per block) is wasted.

Selecting a block size of 512 allows seven records per block ((512-16)/70) with a remainder of six bytes. If the data set were created with two free records per block, the ratio of five base records to two free records would overestimate the insert activity. The larger block size requires a larger buffer but increases I/O efficiency. In addition, fewer bytes are wasted (six bytes).

Assume that the user has entered the DF subcommand to allocate the file using the specifications shown in Example 2. Name the file IDATA and placed it on EDX002.

Example 1

```
ENTER COMMAND (?): SE

ENTER BASEREC 10000
ENTER BLKSIZE 256
ENTER RECSIZE 70
ENTER KEYSIZE 10
ENTER KEYPOS  1
ENTER FREEREC 1
ENTER FREEBLK 0
ENTER RSVBLK  0
ENTER RSVIX   0
ENTER FPOOL   0
ENTER DELTHR  0

TOTAL LOGICAL RECORDS/DATA BLOCK:              3
FULL RECORDS/DATA BLOCK:                       2
INDEX ENTRY SIZE:                             14
TOTAL ENTRIES/INDEX BLOCK:                    17
FREE ENTRIES/PIXB:                             0
RESERVE ENTRIES/PIXB(BLOCKS):                  0
FULL ENTRIES/PIXB:                            17
RESERVE ENTRIES/SIXB:                          0
FULL ENTRIES/SIXB:                            17
DELETE THRESHOLD ENTRIES:                      1
INITIAL ALLOCATED DATA BLOCKS:              5000
FREE POOL SIZE IN BLOCKS:                      0
# OF INDEX BLOCKS AT LEVEL   1:              295
# OF INDEX BLOCKS AT LEVEL   2:               18
# OF INDEX BLOCKS AT LEVEL   3:                2
# OF INDEX BLOCKS AT LEVEL   4:                1

DATA SET SIZE IN EDX RECORDS:               5317
INDEXED ACCESS METHOD RETURN CODE:            -1
SYSTEM RETURN CODE:                           -1
```

Example 2

```
ENTER COMMAND (?): SE

ENTER BASEREC
ENTER BLKSIZE 512
ENTER RECSIZE
ENTER KEYSIZE
ENTER KEYPOS
ENTER FREEREC 2
ENTER FREEBLK
ENTER RSVBLK
ENTER RSVIX
ENTER FPOOL
ENTER DELTHR

TOTAL LOGICAL RECORDS/DATA BLOCK:                7
FULL RECORDS/DATA BLOCK:                         5
INDEX ENTRY SIZE:                               14
TOTAL ENTRIES/INDEX BLOCK:                      35
FREE ENTRIES/PIXB:                               0
RESERVE ENTRIES/PIXB(BLOCKS):                    0
FULL ENTRIES/PIXB:                              35
RESERVE ENTRIES/SIXB:                            0
FULL ENTRIES/SIXB:                              35
DELETE THRESHOLD ENTRIES:                        1
INITIAL ALLOCATED DATA BLOCKS:                2000
FREE POOL SIZE IN BLOCKS:                        0
# OF INDEX BLOCKS AT LEVEL   1:                 58
# OF INDEX BLOCKS AT LEVEL   2:                  2
# OF INDEX BLOCKS AT LEVEL   3:                  1

DATA SET SIZE IN EDX RECORDS:                 4124
INDEXED ACCESS METHOD RETURN CODE:              -1
SYSTEM RETURN CODE:                             -1
```

Note: Respond to the prompts with the values you wish to change. The utility reuses the values from previous execution.

Example 3

```
ENTER COMMAND (?): DF
DO YOU WANT IMMEDIATE WRITE-BACK? N
ENTER (NAME,VOLUME): IDATA,EDX002

TOTAL LOGICAL RECORDS/DATA BLOCK:                7
FULL RECORDS/DATA BLOCK:                         5
INDEX ENTRY SIZE:                               14
TOTAL ENTRIES/INDEX BLOCK:                      35
FREE ENTRIES/PIXB:                               0
RESERVE ENTRIES/PIXB(BLOCKS):                    0
FULL ENTRIES/PIXB:                              35
RESERVE ENTRIES/SIXB:                            0
FULL ENTRIES/SIXB:                              35
DELETE THRESHOLD ENTRIES:                        1
INITIAL ALLOCATED DATA BLOCKS:                2000
FREE POOL SIZE IN BLOCKS:                        0
# OF INDEX BLOCKS AT LEVEL   1:                 58
# OF INDEX BLOCKS AT LEVEL   2:                  2
# OF INDEX BLOCKS AT LEVEL   3:                  1

DATA SET SIZE IN EDX RECORDS:                 4124
INDEXED ACCESS METHOD RETURN CODE:              -1
SYSTEM RETURN CODE:                             -1

ENTER COMMAND (?): EN

$IAMUT1  ENDED AT 00:38:47
```

The key differences between Example 1 and Example 2 are:

- Fewer records (256-byte blocks) are required for Example 2.
- The index in Example 2 is a three-level index, while in Example 1 it is a four-level index. This eliminates one disk access, improving performance slightly.
- Each data block has two free records in Example 2. In example 1 each data block has one free record.

Data Set Format

The define command of the $IAMUT1 utility formats and creates an indexed data set.

Use the DF command to format the data set. You are prompted for the volume and data set names and the immediate write-back option. (Note: the data set must have been previously created using the CR command of the $IAMUT1 utility or the AL command of the $DISKUT1 utility.) The data set is connected and then formatted by the define function. If the data set does not contain sufficient space to support the specified format, $IAMUT1 returns the amount of space required. Knowing the available space and using the SE The information required to establish the format and the number of blocks in a data set is provided by ten parameters of the SE command.

Parameter	Definition
BASEREC	Number of base records
BLKSIZE	Block size
RECSIZE	Record size
KEYSIZE	Key size
KEYPOS	Key position
FREEREC	Number of free records per block
FREEBLK	Percentage of free blocks
RSVBLK	Percentage of reserved blocks
RSVIX	Percentage of reserved index
FPOOL	Percentage of free pool
DELTHR	Percentage of blocks to retain when deleting records

Blocks

The indexed data set is composed of a number of fixed length blocks. The block is the unit of data transferred by the Indexed Access Method. Block size must be a multiple of 256. A block is addressed by its relative block number (RBN). The first block in the data set is located at RBN 0.

Note that the RBN is used only in indexed data sets by the Indexed Access Method. An Indexed Access Method block differs from an Event Driven Executive record in the following ways:

1. The size of a block is not limited to 256 bytes; its length can be a multiple of 256.

2. The RBN of the first block in an indexed data set is 0. The record number of the first Event Driven Executive record in a data set is 1.

The size, in 256-byte records, of the data set is calculated by the define command of the $IAMUT1 utility.

Three kinds of blocks exist in an indexed data set: a file control block (FCB), index blocks, and data blocks. These blocks are all the same length, as defined by BLKSIZE, but they contain different kinds of information. The FCB contains control information, index blocks contain index entries, and data blocks contain data records. The control information is also contained in block headers; a description of control information is contained in Internal Design. Figure 23 also shows examples of the three block types.

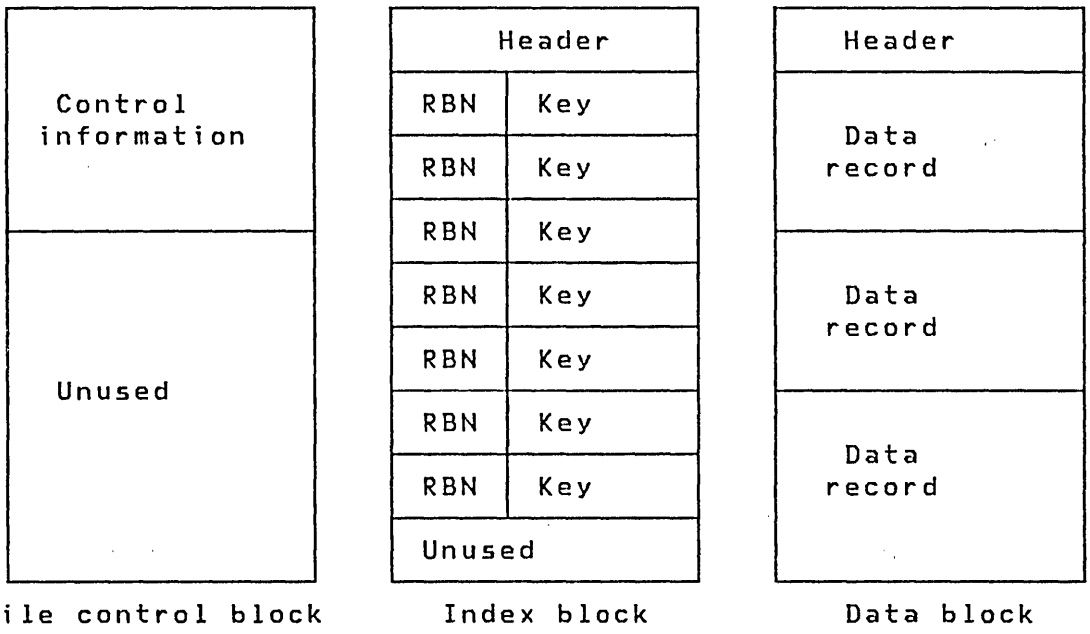


Figure 23. Indexed Data Set Block Types

The File Control Block

The file control block (FCB) is the first block in the data set (RBN 0); it contains control information. The field names in the FCB can be seen by examining a listing of FCBEQU, a copy code module that is supplied as part of the Indexed Access Method.

Index Block

An index block contains a header followed by a number of index entries. Each index entry consists of a key and a pointer. The key is the highest key associated with a block; the pointer is the RBN of that block. The number of entries contained in each index block depends on block size and key size. The header of the block is 16 bytes. The RBN field in each entry is 4 bytes. The key field in each entry must be an even number of bytes in length; if the key field is an odd number of bytes in length, the field is padded with one byte to make it even. The number of index entries in an index block is:

$$\frac{\text{block size} - 16}{4 + \text{key length}}$$

The result is truncated; any remainder represents the number of unused bytes in the block. For example, if block size is 256 and key length is 28, then each index entry is 32 bytes, there are 7 entries in a block, and the last 16 bytes of the block are unused.

Data Block

A data block contains a header followed by a minimum of two records. The number of records that can be contained in a data block depends on the size of the data block and the size of the record. The header of the block is 16 bytes. The number of record areas in the block is:

$$\frac{\text{block size} - 16}{\text{record size}}$$

The result is truncated; any remainder represents the number of unused bytes in the block. For example, if block size is 256 and record size is 80, the data block can accommodate three records and there is no unused area. The key field of the last record slot in an index block is the high key for the data block. If some records of the data block are not currently used, the key field of the last record slot is the same as the key field of

the last used record in the block. However, if the last record of the block has been deleted, the key field of the last record slot will contain a key higher than that of any other record in the block. Deletion of a record does not reduce the key range for the block.

The Index

The index of an indexed data set is constructed in several levels so that, given a key, there is a single path (one index block per level) cascading through the index levels that leads to the data block associated with that key. The index is built from the bottom up. At the lowest level are the primary-level index blocks. At the second level are index blocks containing entries that point to the primary-level index blocks. There are enough levels so that the highest level consists of a single index block.

Primary-Level Index Blocks

Entries in a primary-level index block point to data blocks. Each entry in a primary-level index block is one of three possible types:

- An allocated entry points to an active data block. The key portion of the entry is initialized to binary ones by the $IAMUT1 utility. After records have been loaded or written to a data block, the key portion of the entry which points to the data block contains the highest key from the data block.

  The pointer portion contains the RBN of the data block. Allocated entries are the first entries in an index block. The number of index entries allocated when the indexed data set is loaded is the total number of entries per index block, less the number of entries of the other two types (free block entry and reserve block entry). (Refer to Figure 24 on page 196 for an example of a primary-level index block.)

- A free block entry points to a free data block. The key portion of the entry contains binary zeros. The pointer portion contains the RBN of the free block. Free block entries follow the allocated entries in the index block. The number of index entries formatted as free entries when the indexed data set is loaded is the specified percentage (FREEBLK) of the total number of entries, with the result rounded up if there is a remainder.

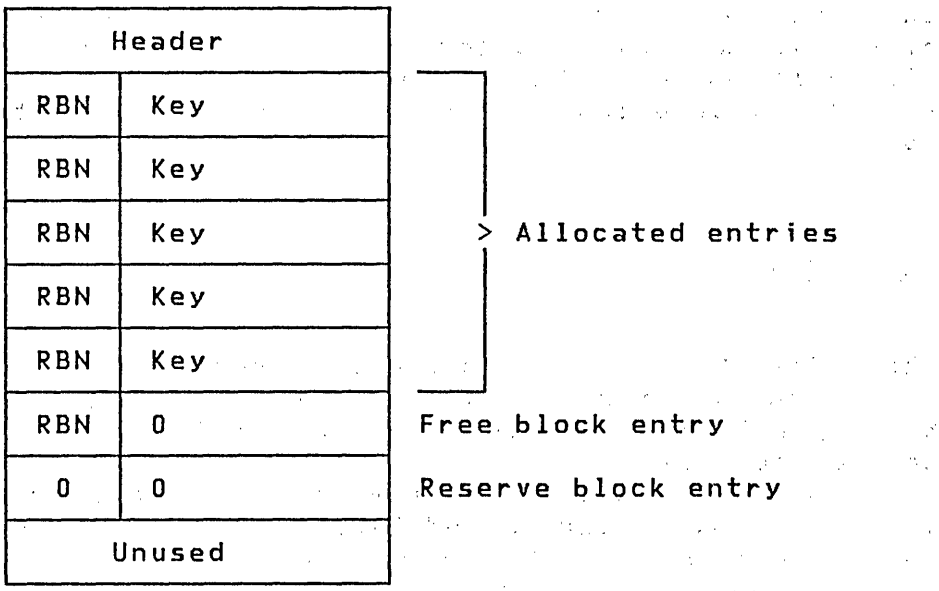


Figure 24. Example of Primary-Level Index Block

- A reserve block entry does not point to a block but is reserved for later use as a pointer to a data block which can be taken from the free pool. Both the key and pointer portions of a reserve block entry are binary zeros. Reserve block entries are at the end of the index block. When a reserve block entry is converted to a used entry, the index block is reformatted to move the entry to the allocated entry area of the block.

  The number of index entries initially formatted as reserve block entries is the specified percentage (RSVBLK) of the total number of entries, with the result rounded up if there is a remainder. However, if the number of free block entries plus the number of reserve block entries require all index entries, the number of reserve block entries is reduced by 1, providing at least one allocated entry per index block.

To calculate the number of primary-level index blocks in an indexed data set, you must know the initial number of data blocks allocated in the indexed data set. The initial number of data blocks is the specified number of base records (BASEREC) divided by the number of allocated records in a data block, with the result rounded up if there is a remainder. The number of primary-level index blocks is the initial number of allocated data blocks divided by the number of allocated entries per primary-level index block, with the result rounded up if up if there is a remainder.

Second-Level Index Block

Entries in a second-level index block point to primary-level index blocks. Each entry in a second-level index block is one of two possible types:

- An allocated entry points to an existing primary-level index block. The key portion of the entry is initialized to binary ones by the $IAMUT1 utility. After records have been loaded or written, the key portion of the entry contains the highest key from the primary-level index block. The pointer portion contains the RBN of the primary-level index block. Allocated entries are the first entries in the index block. The number of index entries allocated when the indexed data set is loaded is calculated as the total number of entries per index block, less the number of reserve index entries.

- A reserve index entry does not point to a block but is reserved for later use as a pointer to a primary-level index block that can be taken from the free pool. Both the key and pointer portions of a reserve index entry are binary zeros. Reserve index entries are at the end of the index block. The number of index entries initially formatted as reserve index entries is the specified percentage (RSVIX) of the total number of entries, with the result rounded up if there is a remainder. However, if the number of reserve index entries is the same as the total number of entries in an index block, the number of reserve index entries is reduced by 1, providing at least one allocated entry per second-level index block.

The number of second-level index blocks is the number of primary-level index blocks divided by the number of allocated entries per second-level index block, with the result rounded up if there is a remainder. (Refer to Figure 25 on page 198 for an example of a second-level index block.)

Higher-Level Index Block

Entries in a higher-level index block point to index blocks at the next lower level. All entries in higher-level index blocks are allocated entries. The key portion of the entry contains the highest key from the index block of the next lower level. The pointer portion contains the RBN of the next lower level index block. The number of blocks at any higher index level is the number of index blocks at the next lower level divided by the total number of entries per index block, with the result rounded up if there is a remainder. (Refer to Figure 26 on page 199 for an example of a higher-level index block.)

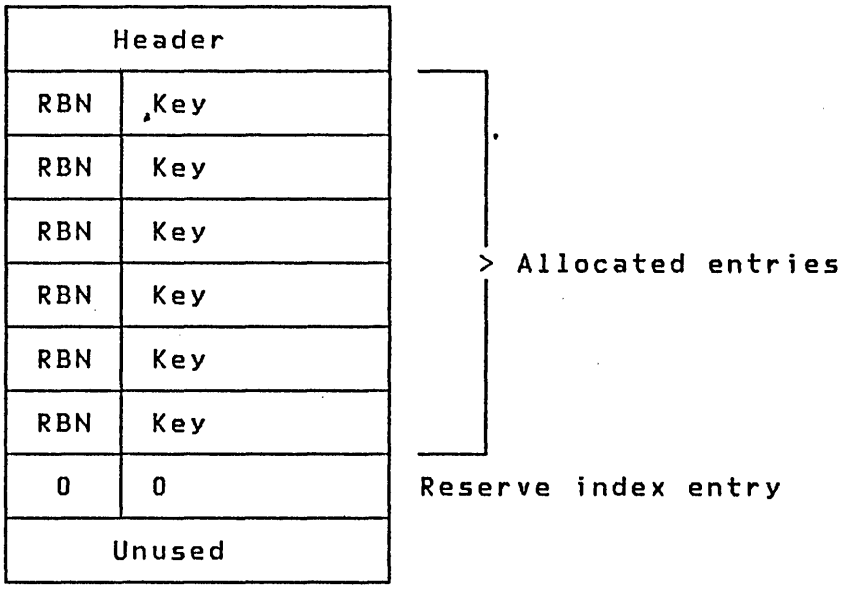


Figure 25. Example of Second-Level Index Block

If the number of index blocks at any level is one, that level is the top level of the index. Although the Indexed Access Method is capable of supporting 17 levels of index, an indexed data set is formatted with only as many index levels as are required for the number of records. If an indexed data set has not been fully loaded and one or more higher index levels have not yet been required, the unnecessary higher levels are not used, even though they exist in the file structure.

Index Example

Assume that 500 data blocks are allocated to a data set and that each primary-level index block contains one free block entry, one reserve block entry, and five allocated entries. Therefore, the total number of primary-level index blocks is 100. Each second-level index block contains one reserve index entry and six allocated entries; therefore, the number of second-level index blocks is 17. The number of entries in higher level index blocks is seven, resulting in three index blocks at the third level and one at the fourth level.

Therefore the data set contains a total of 121 index blocks of which 100 are primary-level index blocks, 17 are second-level index blocks, 3 are third-level index blocks, and 1 is a fourth-level index block. This distinction is important because, as shown later in this chapter, high-level index blocks are located contiguously at the beginning of the data

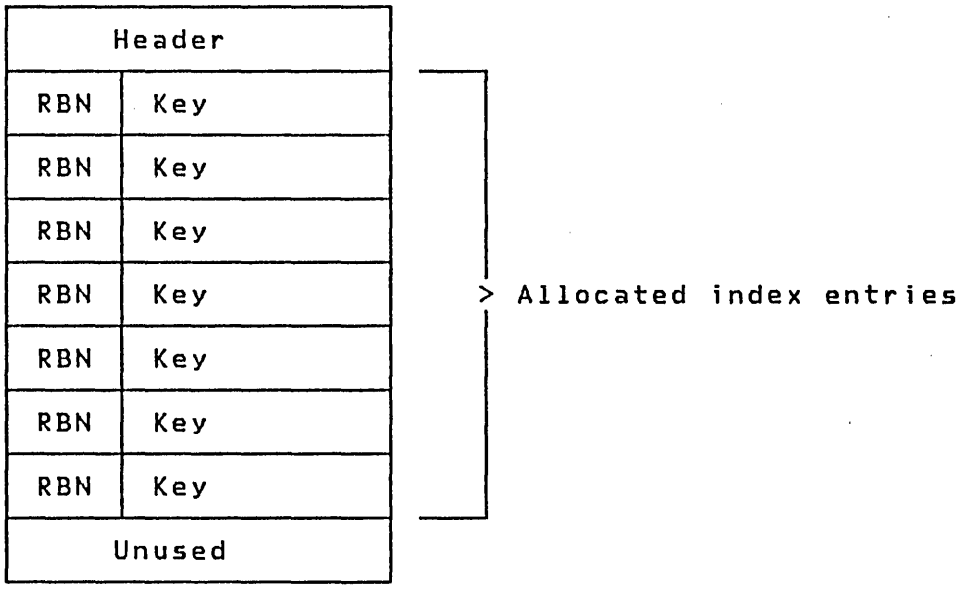


Figure 26. Example of Higher-Level Index Block

set (after the FCB), while primary-level index blocks are scattered throughout the file with the data blocks. Figure 27 shows the structure of the higher-level index blocks.

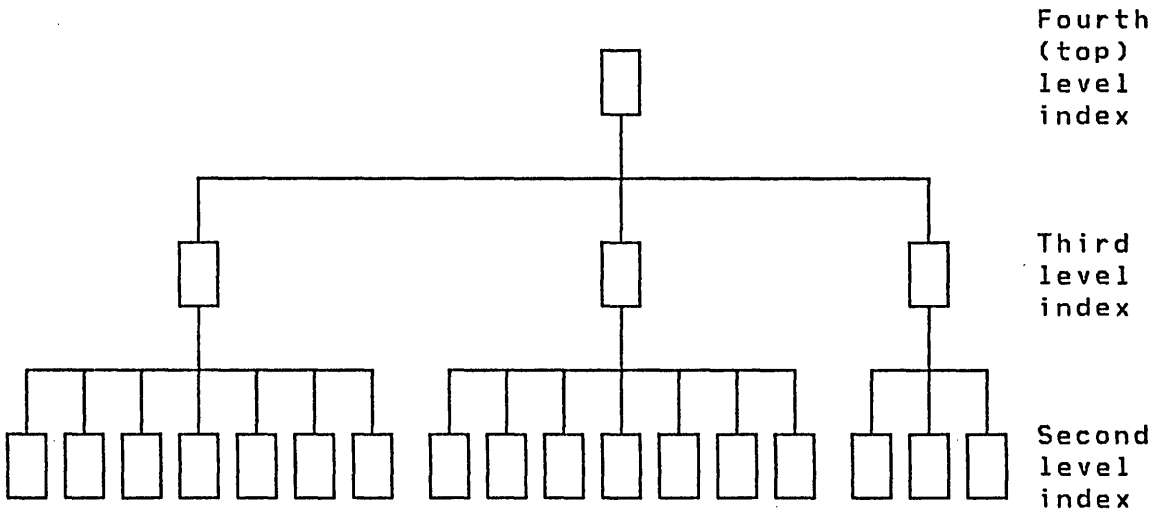


Figure 27. High-Level Index Block Structure

Cluster

Primary-level index blocks and data blocks are stored together in the data set in groups called clusters. Each cluster consists of a primary-level index block and as many data blocks as are allocated or free entries in the index block. For example, if there are seven entries in an index block, there are eight blocks in a cluster: one primary-level index block and up to 7 data blocks. If reserve blocks have been specified, the blocks represented by the reserve block entries are not included until insert activity has taken place and the required blocks have been obtained from the free pool. For example, if there are seven entries in an index block and one of the entries is a reserve block entry, the cluster consists of seven blocks (one index block and six data blocks).

Free Space

When an indexed data set is loaded with data records, free space is reserved for records that may be inserted during processing. There are four kinds of free space: free records, free blocks, reserve blocks, and reserve index entries.

Free Records: Free records are areas reserved at the end of each data block. The FREEREC parameter of define command of the $IAMUT1 utility specifies the number of free records that are reserved in each data block. The remaining record areas are called allocated records.

For example, if a block contains three data record areas and you specify one free record per block, then there are two allocated records per block. Refer to Figure 28 on page 201.

When records are loaded, the allocated records are filled, and the free records are skipped. During processing, a record can be inserted in a block that contains a free record.

Free Blocks: Free blocks follow the allocated data blocks within each cluster. For example, if the cluster contains six data blocks and you specify 10 as the percentage of free blocks, then there are five allocated blocks and one free block in each cluster.

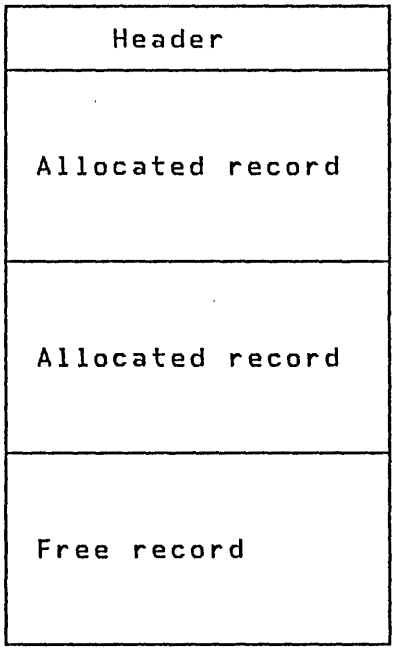


Figure 28. Example of a data block

Primary-level index block
Allocated data block
Allocated data block
Allocated data block
Allocated data block
Allocated data block
Free data block

When records are loaded, the allocated record areas in the allocated data blocks are filled, and the free blocks are skipped. During processing, as data blocks become full, a free block provides space for insertions.

Reserve Blocks: Reserve blocks do not exist in the cluster. When all data blocks in a cluster are used and another data block is needed, a data block can be created from the free pool, if the primary-level index block contains a reserve block entry. The reserve block entry in the primary-level index block points to the block, and the data block becomes an allocated data block.

Reserve Index Entries: Reserve index entries in second-level index blocks allow the index structure to be expanded by adding new primary-level index blocks. These, in turn, can have data blocks associated with them, thus forming new clusters. This process of forming a new cluster is sometimes called a cluster split.

Calculating Allocated Data Blocks, Clusters, and Free Blocks

The number of allocated data blocks in a data set is the specified number of base records (BASEREC) divided by the number of allocated records per data block, with the result rounded up if there is a remainder.

For example, suppose you intend to load 1000 records in an indexed data set that is formatted for two allocated records and one free record per block and five allocated blocks and one free block per cluster. The number of allocated blocks in a data set is:

$$\frac{\text{number of base records}}{\text{number of allocated records per block}}$$

The number of allocated blocks in this example is 1000/2 or 500 blocks.

The number of clusters in a data set is the number of allocated data blocks divided by the number of allocated entries in each primary-level index block, with the result rounded up if there is a remainder.

$$\frac{\text{number of allocated blocks}}{\text{number of allocated blocks per cluster}}$$

The number of clusters in the above example is 500/5 or 100 clusters.

Note that in both calculations, if the quotient is not an integer, it is rounded up (rather than truncated) in order to accommodate all of the base records.

The number of free blocks in the data set (not including the free pool) is the number of clusters in the data set multiplied by the number of free entries in each primary-level index block.

The Last Cluster

The last cluster in the data set may be different from the other clusters. It contains the same number of free blocks as the other clusters but only enough allocated blocks to accommodate the records that you have specified with the parameter BASEREC. Because rounding occurs in calculating the number of clusters, a few more allocated records than required may exist in the last allocated block. The last cluster can be a short one because only the required number of blocks are used.

If the number of allocated blocks divided by the number of allocated blocks per cluster leaves a remainder, the remainder represents the number of allocated entries in the primary-level index block in the last cluster. Unused entries in the last primary-level index block are treated as reserve block entries.

Sequential Chaining

Data blocks in an indexed data set are chained together by forward pointers located in the headers of data blocks. Only allocated data blocks are included in the sequential chain. Chaining allows sequential processing of the data set with no need to reference the index. When a free block is converted to an allocated block, the free block is included in the chain.

Free Pool

If you specify that you want a free pool (with the FPOOL parameter of the define command of the $IAMUT1 utility), your indexed data set contains a pool of free blocks. The file control block contains a pointer to the first block of the free pool, and all blocks in the free pool are chained together by forward pointers.

A block can be taken from the free pool to become either a data block or a primary-level index block. The block is taken from the beginning of the chain, and its address (RBN) is placed in the appropriate primary-level index block (if the new block is to become a data block) or in the second level index block (if the new block is to become a primary-level index block). Any block in the free pool can be used as either a data block or as a primary-level index block.

When a data block becomes empty because of record deletions, the data block may return to the free pool (depending on the delete threshold (DELTHR) parameter). If the data block is

returned to the free pool, reference to the block is removed from the primary-level index block, and the block is placed at the beginning of the free pool chain. Index blocks are never returned to the free pool.

Calculating the initial size of the free pool consists of the following steps:

- Each reserve block entry in a primary-level index block represents a potential data block from the free pool. The number of data blocks that can be assigned to initial clusters is the number of primary-level index blocks times the number of reserve block entries in each primary-level index block.

- Each reserve index entry in a second-level index block represents a potential primary-level index block from the free pool. The number of primary-level index blocks that can be assigned from the free pool is the number of second-level index blocks times the number of reserve index entries in each second-level index block.

- Each primary-level index block taken from the free pool consists entirely of empty (reserve block) entries. New data blocks can be taken from the free pool for the entries in the new primary-level index block. The number of data blocks is the number of entries per index block times the number of new primary-level index blocks (calculated in the previous step).

- The maximum number of blocks that can be taken from the free pool is the sum of the above three calculations.

- The actual number of blocks in the free pool is the specified percentage (FPOOL) of the maximum possible free pool, with the result rounded up if there is a remainder.

STORAGE AND PERFORMANCE

Storage Requirements

The minimum amount of storage required by the Indexed Access Method to perform all functions is about 14KB, not including the link module or any error exit routine you may have written. The storage estimate is based on the following assumptions:

- A maximum block size of 256 bytes for any indexed data set. Since the buffer must be large enough for two blocks, a 512-byte buffer is required. If your maximum block size is larger than 256 bytes, the buffer size is twice your block

size. You can improve performance by making the buffer larger. The program directory that is shipped with your PID material contains a description of the size and capacity of the buffer and information on how to modify it. The buffer that is defined in $IAM should provide adequate performance for most applications.

- One user connected to an indexed file at a time. If more than one user is connected, add about 625 bytes per user.

The size of the IBM-supplied link module which is included in your application program is about 250 bytes.

Indexed File Size

The structure of an indexed file is highly dependent on parameters you specify when you create the file. These parameters are described in "Data Set Format" on page 192.

Performance

Performance of the Indexed Access Method is primarily determined by the structure of the indexed data set being used. This structure is determined by parameters you specify when you create the data set (refer to "Data Set Format" on page 192). The following factors affect performance:

- File size. A large file spans more cylinders of the direct access device, so the average seek to get the the record you want is longer.

- Number of index levels. A file with many index levels requires more accesses to get to the desired data record, thus degrading performance. Factors which influence the number of index levels are:

  - Number of records in data set.

  - Amount and type of free space.

  - Block size.

  - Key size.

  - Data record size.

Use the $IAMUT1 utility to see the affects of the varicus parameters on the file structure. (Refer to "Using the $IAMUT1 Utility - An Example" on page 188 for an example.)

In addition to file structure, the following factors also influence performance:

- Buffer size. If you provide a large buffer when you install the Indexed Access Method, it is more likely that blocks (especially high-level index blocks) needed are already in storage and need not be recalled from the data set.

- Contention. If many tasks are using the Indexed Access Method concurrently, resource contention can result, and performance is degraded.

PART IV - EXTENDING THE SYSTEM CAPABILITIES

This part gives detailed information on how to extend the capabilities of your system.

CHAPTER 10. THE SESSION MANAGER

The session manager provides access to system functions and your applications. Full screen images, called menus, and their associated procedures invoke the functions you request. Because you control the session manager, you can modify it to meet your specific needs. You can add new options to an existing menu or create a new menu.

To add a new option (or tailor) the session manager requires an understanding of the one-to-one relationship between menus and procedures. Once you have acquired this understanding, you can:

- Use the $IMAGE utility to add a new option to an existing menu
- Use the $FSEDIT utility to add the new option to the procedure associated with the menu
- Build the procedure for the new function, which requests its execution

OPERATIONAL OVERVIEW

The session manager can be invoked in two ways:

- As part of the IPL procedure
- With the $L (load) facility

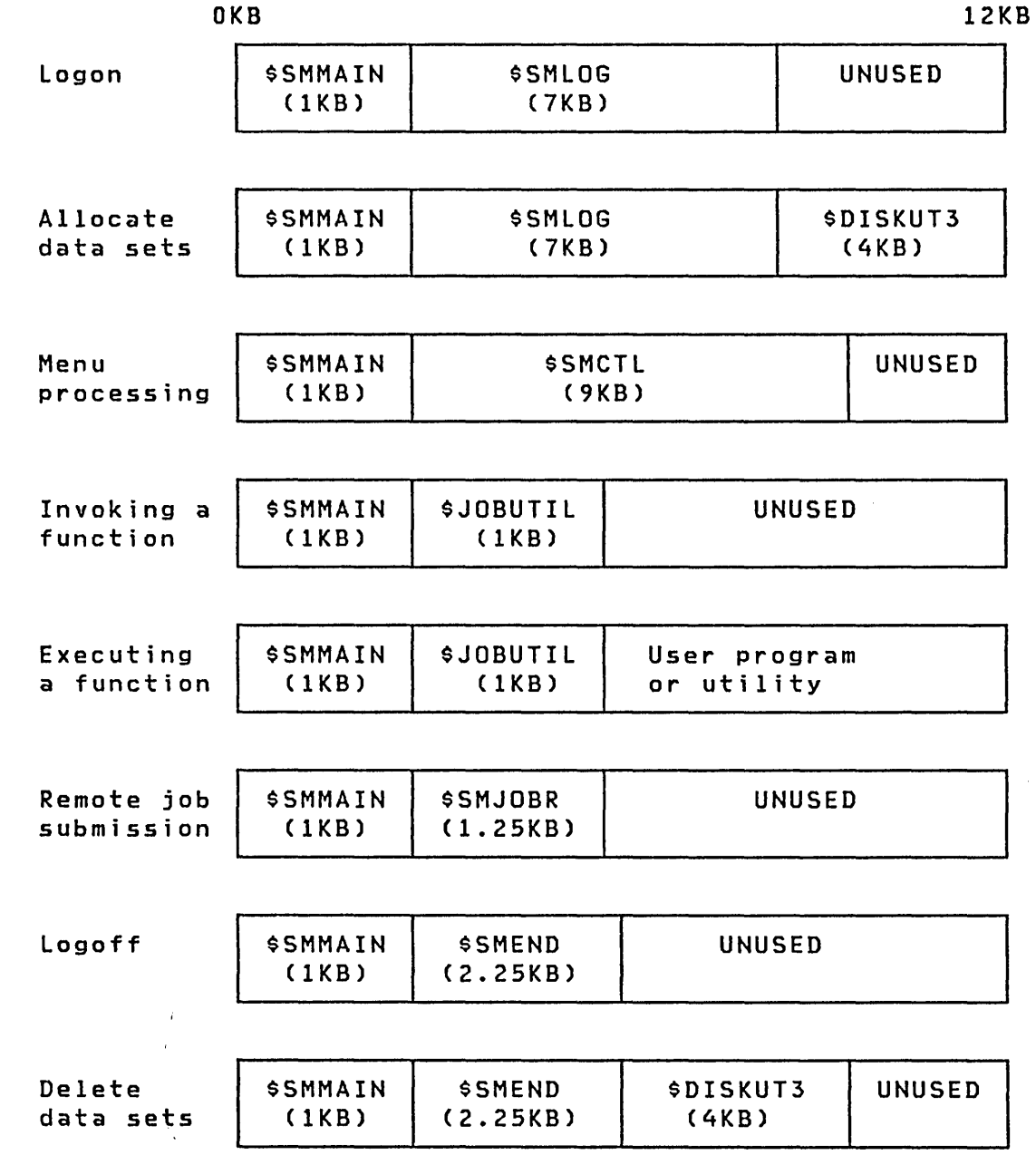
As a part of the IPL procedure, a copy of the session manager is loaded for each active 4978 or 4979 display terminal. To accomplish this, the program $SMINIT is made a part of the IPL stream by renaming it to be $INITIAL. The program $INITIAL is part of the IPL stream and is automatically loaded if it is present. $SMINIT ($INITIAL) loads a copy of the session manager's main program ($SMMAIN) for each terminal that is active. If only selected terminals are to used with the session manager, you must load the session manager as required.

The menu processing program, $SMCTL, processes menus and builds procedures that are passed to the job stream processor, $JOBUTIL, by the main program $SMMAIN. Copies of all procedures submitted to the job stream processor are stored on the same disk volume as the menus, usually EDX002.

$SMCTL reads a copy of the $JOBUTIL procedure into storage, and adds any parameters that you supply. $SMCTL then returns control to $SMMAIN, which invokes $JOBUTIL, which executes the

requested function. Figure 30 on page 213 and the tables below list all of the session manager menus, the associated procedures, and the function of each procedure.

The session manager requires a minimum partition size of 10K bytes to process menus and your requests. However, only 2K bytes remain resident when the requested functions are executing. A root phase (main program), called $SMMAIN, is resident in the partition associated with the 4978/4979 display station. $SMMAIN requires approximately 1K bytes of storage and is resident when the functions are executing; when invoked, it loads one of five programs into storage. The list below and Figure 29 on page 211 illustrate storage usage.



Note: All storage sizes are approximate.

Figure 29. Session manager storage usage

Program	Functions
$SMMAIN	Storage resident program that loads programs and invokes functions
$SMLOG	Processes your ID and alternate menu Allocates dynamic work data sets
$SMCTL	Displays menus Processes the parameters you enter Builds procedures to invoke functions Saves parameters
$JOBUTIL	Invokes the functions you request
$SMJOBR	Submits the control statements that you build to a host system with the Host Communication Facility
$SMEND	Deallocates dynamic work data sets
$SMINIT	Loads a copy of the session manager ($SMMAIN) for each active 4978/4979 terminal. It can be renamed $INITIAL (part of the IPL stream), making session manager startup automatic.

MENUS AND PROCEDURES

Menu names begin with the prefix $SMM and procedures begin with $SMP. Therefore, any menus and procedures that you create must use the $SMM and $SMP prefixes. Except for the logon menu, $SMMLOG, and the primary option menu, $SMMPRIM, and procedure, $SMPPRIM, the menus and procedures are numbered to indicate the primary option number and the secondary option number, when appropriate. Menu names are always displayed in the upper left corner of the screen.

Figure 30 on page 213 and the following tables list all of the session manager menus. Each is listed as a procedure name with its corresponding function. They are broken down into functional categories.

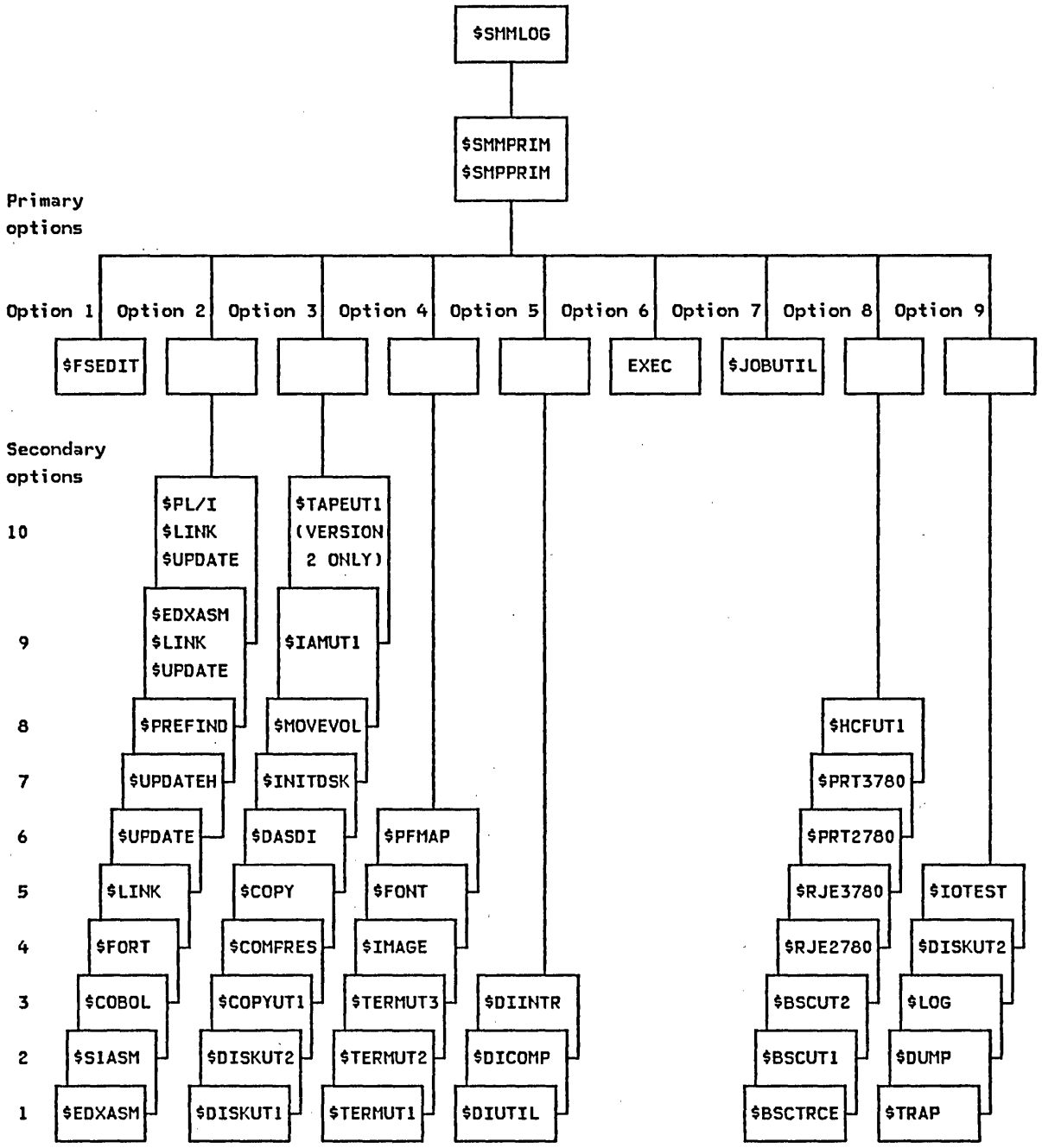


Figure 30. Session manager primary and secondary options

PRIMARY MENUS

Procedure Name	Function
$SMMLOG	LOGON MENU
$SMMPRIM	PRIMARY OPTION MENU
$SMPPRIM	PRIMARY OPTION DECISION TABLE PROCEDURE

PROGRAM PREPARATION UTILITIES

Option	Menu/ Procedure Name	Function
1	$SMP01	EXECUTE $FSEDIT PROCEDURE
2	$SMM02	PROGRAM PREPARATION SECONDARY OPTION MENU
	$SMP02	PROGRAM PREPARATION DECISION TABLE
2.1	$SMM0201	$EDXASM PARAMETER INPUT MENU
	$SMP0201	EXECUTE $EDXASM PROCEDURE
2.2	$SMM0202	$S1ASM PARAMETER INPUT MENU
	$SMP0202	EXECUTE $S1ASM PROCEDURE
2.3	$SMM0203	$COBOL PARAMETER INPUT MENU
	$SMP0203	EXECUTE $COBOL PROCEDURE
2.4	$SMM0204	$FORT PARAMETER INPUT MENU
	$SMP0204	EXECUTE $FORT PROCEDURE
2.5	$SMM0205	$LINK PARAMETER INPUT MENU
	$SMP0205	EXECUTE $LINK PROCEDURE
2.6	$SMM0206	$UPDATE PARAMETER INPUT MENU
	$SMP0206	EXECUTE $UPDATE PROCEDURE
2.7	$SMM0207	$UPDATEH PARAMETER INPUT MENU
	$SMP0207	EXECUTE $UPDATEH PROCEDURE
2.8	$SMM0208	$PREFIND PARAMETER INPUT MENU
	$SMP0208	EXECUTE $PREFIND PROCEDURE
2.9	$SMM0209	$EDXASM/$LINK/$UPDATE PARAMETER INPUT MENU
	$SMP0209	EXECUTE $EDXASM/$LINK/$UPDATE PROCEDURE
2.10	$SMM0210	$PL/I/$LINK/$UPDATE PARAMETER INPUT MENU
	$SMP0210	EXECUTE $PL/I/$LINK/$UPDATE PROCEDURE

DATA MANAGEMENT UTILITIES OPTIONS

Option	Menu/ Procedure Name	Function
3	$SMM03	DATA MANAGEMENT UTILITIES SECONDARY OPTION MENU
	$SMP03	DATA MANAGEMENT UTILITIES OPTION DECISION
3.1	$SMP0301	EXECUTE $DISKUT1 PROCEDURE
3.2	$SMP0302	EXECUTE $DISKUT2 PROCEDURE
3.3	$SMP0303	EXECUTE $COPYUT1 PROCEDURE
3.4	$SMP0304	EXECUTE $COMPRES PROCEDURE
3.5	$SMP0305	EXECUTE $COPY PROCEDURE
3.6	$SMP0306	EXECUTE $DASDI PROCEDURE
3.7	$SMP0307	EXECUTE $INITDSK PROCEDURE
3.8	$SMM0308	$MOVEVOL PARAMETER INPUT MENU
	$SMP0308	EXECUTE $MOVEVOL PROCEDURE
3.9	$SMP0309	EXECUTE $IAMUT1 PROCEDURE
3.10	$SMP0310	EXECUTE $TAPEUT1 PROCEDURE (Version 2 only)

TERMINAL UTILITIES OPTIONS

Option	Menu/ Procedure Name	Function
4	$SMM04	TERMINAL UTILITIES SECONDARY OPTION MENU
	$SMP04	TERMINAL UTILITIES OPTION DECISION TABLE
4.1	$SMP0401	EXECUTE $TERMUT1 PROCEDURE
4.2	$SMP0402	EXECUTE $TERMUT2 PROCEDURE
4.3	$SMP0403	EXECUTE $TERMUT3 PROCEDURE
4.4	$SMP0404	EXECUTE $IMAGE PROCEDURE
4.5	$SMP0405	EXECUTE $FONT PROCEDURE
4.6	$SMP0406	EXECUTE $PFMAP PROCEDURE

GRAPHIC UTILITIES OPTIONS

Option	Menu/ Procedure Name	Function
5	$SMM05	GRAPHIC UTILITIES SECONDARY OPTION MENU
	$SMP05	GRAPHIC UTILITIES OPTION DECISION TABLE
5.1	$SMM0501	$DIUTIL PARAMETER INPUT MENU
	$SMP0501	EXECUTE $DIUTIL PROCEDURE
5.2	$SMM0502	$DICOMP PARAMETER INPUT MENU
	$SMP0502	EXECUTE $DICOMP PROCEDURE
5.3	$SMP0503	EXECUTE $DIINTR PROCEDURE

EXECUTE PROGRAM UTILITIES OPTIONS

Option	Menu/ Procedure Name	Function
6	$SMM06	EXECUTE PROGRAM PARAMETER INPUT MENU
	$SMP06	EXECUTE PROGRAM PROCEDURE

JOB STREAM PROCESSOR UTILITIES OPTIONS

Option	Menu/ Procedure Name	Function
7	$SMM07	$JOBUTIL PARAMETER INPUT MENU
	$SMP07	EXECUTE $JOBUTIL PROCEDURE

COMMUNICATIONS UTILITIES OPTIONS

Option	Menu/ Procedure Name	Function
8	$SMM08	COMMUNICATION UTILITIES SECONDARY OPTION MENU
	$SMP08	COMMUNICATION UTILITIES OPTION DECISION TABLE
8.1	$SMM0801	$BSCTRACE PARAMETER INPUT MENU
	$SMP0801	EXECUTE $BSCTRACE PROCEDURE
8.2	$SMP0802	EXECUTE $BSCUT1 PROCEDURE
8.3	$SMP0803	EXECUTE $BSCUT2 PROCEDURE
8.4	$SMP0804	EXECUTE $RJE2780 PROCEDURE
8.5	$SMP0805	EXECUTE $RJE3780 PROCEDURE
8.6	$SMM0806	$PRT2780 PARAMETER INPUT MENU
	$SMP0806	EXECUTE $PRT2780 PROCEDURE
8.7	$SMM0807	$PRT3780 PARAMETER INPUT MENU
	$SMP0807	EXECUTE $PRT3780 PROCEDURE
8.8	$SMM0808	$HCFUT1 PARAMETER INPUT MENU
	$SMP0808	EXECUTE $HCFUT1 PROCEDURE

DIAGNOSTIC UTILITIES OPTIONS

Option	Menu/ Procedure Name	Function
9.0	$SMM09	DIAGNOSTICS SECONDARY OPTION MENU
	$SMP09	DIAGNOSTICS DECISION TABLE
9.1	$SMM0901	$TRAP PARAMETER INPUT MENU
	$SMP0901	EXECUTE $TRAP PROCEDURE
9.2	$SMM0902	$DUMP PARAMETER INPUT MENU
	$SMP0902	EXECUTE $DUMP PROCEDURE
9.3	$SMM0903	$LOG PARAMETER INPUT MENU
	$SMP0903	EXECUTE $LOG PROCEDURE
9.4	$SMP0904	EXECUTE $DISKUT2 PROCEDURE
9.5	$SMP0905	EXECUTE $IOTEST PROCEDURE

Primary Option Menu

The first screen to appear after the logon menu is a primary option menu called $SMMPRIM. From this menu you select the utility or function you wish to use. This selection causes a secondary option menu to appear.

You may add your applications to the session manager by creating your own primary option menu. If you create your own primary option menu, you can specify its name on the logon screen as an alternate session menu. This causes your menu to appear instead of $SMMPRIM.

Secondary Option Menus

The procedure associated with an option menu is a decision table which points to secondary menus and procedures. Figure 31 on page 219 illustrates all menus and procedures.

OPTION	PROCEDURE/ MENU	DESCRIPTION
1	$SMP01	EXECUTE $FSEDIT
2	$SMM02	PROGRAM PREP SECONDARY OPTION MENU
2.1	$SMM0201	$EDXASM PARAMETER INPUT MENU
2.2	$SMM0202	$S1ASM PARAMETER INPUT MENU
2.3	$SMM0203	$COBOL PARAMETER INPUT MENU
2.4	$SMM0204	$FORT PARAMETER INPUT MENU
2.5	$SMM0205	$LINK PARAMETER INPUT MENU
2.6	$SMM0206	$UPDATE PARAMETER INPUT MENU
2.7	$SMM0207	$UPDATEH PARAMETER INPUT MENU
2.8	$SMM0208	$PREFIND PARAMETER INPUT MENU
2.9	$SMM0209	$EDXASM/$LINK/$UPDATE PARAMETER INPUT MENU
2.10	$SMM0210	$PL/I/$LINK/$UPDATE PARAMETER INPUT MENU
3	$SMM03	DATA MANAGEMENT UTILITIES SECONDARY OPTION MENU
3.1	$SMP0301	EXECUTE $DISKUT1
3.2	$SMP0302	EXECUTE $DISKUT2
3.3	$SMP0303	EXECUTE $COPYUT1
3.4	$SMP0304	EXECUTE $COMPRES
3.5	$SMP0305	EXECUTE $COPY
3.6	$SMP0306	EXECUTE $DASDI
3.7	$SMP0307	EXECUTE $INITDSK
3.8	$SMM0308	$MOVEVOL PARAMETER INPUT MENU
3.9	$SMP0309	EXECUTE $IAMUT1
3.10	$SMM0310	EXECUTE $TAPEUT1 (Version 2 only)
4	$SMM04	TERMINAL UTILITIES SECONDARY OPTION MENU
4.1	$SMP0401	EXECUTE $TERMUT1
4.2	$SMP0402	EXECUTE $TERMUT2
4.3	$SMP0403	EXECUTE $TERMUT3
4.4	$SMP0404	EXECUTE $IMAGE
4.5	$SMP0405	EXECUTE $FONT
4.6	$SMP0406	EXECUTE $PFMAP
5	$SMM05	GRAPHICS UTILITIES SECONDARY OPTION MENU
5.1	$SMM0501	$DIUTIL PARAMETER INPUT MENU
5.2	$SMM0502	$DICOMP PARAMETER INPUT MENU
5.3	$SMP0503	EXECUTE $DIINTR
6	$SMM06	SELECTED PROGRAM PARAMETER INPUT MENU
7	$SMM07	$JOBUTIL PARAMETER INPUT MENU

Figure 31. (Part 1 of 2) Menus and Procedures

OPTION	PROCEDURE/ MENU	DESCRIPTION
8	$SMM08	COMMUNICATION UTILITIES OPTION MENU
8.1	$SMM0801	$BSCTRACE PARAMETER INPUT MENU
8.2	$SMP0802	EXECUTE $BSCUT1
8.3	$SMP0803	EXECUTE $BSCUT2
8.4	$SMP0804	EXECUTE $RJE2780
8.5	$SMP0805	EXECUTE $RJE3780
8.6	$SMM0806	$PRT2780 PARAMETER INPUT MENU
8.7	$SMM0807	$PRT3780 PARAMETER INPUT MENU
8.8	$SMM0808	$HCFUT1 PARAMETER INPUT MENU
9	$SMM09	DIAGNOSTICS SECONDARY OPTION MENU
9.1	$SMM0901	$TRAP PARAMETER INPUT MENU
9.2	$SMM0902	$DUMP PARAMETER INPUT MENU
9.3	$SMM0903	$LOG PARAMETER INPUT MENU
9.4	$SMM0904	EXECUTE $DISKUT2
9.5	$SMM0905	EXECUTE $IOTEST

Figure 32. (Part 2 of 2) Menus and Procedures

If no further inputs are required, as in the first option, then a procedure name is listed alongside the option number (such as, Option 1-$SMP01). If, as in Option 2, the option number is listed with the name of a menu, then it indicates the name of a secondary menu (such as, Option 2-$SMM02).

Procedures

Procedures are divided into two types -- option decision tables and procedures passed to $JOBUTIL, the job stream processor, to invoke the function. The $JOBUTIL procedures can contain two parts -- the first part saves parameters from session to session; the second part is the actual procedure passed to $JOBUTIL.

Figure 33 on page 222 is the procedure used to invoke the assembler $EDXASM. The parameter part begins with the key word PARAMETER and ends with the first END statement. The PARAMETER and END keywords are peculiar to the session manager, not part of the normal $JOBUTIL procedure. Within the procedure, each parameter variable is assigned a name &PARMnn., where nn is a positional index associated with an FTAB table built by the system routine $IMPROT for the menu associated with this procedure (such as, $SMM0201).

The FTAB table provides the screen location (line and spaces) and size (characters) of each parameter field on the menu, in ascending order. The session manager program $SMCTL uses the FTAB table to retrieve the parameters it uses to replace the &PARMnn. fields before passing the procedure to $JOBUTIL. The parameter &PARM00. always represents your one to four character logon ID.

The &SAVEmm fields in the parameter part of the procedure point to fields in the parameter save data set $SMPnnnn (where nnnn is the logon ID) where the parameters you enter are saved from session to session. The two digits, mm, are used to index into the data set.

Note that multiple &PARMnn. fields between PARAMETER and END are sequential, beginning with $PARM01.

The following table lists the $SAVEmm fields, the procedure with which they are associated, and the utility or function invoked. When assigning values to the index digits (mm) in your procedure, start with 90 and work backwards to 61.

FIELD #	PROCEDURE	UTILITY/FUNCTION
$SAVE01-03	$SMP0201	$EDXASM
$SAVE04-06	$SMP0202	$S1ASM
$SAVE07-13	$SMP0203	$COBOL
$SAVE14-16	$SMP0204	$FORT
$SAVE17-18	$SMP0205	$LINK
$SAVE19-22	$SMP0206	$UPDATE
$SAVE23-24	$SMP0208	$PREFIND
$SAVE25-26	$SMP0308	$MOVEVOL
$SAVE27	$SMP0405	$FONT
$SAVE28	$SMP0501	$DIUTIL
$SAVE29	$SMP0502	$DICOMP
$SAVE30	$SMP0503	$DIINTR
$SAVE31-35	$SMP06	Execute application program
$SAVE36	$SMP0801	$BSCTRCE
$SAVE37	$SMP0806	$PRT2780
$SAVE38	$SMP0807	$PRT3780
$SAVE39	$SMP0808	$HCFUT1
$SAVE40-41	$SMP0208	$PREFIND
$SAVE42	$SMP0901	$TRAP
$SAVE43	$SMP0902	$DUMP
$SAVE44	$SMP0903	$LOG
$SAVE45-49	$SMP0210	$PLI
$SAVE50-60	Reserved	

```
PARAMETER
&PARM01,&SAVE01
&PARM02,&SAVE02
&PARM03,&SAVE03
END
LOG        OFF
REMARK    @ASSEMBLE &PARM01. TO &PARM02. USERID=&PARM00.
JOB        $SMP0201
PROGRAM    $EDXASM,ASMLIB
PARM       &PARM03.
DS         &PARM01.
DS         &SM1&PARM00.,EDX003
DS         &PARM02.
EXEC
EOJ
END
```

Figure 33. Invoking EDXASM

Parameters that have been saved are retrieved from the $SMPnnnn data set according to the relationships in the first part of the procedure. These parameters are displayed on the terminal. Then any parameters you enter from the terminal are used to update the procedure.

ALLOCATING AND DELETING WORK DATA SETS

The session manager allocates work data sets at logon time. They may be deleted at logoff time with one of the text editors. Two data sets, $SMALLOC and $SMDELET, are provided which are used in allocating and deleting data sets. $SMALLOC contains the data sets to be allocated and $SMDELET contains the data sets to be deleted. Figure 34 on page 223 lists the contents of $SMALLOC and Figure 35 on page 224 lists the contents of $SMDELET.

You may tailor the work data set allocations and deletions by modifying the $SMALLOC and $SMDELET data sets via the $FSEDIT utility. Modifications usually consists of changing the size or volume of a data set. However, you may allocate and delete up to four additional data sets. By moving the END terminator below $SM7 (statement 00120), you may allocate data sets $SM4, $SM5, $SM6, and $SM7. If you modify $SMALLOC, you should also modify $SMDELET to be consistent.

If the volume name of a work data set is to be changed within the $SMALLOC and $SMDELET data sets, then all of the Session Manager procedures which use the work datasets should be modified to reflect the change as well.

```
00010 $SMP   00     EDX003    NAME AND VOLUME FOR OPEN
00020 $SMP   30     EDX003    SIZE AND VOLUME TO ALLOCATE
00030 $SMW   30     EDX003    SIZE AND VOLUME TO ALLOCATE
00040 $SME   400    EDX003    SIZE AND VOLUME TO ALLOCATE
00050 $SM1   400    EDX003    SIZE AND VOLUME TO ALLOCATE
00060 $SM2   400    EDX003    SIZE AND VOLUME TO ALLOCATE
00070 $SM3   250    EDX003    SIZE AND VOLUME TO ALLOCATE
00080 END    *** TERMINATOR - END OF ALLOCATED DATA SETS
00090 $SM4   100    EDX003    SIZE AND VOLUME TO ALLOCATE
00100 $SM5   100    EDX003    SIZE AND VOLUME TO ALLOCATE
00110 $SM6   100    EDX003    SIZE AND VOLUME TO ALLOCATE
00120 $SM7   100    EDX003    SIZE AND VOLUME TO ALLOCATE
00130 ************************************************************
00140 ************************************************************
00150 ** $SMLOG WORK DATA SET PARAMETER VALUES FOR ALLOCATE **
00160 **  FUNCTION                                            **
00170 **    NOTE: THE DATA SETS $SMW AND $SMP MUST RESIDE ON **
00180 **          THE VOLUME EDX003.  ALL OTHERS MAY BE      **
00190 **          REASSIGNED.                                **
00200 **    NOTE: THE FIRST ENTRY IN THIS LIST IS USED TO    **
00210 **          TEST FOR THE EXISTENCE OF THE $SMP DATA    **
00260 ************************************************************
00270 ************************************************************
00230          END
00270 ************************************************************
```

Figure 34. $SMALLOC data set

```
00010 $SME  00    EDX003   PREFIX NAME AND VOLUME TO DELETE
00020 $SM1        EDX003   PREFIX NAME AND VOLUME TO DELETE
00030 $SM2        EDX003   PREFIX NAME AND VOLUME TO DELETE
00040 $SM3        EDX003   PREFIX NAME AND VOLUME TO DELETE
00050 $SMW        EDX003   PREFIX NAME AND VOLUME TO DELETE
00060 END   *** TERMINATOR - END OF DATA SETS TO BE DELETED
00070 $SM4        EDX003   PREFIX NAME AND VOLUME TO DELETE
00080 $SM5        EDX003   PREFIX NAME AND VOLUME TO DELETE
00090 $SM6        EDX003   PREFIX NAME AND VOLUME TO DELETE
00100 $SM7        EDX003   PREFIX NAME AND VOLUME TO DELETE
00110 ************************************************************
00120 ************************************************************
00130 **  $SMEND WORK DATA SET PARAMETER VALUES FOR DELETE  **
00140 **    FUNCTION                                        **
00160 ************************************************************
00170 ************************************************************
00180           END
00190 ************************************************************
```

Figure 35. $SMDELET data set

ADDING AN OPTION TO THE SESSION MANAGER

The session manager can invoke your programs in the following ways:

- Under primary option 6 (Execute Program Utilities Options)
- Through a menu that you have created, that is specified on the logon menu as an alternate to the primary option menu
- By adding a new option to an existing session manager menu

In the following example, a new option is added to the primary option menu. It is to be used to execute a hypothetical application program called PAYROLL. No parameters are required and it can be invoked directly from the primary option menu. The example illustrates:

- How to update the primary option menu $SMMPRIM, using the $IMAGE utility, to add a new option called PAYROLL
- How to update the associated procedure $SMPPRIM to include the new option, using the utility $FSEDIT
- How to build a new procedure called $SMPPAY, using the utilities $DISKUT1 and $FSEDIT, that will be submitted to $JOBUTIL to execute the program PAYROLL.

All of these steps will be done using the session manager.

The same procedure can be used to modify secondary options menus.

Updating the Primary Option Menu

The following steps add a new option, called PAYROLL, to the primary option menu:

- Step 1. IPL the Series/1 and load the session manager; enter your ID on the logon menu.
- Step 2. When the primary option menu is displayed, select option 4 -- the terminal utilities. When the next menu is displayed for the terminal utilities, it is the secondary option menu $SMM04.
- Step 3. Select option 4, the screen formatting utility, to invoke the utility $IMAGE. The utility gets control and prompts you for a command. The first command entered is to define a null character. It is entered as follows:

```
COMMAND(?): NULL @
```

- Step 4. Edit menu $SMMPRIM. After the next command prompt enter:

```
COMMAND(?): EDIT  $SMMPRIM,EDX002
```

  The primary option menu, $SMMPRIM, appears next on the terminal screen.

- Step 5. To update the menu, press the PF1 key; this causes the protected fields of menu $SMMPRIM to be displayed as non-protected, so that they can be redefined. The data fields now are represented by the null character, @, defined in Step 3. There must be eight null characters for option selection menus. Parameter selection menus may contains field of 1 to 64 characters.

```
$SMMPRIM: SESSION MANAGER PRIMARY OPTION MENU-----------
ENTER/SELECT PARAMETERS:                DEPRESS PF3 TO EXIT


       SELECT OPTION ==> @@@@@@@@

             1 - TEXT EDITING
             2 - PROGRAM PREPARATION
             3 - DATA MANAGEMENT UTILITIES
             4 - TERMINAL UTILITIES
             5 - GRAPHICS UTILITIES
             6 - EXEC PROGRAM/UTILITY
             7 - EXEC $JOBUTIL PROC
             8 - SENSOR/COMMUNICATION UTILITIES
             9 - DIAGNOSTICS AIDS
            10 - EXECUTE PAYROLL
```

Figure 36. Session manager primary option menu

- Step 6. Add the new PAYROLL option, option 10 - EXECUTE PAYROLL Press the ENTER key to display the newly-defined menu image (Figure 36).

- Step 7. Press the PF3 key to return to the $IMAGE command mode. In response to the command prompt, enter:

```
COMMAND(?): SAVE  $SMMPRIM,EDX002
```

The menu is saved and is ready to use. Terminate $IMAGE and the updated primary option menu will be displayed.

Updating the Procedure

The following steps update the procedure associated with the primary option menu:

- Step 1. Select option 1 (text editing) on the primary option menu and press the ENTER key. The utility, $FSEDIT, is loaded and control is passed to it. The next menu on the terminal screen is the primary option menu for $FSEDIT.

- Step 2. Select option 3 and press the ENTER key to read the procedure $SMPPRIM. Specify volume EDX002 for the VOLUME prompt, and $SMPPRIM for the data set prompt as shown in Figure 37 on page 227.

```
----------  $FSEDIT PRIMARY OPTION MENU  --------------
 SELECT OPTION ===> 3

1  BROWSE - DISPLAY DATASET
2  EDIT   - CREATE OR CHANGE DATASET
3  READ   - READ DATASET FROM HOST/NATIVE (H/N)
4  WRITE  - WRITE DATASET TO HOST/NATIVE (H/N)
5  SUBMIT - SUBMIT BATCH JOB TO HOST SYSTEM
6  LIST   - PRINT DATASET ON SYSTEM PRINTER
7  MERGE  - MERGE DATA FROM A SOURCE DATASET
8  END    - TERMINATE $FSEDIT
9  HELP   - DISPLAY TUTORIAL

------------------------------------------------------------

ENTER DATASET (NAME,VOLUME): $SMPPRIM,EDX002
```

Figure 37. Session manager $FSEDIT primary option menu

- Step 3. After the utility has read the procedure $SMPPRIM into your edit work data set, enter option 2 to update the procedure. At the bottom of the procedure, add the new option number and the name of the new procedure called $SMPPAY. Figure 38 on page 228 illustrates how the procedure should appear after the update has been made.

- Step 4. Return to the primary option menu by entering the word MENU in the command field. Then specify option 4 on the $FSEDIT primary option menu when it is displayed again on the screen. Respond YES to the prompt message which asks if you want the procedure written back to the same data set and volume. This places the updated procedure $SMPPRIM back on the disk volume EDX002.

- Step 5. Enter option 8 to terminate $FSEDIT and return to the primary option menu for the session by pressing ENTER.

OPTION	PROCEDURE/ MENU	DESCRIPTION
1	$SMP01	EXECUTE $FSEDIT
2	$SMM02	PROGRAM PREP SECONDARY OPTION MENU
2.1	$SMM0201	$EDXASM PARAMETER INPUT MENU
2.2	$SMM0202	$S1ASM PARAMETER INPUT MENU
2.3	$SMM0203	$COBOL PARAMETER INPUT MENU
2.4	$SMM0204	$FORT PARAMETER INPUT MENU
2.5	$SMM0205	$LINK PARAMETER INPUT MENU
2.6	$SMM0206	$UPDATE PARAMETER INPUT MENU
2.7	$SMM0207	$UPDATEH PARAMETER INPUT MENU
2.8	$SMM0208	$PREFIND PARAMETER INPUT MENU
2.9	$SMM0209	$EDXASM/$LINK/$UPDATE PARAMETER INPUT MENU
2.10	$SMM0210	$PL/I/$LINK/$UPDATE PARAMETER INPUT MENU
3	$SMM03	DATA MANAGEMENT UTILITIES SECONDARY OPTION MENU
3.1	$SMP0301	EXECUTE $DISKUT1
3.2	$SMP0302	EXECUTE $DISKUT2
3.3	$SMP0303	EXECUTE $COPYUT1
3.4	$SMP0304	EXECUTE $COMPRES
3.5	$SMP0305	EXECUTE $COPY
3.6	$SMP0306	EXECUTE $DASDI
3.7	$SMP0307	EXECUTE $INITDSK
3.8	$SMM0308	$MOVEVOL PARAMETER INPUT MENU
3.9	$SMP0309	EXECUTE $IAMUT1
3.10	$SMP0310	EXECUTE $TAPEUT1 (Version 2 only)
4	$SMM04	TERMINAL UTILITIES SECONDARY OPTION MENU
4.1	$SMP0401	EXECUTE $TERMUT1
4.2	$SMP0402	EXECUTE $TERMUT2
4.3	$SMP0403	EXECUTE $TERMUT3
4.4	$SMP0404	EXECUTE $IMAGE
4.5	$SMP0405	EXECUTE $FONT
4.6	$SMP0406	EXECUTE $PFMAP
5	$SMM05	GRAPHICS UTILITIES SECONDARY OPTION MENU
5.1	$SMM0501	$DIUTIL PARAMETER INPUT MENU
5.2	$SMM0502	$DICOMP PARAMETER INPUT MENU
5.3	$SMP0503	EXECUTE $DIINTR
6	$SMM06	SELECTED PROGRAM PARAMETER INPUT MENU
7	$SMM07	$JOBUTIL PARAMETER INPUT MENU

Figure 38. $SMPPRIM option menu with option 10 added (Part 1 of 2)

OPTION	PROCEDURE/ MENU	DESCRIPTION
8	$SMM08	COMMUNICATION UTILITIES OPTION MENU
8.1	$SMM0801	$BSCTRACE PARAMETER INPUT MENU
8.2	$SMP0802	EXECUTE $BSCUT1
8.3	$SMP0803	EXECUTE $BSCUT2
8.4	$SMP0804	EXECUTE $RJE2780
8.5	$SMP0805	EXECUTE $RJE3780
8.6	$SMM0806	$PRT2780 PARAMETER INPUT MENU
8.7	$SMM0807	$PRT3780 PARAMETER INPUT MENU
8.8	$SMM0808	$HCFUT1 PARAMETER INPUT MENU
9	$SMM09	DIAGNOSTICS SECONDARY OPTION MENU
9.1	$SMM0901	$TRAP PARAMETER INPUT MENU
9.2	$SMM0902	$DUMP PARAMETER INPUT MENU
9.3	$SMM0903	$LOG PARAMETER INPUT MENU
9.4	$SMM0904	EXECUTE $DISKUT2
9.5	$SMM0905	EXECUTE $IOTEST
10	$SMPPAY	EXECUTE APPLICATION PROGRAM PAYROLL

Figure 39. $SMPPRIM option menu with option 10 added (Part 2 of 2)

Building a $JOBUTIL Procedure

A procedure may be created that will be passed to the job stream processor utility to request execution of the program PAYROLL.

For this example, assume that the program PAYROLL already exists, requires no parameters, and resides on disk volume EDX002.

1. From the primary option menu, select both the primary and secondary options by entering 3.1. This invokes the disk utility $DISKUT1.

2. In response to the command prompts, enter the responses as shown in Figure 40 on page 230. This will allocate a data set called $SMPPAY to hold the new procedure.

```
JOB        $DISKUT1
USING VOLUME EDX002

COMMAND (?): AL $SMPPAY 10
DEFAULT TYPE = DATA - OK? Y
```

Figure 40. Job $DISKUT1

3. Enter EN for the next command prompt and the session manager's primary option menu is again displayed on the terminal screen.

4. Select option 1, text editing, to invoke the $FSEDIT utility.

5. Select option 2 and create the new procedure as shown in Figure 41.

```
PARAMETER
END
LOG        OFF
REMARK   @PAYROLL   USERID=&PARM00.
JOB        $SMPPAY
PROGRAM    PAYROLL,EDX002
EXEC
EOJ
END
```

Figure 41. Job $SMPPAY

6. When the procedure is completely entered, return to the $FSEDIT primary option menu by entering MENU on the command line.

7. Select option 4 and write the new procedure back to the disk volume EDX002.

8. Enter the same responses shown in Figure 37 on page 227 with the previous exception that the data set name specified is $SMPPAY.

9. When control returns, enter option 8 to exit $FSEDIT and return to the session manager.

Now the session manager is completely tailored. The new option can be specified on the primary option menu and the program PAYROLL will execute.

More sophisticated procedures can be built by copying existing session manager procedures and updating them with the $FSEDIT utility to invoke different programs and save parameters in unused fields in the data set $SMPnnnn (nnnn is your ID). The parameter data set can save up to 90 parameters (three per 256 byte block). The session manager uses the first sixty parameter locations, labeled &SAVE01 through &SAVE60. The remaining 30 parameter fields (&SAVE61 through &SAVE90) are available for your use. Many procedure formats are used by the session manager and these should provide you with some valuable guidelines for building your own procedures.

CHAPTER 11. TAPE ORGANIZATION

This chapter explains the types of label processing provided with the IBM 4969 Magnetic Tape Subsystem and the organization of tape data sets. Figures are included to show the layout, format, and content of labels supported.

The $TAPEUT1 utility allows you to allocate tape data sets, copy data sets from one medium to another, and change tape attributes. For detailed information on the $TAPEUT1 utility see Utilities, Operator Commands, Program Preparation, Messages and Codes.

For detailed information on Tape I/O instructions, see the Language Reference.

Note: Tapes are supported by Version 2 only.

EXTERNAL AND INTERNAL LABELS

When data records are stored on volumes of magnetic tape, those volumes must be identified (labeled) for future reference to the data. Two kinds of labeling can be used: external and internal.

External labels are attached to the outside of the tape to enable you to identify the tape visually.

Internal labels are written directly on the tape itself and are read the same way data records are read.

When a volume of tape is first received, it should be assigned a unique number, a **volume serial number**. The volume serial number is the identifying number for the volume. The volume serial number should be written on an external label, and recorded on an internal label. Other information contained in external and internal labels is determined by the files on the tape. This information usually changes from time to time.

In addition to the volume number, the external label could contain such items of identification as:

- file name
- file number
- file creation date

- number of volumes, if more than one volume is required (Even though EDX does not support multi-volume processing, an application programmer can implement a form of multi-volume processing.)

- department number

Internal labels usually contain the same information that is written on external labels; such as the volume serial number and the names of the files contained on the volume. Internal labels can be checked by the system when a tape is mounted, helping to ensure that the correct volume is being used for each application.

Control characters called **tapemarks** are used to separate files. The arrangement of label records, tapemarks, and data records is called the **volume layout**, which is discussed and illustrated in this chapter.

TYPES OF INTERNAL LABELS

Tape files may be either labeled or non-labeled. EDX supports a subset of DOS/VS labels. Tapes created on DOS may be used on EDX. Label processing may be bypassed, and non-standard, user, or ANSI labels are not supported.

Standard Labels

Standard labels are fixed length 80-character records. There are three types of label records:

- A volume label identifying the tape volume.
- A header label preceding the data records.
- A trailer label following the data records.

The first four characters of each label type identify the particular label record and its logical location.

Volume label	VOL1 at the beginning of the volume
Header label	HDR1 at the beginning of a file
Trailer label	EOF1 at the end of the file EOV1 at the end of a volume in a multivolume file

If two or more files are written on a single volume, each file has a header and trailer label. The volume label exists only once at the beginning of the volume.

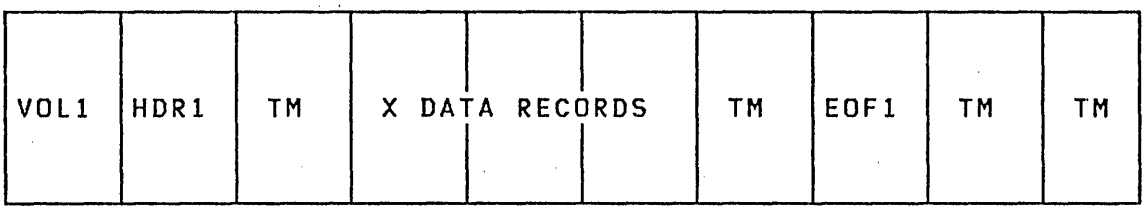
Provision is made in the IBM standard label set for additional standard volume labels (VOL2,VOL8) and additional standard file labels (HDR2 HDR8, EOF2 EOF8, and EOV2 EOV8). These labels are allowed but are not processed. User labels are also bypassed.

The Event Driven Executive system does not provide EOV processing. However, you may perform EOV processing. See "Processing the EOV Condition" on page 326 for a detailed example.

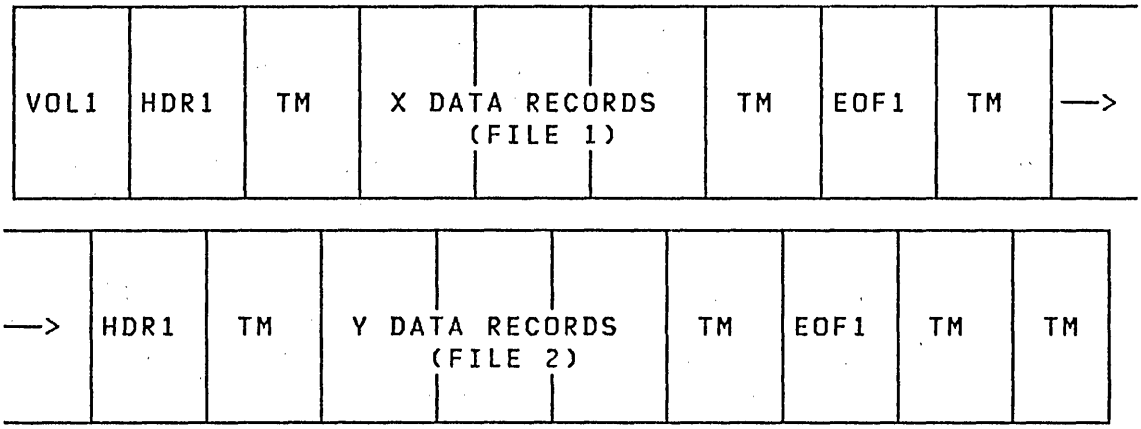
In general, the following factors apply to most volume layouts:

- When standard labels are written on a volume, the first record must be the volume label (VOL1).
- When labeled files are written, a header label (HDR1) precedes each file and a trailer label follows each file (EOF1 or EOV1).

Single File Volume



Multi-file Volume



Labeled Tape Processing

Labels are processed (checked or written) when files are opened or closed.

On output tapes expiration date checking is done if the system contains time-of-day support.

On input or output tapes, an error will occur during file search if:

- A non-standard label is found
- An non-labeled file is found
- A specified data set is not found

When a tape is varied online, the tape unit is defined for standard labels. The VOL1 is read and the tape is positioned at the HDR1 for the desired file number according to the following formula:

SPACE FILE FORWARD = (n-1)*3

where n is the specified file number

Note: If the file number being searched for is not on the volume mounted, the positioning of the tape is unpredictable.

Labeled Tape Label Fields

The following charts show the layout of the tape label record and the contents of each after the $TAPEUT1 utility has written the label (refer to Utilities, Operator Commands, Program Preparation, Messages and Codes for details on $TAPEUT1). These charts also show which fields are verified ($VARYON verifies VOL1, OPEN verifies HDR1) and initialized ($TAPEUT1) during standard label processing.

VOL1			
NAME	BYTES	INITIALIZED CONTENTS	VERIFIED
LABEL IDENTIFIER	3	'VOL'	YES
VOL LABEL NUMBER	1	'1'	YES
VOLUME SERIAL	6	'XXXXXX'	YES
VOLUME SECURITY	1	'0'	NO
DATA FILE DIRECTORY	10	BLANKS	NO
RESERVED	10	BLANKS	NO
RESERVED	10	BLANKS	NO
OWNER NAME	10	NAME	NO
RESERVED	29	BLANKS	NO

HDR1			
NAME	BYTES	INITIALIZED CONTENTS	VERIFIED
LABEL IDENTIFIER	3	'HDR'	YES
FILE LABEL NUMBER	1	'1'	YES
FILE IDENTIFIER (DSN)	17*	DATASET NAME (DSN)	YES
FILE SERIAL NUMBER	6	'XXXXXX'	NO
VOLUME SEQUENCE NUMBER	4	'0001'	NO
FILE SEQUENCE NUMBER	4	'00NN'	NO
GENERATION NUMBER	4	BLANKS	NO
GENERATION VERSION NUMBER	2	BLANKS	NO
CREATION DATE	6	YYDDD	NO
EXPIRATION DATE	6	YYDDD	YES
FILE SECURITY	1	'0'	NO
BLOCK COUNT	6	'000000'	NO
SYSTEM CODE	13	'IBMEDXS1'	NO
RESERVED	7	BLANKS	NO

EOF1			
NAME	BYTES	INITIALIZED CONTENTS	VERIFIED
LABEL IDENTIFIER	3	'EOF'	YES
FILE LABEL NUMBER	1	'1'	YES
FILE IDENTIFIER (DSN)	17*	DATASET NAME (DSN)	YES
FILE SERIAL NUMBER	6	'NNNNNN'	NO
VOLUME SEQUENCE NUMBER	4	'0001'	NO
FILE SEQUENCE NUMBER	4	'00NN'	NO
GENERATION NUMBER	4	BLANKS	NO
GENERATION VERSION NUMBER	2	BLANKS	NO
CREATION DATE	6	BLANKS	NO
EXPIRATION DATE	6	BLANKS	YES
FILE SECURITY	1	'0'	NO
BLOCK COUNT	6	'NNNNNN'	NO
SYSTEM CODE	13	'IBMEDXS1'	NO
RESERVED	7	BLANKS	NO

* EDX supports an eight byte non-blank DSN. The remaining nine bytes of the DSN field are not supported by EDX.

EOV1			
NAME	BYTES	INITIALIZED CONTENTS	VERIFIED
LABEL IDENTIFIER	3	'EOV'	YES
FILE LABEL NUMBER	1	'1'	YES
FILE IDENTIFIER (DSN)	17*	DATASET NAME (DSN)	NO
FILE SERIAL NUMBER	6	'NNNNNN'	NO
VOLUME SEQUENCE NUMBER	4	'0001'	NO
FILE SEQUENCE NUMBER	4	'00NN'	NO
GENERATION NUMBER	4	BLANKS	NO
GENERATION VERSION NUMBER	2	BLANKS	NO
CREATION DATE	6	BLANKS	NO
EXPIRATION DATE	6	BLANKS	NO
FILE SECURITY	1	'0'	NO
BLOCK COUNT	6	'NNNNNN'	NO
SYSTEM CODE	13	'IBMEDXS1'	NO
RESERVED	7	BLANKS	NO

* EDX supports an eight byte non-blank DSN. The remaining nine bytes of the DSN field are not supported by EDX.

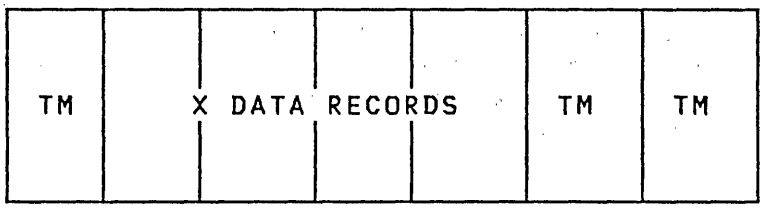
Non-Labeled Tapes

Non-labeled tape volumes consist of files separated by tapemarks. An additional tapemark may be placed at the beginning of the tape.

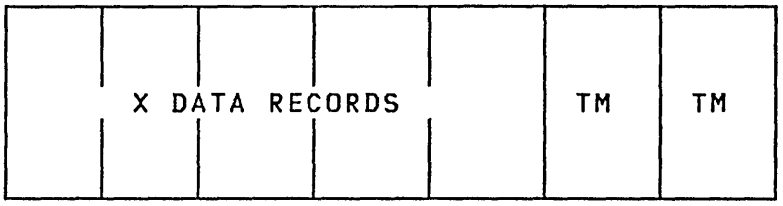
Non-labeled tapes allow the exchange of tapes with unknown configuration. The disadvantages of using non-labeled tapes are loss of control and the difficulty of maintaining data security. The following illustrations depict the non-labeled tape layouts that are supported.

Non-Labeled Tape Layouts

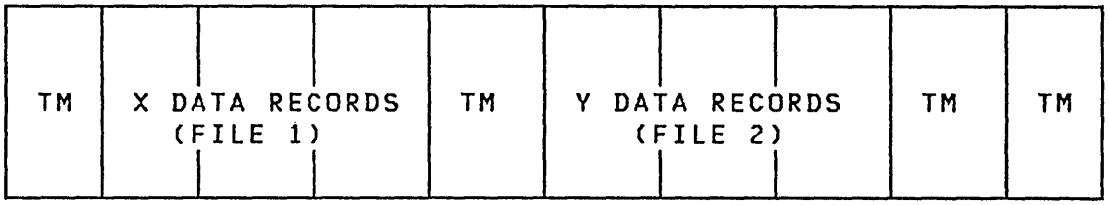
tapemark before the data records (one file)



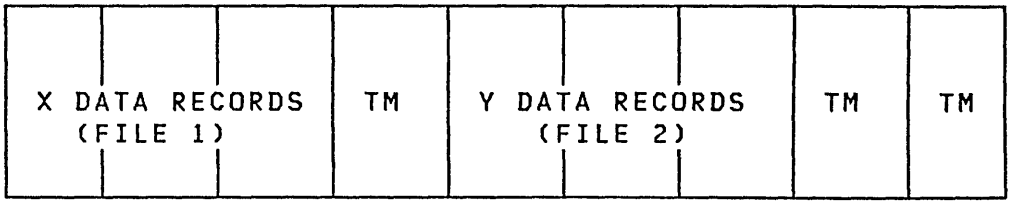
no tapemark before the data records (one file)



tapemark before the data records (multiple files)



no tapemark before the data records (multiple files)



Non-Labeled Tape Processing

The tape is checked to verify that it is not a standard labeled tape. The tape support positions the tape to the requested file sequence. If the volume starts with a tapemark, the tape is positioned after the tapemark. In all cases, the tape is positioned at the first data record in the file.

When a tape drive with the non-labeled attribute is varied online, the first record is examined. If the record is a VOL1, the $VARYON will fail. If the record is not a VOL1 or a tapemark, the tape is backspaced. If you specify a file number greater than 1, the tape is positioned according to the formula:

SPACE FILE FORWARD = n-1

where n is the specified file number.

Note: If the file number being searched for is not on the volume mounted, the positioning of the tape is unpredictable.

Bypass Label Processing (BLP)

Use the bypass label processing option when you want to access the tape but do not want the labels to be processed by EDX. When the tape unit is defined for BLP, no initial tape motion occurs. The application program must move the tape to the first data record. Recognizing and processing labels and tape marks is the responsibility of the application program. Event Driven Executive support simply transfers records; no other processing is performed.

When a tape unit defined for bypass label processing (BLP) is varied online, no tape motion occurs. If a file number greater than 1 is specified, the tape is positioned according to the formula:

SPACE FILE FORWARD = n-1

where n is the specified file number.

Note: If the file number being searched for is not on the volume mounted, the positioning of the tape is unpredictable.

For an example of BLP processing see "Sample Use of BLP to Access All Label Fields" on page 331.

TAPE RECORDS

Tape data sets may contain records of differing sizes, called variable length records and undefined length records. Application programs can write variable length records by specifing the size of each record on the WRITE statement and can read them by specifing the maximum size record on the READ statement. After the READ statement is complete, the TCB contains the actual size of the record.

The $TAPEUT1 utility also processes variable length records.

Variable Length Records

The first four bytes of each variable length record contains a two-byte hexadecimal value defining the actual length of the record followed by two bytes of zeros (reserved for future use). The length value includes the four- byte header. These records are written and read as described in "Tape Records." The application program can obtain the length of the record read from the record itself. You are responsible for placing the header value in each record when it is written.

Undefined Length Records

Each undefinied length record can be of a different size up to some maximum established by mutual agreement among the applications which must process the data. Processing is the same as described in "Tape Records" on page 244.

TAPE LOG ENTRIES

The error log entry for the 4969 tape device is similar to all other I/O device log entries except for seven words of dependent information:

- VOL1 - The VOL1 of the standard label tape or the ID for a no-label or BLP tape (3 words).
- ID - the ID, assigned by system generation, of the tape device (3 words).
- Flags - A one-word entry whose bits are defined as follows:

BIT	CONTENTS
0	Ignore expiration date (1 = Yes, 0 = no)
1	Must be zero
2	Data set expiration (1 = no, 0 = yes)
3	Output has occurred (1 = yes, 0 = no)
4-5	Density (defined by system generation): 00 = Undefined 01 = 800 BPI only 10 = 1600 BPI only 11 = Dual density
6-7	Label type: 00 = Undefined 01 = Standard label 10 = Bypass label processing 11 = No label
8-15	Logical file number that was open at the time of the error

CHAPTER 12. USING PARTITIONED DATA SETS

The display processor utility program uses a utility program, $PDS, to make partitioned data sets available for its use. Your programs can also use $PDS to access these members (for example, report data members and realtime data members). $PDS can also be used by your programs for all available functions.

Execution of $PDS by your program is through the use of the LOAD instruction. $PDS can be used as an overlay program as well as a normally loaded extension of an application program. $PDS allows you to:

- Open a member
- Allocate a member for a fixed number of records
- Allocate a member for the maximum number of records
- Release unused space from a member
- Delete a member
- Store the next record
- Store a record
- Fetch a record

Members can be created by these methods:

1. Use $DIUTIL program
    - Data member, member codes 4,5,6
    - User data members, member codes 7,8,9
    - User defined members, member codes 10 and up
    - Member codes 1,2,3 cannot be created by $DIUTIL
2. Use $DICOMP program
    - Report member, member code 1
    - Graphic member, member code 2
    - Graphic 3D member, member code 3
3. Use $PDS
    - All member types

Member codes are assigned as follows:

1	Report member
2	Graphic member
3	Graphic member 3D
4	Report data member
5	Plot curve data member
6	Realtime data member
7,8,9	User defined data members
10-n	User defined

Member types 1,2,3 store commands that will be used by $DIINTR to create a display. Member types 4,5,6 contain data that is saved by your application program. Member types 7,8,9 have the same format as member types 4,5,6 but are for use by application programs. Member types 10 and up are for use by applications programs.

Member types 4 through 9 are special members because they contain multiple records with a length of 1 to 32767 bytes. This feature allows the application program to Fetch and Store data by record number within a member. This technique could be used by an application program to update data members defined with the Display Utility Program Set.

Data Set Allocation

A data set that is to be used by $PDS must be allocated using $DISKUT1. Records should be allocated for the directory as well as members. Each record can contain eight directory entries except the first record which can contain seven. For example, if space is required for 40 members each with five records of space, you should allocate 206 records, 200 for members and six for the directory.

After a data set has been defined by $DISKUT1, it must be formatted for use by $PDS. $DIUTIL functions IN (Initialize), AL (Allocate), and BU (Build Data) are used for this purpose. $PDS can also be used to allocate members. Once members are allocated, they can be used by the application program. The $DIUTIL program is used to maintain of the data set.

Data Set Format

The data set to be used with $PDS consists of:

- Directory area
- Member area

Directory Area

The first entry in the directory describes the data set and has the following format:

Byte	Usage
0-1	Next available record number for member
2-3	Total size of data set in records
4-5	Number of next directory entry
6-7	Total available directory entries (allocated and unallocated)
8-15	Unused space

Each succeeding directory entry is 16 bytes with the following format:

Byte	Usage
0-7	EBCDIC member name
8-9	First record number (relative to start of data set)
10-11	Number of records in member
12-13	Member code
14-15	User code or clear screen indicator

Member Code (bytes 12-13)	
-1	Deleted member
0	Available space
1	Report member
2	Graphic member
3	Reserved
4	Report data member
5	Plot curve data member
6	Realtime data member
7-9	User defined data member
10-n	User defined members

Users code (bytes 14-15)
Defined by you and stored by $PDS allocate
or value of 1 if clear screen (ESC,FF) is not
to be sent on $DIINTR invocation.

$DIUTIL can be used to display this data for reference.

Member Area

Each member type has a unique format.

Member types 1-3	Display Control Member
No specific format is defined. The data is freeform, generated by the $DICOMP Utility Program. See Display Control Member format for information as to content of these members.	

Member Type 4	Report Data Member
Byte	Usage
0-7	Unused
8-9	Number of records
10-11	Record length in bytes (1-132)
12-13	Number of records available
14-15	Unused
16-n	Data Area

Member Type 5	Plot Curve Data Member
Byte	Usage
0-1	X Axis Range
2-3	Y Axis Range
4-5	X Base Line Value
6-7	Y Base Line Value
8-9	Number of records
10-11	Record length in bytes (1-32767)
12-13	Number of records available
14-15	Unused
16-n	Data Area

Note: Each record can be larger than 4 bytes, however relative bytes 0,1 must contain the X-coordinate value and bytes 2,3 must contain the Y-coordinate value.

Member Type 6	Realtime Data Member
Byte	Usage
0-7	Unused
8-9	Number of records
10-11	Record length in bytes (must be 16)
12-13	Number of records available
14-15	Unused
16-n	Data Area

Member Type 7,8,9	User Defined Data Member
Byte	Usage
0-7	Unused
8-9	Number of records
10-11	Record length in bytes (1-32767)
12-13	Number of records available
14-15	Unused
16-n	Data Area

Member type 10-n	User Defined Member

Display Control Member Format

Each of the display profile elements contained in the control members, type codes (1,2,3), is shown in this section. You may wish to use $PDS to access a control member. The application program could then generate a display profile command string and use $DIINTR to display the results. Following is the format of each of the display profile elements.

LB	Display Characters		
Byte	Bits	Value	Content
0	0-3	1	Display characters code
0	4-7	0	Unused
1	0-7	1-72	Number of characters to display
2-n	0-7	EBCDIC	EBCDIC data to display

MP Move Position

Byte	Bits	Value	Content
0	0-3	2	Move Position Code
0-1	4-7/0-7	0-1023	X Coordinate Value
2-3	0-7	0-1023	Y Coordinate Value

For 3D Members:

Byte	Bits	Value	Content
0	0-3	2	Move Position Code
0-1	4-15	0	Unused
2-3	0-15	-32768 - +32767	X Coordinate Value
4-5	0-15	-32768 - +32767	Y Coordinate Value
6-7	0-15	-32768 - +32767	Z Coordinate Value

LI Draw Line

Byte	Bits	Value	Content
0	0-3	3	Draw Line Code
0-1	4-7/0-7	0-1023	X Coordinate Value
2-3	0-7	0-1023	Y Coordinate Value

For 3D Members:

Byte	Bits	Value	Content
0	0-3	3	Move Position Code
0-1	4-15	0	Unused
2-3	0-15	-32768 - +32767	X Coordinate Value
4-5	0-15	-32768 - +32767	Y Coordinate Value
6-7	0-15	-32768 - +32767	Z Coordinate Value

DR Draw Symbol

Byte	Bits	Value	Content
0	0-3	4	Draw Symbol Code
0	4-7	1-15	Symbol ID
1	0-7	0-255	Symbol Modifier
2-3	0-7	0-32767	Users Symbol Number
		OR	
2	0-5	0	Unused
2	6	0-1	Start top (0) or bottom (1) for Arc
2-3	7/0-7	0-508	# of Y units in Arc

VA Display Variable

Byte	Bits	Value	Content
0	0-3	5	Display Variable Code
0	4-7	0-7	Word Number within record
1	0-3	0-15	Function Code
1	4-7	0-3	Type Code
2-3	0-7	1-32767	Record number in Realtime Data Member
4	0-7	1-40	Field Width
5	0-7	0-39	Number of Decimals

JP Jump

Byte	Bits	Value	Content
0	0-3	6	Jump Code
0	4-7	0-7	Word number within record
1	0-7	0-2	Jump Modifier 0=Unconditional 1=Zero 2=Non Zero
2-3	0-7	1-32767	Record number in Realtime Data Member
4-5	0-7	0-32767	Jump to Address (offset in Words from beginning of Control Member)

DI Direct Output to Another Device

Byte	Bits	Value	Content
0	0-3	8	Direct Output Code
0	4-7	0	Unused
1	0-7	0	Unused
2-9	0-7	EBCDIC	8 character name of output device. Refer to ENQT Instruction

PC Plot Curve from Plot Curve Data Member

Byte	Bits	Value	Content
0	0-3	9	Plot Curve Code
0	4-7	0	Unused
1	0-7	0 or EBCDIC	EBCDIC character for plot if character plot
2-9	0-7	EBCDIC	8 character member name of a plot data member

** Display Report Line Items

Byte	Bits	Value	Content
0	0-3	10	Display Report Line Items
0	4-7	0	Unused
1	0-7	0	Unused
2-9	0-7	EBCDIC	8 character member name of a report data member

AD Advance X,Y

Byte	Bits	Value	Content
0	0-3	11	Advance X,Y code
0-1	4-7/0-7	0-1023	X advance value (adjusted by +512)
2-3	0-7	0-1023	Y advance value (adjusted by +512)

For 3D Members:

Byte	Bits	Value	Content
0	0-3	11	Advance X,Y,Z Code
0-1	4-7/0-7	0-1023	X Advance Value (adjusted by +512)
2-3	0-7	0-1023	Y Advance Value (adjusted by +512)
4-5	0-7	0-1023	Z Advance Value (adjusted by +512)

IM Insert Member

Byte	Bits	Value	Content
0	0-3	12	Insert Member Code
0	4-7	0	Unused
1	0-7	0	Unused
2-9	0-7	EBCDIC	8 character member name of a central member

LR Draw Line Relative

Byte	Bits	Value	Content
0	0-3	13	Draw Line relative code
0-1	4-7/0-7	0-1023	Delta X Value (adjusted by +512)
2-3	0-7	0-1023	Delta Y Value (adjusted by +512)

For 3D Members:

Byte	Bits	Value	Content
0	0-3	13	Draw Line Relative Code
0-1	4-7/0-7	0-1023	Delta X Value (adjusted by +512)
2-3	0-7	0-1023	Delta Y Value (adjusted by +512)
4-5	0-7	0-1023	Delta Z Value (adjusted by +512)

RT Change Realtime Data Member Name			
Byte	Bits	Value	Content
0	0-3	14	Change Realtime Data Member Code
0	4-7	0	Unused
1	0-7	0	Unused
2-9	0-7	EBCDIC	8 character member name of a new realtime data member (for VA and +P codes)

TD Display Time and Data			
Byte	Bits	Value	Content
0	0-3	15	Display time and data code
0	4-7	0	Unused
1	0-7	0	Unused

Using $PDS in Your Program

Access to the $PDS utility is through a LOAD instruction in your application program. The following example shows how to open a member.

```
XYZ       PROGRAM    START, DS=(??)
              .
              .
              .
START     EQU        *
              .
              .
              .
          READTEXT   #MCB,'ENTER MEMBER NAME@'.
              .
              .
              .
          LOAD       $PDS,$MCB,DS=(DS1),EVENT=#PDS,     C
                     LOGMSG=NO
              .
              .
              .
          WAIT       #PDS
          IF         (#PDS,NE,-1),GOTO,ERROR
              .
              .
              .
 *  NORMAL PROCESSING OF OPENED MEMBER  *
              .
              .
              .
          READ       MBR,BUFF
              .
              .
              .
          WRITE      MBR,BUFF
              .
              .
              .
          PROGSTOP
              .
              .
              .
BUFF      DATA       128F'0'        DISK I/O BUFFER
$MCB      DATA       A(#MCB)        POINTER TO MEMBER CONTROL
                                    BLOCK
              .
              .
              .
#MCB      TEXT       LENGTH=8       MEMBER NAME
#MCBCMD   DATA       F'1'           $PDS COMMAND(OPEN)
#MCBDSA   DATA       A(MBR)         ADDRESS OF DSCB
```

```
#MCBDT0  DATA        F'0'              Data Field 0
#MCBDT1  DATA        F'0'              Data Field 1
#MCBDT2  DATA        F'0'              Data Field 2
#MCBDT3  DATA        F'0'              Data Field 3
            .
            .
            .
         DSCB        DS#=MBR,DSNAME=DUMMY,VOLSER=DUMMY
            .
            .
            .
         ENDPROG
         END
```

Member Control Block

The member control block that is passed to the $PDS utility program by the use of the PARM facility is a 20 byte field. This member control block (MCB) is filled in by your application program. The format of the MCB is as follows:

Byte	Usage
0-7	EBCDIC Member Name
8-9	$PDS Command (see below)
10-11	Address of Callers DSCB
12-19	Data field 0 through 3 (see below)

$PDS Commands (bytes 8-9)	
Command	Function
1	Open Member
2	Allocate Member
3	Allocate Member (Maximum Space)
4	Release Space
5	Delete Member
6	Store Next Record
7	Store Record
8	Fetch Record

Open Member

The member specified in bytes 0-7 of the MCB is located and the DSCB specified in bytes 10-11 is filled in to point to the member.

Allocate Member

The member specified in bytes 0-7 of the MCB is dynamically allocated with the parameter specified in bytes 14-19.

Allocate Member (maximum space)

The member specified in bytes 0-7 of the MCB is dynamically allocated with the parameter specified in bytes 16-19. Maximum space is allocated.

Release Space

The member specified in bytes 0-7 of the MCB is located and unused space is returned to the available space in the data set. Bytes 14-15 must contain the number of records that the member will contain.

Delete Member

The member specified in bytes 0-7 of the MCB is located and marked for deletion. Note: the space occupied by the deleted member is NOT returned to the available space in the data set. Use the utility $DIUTIL to reclaim deleted space.

Store Next Record

The member specified in bytes 0-7 of the MCB is located. The member header is used to determine which record is next and data is stored in that record. Your data buffer address is located in bytes 14-15 of the MCB.

Store Record

The member specified in bytes 0-7 of the MCB is located. The record specified in bytes 12-13 is located and the data is stored in that record. Your data buffer address is located in bytes 14-15 of the MCB.

Fetch Record

The member specified in bytes 0-7 of the MCB is

located. The record specified in bytes 12-13 is located. All the data is retrieved and stored in your data buffer. The data buffer address is located in bytes 14-15 of the MCB.

Data Field 0 through Data Field 3 must contain modifier information for the various $PDS commands. Also, these areas contain data following the action taken by the $PDS program. The following table shows the data required prior to the execution of $PDS and the data returned following the execution of $PDS.

Before Execution of $PDS:				
	DATA WORDS			
COMMAND	0	1	2	3
1-Open	N/A	N/A	N/A	N/A
2-Allocate	N/A	Records	Member Type Code	User Code
3-Allocate Max	N/A	N/A	Member Type Code	User Code
4-Release	N/A	Records	N/A	N/A
5-Delete	N/A	N/A	N/A	N/A
6-Store Next	N/A	Data Buffer Address	N/A	N/A
7-Store	Record	Data Buffer Address	N/A	N/A
8-Fetch	Record	Data Buffer Address	N/A	N/A

Note: N/A = Not Applicable

After Execution of $PDS:				
	DATA WORDS			
COMMAND	0	1	2	3
1-Open	1st Record	Records	Member Type Code	User Code
2-Allocate	1st Record	Records	Member Type Code	User Code
3-Allocate Max	1st Record	Records	Member Type Code	User Code
4-Release	N/A	N/A	N/A	N/A
5-Delete	N/A	N/A	N/A	N/A
6-Store Next	Record	Data Buffer Address	Records In Member	N/A
7-Store	Record	Data Buffer Address	Records In Member	N/A
8-Fetch	Record	Data Buffer	Records	N/A

Note: N/A = Not Applicable

THE SOFTWARE TRACE TABLE

An analysis of the software trace table is the first step in the determination of a software problem. This table contains program check, soft exception, and machine check trace data, and offers information on the types of errors that have been occurring. This information may lead to more efficient utilization of other problem determination aids such as a $TRAP, $DUMP, $DEBUG, $D, and the hardware error log. The structure of this table is seen in Figure 42 on page 267.

When an errors occurs, an entry is placed into the next available trace table slot. This entry includes a state variable, the TCB address, the Processor Status Word (PSW), the Storage Address Register (SAR), and the Level Status Block (LSB). After the table is filled, the oldest entry is overlaid by the new entry. The second word of the control table points to the oldest entry. Hence the newest entry precedes the oldest.

Software trace table support is included at system generation time by adding an INCLUDE CIRCBUFF entry to the link control data set ($LNKCNTL). The default trace table has room for eight fifteen word entries. This number may be modified if you:

- Alter the replication factor on the CIRESTR variable (in CIRCBUFF source module) by a multiple of the entry size to reflect the new trace entry space desired
- Reassemble CIRCBUFF module
- Relink the supervisor

The $D operator command may be used to obtain a dump of the software trace table information. Upon issuing the $D Command, you will be prompted for:

1. Origin. ENTER 0.
2. The beginning address of the trace table. Enter the CIRCBUFF entry point from the system generation link listing.
3. Number of addresses to be displayed. This value is:

(10 + (30n))/2

Where n equals the number of entry spaces that you have allowed for. The default for n equals 8. Therefore, the default number of addresses equals:

(10 + (30x8))/2 = 250/2 = 125

Example: The first entry in the trace table storage dump below reflects the following program check information as displayed on the console:

```
PGM CHECK:
 PLP  TCB  PSW  LSB
 6B00 0138 8002 1E6A 0000 88D0 6C30 6B7E 6C38 6C31 6C32 005C 00B8 00

> $D
ENTER ORIGIN:  0
ENTER ADDRESS,COUNT:  62EE,125
 62EE: 62F8 6370 63E8 0004 001E 0100 0138 8002
 62FE: 6C31 1E6A 0000 88D0 6C30 6B7E 6C38 6C31
 630E: 6C32 005C 00B8 0000 0100 0138 8002 6C31
 631E: 1E6A 0000 88D0 6C30 6B7E 6C38 6C31 6C32
 632E: 005C 00B8 0000 0100 0138 8002 6C31 1E6A
 633E: 0000 88D0 6C30 6B7E 6C38 6C31 6C32 005C

 634E: 00B8 0000 0100 0138 8002 6C31 1E6A 0000
 635E: 88D0 6C30 6B7E 6C38 6C31 6C32 005C 00B8
 636E: 0000 0000 0000 0000 0000 0000 0000 0000
 637E: 0000 0000 0000 0000 0000 0000 0000 0000
 638E: 0000 0000 0000 0000 0000 0000 0000 0000
 639E: 0000 0000 0000 0000 0000 0000 0000 0000
 63AE: 0000 0000 0000 0000 0000 0000 0000 0000
 63BE: 0000 0000 0000 0000 0000 0000 0000 0000
 63CE: 0000 0000 0000 0000 0000 0000 0000 0000
 63DE: 0000 0000 0000 0000 0000
ANOTHER DISPLAY?
```

ADDRESS	CONTENTS	VALUE	DESCRIPTION
62EE through 62F5	Control table	62F8	Address of first entry
		6370	Address of next entry
		63E8	Address of end of table
		0004	Use count
		001E	Size of each entry
62F8 through 6315	A trace entry	01	State variable
		00	Address key
		0138	TCB address
		8002	PSW
		6C31	SAR
		1E6A	IAR
		0000	AKR
		88D0	LSR
		6C30	GPR0
		6B7E	GPR1
		6C38	GPR2
		6C31	GPR3
		6C32	GPR4
		005C	GPR5
		00B8	GPR6
		0000	GPR7
6316 through 6333	A trace entry		
6334 through 6651	A trace entry		
6352 through 636F	A trace entry		

* STATE VARIABLE VALUES:
0 = no interupt in process
1 = standard (default) processing
2 = now processing task error exit
3 = undefined

Figure 42. Software Trace Table Structure

Check and Trap Handling - An Overview

During the execution of a task, exception conditions may arise either in the task itself or in the Series/1 hardware. These conditions are known as machine checks (indicating a hardware malfunction such as a storage parity check), program checks (indicating a software malfunction such as a specification check), and soft exception traps (indicating an unusual, but generally recoverable, software condition such as a floating-point exception). When any of these exceptions occurs, the processor saves its state in an area of storage set aside for that purpose by the supervisor and gives control to the supervisor through special entry points, one for each type of exception. For details of the hardware handling of exceptions, see the processor description manuals.

The major goal of the supervisor in handling exceptions is to keep the system running. The usual response to program and machine check exceptions is to cancel the program that the offending task is a part of. The response for soft exception traps is to resume processing at the instruction following the one that caused the exception.

The supervisor approach to exception handling, while appropriate for a large number of programs, is sometimes inadequate. For example, if your task shares resources (e.g. QCBs or ECBs) with a task in another program, when an error is encountered the supervisor will not release the resources. If your task were unable to continue because of the error, it may be necessary to release the resources and inform the other task of the error.

Whenever an exception occurs in a task with a task error exit, the supervisor, instead of processing the exception in the usual way, stores the hardware status at the time of the exception in a block of storage designated by the task. It then passes control to the task at its task error exit entry point. The task's error routine is then in control and must interrogate the stored hardware status to make an appropriate response.

Note that a task error exit routine is a part of the task it serves - the supervisor passes control to it at the task level; it is not a subroutine of the supervisor's error handler. There are important ramifications of this fact. The registers (including the EDL software registers, #1 and #2) used by the error exit routine are those normally used by the task. To resume executing the task following corrective action by task error exit, simply branch (if in Series/1 instruction mode) or GOTO (if in EDL mode) the appropriate location. If the error exit is unable to recover from the exception, it should issue a

PROGSTOP instruction.

Using the Task Error Exit Facility in Your Task

Linkage Conventions

To make use of the Task Error Exit facility in your task, you must code a small control block and the error exit routine. In addition, you must set aside the block of storage that will be filled with the hardware status when an exception occurs.

The control block, called the task error exit control block (TEECB), provides the linkage between the supervisor and your error exit. The TEECB must be aligned on a fullword boundary.

To allow the supervisor to find your TEECB, you should code its address as the value of the ERRXIT keyword parameter of the PROGRAM or TASK EDL statement that defines your task.

The format of the TEECB is as follows:

```
TEECB    DS  0F          TASK ERROR EXIT CONTROL BLOCK
TEECTL   DC  X'0002'      ------------------------
TEESIA   DC  A(XITRTN)   |_____0_____|_____2_____|
TEEHSA   DC  A(HSA)      |___________SIA___________|
                         |___________HSA___________|
```

In the first word (TEECTL), bits 0 - 7 are reserved and must be zero. Bits 8-15 state the number of data words that follow. This value must be two. The second word (TEESIA) contains the address of the starting instruction of your Error Exit routine. The last word (TEEHSA) contains the address of the block of storage you have reserved to receive the hardware status when an exception occurs. This block is called the Hardware Status Area (HSA) and is 24 bytes long.

The format of the HSA is:

```
*       HARDWARE STATUS AREA
HSA       DS      0F     ALIGN ON FULL WORD BOUNDARY
HSAPSW    DS      1F     PROGRAM STATUS WORD
HSALSB    EQU     *      LEVEL STATUS BLOCK
HSAAKR    DS      1F     ADDRESS KEY REGISTER
HSAIAR    DS      1F     INSTRUCTION ADDRESS REGISTER
HSALSR    DS      1F     LEVEL STATUS REGISTER
HSAREGS   DS      8F     GENERAL REGISTERS 0 - 7
```

The contents of the various HSA locations (PSW,AKR,Etc,) will contain, at entry to your error exit routine, the values that were in the corresponding hardware registers at the time of the

exception. Upon entry to your error routine, general registers 1 and 2 will have been set to the SIA of your routine and to the address of your task's TCB, respectively.

Since entry to your error exit routine is made at the Event Driven Language level, the contents of the remaining general registers is dependent upon what instructions are executed.

What Happens When an Exception Occurs

If an exception (machine check, program check or soft exception trap) occurs during the execution of your task and you have specified a task error exit, as outlined above, the supervisor locates your TEECB. It then uses the TEEHSA pointer to locate your HSA and stores the hardware status information in it. Next, it retrieves the TEESIA pointer and sets it to zero to prevent recursive exceptions. Finally, it starts your task at the address it retrieved if that address is non-zero. If the TEESIA is zero or an exception occurs during any of this processing (if, for example, the TEECB is invalid), the supervisor treats the error as though no task Error Exit had been specified. Note that even if the TEESIA is zero, the supervisor still attempts to store the hardware status.

Since the supervisor zeroes TEESIA prior to starting your task, your error exit routine only gets control on the first exception that occurs, unless your routine restores TEESIA to its original condition. Zeroing TEESIA allows the supervisor to handle exceptions that occur in error exit routines, thus preventing recursion in the error handling process. When you implement a task error exit, do not restore TEESIA until the error exit routine has completed.

I/O ERROR LOGGING

The Event Driven Executive provides the capability to record device I/O errors into a log data set on disk or diskette and to display the log data set. The support is provided with a set of utilities and subroutines.

Recording the Errors

To activate I/O logging, the utility $LOG is loaded into any partition. The logging function can be deactivated, reactivated, and terminated after it becomes active.

The logging function is performed in three steps. First, the device handler gathers all the pertinent information required by the LOG interface. Among the required parameters is the address of the log data. This parameter points to the device data block (DDB) containing the IDCB, DCBs, and device status to be logged. Optionally, you may provide the default log record format or a control block containing all the necessary information. Second, control is passed to the logging subroutines which build a log record by copying the DDB and other parameters into a buffer. The log record is then placed on the log queue, and control is returned to the caller after the logging utility is posted. Last, the utility $LOG, having been posted, receives the log entry from the queue and writes it to the log data set.

The I/O error logging function can be called by both system and application programs. If a program requires logging but is not link- edited with the system, then the modules $DEVLOG and $$RETURN must be link- edited with the program. There are two interfaces to perform logging. The LOG macro is provided for programs that are assembled with $S1ASM. A similar interface using the USER instruction is provided for use with Event Driven Language and is not available to programs link-edited with the system. The interfaces available for performing I/O error logging are described below.

The LOG Macro

The LOG macro is used to generate a BAL instruction (using R7) to the routines that perform the logging function. The macro also expands a parameter list which can be modified at execution time by using a set of equates. The equates are generated by coding the LOG macro with the TYPE=EQUATES parameter specified. Before a call is made to the LOG function from a task, the OP1 key must have the storage key value of the TCB and the OP2 key must be equal to the ISK key. If the LOG function is called from as the result of an interrupt, then the OP2 key must have a zero value. The caller's registers, except R7, are restored when control is returned to the calling program.

Syntax

```
Label LOG  logtype,datatype,dataaddr,
           datakey,devaddr,iib,intcc,
           REQTYPE=,TYPE=,RETURN=,P1=,P2=,
           P3=,P4=,P5=,P6=,P7=

Required:  logtype,datatype,dataaddr,datakey,devaddr
Defaults:  None
Indexable: None
```

Operands	Description
logtype	Indicator of the type of log record. Use one of the following values: • $SE - Soft error (device recoverable) • $HE - Hard (unrecoverable) error • $RE - Software recoverable error P1 names a byte field which can be modified at execution time using the above values.
datatype	Indicator of the type of control block data being logged. The following values are used by the Event Driven Executive supervisor. • $LTERM - For terminal control block • $49789 - For terminal control block • $49734 - For terminal control block • $LDISK - For disk and diskette control block • $LDISK1 - For disk and diskette control block • $L4969 - For tape control block (Version 2.0 only) • $LBSC - For BSC control block • $LDFLT - For the default log data format P2 names a byte field which can be modified at execution time using the above values. The values 128 to 255 are reserved for your control blocks.

dataaddr The address of the log data. P3 names a two-byte field which can be modified at execution time.

datakey The address space key (0-7) of the log data address. P4 names a one-byte field which can be modified at execution time.

devaddr The device address. P5 names a byte field which can be modified at execution time.

iib The interrupt information byte. This is an optional parameter which should be specified only if the IIB is not found in the log data. P6 names a byte field which can be modified at execution time.

intcc The condition code presented by the hardware when the I/O interrupt was accepted. This is an optional parameter which should be specified only if the condition code is not found in the log data. P7 names a byte field which can be modified at execution time.

REQTYPE= The caller's execution state

- IA indicates that the call was made as the result of an interrupt.
- TASK indicates that the call was made from a task.

TYPE= Specify EQUATES to indicate that only the list of equates for setting the various parameter values is provided. Specify DSECT to generate both the equates and a DSECT.

RETURN= Specify a label which is assigned to the location in the log parameter list containing the return address. The two-byte return address is initialized with the address of the location following the parameter list. The return address may be modified at execution time by moving the desired return address into the location addressed by the label specified for this parameter.

Passing Control to the Log Routines

To allow the log function to be called from the Event Driven Language, the USER instruction can be used to pass control to the log routines. In the calling sequence you must pass certain parameters which are similar to those specified in the LOG macro. When a specific equate value must be coded for a parameter, generate the equates by coding COPY LOGEQU or LOG

TYPE=EQUATES. As with the LOG macro, the parameters may be modified at execution time. Note that all parameters are two-byte fields, unlike the parameters in the LOG macro.

```
USER  $USRLOG,PARM=(logtype,datatype,                C
      dataaddr,datakey,devaddr)
```

Operands	Description
logtype	Indicator of the type of log record. Use one of the following values: 1. $SE - Soft error (device recoverable) 2. $HE - Hard (unrecoverable) error 3. $RE - Software recoverable error
datatype	Indicator of the type of control block data being logged. The following values are used by the Event Driven Executive. • $LTERM - For terminal control block • $LDISK - For disk and diskette control block • $LHSTCOM - For HOSTCOMM control block • $LBSC - For BSC control block • $LTIMER - For timer control block • $LPI - For PI control block • $LDI - For digital input control block • $LDO - For digital output control block • $4969 - For tape control block (Version 2.0 only)
dataaddr	The address of the log data.
datakey	The address space key of the log data address.
devaddr	The device address.

In order to make the invocation of the log function reentrant, you may need to disable the system while your program is modifying the parameter list. Note that the logging routine disables the system for a small period of time immediately after logging is invoked. Control is passed back to the caller with the system enabled.

Printing the Errors

The I/O error log data set contains unformatted log records which the system I/O device handlers, through $LOG, have written to the log data set. These unformatted data records are interpreted, formatted, and printed by the $DISKUT2 utility. $DISKUT2 processes the log records of the following I/O devices:

- 4962 and 4963 disks
- 4964 and 4966 diskettes
- 4973 and 4974 printers
- 4978 and 4979 displays
- BSCA
- 4969 tapes (Version 2.0 only)

$DISKUT2, upon recognition of the LIST LOG command, reads the log data set control record and verifies that the log data set is valid. Using pointers in the control record, it determines whether the data set has wrapped around and the number of records in the log.

The first record of the log data set is a control record which contains the device table and initialization data. It also contains the size of the log data set and the number of the next record to be written. Initially, the next record to be written is set to "3". If the size of the data set is ten records and the tenth record has been written, the next record to be written is set to "3" and the data set is said to have "wrapped around". When this occurs, a message is printed.

The second record is a set of 256 one-byte counters (one byte per device address). Each time an error occurs, the appropriate counter is incremented. For example, an error at device address 4 causes byte 4 to be incremented. The counters cannot be incremented past hex '7F'.

Each record contains a device address and a "pseudo" device type at a fixed location. Depending upon the options you select, the device address determines **whether** to process the

record, and the pseudo device type determines how to process the record.

Pseudo device types X'01' thru X'7E' are reserved for system I/O devices. If any of these values are present, $DISKUT2 will load the overlay program $LOGUT00 which interprets the log record and moves critical information into a format buffer. $LOGUT00 then returns to $DISKUT2 which prints the information from the format buffer. Refer to Utilities, Operator Commands, Program Preparation, Messages and Codes for further information on using DISKUT2.

Log records with a pseudo types of X'7F' contain already formatted log data; no overlay program is called. The data is printed directly from the log record.

Log records with pseudo types X'80' thru X'FF' are is considered alien device log records. You must supply an overlay program to format alien device log record into printable format. The overlay program must be located in the IPL volume and must begin with the characters "$LOGUT". The last two characters are located through the device table (bytes 2 and 3 of the entry). Note that the pseudo type is used only to look up the table entry and need not be the same as the overlay name suffix.

The Device Table

The log control record (the first record of the log data set) contains a table of up to 30 entries (8 bytes per entry). This table must be patched with the alien device pseudo type, the size of the DDB (data block to be logged), and the two-character overlay program suffix.

The device table is provided so that devices other than system devices (such as program products and RPQs) can be supported by error logging with no change to the supervisor. The device handler logs a pseudo type of X'80' through X'FF' and patches this information into the device table.

Note that it is the task which initializes the $LOG data set that should be patched, and not the log data set. The procedure for patching is as follows:

1. Dump the first two records of $LOG (512 bytes).

2. Locate the characters "DEVICE TABLE". Immediately following these characters are 240 bytes of the alien device table.

3. Patch in the above data as required.

The format of an entry is as follows:

```
byte 0 = pseudo type (X'80' thru X'FF')
     1 = data block size in bytes
   2-3 = overlay suffix characters.
   4-7 = reserved - do not use.
```

The pseudo type is the indicator of the type of control block being logged (the datatype parameter on the LOG macro).

The next IPL of the system causes the log data set to be updated with the new device table entries.

Display Program

$DISKUT2 uses the overlay suffix in the device table to load the overlay program to format the log record. If the program cannot be loaded, the 256-byte record is dumped in hexadecimal. If an error is returned by the overlay program, the record is dumped in hexadecimal.

A check is made to see if the overlay program requested is already in storage. If it is not in storage, a LOAD instruction is executed to bring it into storage. A WAIT for the end event of the loaded program is then executed. The overlay program should not issue a PROGSTOP. The overlay program should end with a DETACH followed by a branch to the first executable instruction of the overlay program.

If the required overlay program is already in storage, its main task is attached and a WAIT is issued for the task end event. This technique minimizes disk access. Since a fresh copy of the overlay program is not loaded for every invocation, the overlay program must be reusable, making it important to initialize data areas upon entry.

A suggested sequence follows:

```
PGM      PROGRAM  INIT,300,PARMS=3
INIT     EQU      *
         USER     GO
         DETACH
         GOTO     INIT
GO       EQU      *
         MVW      R1,SAVER1      * SAVE R1
         MVW      $PARM1,R1      * A(RAW BUFFER)
         MVW      $PARM2,R3      * A(FORMAT)
          .
          .
          .
*         PROCESS DEVICE DEPENDENT DATA
          .
          .
          .
         MVW     $PARM3,R1       * A(STATUS)
         MVWI    0,(R1)          * SET STATUS OK
         MVW     SAVER1,R1       * RESTORE R1
         B       RETURN          * RETURN TO DETACH
SAVER1   DC       A(*-*)
         ENDPROG
         END
```

LOG Macro Equates

Coding LOG TYPE=ALL provides a listing of all equated symbols associated with logging. Symbols beginning with $LGC define the control record, and symbols beginning with $LOG define the log record. $LOGEQU through $LOGDDBA are placed into the log record by $LOG and are moved to the formatted record by $DISKUT2 before calling the overlay program. The overlay program must fill in the data defined by $LOGDDB through $LOGDEP.

CHAPTER 14. INTER-PROGRAM COMMUNICATIONS

A program may communicate with a terminal operator or with another program. There are several ways to communicate:

- Passing parameters, using the LOAD instruction.
- Communicating with a terminal operator.
- Communicating with another program.

For a description of the LOAD instruction, refer to the Language Reference.

To communicate with a terminal operator, use the PRINTEXT and READTEXT instructions. For a description of the PRINTEXT and READTEXT instructions, refer to the Language Reference.

To communicate with another program, you may use either:

- Virtual terminal support in conjunction with the PRINTEXT and READTEXT instructions.
- Cross partition services.

Virtual terminal support uses the PRINTEXT/READTEXT instructions to allow programs to communicate with each other. It requires two TERMINAL configuration statements and the supervisor module IOSVIRT. Refer to "Appendix A. Storage Estimating" on page 333 to estimate the storage required. Virtual terminal support provides synchronization logic.

Cross partition services are general purpose services that require synchronization logic in your programs but no additional storage in your supervisor.

Communication is possible between two programs within the same partition and between programs in different partitions.

VIRTUAL TERMINALS

The virtual terminal mechanism allows programs to communicate across partitions as well as within the same partition.

Creating a Virtual Channel

In the discussion of the TERMINAL statement in Chapter 2, it was noted that the combination

```
DEVICE=VIRT,ADDRESS=label
```

referred to a virtual device, and that the device could be defined by another TERMINAL statement referenced by the ADDRESS parameter. A virtual channel consists of two such virtual devices, each referencing the other.

For example,

```
A       TERMINAL  DEVICE=VIRT,ADDRESS=B,SYNC=YES
```

and

```
B       TERMINAL  DEVICE=VIRT,ADDRESS=A
```

constitute a virtual channel. The SYNC parameter on terminal A designates that terminal as the one to which 'synchronization' events will be posted. The synchronization mechanism will be discussed in "Inter-program Dialogue" on page 282.

A TERMINAL statement specifying DEVICE=VIRT can be entered in an application program provided exactly the same statement is entered in the system configuration program. All TERMINAL statements within the application program are automatically converted to an IOCB statement. The label on the TERMINAL statement is used for the label and the operand of the IOCB statement.

Establishing the Connection

Each program connects to its side of the channel in a way which is analogous to the way in which real terminals are accessed,

i.e., the program can be 'loaded' from a virtual device, or it can acquire the device by issuing the ENQT instruction.

Accessing the Virtual Terminal

The application program can acquire a virtual terminal through an IOCB simply by issuing

```
          ENQT   IO1
            .
            .
            .
IO1       IOCB   VCHAN1
```

Loading from a Virtual Terminal

When an Event Driven Executive program is loaded from a real terminal, that terminal becomes its 'primary' communication port. When one program loads another, the current terminal of the first program is 'passed' and becomes the primary terminal of the second. It is this convention which allows a new program to establish a virtual terminal as the primary port for the loaded program. This is done with the following sequence:

Definition contained in supervisor system generation:

```
A         TERMINAL   DEVICE=VIRT,ADDRESS=B,SYNC=YES
B         TERMINAL   DEVICE=VIRT,ADDRESS=A,END=YES
```

Program execution:

```
            ENQT        B
            LOAD        $TERMUT1,LOGMSG=NO,END=ENDWAIT
            ENQT        A
                        .
                        .
                        .
  A         IOCB        A
  B         IOCB        B
```

After this sequence, $TERMUT1 has B (the 'other' end of the channel) as its primary port, and the loading program has A ('this' end of the channel) as its current port.

Inter-program Dialogue

Once the two communicating programs have connected to their respective ends of the channel, the normal terminal I/O instructions can be used to send and receive data. These instructions include PRINTEXT, READTEXT, PRINTNUM and GETVALUE. Also, attention interrupts can be generated by means of the TERMCTRL PF instruction described in the Language Reference. The usual conventions with respect to output buffering and advance input apply. There are some questions of communication protocol (such as knowing when a program is ready for input, has ended, etc.) which need not be made explicit for dialogue with a human operator with a real terminal but which, for virtual terminals, require the condition to be explicitly communicated through the task code word. The relevant code word values are listed below along with their meanings.

Value	Transmit	Receive
X'8Fnn'	NA	LINE=nn received
X'8Enn'	NA	SKIP=nn received
-2	NA	Line received (no CR)
-1	Normal completion	New line received
1	Not attached	Not attached
5	Disconnect	Disconnect
8	Break	Break

LINE=nn (X'8Fnn') This code is posted for READTEXT or GETVALUE instructions if the other side sent the LINE forms control operation; it is transmitted so that the receiving program can reproduce on a real terminal (for printer spooling applications for example) the output format intended by the sending program.

SKIP=nn (X'8Enn') The sending program transmitted SKIP=nn.

Line Received (-2) This code indicated that the sending program did not send a new-line indicator, but that the line was transmitted because of execution of a control operation or a transition to the read state. This is how, for example, a prompt message is usually transmitted with READTEXT or GETVALUE.

New Line Received (-1) This code indicates transmission of the carriage return at the end of the data. The distinction between a new-line indicator and a simple line indicator is made only to allow preservation of the original output format.

Not attached (1) If the virtual terminal accessed for the operation does not reference another virtual terminal, then this code is returned.

Disconnect (5) This code value corresponds to the not ready indication for real terminals; its specific meaning for virtual terminals is that the program at the other end of the channel terminated either through PROGSTOP or operator intervention.

Break (8) The break code indicates that the other side of the channel is in a state (transmit or receive) which is incompatible with the attempted operation. If only one end of the channel is defined with SYNC=YES, then the task on that end will always receive the break code, whether or not it attempted the operation first. If both ends are defined with SYNC=YES, then the code will be posted to the task which last attempted the operation. The break code can thus be understood as follows: when reading (READTEXT or GETVALUE), the

other program has stopped sending and is waiting for input; when writing (PRINTEXT or PRINTNUM), the other program is also attempting to write. Note that current Event Driven Executive programs, or future programs which do not interpret the break code, must always communicate through a virtual terminal which is defined with SYNC=NO (the default).

Example: The following program exhibits virtual terminal communications in a general way. Its function is to load a program designated by the operator, communicate with that program through a virtual channel and relay messages between it and the real terminal.

```
VIRTCHAN PROGRAM   BEGIN
*-------------------------------------------------------
*
*         THIS EXAMPLE EXHIBITS VIRTUAL TERMINAL
*         IN A GENERAL WAY.  ITS FUNCTION IS TO
*         LOAD A PROGRAM DESIGNATED BY THE OPERATOR,
*         COMMUNICATE WITH THE PROGRAM THROUGH A
*         VIRTUAL CHANNEL, AND RELAY MESSAGES BETWEEN
*         IT AND THE REAL TERMINAL.
*
*-------------------------------------------------------
          COPY      PROGEQU  (REQUIRED IF USING $EDXASM)
A         TERMINAL  DEVICE=VIRT,ADDRESS=B,SYNC=YES
B         TERMINAL  DEVICE=VIRT,ADDRESS=A,END=YES
DEND      ECB
RETCODE   DC        F'0'
LINE      TEXT      LENGTH=80
*
BEGIN     READTEXT  LINE,'UTILITY: '     * GET PGM NAME
          IF        (LINE,EQ,C' ',BYTE),GOTO,END
          MOVE      LOAD+10,LINE,(14,BYTES)  * TO LOAD INSTR
          ENQT      B                 * GET END B
LOAD      LOAD      DUMMY,LOGMSG=NO,EVENT=DEND,ERROR=NEXTPROG
          ENQT      A                   * GET END A
NEXTREAD  READTEXT  LINE,MODE=LINE      * NEXT LINE FROM PGM
          MOVE      RETCODE,VIRTCHAN    * GET RETURN CODE
          DEQT                        * DEQUEUE FROM CHANNEL
          IF        (RETCODE,EQ,5),GOTO,ENDWAIT   * IF END
          IF        RETCODE,NE,8      * IF NOT END OF OUTPUT
            IF        (RETCODE,EQ,-1),OR,(RETCODE,EQ,-2)
              PRINTEXT  LINE,MODE=LINE     * TO TERMINAL
              IF        RETCODE,EQ,-1      * IF NEWLINE
                PRINTEXT  SKIP=1           * THEN DO NEWLINE
              ENDIF
            ELSE
              IF        (RETCODE,EQ,X'8F',BYTE)  * IF LINE=
                AND       RETCODE,X'00FF'          * THEN DO IT
                PRINTEXT   LINE=RETCODE            * ON TERMINAL
              ELSE
                IF          (RETCODE,EQ,X'8E',BYTE)  * IF SKIP=
                  AND         RETCODE,X'00FF'        * ETC.
                  PRINTEXT    SKIP=RETCODE
                ENDIF
              ENDIF
            ENDIF
          ENQT      A                     * RETURN TO CHANNEL
```

```
          ELSE
            READTEXT  LINE,MODE=LINE        * IF BREAK, READ LINE
            ENQT      A                     * RETURN TO CHANNEL
            PRINTEXT  LINE,MODE=LINE        * SEND LINE
            PRINTEXT  SKIP=1
          ENDIF
          GOTO      NEXTREAD                * GO GET NEXT LINE
ENDWAIT   WAIT      DEND                    * WAIT FOR END EVENT
NEXTPROG  DEQT                              * DEQUEUE FR CHANNEL
          GOTO      BEGIN                   * GO GET NEXT PGM
END       PROGSTOP
          ENDPROG
          END
```

CROSS PARTITION SERVICES

Cross partition services permit asynchronous but coordinated execution of application programs running in different partitions.

These services can be used when interrelated programs and tasks in your application cannot be accommodated in a single partition.

When your task is attached, its TCB ($TCBADS) is updated to contain the number of the address space in which it is executing. The address space value (the partition number minus one) is also known as the hardware address key. This key, along with an address you supply, is used to calculate the target address used in cross partition services. For some functions, you put the address key of the target partition in $TCBADS.

Cross partition services provide the following:

- Loading other programs via the LOAD instruction (using the PART= operand).

```
PROGA     PROGRAM START,DS=DATASET1
          COPY    PROGEQU
          COPY    TCBEQU
START     LOAD    PROGB,PARML1,EVENT=PROGBEND,PART=2
          WAIT    PROGBEND
          .
          .
          .
PROGBEND  ECB     0           DEFINE ECB W/VALUE OF ZERO
PARML1    DATA    F'0'        (AS REQUIRED)
```

In this example, PROGB is loaded into partition two and the parameters at PARML1 are passed to it. When PROGB terminates, the supervisor will post the ECB at PROGBEND, signaling PROGA that PROGB has ended.

$TCBADS is not modified by the LOAD instruction.

Note: In this and the following examples, the same data areas are referenced to show interrelationships.

- Finding other programs via the WHERES instruction which returns the address key and the load point of a program

```
          WHERES  PROGB,ADDRB,KEY=KEYB
          .
          .
          .
PROGB     DATA    C'PROGB    '  PROGRAM NAME
ADDRB     DATA    F'0'          FOR PGM HDR ADDRESS
KEYB      DATA    F'0'          FOR PGM ADDRESS KEY
```

The above instruction causes the address key and program header address of PROGB to be placed into KEYB and ADDRB.

$TCBADS is not modified by the WHERES instruction.

- Data movement via the MOVE instructions using the FKEY= and TKEY= operands. FKEY designates the address key of the partition containing the "from" address (operand2 of the MOVE instruction). TKEY designates the address key of the partition containing the "to" address (operand1 of the MOVE instruction).

```
          MOVE  #2,ADDRB      ADDRESS OF HEADER
*
*   GET TCB ADDRESS FROM PROGB HEADER
*
          MOVE  TASKADDR,($PRGTCB,#2),FKEY=KEYB
TASKADDR  DATA  F'0'           PROGB'S PRIMARY TCB
```

Using the address and address key obtained from the WHERES instruction in the previous example, the above cross-partition MOVE instruction obtains the primary TCB address from a program in another partition.

- Starting other tasks via the ATTACH instruction. $TCBADS and the supplied task name are used to calculate the partition and address of the task to be attached.

Assume that PROGB's primary task does not terminate but issues a DETACH instruction. PROGA can cause the primary task of PROGB to become active by issuing an ATTACH, specifying the address key and the address of the TCB as follows:

```
          MOVE  KEYSAV,$TCBADS+PROGA  SAVE MY KEY
          MOVE  $TCBADS+PROGA,KEYB    SET UP KEY VAL
*
*  REATTACH PROGB'S PRIMARY TASK
*
          ATTACH *,500,CODE=-5,P1=TASKADDR
          MOVE  $TCBADS+PROGA,KEYSAV  RESTORE MY KEY
KEYSAV    DATA  F'0'                  KEY SAVE AREA
```

This sequence of instructions saves the address space key, replaces it with the key of PROGB, and issues an ATTACH instruction to reactivate PROGB's primary TCB. In the above example, TASKADDR is set to the appropriate TCB address.

- Sharing resources via the ENQ/DEQ instructions. $TCBADS and the QCB address are used to calculate the partition and the address of the resource to be enqueued or dequeued.

  Assume that PROGA has passed to PROGB the address of two QCBs and that PROGB has saved these addresses in RES1 and RES2 respectively. Sharing these resources can be accomplished as follows:

```
          MOVE  SAVKEY,$TCBADS+PROGB   SAVE KEY
          MOVE  RESNAME,RES1           RESOURCE ADDRESS
          MOVE  $TCBADS+PROGB,PROGAKEY SET KEY TO PROGA
          ENQ   *,BUSY=CANTHAVE,P1=RESNAME ISSUE ENQ
          MOVE  $TCBADS+PROGB,SAVKEY   RESTORE MY KEY
          .
          .
          .
CANTHAVE  DATA  F'0'    ROUTINE TO:
*                         1. RESET $TCBADS FROM SAVKEY
*                         2. HANDLE NON-AVAILABLE
*                            RESOURCE
          .
          .
          .
RES1      DATA  F'0'      RESOURCE ONE ADDRESS
RES2      DATA  F'0'      RESOURCE TWO ADDRESS
SAVKEY    DATA  F'0'      SLOT FOR KEY SAVE AREA
```

  This example acquires exclusive use of the resource defined as RES1. If the resource is not available, execution of PROGB resumes at location CANTHAVE. It is also possible to wait for PROGA to free (dequeue) RES1. In this case, a BUSY keyword is not entered, causing PROGB to be suspended until PROGA issues a DEQ for the resource.

  A DEQ instruction is set up in an identical manner.

- Synchronizing tasks via the WAIT/POST instructions. $TCBADS and the ECB address are used to calculate the partition number and the address of the ECB to be waited on

or posted.

Assume that PROGA has passed parameters to PROGB, among them an ECB address and the key of PROGA. The parameters are saved in PROGAECB and PROGAKEY respectively. PROGB can signal (post) PROGA as follows:

```
          MOVE   KEYHOLD,$TCBADS+PROGB  SAVE $TCBADS
          MOVE   PROGA,PROGAECB         SET ADDR OF ECB
          MOVE   $TCBADS+PROGB,PROGAKEY SET PROGA'S KEY
          POST   *,-1,P1=PROGA          POST PROGA
          MOVE   $TCBADS+PROGB,KEYHOLD  RESTORE $TCBADS
          .
          .
          .
KEYHOLD   DATA   F'0'                   SAVE AREA FOR KEY
PROGAECB  DATA   F'0'                   ECB ADDR IN PROGA
PROGAKEY  DATA   F'0'                   PROGA'S KEY
```

In this example, PROGB saves its key, inserts the address of PROGA's ECB, set $TCBADS to the key of PROGA, issues a POST to PROGA, and restores its $TCBADS, using the value saved.

A WAIT instruction is set up in an identical manner.

- I/O services via the READ/WRITE or BSCREAD/BSCWRITE instructions. $TCBADS is used to calculate the partition and address to/from which data will be transferred.

  Assume that PROGB had STG=1024 on its PROGRAM statement. This causes a 1024 byte area of storage to be acquired for PROGB when it is loaded. The address of this area is in PROGB's program header (at $PRGSTG). PROGA can acquire this address as follows:

```
          MOVE   PROGBBUF,($PRGSTG,#2),FKEY=KEYB #2 HAS
*                          THE ADDRESS OF PROGB'S HEADER
          .
          .
          .
PROGBBUF  DATA   F'0'       ADDR OF PROGB'S DYNAMIC AREA
```

PROGA can then read data directly into PROGB's storage as follows:

```
            MOVE  SAVAKEY,$TCBADS+PROGA SAVE $TCBADS
            MOVE  BUFAD,PROGBBUF        PROGB BUFFER
            MOVE  RELREC,RECN           START WITH REC 1
            MOVE  $TCBADS+PROGA,KEYB    SET PROGB KEY
*
*  READ FROM DS1 INTO PROGB'S BUFFER
*
            READ  DS1,*,4,*,END=EOD,ERROR=RDERR,        C
                  P2=BUFAD,P4=RELREC
*
*  RESTORE PROGA'S KEY
*
            MOVE  $TCBADS+PROGA,SAVAKEY
                  .
                  .
                  .
EOD         DATA  F'0'    ROUTINE TO:
*                             1. RESET $TCBADS
*                             2. HANDLE END OF DATA
                  .
                  .
                  .
RDERR       DATA  F'0'    ROUTINE TO:
*                             1. RESET $TCBADS
*                             2. HANDLE ERROR CONDITIONS
                  .
                  .
                  .
RECN        DATA  F'1'    RELATIVE RECORD NUMBER
```

In this example, four 256-byte records are transferred (from the data set described by DS1 in PROGA's program header) to the storage address obtained from PROGB's header.

Notes:

1. After issuing the cross partition service request, $TCBADS was immediately restored to its original value. It is recommended that this practice be implemented in your application. Doing so will preclude unexpected or unpredictable results such as overlaying other applications with data or waiting indefinitely because of ECBs that were never posted or DEQs that were never issued.

2. In the READ example, only the LOC operand of the READ instruction (or the BUFFER operand of the BSCIOCB instruction) is affected by $TCBADS. All other operand values or addresses are contained in the address space of the issuing program. Therefore, the END operand specifies a routine in your program which is to be invoked if an end-of-data condition occurs.

3. When an ATTACH instruction is executed, the default terminal address or the currently active terminal address of the task issuing the ATTACH is placed into $TCBCCB. This address is a CCB address.

4. When a LOAD instruction is executed for an overlay or non-overlay program, the default terminal address or the currently active terminal address of the program issuing the LOAD is placed in the program header of the loaded program. This address is taken from $PRGCCB in the issuing program's program header and placed into $PRGCCB of the loaded program's program header. This address is a CCB address.

Logical screens can be defined either during system generation or at the time an ENQT instruction is executed. Examples of the TERMINAL statement for the 4978/4979 Display are given below.

```
TERM1     TERMINAL  DEVICE=4979,ADDRESS=04
```

Defines the default configuration, to be used for general purpose program loading and execution. The entire screen simulates a typewriter terminal.

```
TERM2     TERMINAL  DEVICE=4979,ADDRESS=14,SCREEN=STATIC
```

Defines a full screen static configuration to be used for data entry and display. Programs can be loaded from the terminal, but the terminal I/O instructions issued will be interpreted for a STATIC screen unless the configuration is changed to ROLL by an IOCB. This configuration would normally be used when the terminal is to be used only as a data entry device.

```
TERM3     TERMINAL  DEVICE=4978,ADDRESS=24,TOPM=12,NHIST=6
```

This represents a split screen configuration. The area of roll screen operation will be limited to the bottom 12 lines of the screen, leaving the top half for other logical screens to be defined upon execution of ENQT.

```
TERM4     TERMINAL  DEVICE=4979,ADDRESS=34,LEFTM=39,      C
                    BOTM=11,NHIST=0
```

This statement defines a roll screen occupying the upper right quadrant of the screen. In general, logical screens with less than an 80-character line size suffer some performance disadvantages (such as slower erasure) but can be useful for special applications. Note that NHIST is zero here because screen shifting will not be performed; a non-zero value for NHIST would merely cause the history area to be unused.

Logical screens can also be established by the ENQT instruction referencing an IOCB. Examples follow.

```
DISPLAY   PROGRAM   BEGIN
TOPHALF   IOCB      BOTM=11,SCREEN=STATIC
BEGIN     ENQT      TOPHALF
            .
            .
            .
          DEQT
          PROGSTOP
          ENDPROG
          END
```

The IOCB labeled TOPHALF defines the top half of the screen from which the program was loaded as a static screen. If, for example, the terminal was configured as in TERM3 on the previous page, the program could have been loaded by entry of $L (program name) in the roll screen area on the bottom half of the screen. Since no terminal name is specified on the IOCB statement, the ENQT refers to the loading terminal. The program then might display tabular information on the static screen, execute DEQT and then end. The result of this is that the information displayed can remain on the screen while the terminal operator performs other operations using the roll screen.

```
NOTICE   PROGRAM   BEGIN
TERMX    IOCB      SCREEN=STATIC
NAMETAB  DATA      CL8'TERM1'
         DATA      CL8'TERM2'
         DATA      CL8'TERM3'
         DATA      CL8'TERM4'
BEGIN    MOVEA     #1,NAMETAB
         DO        4
         MOVE      TERMX,(0,#1),(8,BYTES)
         ENQT      TERMX
         PRINTEXT  'SYSTEM ACTIVE',LINE=0
         DEQT
         ADD       #1,8
         ENDDO
         PROGSTOP
         ENDPROG
         END
```

This example illustrates terminal access by using the name of the terminal. TERM1, TERM2, TERM3, and TERM4 must have been defined on a TERMINAL configuration statement. The use of the static screen mode insures that only physical line 0 of each screen will be altered. (LINE=0 for roll screens causes a page eject and erasure of information.)

Modifying the IOCB

Elements of the IOCB which may be modified by an application program are the terminal name, roll to static, and NHIST. The structure given here is provided for those special applications in which other elements may need to be modified; note that the structure may change with future versions of the Event Driven Executive.

BYTE(S)	ELEMENT	COMMENTS
0-7	Terminal Name	EBCDIC, blank filled
8	Flags	#CCBFLGS is described in the Internal Design manual under "Terminal I/O". Bit 0 off indicates that the name is the only element of the IOCB.
9	Top of working area	Equal to TOPM+NHIST
10	Top margin	TOPM or zero
11	Bottom margin	BOTM, or X'FF' if unspecified
12	Left margin	LEFTM or zero
13	Page size	Equal to X'00' if unspecified
14-15	Line size	Equal to X'7FFF' if unspecified
16	Current line	Initialized to TOPM+NHIST
17	Current indent	Left margin included
18-19	Buffer address	Zero if unspecified

Accessing a Static Screen

Line-oriented input/output instructions provide the most straightforward means for constructing and reading data from static screens. However, when individual data fields are accessed frequently, excessive screen flicker can result. This problem can be eliminated by transferring an entire screen image to the display device with one I/O operation. The following program will illustrate this procedure as well as some other important points relating to programming for static screens.

```
DISPLAY  PROGRAM   BEGIN
SCREEN   IOCB      SCREEN=STATIC,BOTM=5,                C  2
                   BUFFER=BUFF,RIGHTM=479
I        DATA      F'0'
BUFF     BUFFER    480,BYTES                              5
         DATA      X'0202'                                6
NULLS    DATA      X'0000'                                7
NUMS     DATA      48F'0'                                 8
VALS     TEXT      LENGTH=254                             9
BEGIN    ENQT      SCREEN                                10
         TERMCTRL  BLANK                                 11
         PRINTEXT  LINE=0                                12
*
* THIS DO LOOP PLACES THE WORD "FIELD" AND THE VALUE
*  OF "I" INTO THE TERMINAL BUFFER 48 TIMES.  THE
*  ACTUAL CONTENTS OF THE TERMINAL BUFFER IS PRINTED
*  WHEN THE "TERMCTRL DISPLAY" STATEMENT IS REACHED.
*
         DO        48,INDEX=I                            13
         PRINTEXT  'FIELD',PROTECT=YES                   14
         PUTEDIT   FORMAT1,VALS,((I)),PROTECT=YES        15
         PRINTEXT  ' ',PROTECT=YES                       16
         PRINTEXT  NULLS,PROTECT=YES                     17
         ENDDO
         PRINTEXT  LINE=0                                19
         PUTEDIT   FORMAT1,VALS,((NUMS,48)),          C  20
                   ACTION=STG
         PRINTEXT  VALS,MODE=LINE,LINE=0                 22
         PRINTEXT  LINE=0,SPACES=8                       23
         TERMCTRL  DISPLAY                               24
         DEQT                                            25
         PROGSTOP
FORMAT1  FORMAT (I2)
         ENDPROG
         END
```

This program accesses the top six lines of the screen in static mode and initially formats it with a sequence of protected fields. An array of integers is displayed on lines 0-5 and a pause is executed to allow the operator to enter a new set of values in corresponding positions of lines 6-11. The new values are then displayed on lines 0-5.

Numbers refer to lines in the preceding example program.

2 Define the static screen with the terminal I/O buffer to be in the application program at BUFF, with a length of 480 bytes (one-quarter of the 4979 display screen).

5 Allocate storage for the buffer. Note that in this program the buffer is never accessed directly; the space is merely allocated here for use by the supervisor.

6 and 7 Define a TEXT message consisting of two null characters (EBCDIC code X'00').

8 and 9 Define the array of integers (initially zero) and the TEXT buffer which will be used for input and output of the data in EBCDIC form.

10 to 12 Acquire the terminal, erase all data and establish the screen position for the first I/O operation. Since several text strings will be concatenated to form the first output line, the screen position must be established in advance.

13 Begin a DO loop to construct the initial screen image. This will consist of 48 protected fields of the form FIELDxx, where xx is a sequential field number, each followed by one protected blank and two unprotected data positions. Note here the conditions required for forming a concatenated line; the protection mode of the PRINTEXT instructions must not change (either all PROTECT=YES or all PROTECT=NO), and no intervening forms control operations can be executed.

14 Write 'FIELD' to the buffer.

15 Convert the sequence number to two EBCDIC characters and write it to the buffer.

16 Write a protected separation character.

17 Write the two null characters to define the data positions. Null characters will always generate unprotected positions on the screen; PROTECT=YES is nevertheless required here in order to maintain concatenation.

19 Write the concatenated line to the display. Any convenient line control operation or the DEQT instruction will accomplish this.

20 Convert the integer array to two-character EBCDIC values and store the resulting line in the TEXT buffer VALS.

22 Write the values into successive unprotected positions of the display beginning at LINE=0,SPACES=0. This 'scatter write' mode is defined by MODE=LINE; without MODE=LINE the protected fields of the display would be overwritten.

23 Define the cursor to be at the first unprotected position.

24 Display the cursor at its defined position.

25 Release the terminal. The buffer designated in the IOCB will be released and the screen configuration restored to that defined by the TERMINAL statement.

In the program described above, the terminal I/O operations were all conveniently performed through the concatenation of TEXT strings. If the application requires more complex formatting of the screen image, or if input of more than 254 bytes at a time is necessary, then direct access to the buffer is appropriate. See PRINTEXT and READTEXT.

Using Formatted Screen Images

Formatted screen images can be created and saved in disk or diskette data sets with the utility program $IMAGE. The retrieval and display of such images can be simplified by employing a set of subroutines. An EXTRN statement must be coded for each subroutine name which is referenced, and AUTO=$AUTO,ASMLIB must be coded on the OUTPUT statement of the link-edit control data set.

In the calling formats given below, arguments which represent addresses to be passed to a subroutine must be enclosed within parentheses as shown. If the desired address is contained within a variable, then the name of that variable must be written without parentheses.

$IMOPEN Subroutine

This subroutine reads the designated image from disk or diskette into your buffer. You can also perform this operation by using DSOPEN or defining the data set at program load time, and issuing the disk READ instruction. Refer to the format description at the end of this section for data set size determination.

Syntax

```
label     CALL  $IMOPEN,(dsname,volume),(buffer),P2=,P3=

Required:  dsname,buffer
Defaults:  None
Indexable: None
```

Operands	Description
dsname	The address of a TEXT statement which contains the name of the data set. A volume label can be included, separated from the name by a comma.
buffer	The address of a BUFFER statement allocating the storage into which the image data will be read. The storage should be allocated in bytes, as in the following example:

```
label    BUFFER  256,BYTES
```

Error Codes (Returned in taskname +2)	
-1	Successful completion
1	Disk I/O error
2	Invalid data set name
3	Data set not found
4	Incorrect header or data set length
5	Input buffer too small
6	Invalid volume name

$IMDEFN Subroutine

Subroutine $IMDEFN is used to construct an IOCB for a formatted screen image. The IOCB can also be coded directly, but the use of $IMDEFN allows the image dimensions to be modified with $IMAGE without requiring a change to the application program.

Syntax

```
label      CALL  $IMDEFN,(iocb),(buffer),topm,leftm,
                 P2=,P3=,P4=,P5=

Required:  iocb,buffer
Defaults:  None
Indexable: None
```

Operands	Description
iocb	The address of an IOCB statement defining a static screen. The IOCB need not specify the TOPM, BOTM, LEFTM nor RIGHTM parameters; these are 'filled in' by the subroutine. The following IOCB would normally suffice:

```
label   IOCB   terminal,SCREEN=STATIC
```

buffer — The address of an area containing the screen image in disk storage format. (The format is described at the end of this section.)

topm — This parameter indicates the screen position at which line 0 will appear. If its value is such that lines would be lost at the bottom of the screen, then it is forced to zero.

leftm — This parameter indicates the screen position at which the left edge of the image will appear. If its value is such that characters would be lost at the right of the screen, then it is forced to zero.

Once an IOCB for the static screen area has been defined, the program can then acquire that screen through ENQT and call one or both of the following subroutines in order to display the image.

$IMPROT Subroutine

This subroutine displays the protected and null fields of an image which is in disk-storage format. A field table giving the location (line,spaces) and size of each data field of the image can also be constructed at the option of the calling program.

Syntax

```
label      CALL  $IMPROT,(buffer),(ftab),P2=,P3=

Required:  buffer,ftab
Defaults:  None
Indexable: None
```

Operands Description

buffer The address of an area containing the screen image in disk storage format. The format is described at the end of this section.

ftab The address of a table to be constructed giving the location (lines,spaces) and size (characters) of each data field of the image.

The table has the following form:

```
label          line        ** field 1
               spaces
               size
label+6        line        ** field 2
               spaces
               size
                 .
                 .
                 .
label+6(n-1)   line        ** field n
               spaces
               size
```

The field numbers correspond to the following ordering: left to right in the top line, left to right in the second line, and so on to the last field in the last line. Storage for the field table should be allocated with a BUFFER statement specifying the desired number of words. The buffer control word at label-2 will be used to limit the amount of field information stored, and the buffer index word at buffer-4 will be set with the number of fields for which information was stored, the total number of words being three times that value. If the field table is not desired, code zero for this parameter.

$IMDATA Subroutine

$IMDATA can be called to display the initial data values for an image which is in disk storage format.

Syntax

```
label     CALL  $IMDATA,(buffer),P2=

Required:  buffer
Defaults:  None
Indexable: None
```

Operands	Description
buffer	The address of an area containing the image in disk-storage format.

Example: Formatted Screen Images

The following program illustrates use of the image generation subroutines in a general application. Under direction of the terminal operator, this program displays any image stored on disk, at any (4978 or 4979) terminal, and at any screen coordinates. For each image, a field table is constructed and used to modify the initial data values.

```
IMDISP   PROGRAM   BEGIN
*-------------------------------------------------
*
*        THIS IS AN EXAMPLE OF THE USE OF
*        IMAGE GENERATION SUBROUTINES.
*        UNDER DIRECTION OF THE TERMINAL
*        OPERATOR, THIS PROGRAM DISPLAYS
*        ANY IMAGE STORED ON DISK AT ANY
*        4978 OR 4979 TERMINAL.
*
*-------------------------------------------------
         EXTRN     $IMOPEN,$IMDEFN,$IMPROT,$IMDATA
*     ** Terminal name to IOCB
BEGIN    READTEXT  IMAGE,'TERMINAL: '
*     ** Get data set name
         READTEXT  DSNAME,'DATA SET: ',PROMPT=COND
*     ** Get origin values
         GETVALUE  ORG,'ORIGIN(LINE,SPACES): ',2,          C
                   PROMPT=COND
*     ** Open image data set
         CALL      $IMOPEN,(DSNAME),(DISKBFR)
*     ** If open is unsuccessful,
*     ** Print error code and query
         IF        IMDISP+2,NE,-1
           MOVE      CODE,IMDISP+2
           PRINTEXT  '@OPEN ERROR CODE'
           PRINTNUM  CODE
           GOTO      NEXT
         ENDIF
*     ** Construct IOCB
         CALL      $IMDEFN,(IMAGE),(DISKBFR),ORG,ORG+2
         ENQT      IMAGE       ** Acquire static screen
         TERMCTRL  BLANK       ** Blank screen
*                              ** Write protected fields
*                              ** and build field table
*                              ** at FTAB
         CALL      $IMPROT,(DISKBFR),(FTAB)
         CALL      $IMDATA,(DISKBFR)  ** Write data fields
*     ** Set cursor at first field
         PRINTEXT  LINE=FTAB,SPACES=FTAB+2
         TERMCTRL  DISPLAY     ** Unblank screen
         DEQT                  ** Return to this terminal
         WAIT      KEY         ** WAIT for operator
         ENQT      IMAGE       ** Back to target terminal
         TERMCTRL  BLANK       ** Blank screen
         MOVEA     #1,FTAB     ** Point #1 to field table
*     ** Build line of @S
         MOVE      LINE,C'@',(80,BYTES)
         DO        FTAB-4      ** DO NR of fields
*                              ** Field size to text size
           MOVE      LINE-1,(5,#1),BYTE
```

```
*        ** Print @S in field
            PRINTEXT  LINE,LINE=(0,#1),SPACES=(2,#1)
            ADD       #1,6      ** Next field
         ENDDO
         TERMCTRL  DISPLAY    ** Unblank screen
         DEQT                 ** Back to this terminal
         WAIT      KEY        ** Allow viewing time
         ENQT      IMAGE      ** Acquire target image
         ERASE     LINE=0,MODE=SCREEN,TYPE=ALL  ** Erase
         DEQT                 ** Back to roll screen
NEXT     QUESTION  'ANOTHER IMAGE? ',YES=BEGIN
         PROGSTOP
DSNAME   TEXT      LENGTH=16          ** Data set name
DISKBFR  BUFFER    512,BYTES          ** Disk buffer
         DC        X'0808'            ** Text control for name
IMAGE    IOCB      SCREEN=STATIC      ** IOCB for image
CODE     DC        F'0'               ** Return code
ORG      DC        2F'0'              ** Origin (LINE,SPACES)
FTAB     BUFFER    300
LINE     TEXT      LENGTH=80
         ENDPROG
         END
```

In the above example, use of the field size from the field table is for illustrative purposes only. Each non-protected output operation is terminated by the beginning of the next protected field, unless MODE=LINE is coded.

The size of the disk buffer can vary between 256 bytes and 2048 bytes. Because of the data compression used in storage of the images, many images will require only 256 bytes (1 sector), and 512 bytes (2 sectors) will be adequate for typical applications.

The display subroutines normally write images to the terminal in line-by-line fashion. If a defined image consists of 80-byte lines, however, then a performance improvement can be obtained by providing a terminal buffer large enough to contain more than one line. Since the display subroutines will perform concatenated write operations whenever possible, larger buffers result in fewer such operations and, therefore, faster generation of the display image. For a full screen image, for example, a time for space trade-off can be obtained by choosing a buffer size between 80 bytes (1 line) and 1920 bytes (24 lines). A temporary buffer can be defined by coding the BUFFER= parameter on the IOCB which is used to access the screen.

End of Forms on 4973 and 4974 Printers

Terminal I/O goes into a wait state trying to print when one of the following situations occurs:

- The printer is in DISABLE (4973) or WAIT (4974) mode.
- The printer is out of paper.
- The paper is jammed in the printer.

You can correct this problem by doing the following:

- If in DISABLE or WAIT mode, set the switch to ENABLE (on 4973) or to PRINT (on 4974) to resume printing.
- If the printer is out of paper or the paper is jammed, set the mode switch to DISABLE or WAIT, add new paper or fix paper jam, and reset the mode switch to ENABLE or PRINT.

Reading Modified Data on the 4978 Display

Both protected and unprotected fields on the 4978 are defined as a set of contiguous characters that may span line boundaries. A protected field ends when an unprotected field is encountered. Similarly, an unprotected field ends when a protected field is encountered. Neither an unprotected nor a protected field necessarily ends at an EDX partial screen boundary.

An unprotected field is considered a modified field when any character within the field is modified by the operator. The field may be read by the Event Driven Language READTEXT instruction with TYPE=MODDATA. Reading the field leaves its modified status unchanged. A modified field becomes an unmodified field by either writing or erasing all the characters in the field. For additional information, refer to IBM Series/1 4978-1 Display Station (RPQ D02055) and Attachment (RPQ D02038), General Information, GA34-1550.

TRANSLATING COMPRESSED/NONCOMPRESSED BYTE STRINGS

The following two subroutines are used internally by $IMPROT and $IMDATA as well as by the utility program $IMAGE. They can also, however, be called directly by an application program and are described here because of their general utility.

$UNPACK Subroutine

This subroutine moves a compressed byte string and translates it to noncompressed form.

Syntax

```
label     CALL  $UNPACK,(source),(dest),P2=,P3=

Required:  source,dest
Defaults:  None
Indexable: None
```

Operands	Description
source	The address of a compressed byte string. (See Figure 43 on page 311 for the compressed format.) At completion of the operation, this parameter is incremented by the length of the compressed string.
dest	The address at which the expanded string is to be placed. The length of the expanded string is placed in the byte preceding this location. The $UNPACK subroutine can, therefore, conveniently be used to move and expand a compressed byte string into a TEXT buffer.

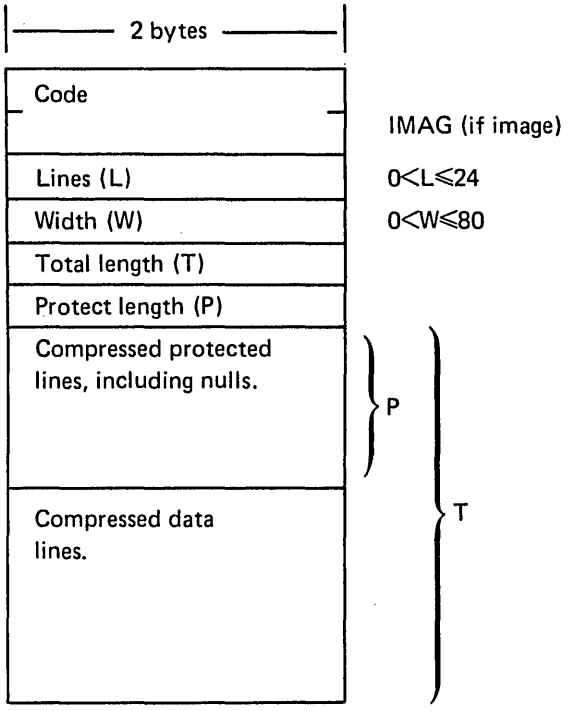
$PACK Subroutine

This subroutine moves a byte string and translates it to compressed form.

Syntax

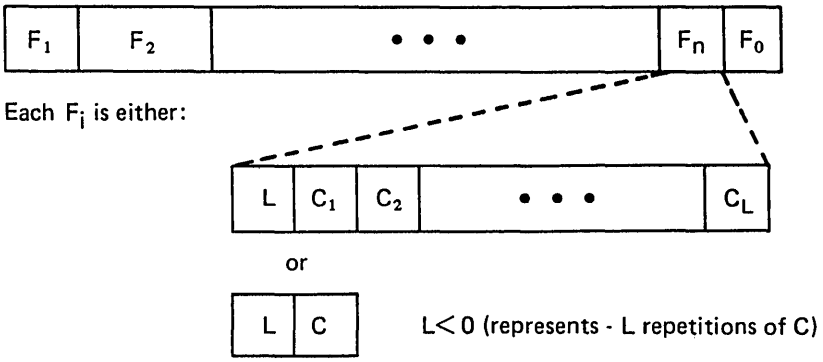
```
label     CALL  $PACK,(source),(dest),P2=,P3=

Required:  source,dest
Defaults:  None
Indexable: None
```

Operands	Description
source	The address of the string to be compressed. The length of the string is taken from the byte preceding this location, and the string could, therefore, be the contents of a TEXT buffer.
dest	The address at which the compressed string is to be stored. At completion of the operation, this parameter is incremented by the length of the compressed string.



Total length of data set = T+12 bytes, or $\frac{T+267}{256}$

Compressed line format:



Where L and C_i are bytes.

Figure 43. $IMAGE Disk Storage Format

TERMINALS CONNECTED VIA DIGITAL I/O

Terminal support is provided for digital I/O devices such as the Tektronix[5] 4010 Series of Display Terminals equipped with the General Purpose Parallel Interface (Tektronix Custom Feature Number CM021-0109-03) or terminals having equivalent hardware interfaces. The software provides addressing logic such that up to eight terminals may be shared on one digital input group and one digital output group, with one process interrupt bit for each terminal.

The parallel interface is intended to connect directly to the integrated digital input/output feature or the 4982 nonisolated digital input/output features. This interface consists of a driver and a receiver card, each of which has several selectable options. These "patch options" allow you to customize the interface to the requirements. You must refer to the manufacturer's manuals for detailed installation procedures. The following description is intended only to supplement those manuals and guide you in the use of the Event Driven Executive terminal support on the Series/1. The following options should be selected:

Receiver Card	
INTR (interrupt)	PROG
ADDRESS	000(0)-111(7) to match TERMINAL definition
PERM ADD	OFF
PARITY	EVEN
DELAY	3.5-18 (depends on distance)
LOGIC SENSE (3) HANDSHAKE CONTROL DATA	
THRESHOLD	+2 volts
MASTER OPTION	None

[5] Registered trademark of the Tektronix Corporation.

Driver Card	
LOGIC SENSE (4) STATUS HANDSHAKE INTERRUPT DATA	Set all to HIGH.
INTERRUPT CHANNEL	Use INTR
AUX TSUP	OUT
ECHO	OUT
PARITY	EVEN, BIT 8 IN AB to A, CD to D

Before the terminal may be used with the computer, some other considerations are necessary. As noted above, the common interrupt line (INTR) should be used. It is recommended that you select the interrupt line (0 - 7) corresponding to the terminal address. If fewer than eight terminals are attached, some of the interrupt lines will not be used. All digital input and process interrupt lines must be terminated for proper operation. If only one terminal is used, the DI terminations may have been installed by the manufacturer. With multiple terminals, all DI lines and PI lines should be terminated at the computer. A 1000-ohm resistor across the DI and PI inputs is recommended.

When the terminal is powered on, it may be necessary to reset the terminal. The procedure is to put the LOCAL/LINE switch to LOCAL, back to LINE, and simultaneously depress the SHIFT and RESET keys. If the terminal does not respond during normal operation, it may be necessary to perform this sequence to reset the internal circuits.

Since all Event Driven Executive terminal input/output is done with upper case ASCII character codes, the TTY LOCK key should be activated when using the terminal with the Series/1. The last items which merit special discussion are the GIN mode and the PAGE FULL BREAK strap options on the terminal control card (TC-2). The GIN termination should be set to NONE. Thus, when GIN mode is used, you must strike the appropriate key followed by carriage return (CR). The PAGE FULL BREAK termination may be set to either OUT or IN, depending on your preference. If it is IN, the terminal will always stop when a full page condition is reached. You must hit the PAGE RESET key in order to continue. If it is OUT, the terminal will automatically go to the

home address and continue printing without erasing the screen.

THE $DISKUT3 DATA MANAGEMENT UTILITY

$DISKUT3 provides certain data management functions for disks and diskettes. It must be invoked from application programs.

$DISKUT3 performs the following functions:

- Allocates new data set(s)
- Opens existing data set(s)
- Deletes existing data set(s)
- Releases unused space in existing data set(s)
- Renames existing data set(s)
- Sets end of data on existing data set(s)

Any combination of these basic functions may be performed in one invocation of $DISKUT3. For example, it is more efficient to open two data sets and allocate two other data sets in one invocation of the program. It is quicker to perform multiple functions with one invocation of the program than to use separate invocations. This eliminates extra program load operations.

Note: $DISKUT3 may be invoked only by the LOAD Event Driven Language instruction. If it is invoked by use of the $L operator command, $DISKUT3 immediately terminates.

Special Considerations

Some special considerations when using $DISKUT3 are:

- An attempt to delete a data set which does not exist is considered to be a successful operation
- An attempt to allocate an existing data set is considered to be a successful operation if the following conditions are met:
  - The type attribute of the existing data set is the same as the allocation request, and
  - The size of the existing data set is the same as the allocation request
- If an allocation request is set for an existing data set and the type attributes match but the sizes do not, a return code is set indicating whether the data set is

smaller or larger than that requested. Refer to "$DISKUT3 Return Codes" on page 319 for the return codes and their meanings.

- $DISKUT3 requires 4352 bytes of storage.

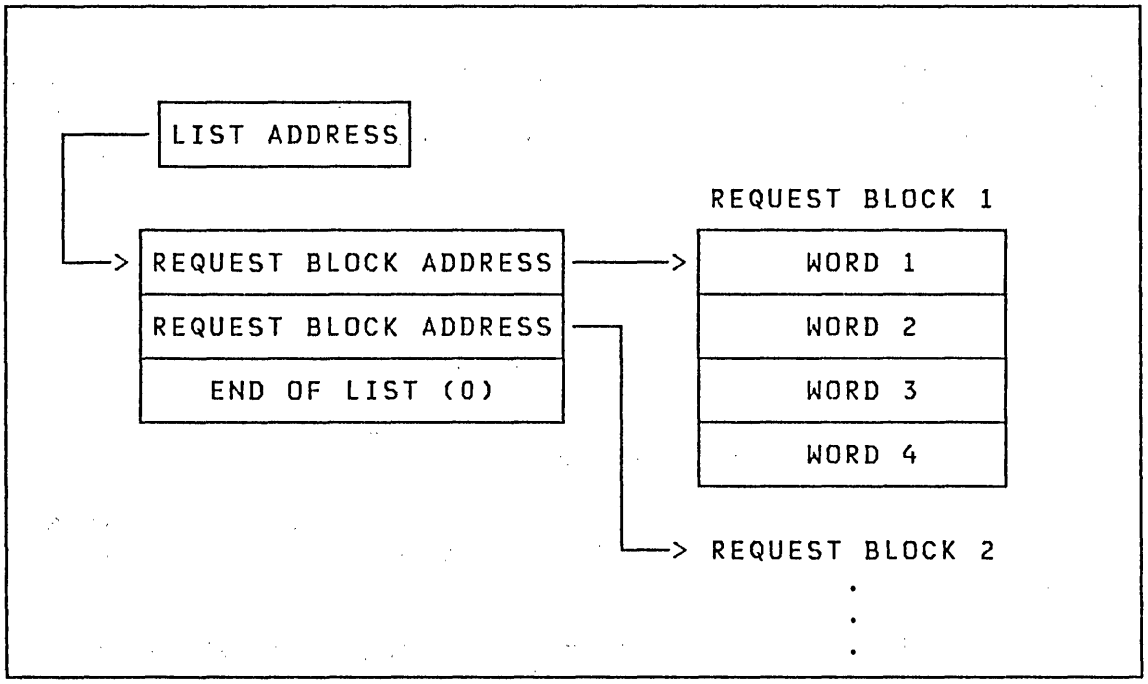
Input to $DISKUT3

Input to $DISKUT3 consists of the following:

Initial Parameter

An initial parameter is passed to $DISKUT3 when it is loaded. It is the address of a list of request block addresses. The end of the request block address list is indicated by a word of zero.

The following diagram shows the relationship between the $DISKUT3 input parameters:



Request Block Contents

A request block is four words.

Word 1: The first word contains a binary value which indicates the function to be performed as follows:

Code	Function
F'1'	Open a data set
F'2'	Allocate a new data set
F'3'	Rename a data set
F'4'	Delete a data set
F'5'	Release unused space in a data set
F'6'	Set end of data in a data set

Word 2: The second word contains the address of the associated Data Set Control Block (DSCB). The DSCB describes the volume and data set being referenced.

A DSCB must be specified for each function requested. The data set name ($DSCBNAM) and volume ($DSCBVOL) fields must be filled in.

Word 3: The content of the third word varies according to the function requested as follows:

Function	Code	Contents
ALLOCATE	F'nr'	The data set size in records
DELETE	F'0'	Unused
OPEN	F'0'	Unused
RELEASE	F'nr'	The new size of the data set in records must be less than the current data
RENAME	A(NEWNAME)	The address of an eight-byte field containing the new data set name
*SETEOD	F'nb'	The number of bytes in the last record

(nr=value in the range of 0 to 32,767)

(nb=number of bytes)

NEWNAME=address of a 1- to 8-character data set name

* For SETEOD, $DISKUT3 assumes that the last record written or read is the end of data, that is, DSCB points to the next available record in the data set.

Word 4: The fourth word specifies the data set type attribute. The types that may be specified are:

Code	Type
F'0'	Undefined
F'1'	Data
F'3'	Program
F'-1'	Unspecified

For an ALLOCATE request, this is the type attribute to be assigned to the data set. For OPEN, RENAME, DELETE and RELEASE, code a -1. This causes $DISKUT3 to return the type attribute in word 4. Upon return from $DISKUT3, word 4 is set to 0, 1, or 3, depending upon the attribute type of the data set specified. If this word is set to any value other than -1, $DISKUT3 compares the type attribute specified with the type attribute specified where the data set was allocated. If they do not match, a return code of 17 is set and $DISKUT3 terminates. In all cases except DELETE, the DSCB is returned in an open condition. Therefore, the ALLOCATE function need not be accompanied by an OPEN parameter list or followed by a DSOPEN.

$DISKUT3 Return Codes

The first word of the DSCB is posted with a return code to indicate whether the function was performed successfully (-1) or unsuccessfully. When a list of more than one function is specified, each function requested is processed in the sequence presented. A return code is posted for each function attempted.

Caution: If more than one function which references the same DSCB is requested on a single $DISKUT3 invocation, the return code set reflects only the results of the last function attempted using that DSCB.

The return codes set by $DISKUT3 and their meanings are as follows:

Code	Condition
1	Invalid request code (not 1-5)
2	Volume does not exist (All functions)
4	Insufficient space in library (ALLOCATE)
5	Insufficient space in directory (ALLOCATE)
6	Data set already exists - smaller than the requested allocation
7	Insufficient contiguous space (ALLOCATE)
8	Disallowed data set name, eg. $EDXVOL or $EDXLIB (All functions)
9	Data set not found (DELETE, RENAME, RELEASE, OPEN)
10	New name pointer is zero (RENAME)
11	Disk is busy (ALLOCATE, DELETE, RENAME, RELEASE)
12	I/O error writing to disk (ALLOCATE, DELETE, RENAME, RELEASE)
13	I/O error reading from disk (All functions)
14	Data set name is all blanks (ALLOCATE, RENAME)
15	Invalid size specification (ALLOCATE)
16	Invalid size specification (RELEASE)
17	Mismatched data set type (OPEN, RENAME, DELETE, RELEASE)
18	Data set already exists - larger than the requested allocation
19	SETEOD only valid for data set of type 'data'
20	Load of $DISKUT3 failed ($RMU only)
21	Tape data sets not supported

Figure 44. $DISKUT3 return codes

Example: The following example illustrates the use of $DISKUT3. The use of all five functions (OPEN, ALLOCATE, RENAME, DELETE, and RELEASE) is shown.

```
TASK     PROGRAM GO,((DS1,EDX002),(DS2,EDX003))
         COPY    DSCBEQU
GO       EQU     *
         .
         .
         .
*  LOAD $DISKUT3 IN THE 'NON-OVERLAY' MODE, TO OPEN
*  DATA SET 'DSY', ALLOCATE A NEW DATA SET 'DSX', AND
*  RENAME AN EXISTING DATA SET 'DS1'
*
         LOAD    $DISKUT3,LISTPTR1,EVENT=$DISKUT3
         WAIT    $DISKUT3
         .
         .
*  COMPUTE CURRENT SIZE OF THE DATA SET AND USE IT AS A
*  CALLING PARAMETER FOR A 'RELEASE' RECORDS CALL TO
*  $DISKUT3.
*  THE ASSUMPTION IS THAT THE DATA SET HAS BEEN WRITTEN
*  SEQUENTIALLY. THEREFORE '$DSCBNXT' POINTS TO THE NEXT
*  RECORD TO BE USED IN THE DATA SET AND $DSCBNXT-1 IS
*  THE NUMBER OF RECORDS CURRENTLY IN USE.
*
         SUBTRACT DSX+$DSCBNXT,1,RESULT=REQUEST5+4
         .
*  LOAD $DISKUT3, DELETE DATA SET 'DS2',
*  AND RELEASE THE UNUSED SPACE IN 'DSX'.
*
         LOAD  $DISKUT3,LISTPTR2,EVENT=$DISKUT3,PART=ANY
         WAIT  $DISKUT3
         .
         .
         PROGSTOP
         .
         .
$DISKUT3 ECB   0              SET INITIAL STATE TO ZERO
*
LISTPTR1 DC    A(LIST1)      POINTER TO LIST OF REQUEST
*                             BLOCK POINTERS
LISTPTR2 DC    A(LIST2)      POINTER TO ANOTHER LIST OF
*                             REQUEST BLOCK POINTERS
```

$DISKUT3 Use Example (Continued)

```
LIST1     DC    A(REQUEST1)
          DC    A(REQUEST2)
          DC    A(REQUEST3)
          DC    F'0'          END OF LIST FLAG
*
LIST2     DC    A(REQUEST4)
          DC    A(REQUEST5)
          DC    F'0'          END OF LIST FLAG
*
REQUEST1  DC    F'1'          REQUEST IS FOR AN 'OPEN'
          DC    A(DSY)        DSCB IS 'DSY'
          DC    F'0'          UNUSED FOR OPEN REQUESTS
          DC    F'0'          UNUSED FOR OPEN REQUESTS
*
REQUEST2  DC    F'2'          REQUEST IS FOR AN 'ALLOCATE'
          DC    A(DSX)        DSCB IS 'DSX'
          DC    F'50'         ALLOCATE 50 RECORDS
          DC    F'1'          DATA SET TYPE IS 'DATA'
*
REQUEST3  DC    F'3'          REQUEST IS FOR A 'RENAME'
          DC    A(DS1)        DSCB IS 'DS1'
          DC    A(NEWNAME)    ADDRESS OF NEW DATA SET NAME
          DC    F'-1'         FOR RENAME REQUESTS
*
REQUEST4  DC    F'4'          REQUEST IS TO 'DELETE'
          DC    A(DS2)        DSCB IS 'DS2'
          DC    F'0'          UNUSED FOR DELETE REQUESTS
          DC    F'-1'         FOR DELETE REQUESTS
*
REQUEST5  DC    F'5'          REQUEST IS TO 'RELEASE' SPACE
          DC    A(DSX)        DSCB IS 'DSX'
          DC    A(*-*)        NEW SIZE OF DATA SET
          DC    F'-1'         FOR RELEASE REQUESTS
*
          DSCB  DS#=DSY,DSNAME=DSY
*
          DSCB  DS#=DSX,DSNAME=DSX
          .
NEWNAME   DC    CL8'RENAMED' NEW DATA SET NAME
          ENDPROG
          END
```

DSOPEN SUBROUTINE

DSOPEN is a subroutine that can be copied into your program. It opens a disk, diskette, or tape data set for input and/or output operations by initializing a DSCB. Only one DSCB can be open to a tape at a time. If a tape has been opened, a close must be issued before another open can be requested. The results of DSOPEN processing are identical to the implicit open performed by $L or LOAD for data sets specified in the PROGRAM statement.

Use DSOPEN to open a data set after the program has begun execution.

The following functions are performed:

- Verifies that the specified volume is online
- Verifies that the specified data set is in the volume
- Initializes the DSCB

Using DSOPEN adds 1056 bytes to the size of your program.

To use DSOPEN, you must first copy the source code into your program by coding:

```
        COPY  PROGEQU
        COPY  DDBEQU
        COPY  DSCBEQU
                .
                .
                .
        COPY  DSOPEN
```

During execution, DSOPEN is invoked via the CALL instruction as follows:

```
        CALL  DSOPEN,(dscb)
```

Four optional parameters are also available. Three are error return addresses and the fourth is the address of an area in which DSOPEN saves a directory control entry (DCE) and the first directory member entry (DME).

The three error exit addresses are:

1. Data set not found

2. Invalid VOLSER

3. I/O error

Since the exit addresses and the area address lie within your program, they must be initialized by your program before it calls DSOPEN. DSOPEN automatically sets them to zero. The labels of these fields can be found in the beginning of the DSOPEN copy code. Since the four parameters are addresses within your program, you must insert (move) them to the beginning of the DSOPEN routine before calling it.

You must have a 256-byte work area labeled DISKBUFR in your program. The DSCB to be opened can be DS1-DS9 or a DSCB defined in your program via the DSCB statement. The DSCB must be initialized with a 6-character volume name in $DSCBVOL and an 8-character data set name in $DSCBNAM. Other fields are ignored. The volume name can be specified as 6 blanks, which causes the IPL volume to be searched for the data set.

After DSOPEN processing, #1 contains the number of the directory record containing the member entry and #2 contains the displacement within DISKBUFR to the member entry. The fields $DSCBR3 and $DSCBR4 contain the next available logical record data, if any, placed in the directory by SETEOD. Refer to the comments in the DSOPEN copy-code for additional details.

SETEOD

SETEOD is a copy code routine that updates the directory member entry (DME) of a disk directory to reflect the last record accessed up to the point in time SETEOD is invoked. Information on the DME can be found in Internal Design. The value in $DSCBNXT (relative record number to be used for next sequential READ or WRITE) is placed in the next available logical record field of the DME, so that it can be retrieved by subsequent calls of DSOPEN.

If the value of $DSCBNXT is 1 when SETEOD is performed, the DME is set to indicate that the data set is empty. Subsequent calls to DSOPEN cause $DSCBEOD to be set to X'FFFF', indicating that the data set is empty. If $DSCBEOD is zero, the length of the data set ($DSCBLEN) is used as the end-of-data (EOD) value.

SETEOD is used to indicate a logical end of file on disk. If your program does not SETEOD when creating or overwriting a file, the READ end of data exception will occur at either the physical end, or the logical end set by some previous use of the data set.

SETEOD can be used before, during or at the end of either input or output. It does not inhibit further I/O and can be used more than once. The only requirement is that the DSCB passed as input must have been previously opened.

The POINT function modifies the $DSCBNXT field. If SETEOD is used after a POINT, the relative record number pointed to becomes the value placed in the directory by SETEOD.

SETEOD requires that the DSOPEN copy code, PROGEQU, and TCBEQU be included in your program. SETEOD uses the 256-byte DISKBUFR that is also used by DSOPEN. You invoke SETEOD as a subroutine through the Event Driven Language CALL statement, passing the DSCB and an I/O error exit routine pointer as parameters.

Using SETEOD adds 318 bytes to the size of your program.

To use SETEOD, you must first copy the source code into your program by coding:

```
         COPY  PROGEQU
         COPY  TCBEQU
         COPY  DDBEQU
         COPY  DSCBEQU
                .
                .
                .
         COPY  DSOPEN
         COPY  SETEOD
```

Calling Sequence

```
         CALL   SETEOD,(DS1),(IOERROR)
```

where

DS1 Names a previously opened DSCB

IOERROR Names the I/O error exit routine

PROCESSING THE EOV CONDITION

Reading End-of-Volume (EOV) Labels

The Event Driven Executive does not provide EOV processing. However, you may elect to add EOV processing to your application. To read a multi-volume data set the following steps can be used:

1. Vary the tape online (specifying the SL option).

2. Execute the program, reading and processing data records.

   When the end of the data set is reached, the END= exit routine of the READ statement will be entered. (If you do not use the END option, check for return code 10.)

3. Perform a CONTROL CLSOFF operation in the END= exit or when return code 10 is encountered.

   If the return code from the CONTROL operation is a +33 (EOV encountered), then the close processing has detected an EOV1 label. This means more data is contained on another reel. The CONTROL completes by rewinding the tape and setting it offline.

4. Issue a message (PRINTEXT) telling the operator to enter the volume serial number of the next tape.

5. Read (READTEXT) the volume serial number supplied by the operator from the terminal and place it in the $DSCBVOL field of the DSCB used to READ the data set.

6. Issue a message (PRINTEXT) telling the operator to place the next volume on an available tape drive and vary it online using $VARYON.

7. After the new tape has been varied online, call the DSOPEN subroutine to ready the data set for READ processing.

   Note: The new volume **must** be online ($VARYON) before DSOPEN is called.

8. Resume reading and processing as soon as the tape is opened

For a sample of the operator console sheet for the reading EOV process, see "Console Output for EOV Processing" on page 327 For a sample of a program to process an EOV condition while reading, see "Input EOV Processing Example" on page 329.

Writing End-of-Volume (EOV) Labels

To write EOV labels for multi-volume data sets the following steps can be used.

1. Allocate or initialize a SL tape.
2. Vary the initialized tape online using BLP and the proper file number to position the tape at the beginning of the data set (not the header label).
3. Write data records to the tape.

   When the end of tape (EOT) is sensed (return code 24 during a WRITE operation), the END= exit specified on the WRITE statement is entered. If you do not wish to use an END exit, check for return code 24.
4. Create the trailer label in the END= exit code of your application. (See "Chapter 11. Tape Organization" on page 233 for the format of the EOV label.)

   - Write one tapemark (CONTROL WTM).
   - Write an 80-byte EOV1 record (WRITE).
   - Write two more tapemarks (CONTROL WTM).
   - Rewind the tape (CONTROL ROFF).
   - Prompt the operator to mount and vary the tape online ($VARYON).

     The next tape must have been SL initialized with a different volume name and the same data set name as previous tape.
5. Use DSOPEN to open tape.
6. Resume writing data records.

Console Output for EOV Processing

The following is the console listing seen by the operator during EOV reading processing, as performed by the sample program "Input EOV Processing Example" on page 329.

```
> $VARYON 4C
123456 ONLINE
> $L TESTEOV
 INPUTDS (NAME,VOLUME): MYDATA,123456
TESTEOV       8P,11:29:52, LP=0000

BEGIN EOV TEST PROGRAM

EOV ENCOUNTERED - ENTER VOL1 OF NEXT VOLUME
654321

REPLY Y WHEN THE VOLUME IS MOUNTED AND ONLINE
?
> $VARYON 4C
654321 ONLINE
?Y

END EOV TEST PROGRAM

TESTEOV ENDED AT 11:34:14
```

Input EOV Processing Example

The following example illustrates input EOV processing:

```
PROGX    PROGRAM  START,DS=((INPUTDS,??))
START    EQU      *
         PRINTEXT '@BEGIN EOV TEST PROGRAM@'
LOOP     EQU      *
         READ     DS1,BUFFER,1,80,ERROR=ERR1,END=CHKEND
*
**
***      PROCESS THE DATA RECORD
**
*
         GOTO     LOOP                GET NEXT RECORD
ENDIT    EQU      *
         PRINTEXT '@END EOV TEST PROGRAM@'
         PROGSTOP
*
**
***     CHECK FOR REAL END OF DATA OR ONLY END OF VOLUME (EOV)
**
*
CHKEND   EQU      *
         CONTROL  DS1,CLSOFF
         IF       (DS1,EQ,33)         IF CLOSE FOUND AN EOV
          PRINTEXT '@EOV ENCOUNTERED - ENTER VOL1 OF NEXT VOLUME@'
          READTEXT NEWVOL
          MOVEA    #1,DS1
          MOVE     ($DSCBVOL,#1),NEWVOL,(3,WORD)
*      SET DSOPEN ERROR EXITS
          MOVEA    $DSNFND,ERRDSN
          MOVEA    $DSBIODA,ERRIODA
          MOVEA    $DSBVOL,ERRVOL
          MOVEA    $DSIOERR,ERRIO
*      OPEN THE NEXT VOLUME
    PRINTEXT '@REPLY Y WHEN THE NEXT VOLUME IS MOUNTED AND ONLINE@'
X         READTEXT REPLY,'?'
          IF       (REPLY-1,EQ=0,BYTE),GOTO,X
          IF       (REPLY,NE,C'Y'),GOTO,ENDIT
          CALL     DSOPEN,(DS1)
          GOTO     LOOP              RESUME PROCESSING DATA
         ENDIF
         GOTO     ENDIT
```

```
*
**    DSOPEN ERROR EXITS
*
ERRDSN    EQU       *
          MOVEA     MSGX,MSG1
          GOTO      ERRMSG
ERRIODA   EQU       *
          MOVEA     MSGX,MSG2
          GOTO      ERRMSG
ERRVOL    EQU       *
          MOVEA     MSGX,MSG3
          GOTO      ERRMSG
ERRIO     EQU       *
          MOVEA     MSGX,MSG4
ERRMSG    EQU       *
          PRINTEXT '@DSOPEN ERROR -@'
          PRINTEXT MSG1,P1=MSGX
          PRINTEXT SKIP=1
          GOTO      ENDIT
MSG1      TEXT      'DATA SET NOT FOUND'
MSG2      TEXT      'INVALID IODA'
MSG3      TEXT      'VOLUME NOT FOUND'
MSG4      TEXT      'I/O ERROR'
*
**
***     DATA AREA
**
*
ERR1      EQU       *              READ ERROR EXIT
          PRINTEXT '@READ ERROR - RC='
          PRINTNUM DS1
          GOTO      ENDIT
*
BUFFER    DATA      40F'0'          80 BYTE BUFFER
NEWVOL    TEXT      LENGTH=6        HOLDS NEW VOLUME #
REPLY     TEXT      LENGTH=2
          COPY      DSOPEN
          COPY      DSCBEQU
          COPY      PROGEQU
          COPY      DDBEQU
DISKBUFR  DC        128F'0'
          ENDPROG
          END
```

SAMPLE USE OF BLP TO ACCESS ALL LABEL FIELDS

The following is an example of using BLP to access label fields.

The program reads the VOL1, HDR1, and EOF1 labels of a standard label tape. The comments in the following example show where code to process labels can be inserted if desired. The sample program reads and ignores labels.

```
*****************************************************************
*     THE TAPE IS MOUNTED ON A BLP DRIVE WITH ID=TAPE01         *
*****************************************************************
PROG8 PROGRAM  START,DS=((XYZ,TAPE01))
START EQU      *
*****************************************************************
*     READ THE VOL1 LABEL (80 BYTES) INTO THE BUFFER            *
*     WHERE IT CAN BE PROCESSED                                 *
*****************************************************************
**
**
      READ     DS1,BUFFER,1,80,ERROR=ERR1
**
***          INTERPRET THE VOL1 RECORDS FIELDS AS
***          DESIRED FOR THE APPLICATION
**
*****************************************************************
*     READ THE HDR1 LABEL (80 BYTES) INTO THE BUFFER            *
*     WHERE IT CAN BE PROCESSED                                 *
*****************************************************************
**
**
      READ     DS1,BUFFER,1,80,ERROR=ERR1
**
***          INTERPRET THE HDR1 RECORDS FIELDS AS
***          DESIRED FOR THE APPLICATION
**
```

```
*****************************************************************
*     SKIP OVER ANY REMAINING BLOCKS IN THE HEADER              *
*     GROUP AND THE TAPEMARK. THIS ALLOWS THE DATA              *
*     TO BE ACCESSED                                            *
*****************************************************************
**
**
     CONTROL  DS1,FSF
**
**
*****************************************************************
*     PROCESS THE APPLICATION DATA ON TAPE                      *
*****************************************************************
**
**
LOOP  EQU      *
      READ     DS1,BUFFER,1,50,ERROR=ERR2,END=ALLDONE
      GOTO     LOOP
ALLDONE EQU    *
*****************************************************************
*     PROCESS THE TRAILER LABEL GROUP                           *
*****************************************************************
**
**
      READ     DS1,BUFFER,1,80,ERROR=ERR1
**
***    INTERPRET THE TRAILER LABELS
***    AS DESIRED FOR THE APPLICATION
**
ENDIT EQU      *
     PROGSTOP
*
ERR1  EQU      *
      PRINTEXT '@LABEL ERROR - RC= '
      PRINTNUM DS1
      GOTO     ENDIT
ERR2  EQU      *
      PRINTEXT '@READ ERROR - RC= '
      PRINTNUM DS1
      QUESTION '@DO YOU WANT TO CONTINUE? ',                    X
               YES=LOOP,NO=ENDIT
*
BUFFER DATA    40F'0'
      ENDPROG
      END
```

APPENDIX A. STORAGE ESTIMATING

To calculate Series/1 storage requirements, you must estimate storage required for:

- The supervisor
- The utility programs
- Your application programs

SUPERVISOR

The supervisor requires storage for each of the items listed in Figure 45 on page 334, Figure 46 on page 336, and Figure 47 on page 337 for V1.0 or items listed in Figure 48 on page 338, Figure 49 on page 340, and Figure 50 on page 341 for V2.0. All numbers are in decimal notation and the unit is bytes. The numbers in the left column are the resident program sizes. The numbers in the right column are initialization routines used only at IPL time. The total of the selected left column numbers, when rounded upward to the next multiple of 256, represent the size of the supervisor program that will reside in storage during system execution. You should allow from three to five per cent more storage than that calculated to provide for error correction. These calculations will be reasonably close to your actual configuration; to get the actual size, perform a system generation of your supervisor. The actual size is the address (in hexadecimal) of EDXINIT as contained in the $LINK output.

Support for	Resident	Initialization
Basic Supervisor		
with Address Translator	6696	212
without Address Translator	5866	
+8*(sum of MAXPROG values)	( )	
Disk or Diskette		
Disk(ette) Basic	1182	1424
4962/4964 Support	1176	
4963 Support	336	
4963 Fixed Head Support		550
4966 Support	1374	
+178 per unit	( )	
+ 32 per secondary logical volume	( )	
+128 per I/O task defined	( )	
Terminals		
Basic		
With Addr Translator	5636	488
Without Addr Translator	4656	488
4979/4978		
With Addr Translator	2164	1634*
Without Addr Translator	2038	1634*
+468 per 4979 or 4978	( )	
4973/4974		
With Addr Translator	716	
Without Addr Translator	640	
+530 per 4973 or 4974	( )	
4013 type devices	480	
+438 per device	( )	
Virtual Terminals	504	
+426 per terminal	( )	
Basic for any TTY, 2741/PROC,ACCA	534	
Teletypewriter	938	
+446 per teletypewriter	( )	
2741/PROC	1446	
+574 per 2741/PROC	( )	
+512 if Correspondence code	( )	
+512 if EBCD code	( )	
ACCA ASCII Terminals	1626	500
+498 per terminal	( )	

* 4978 only

Figure 45. (Part 1 of 3) V1.1 Supervisor Storage Requirements

Notes:

1. The above numbers include 128 bytes per terminal for the optional Keyboard Task (ATTN=YES).

2. Basic ASCII support is required for teletypewriter, ACCA, 2741, and 4013 terminals.

Support for:	Resident	Initialization
Timers		
4953/4955	1080	342
4952	1048	180
Binary Synchronous Access Method		
With Addr Translator	3284	570
Without Addr Translator	3034	570
+136 per line of any type	( )	
+ 22 per multi-line controller	( )	
multi-controller	( )	
Host Communication Facility	1910	362
Sensor Based Input/Output		
Basic		
With Addr Translator	1050	180
Without Addr Translator	876	180
Analog Input	610	
+48 for first AI group	( )	
+16 for each additional group	( )	
Analog Output	66	
+16 per AO	( )	
Digital Input and Output	892	
+38 per DI group	( )	
+16 per DO group in 4982	( )	
+38 per DO group in IDIO	( )	
Process Interrupt	164	
+156 per PI group	( )	
EXIO Control		
Basic	686	64
+(32+x(16+n)) per device (x=maximum number of DCBs, n=number of residual status bytes transferred)	( )	
Error Logging		
Included	330	
Not Included	20	

Figure 46. (Part 2 of 3) V1.1 Supervisor Storage Requirements

Support for:	Resident	Initialization
Program/Machine Check Log	250	
Relocating Loader		
With Addr Translator	4004	2250
Without Addr Translator	3044	2250
Floating Point Support		
Included	610	
Not Included	4	
Support of GETEDIT/PUTEDIT		
With Addr Translator	1402	
Without Addr Translator	1330	
Queue Processing Support	258	
$DEBUG Support	384	
Supervisor Patch Area	256	

Figure 47. (Part 3 of 3) V1.1 Supervisor Storage Requirements

Note: The transient program loader requires an area of 3840 bytes which will be overlaid by the loaded programs.

Version 2.0,5719-XS2 Support for	Resident	Initialization
Basic Supervisor		
with Address Translator	6700	228
without Address Translator	5862	
+8*(sum of MAXPROG values)	( )	
Disk or Diskette		
Disk(ette) Basic	1298	1432
4962/4964 Support	1176	
4963 Support	430	
4963 Fixed Head Support		550
+178 per unit	( )	
+ 32 per secondary volume	( )	
4966 Support	1374	
4966 Tape Support	4362	384
+130 per unit	( )	
+128 per I/O task defined	( )	
Terminals		
Basic		
With Addr Translator	5638	616
Without Addr Translator	4656	616
4979/4978		
With Addr Translator	2202	1634*
Without Addr Translator	2076	1634*
+468 per 4979 or 4978	( )	
4973/4974		
With Addr Translator	716	
Without Addr Translator	640	
+530 per 4973 or 4974	( )	
4013 type devices	480	
+438 per device	( )	
Virtual Terminals	504	
+426 per terminal	( )	
Basic for		
any TTY, 2741/PROC,ACCA	536	
Teletypewriter	438	
+446 per teletypewriter	( )	
2741/PROC	1446	
+574 per 2741/PROC	( )	
+512 if Correspondence code	( )	
+512 if EBCD code	( )	
ACCA ASCII Terminals	1114	500
+506 per terminal	( )	

* 4978 only

Figure 48. (Part 1 of 3) V2.0 Supervisor Storage Requirements

Notes:

1. The above numbers include 128 bytes per terminal for the optional Keyboard Task (ATTN=YES).

2. Basic ASCII support is required for teletypewriter, ACCA, 2741, and 4013 terminals.

Support for:	Resident	Initialization
Timers		
4953/4955	1078	342
4952	1038	180
Binary Synchronous Access Method		
With Addr Translator	3282	570
Without Addr Translator	3032	570
+136 per line of any type	( )	
+ 22 per multi-line controller	( )	
multi-controller	( )	
Host Communication Facility	1928	362
Sensor Based Input/Output		
Basic		
With Addr Translator	1050	178
Without Addr Translator	876	178
Analog Input	610	
+48 for first AI group	( )	
+16 for each additional group	( )	
Analog Output	66	
+16 per AO	( )	
Digital Input and Output	932	
+38 per DI group	( )	
+16 per DO group in 4982	( )	
+38 per DO group in IDIO	( )	
Process Interrupt	164	
+156 per PI group	( )	
EXIO Control		
Basic	698	64
+(32+x(16+n)) per device (x=maximum number of DCBs, n=number of residual status bytes transferred)	( )	
Error Logging		
Included	352	
Not Included	20	

Figure 49. (Part 2 of 3) V2.0 Supervisor Storage Requirements

Support for:	Resident	Initialization
Program/Machine Check Log	250	
Relocating Loader		
With Addr Translator	4016	2352
Without Addr Translator	3068	2352
Floating Point Support		
Included	610	
Not Included	4	
Support of GETEDIT/PUTEDIT		
With Addr Translator	1602	
Without Addr Translator	1330	
Queue Processing Support	258	
$DEBUG Support	384	
Supervisor Patch Area	256	

Figure 50. (Part 3 of 3) V2.0 Supervisor Storage Requirements

Note: The transient program loader requires an area of 3840 bytes which will be overlaid by the loaded programs.

The storage (in bytes rounded up to the next 256 byte increment) required by the Event Driven Executive utility programs:

$BSCTRCE	1792	
$BSCUT1	4864	
$BSCUT2	19712	
$COMPRES	3584	
$COPY	9216	
$COPYUT1	9984	
$DASDI	25600	
$DEBUG	6912	
$DICOMP	11264	
$DIINTR	9728	
$DISKUT1	7680	
$DISKUT2	9728	(+1280 if printing error log)
$DIUTIL	9216	
$DUMP	5888	
$EDIT1	9728	
$EDIT1N	11776	
$EDXASM	18944	(+5632 when assembling TERMINAL statements)
$EDXLIST	6144	
$FONT	5632	
$FSEDIT	22528	
$HCFUT1	2304	
$IAMUT1	12648	
$IMAGE	9728	
$INITDSK	6656	
$IOTEST	8960	
$JOBUTIL	5376	
$LINK	18688	
$LOG	5632	
$MOVEVOL	6144	
$PDS	1792	
$PFMAP	512	
$PREFIND	6144	
$PRT2780	2304	
$PRT3780	2560	
$RJE2780	9728	
$RJE3780	9984	
$TERMUT1	3072	
$TERMUT2	8192	
$TERMUT3	768	
$TRAP	5376	
$UPDATE	7936	
$UPDATEH	6400	

Storage requirements for Version 2 utilities are the same as above except for the addition of the following:

$RMU	7680
$TAPEUT1	5632 (plus size of option)

option	additional storage
EX	5632
CD	3584 (+ additional space requested)
MT	1280
DP	1792 (+ additional space requested)
ST	21248
RT	20480
IT	2560
TA	1792

APPLICATION PROGRAMS

A reasonable estimate of the storage required, in bytes, for a program can be made by totaling the following:

1. Number of source statements * 10 = ( )

   Includes operation code and parameters instruction plus incidental tables and buffers. This estimate was determined from examination of the utility programs, written in Event Driven Executive instructions.

2. Large tables and buffers = ( )

3. Graphics instruction cause subroutines to be added to your program. Add the following for the first occurrence of each instruction.

CONCAT	286	
GIN	158	
PLOTCB	16	
PLOTGIN	186	(+GIN if not already used)
SCREEN	474	
XYPLOT	368	(+SCREEN and CONCAT if not already used)
YTPLOT	368	(+SCREEN if not already used)

   Graphics subroutines = ( )

4. Data formatting instructions cause subroutines to be included in your program. Add the number indicated for each first occurrences of the following specification included in a FORMAT statement referenced in the instructions.

INSTRUCTION	SPECIFICATION	BYTES
GETEDIT,PUTEDIT or FORMAT	any	886
GETEDIT	alphameric	208
GETEDIT	float.pt. F or E	86
GETEDIT	integer	80
PUTEDIT	alphameric	18
PUTEDIT	float.pt. F	88
PUTEDIT	float.pt. E	72
PUTEDIT	integer	66
GETEDIT or PUTEDIT	any parenthetical expression	22

Formatting subroutines = ( )

5. Data formatting instructions cause data areas to be inserted in your program. Add B bytes for each occurrence of the following instructions:

```
GETEDIT
  B=16+(4*V)+A
  where V=the number of variables in list
        A=6 if ACTION+IO, else A=0
```

```
PUTEDIT
  B=16+(4*V)+A
  where V=number of variables in list
        A=4 if ACTION=IO, else A=0
```

```
FORMAT
  B=24+(4*L)
  where L=number of elements in FORMAT list
```

Formatting instructions = ( )

6. When the formatted screen image subroutines are included, program size is increased as follows:

Module	Size (Bytes)
$IMOPEN (includes DSOPEN)	1702
$IMGEN (entry points $IMDEFN $IMPROT $IMDATA $PACK $UNPACK)	1030
Total ( )	

7. Programs using assembler language code will require one of more of the following subroutines to be included in the program:

Module	Size (Bytes)
$$RETURN(entry point RETURN)	38
$$SVC(entry point SVC)	64
$EDXATSR(entry points) SETBUSY, SUPEXIT	40
Total ( )	

8. When a local or a global (or both) ATTNLIST is coded, an extra TCB is generated. The size of this TCB is 128 bytes.

APPENDIX B. V1.1 SUPERVISOR MODULE NAMES (CSECTS)

```
CSECT     EDXSYS
   ENTRY  $START
   ENTRY  RETURN
   ENTRY  $DISKDDB
   ENTRY  $IPLVOL
   ENTRY  $TIMRTBL
   ENTRY  $TESTADR
   ENTRY  $TPDDB
   ENTRY  $BSCADDR
   ENTRY  EDXFLAGS
   ENTRY  SVCFLAGS
   ENTRY  $SBPITAB
   ENTRY  LCBA
   ENTRY  SVCBFIN
   ENTRY  SVCBFOUT
   ENTRY  SVCRTRN
   ENTRY  SVCL1
   ENTRY  SVCLT
   ENTRY  SVCL2
   ENTRY  SVCL3
   ENTRY  SVCLSB
   ENTRY  SVCIAR
   ENTRY  SVCAKR
   ENTRY  SVCLSR
   ENTRY  SVCR0
   ENTRY  SVCR1
   ENTRY  SVCR2
   ENTRY  SVCR3
   ENTRY  SVCR4
   ENTRY  SVCR5
   ENTRY  SVCR6
   ENTRY  SVCR7
   ENTRY  SVCIIAR
   ENTRY  SVCILSB
   ENTRY  SVCIAKR
   ENTRY  SVCILSR
   ENTRY  SVCIR0
   ENTRY  SVCIR1
   ENTRY  SVCIR2
   ENTRY  SVCIR3
   ENTRY  SVCIR4
   ENTRY  SVCIR5
   ENTRY  SVCIR6
   ENTRY  SVCIR7
   ENTRY  UNCHAKR1
   ENTRY  UNCHSAV6
   ENTRY  SYCPARMS
   ENTRY  CMDTABLE
CSECT     EDXSVCX
   ENTRY  SVC
   ENTRY  SVCA
```

```
   ENTRY  SETBUSY
   ENTRY  SVCI
   ENTRY  WAIT
   ENTRY  ENQ
   ENTRY  DEQ
   ENTRY  POST
   ENTRY  ATTACHX
   ENTRY  ATTACH
   ENTRY  DETACH
   ENTRY  SUPEXIT
   ENTRY  SUPEXTRL
   ENTRY  SUPLVLXO
   ENTRY  SATTACH
   ENTRY  SDETACH
   ENTRY  SCWAIT
   ENTRY  SWAIT
   ENTRY  SPOST
   ENTRY  SRESETEV
   ENTRY  SENQ
   ENTRY  SDEQ
   ENTRY  STPTASK1
   ENTRY  STPTASK2
   ENTRY  UNCHAIN
   ENTRY  $CP
   ENTRY  LPGMXP1
   ENTRY  LPGMXP2
CSECT     $DBUGNUC
   ENTRY  $TESTCOM
   ENTRY  STESTIN
   ENTRY  STESTOUT
   ENTRY  $TRCSIA
   ENTRY  $TRCLSB
CSECT     EDXALU
   ENTRY  $EXEC
   ENTRY  #NOP
   ENTRY  CMDSETUP
   ENTRY  CMD$TEST
   ENTRY  #IFB
   ENTRY  #IFW
   ENTRY  #IFDW
   ENTRY  #IFTEST
   ENTRY  $COMPE
   ENTRY  $COMPNE
   ENTRY  $FINDE
   ENTRY  $FINDNE
   ENTRY  CGOTO
   ENTRY  BRANCH
   ENTRY  SDOLOOP
   ENTRY  SCONTINU
   ENTRY  SAV222CR
   ENTRY  SAV424CF
   ENTRY  SAV444CR
   ENTRY  SAV224CR
   ENTRY  SAX222
   ENTY   SA222C
```

```
ENTRY  SA222
ENTRY  SSX222
ENTRY  SS222C
ENTRY  SS222
ENTRY  SM222
ENTRY  SM222C
ENTRY  SD222
ENTRY  SD222C
ENTRY  GETPAR3
ENTRY  GETCNT
ENTRY  SA424
ENTRY  SA424C
ENTRY  SS424
ENTRY  SS424C
ENTRY  SM424
ENTRY  SM424C
ENTRY  SD424
ENTRY  SD424C
ENTRY  SX444
ENTRY  SX444C
ENTRY  SX224R
ENTRY  SX224CR
ENTRY  SD422R
ENTRY  SD422CR
ENTRY  MOV1
ENTRY  MOV1C
ENTRY  AND1
ENTRY  AND1XX
ENTRY  IOR1
ENTRY  IOR1XX
ENTRY  EOR1
ENTRY  EOR1XX
ENTRY  SHR1
ENTRY  SHR1XX
ENTRY  SHL1
ENTRY  SHL1XX
ENTRY  MOV2
ENTRY  MOV2C
ENTRY  AND2
ENTRY  AND2XX
ENTRY  IOR2
ENTRY  IOR2XX
ENTRY  EOR2
ENTRY  EOR2XX
ENTRY  SHR2
ENTRY  SHR2XX
ENTRY  SHL2
ENTRY  SHL2XX
ENTRY  MOV4
ENTRY  MOV4C
ENTRY  AND4
ENTRY  AND4XX
ENTRY  IOR4
ENTRY  IOR4XX
ENTRY  EOR4
```

```
    ENTRY  EOR4XX
    ENTRY  SHR4
    ENTRY  SHR4XX
    ENTRY  SHL4
    ENTRY  SHL4XX
    ENTRY  MOVEXP
    ENTRY  SCALL
    ENTRY  SRETURN
    ENTRY  BFRCK
    ENTRY  USER
CSECT      DISKIO
    ENTRY  DISKRW
    ENTRY  DISKIOOO
    ENTRY  DISKER09
    ENTRY  DCBRETRN
    ENTRY  DISKFLIH
    ENTRY  DFLIH04
    ENTRY  DISKERR1
    ENTRY  DISKERR2
    ENTRY  DISKERR3
    ENTRY  DISKERR5
    ENTRY  DISKER6B
    ENTRY  DISKERR7
    ENTRY  DISKPOST
    ENTRY  VARYON
    ENTRY  VARYOFF
    ENTRY  VARYWORD
    ENTRY  VARYQCB
    ENTRY  VARYDSCB
CSECT      D49624
    ENTRY  D49624AT
    ENTRY  D4962IH1
    ENTRY  DFLIH50
    ENTRY  DISKATTN
    ENTRY  DATTNOO
CSECT      D4963A
    ENTRY  D4963AT
    ENTRY  CNTLBUSY
    ENTRY  D4963IH1
    ENTRY  CNTLEND
    ENTRY  D4963ATN
    ENTRY  D4963AT1
CSECT      D4966A
    ENTRY  D4966AT
    ENTRY  D4966B
    ENTRY  D4966ATN
    ENTRY  D4969A
    ENTRY  VRY4966
CSECT      RLOADER
    ENTRY  LOADPGM
    ENTRY  LPGMXPA
    ENTRY  LOADPGMO
    ENTRY  LOADEXIT
    ENTRY  LPGMXPB
    ENTRY  ENDCODE
```

```
   ENTRY  LOADQCB
   ENTRY  LOADORG
   ENTRY  LCMDKEY
   ENTRY  LCMDTGT
   ENTRY  GETMAIN
   ENTRY  FREEMAIN
   ENTRY  $ACTIVE
   ENTRY  GOTOTABL
   ENTRY  $CANCEL
   ENTRY  STOP
   ENTRY  STOPTASK
CSECT     IOLOADER
   ENTRY  IOLOAD
   ENTRY  IOUNLOAD
CSECT     IOSEXIO
   ENTRY  EXOPEN
   ENTRY  EXIO
   ENTRY  EXFLIH
   ENTRY  EXIOCLEN
CSECT     EDXTIO
   ENTRY  $DPEND
   ENTRY  PRSKSP
   ENTRY  CURCTL
   ENTRY  CTLXFER
   ENTRY  PRTEXT
   ENTRY  NXTCOMD
   ENTRY  NXTCOMD1
   ENTRY  RDTEXTL
   ENTRY  RDTEXT
   ENTRY  QUESTION
   ENTRY  PRTNUM2S
   ENTRY  PRTNUM2
   ENTRY  PRTNUM4S
   ENTRY  PRTNUM4
   ENTRY  GETVAL2
   ENTRY  GETVAL4
   ENTRY  KBTASK
   ENTRY  ENDATTN
   ENTRY  TERMOUT
   ENTRY  TERMINT
   ENTRY  DECSCAN
   ENTRY  FLDCLEAR
   ENTRY  BDCWORD
   ENTRY  DCBWORD
   ENTRY  EBBICVT
CSECT     EDXTERMQ
   ENTRY  ENQDEQT
   ENTRY  QUTERMIN
   ENTRY  QUTERM
   ENTRY  DQTERM
   ENTRY  DQTERMIN
   ENTRY  DQTERMB
   ENTRY  DEQTERMS
CSECT     EDXFLOAT (NOFLOAT has same entry points)
   ENTRY  FADD010
```

```
    ENTRY  FADD000
    ENTRY  FADD100
    ENTRY  FADD001
    ENTRY  FADD011
    ENTRY  FADD101
    ENTRY  FADD110
    ENTRY  FADD111
    ENTRY  FSUB000
    ENTRY  FSUB100
    ENTRY  FSUB001
    ENTRY  FSUB010
    ENTRY  FSUB011
    ENTRY  FSUB101
    ENTRY  FSUB110
    ENTRY  FSUB111
    ENTRY  FLOATERR
    ENTRY  FMPY000
    ENTRY  FMPY001
    ENTRY  FMPY010
    ENTRY  FMPY011
    ENTRY  FMPY110
    ENTRY  FMPY100
    ENTRY  FMPY111
    ENTRY  FMPY101
    ENTRY  FDIV000
    ENTRY  FDIV001
    ENTRY  FDIV010
    ENTRY  FDIV011
    ENTRY  FDIV110
    ENTRY  FDIV100
    ENTRY  FDIV111
    ENTRY  FDIV101
    ENTRY  FLTCONV
    ENTRY  MOVFP4
    ENTRY  MOVFP8
    ENTRY  IFFLOAT
    ENTRY  IFFLOATL
    ENTRY  EDXFLEND
CSECT      EBFLCVY
    ENTRY  EBFLDBL
    ENTRY  EBFLSTD
    ENTRY  FLEBDBL
    ENTRY  FLEBSTD
CSECT      IOSTTY
    ENTRY  WRTTY
    ENTRY  RDTTY
    ENTRY  IATTY
CSECT      IOS4979
    ENTRY  IO4979
    ENTRY  IO4978
    ENTRY  IA4979
    ENTRY  IA4978
CSECT      IOS4974
    ENTRY  IO4974
    ENTRY  IO4973
```

```
   ENTRY  IA4974
   ENTRY  IA4973
CSECT     IOSVIRT
   ENTRY  IOVIRT
   ENTRY  IAVIRT
CSECT     IOS4013
   ENTRY  WR4013
   ENTRY  RD4013
   ENTRY  IA4013
CSECT     IOS2741
   ENTRY  WR2741
   ENTRY  RD2741
   ENTRY  IA2741
   ENTRY  IAPROC
CSECT     IOSTERM
   ENTRY  IOTERM
CSECT     ASCIITAB
   ENTRY  TRASCII
CSECT     EBASCII
   ENTRY  TREBASC
CSECT     EBCDTAB
   ENTRY  TREBCD
CSECT     CRSPTAB
   ENTRY  TRCRSP
CSECT     EDXTIMER
   ENTRY  TIMER0IA
   ENTRY  TIMER1IA
   ENTRY  SETIMER
   ENTRY  WAITIMER
   ENTRY  INTIMEX
   ENTRY  INTIME
   ENTRY  GTIMDATE
   ENTRY  PRINTIME
   ENTRY  WHATIME
   ENTRY  SETCLOCK
CSECT     EDXTIMR2
   ENTRY  TIMER0Ia
   ENTRY  TIMRLSB
   ENTRY  SETIMER
   ENTRY  WAITIMER
   ENTRY  INTIMEX
   ENTRY  INTIME
   ENTRY  GTIMDATE
   ENTRY  PRINTIME
   ENTRY  WHATIME
   ENTRY  SETCLOCK
CSECT     BSCAM
   ENTRY  BSCENTRY
   ENTRY  BSCIA
   ENTRY  DEQBSC
CSECT     IOSACCA
   ENTRY  WRACCA
   ENTRY  RDACCA
   ENTRY  IAACCA
   ENTRY  ACCALS
```

```
CSECT     SBAI
   ENTRY  SAI
   ENTRY  SAIA
   ENTRY  SAIX
   ENTRY  SAIS
   ENTRY  AIIA
CSECT     SBAO
   ENTRY  SAO
   ENTRY  SAOA
   ENTRY  SAOX
CSECT     SBDIDO
   ENTRY  SDI
   ENTRY  SDIA
   ENTRY  SDIX
   ENTRY  SDIS
   ENTRY  SDO
   ENTRY  SDOA
   ENTRY  SDOX
   ENTRY  SDOS
   ENTRY  SDOP
   ENTRY  DIIA
   ENTRY  DOIA
CSECT     SBPI
   ENTRY  PIIALEX
   ENTRY  PIIAG17
   ENTRY  PIIABIT
CSECT     SBCOM
   ENTRY  SBERR
   ENTRY  GETDDB
   ENTRY  CKEXIT
CSECT     QIO
CSECT     TPCOM
   ENTRY  STP
   ENTRY  $TPIA
   ENTRY  $TPDDB1
   ENTRY  TPSTATS
   ENTRY  IDCBSCSS
   ENTRY  IDCBRES
   ENTRY  IDCBSIO
CSECT     SYSLOG
   ENTRY  LCB
   ENTRY  $LOGIA
   ENTRY  $LOGTSK
   ENTRY  $SLOGIA
   ENTRY  $SLOGTSK
   ENTRY  $SLOGPRM
CSECT     NOSYSLOG
CSECT     CIRCBUFF
   ENTRY  CIRSTR
   ENTRY  CIRIN
   ENTRY  CIREND
   ENTRY  CIRCNT
   ENTRY  CIRESIZ
   ENTRY  CIRESTR
CSECT     EDXSTART
```

```
   ENTRY  INITTASK
   ENTRY  PCHKSIA
   ENTRY  PCHKLSB
   ENTRY  SFTKSIA
   ENTRY  STARTPGM
CSECT     EDXINIT
   ENTRY  START
   ENTRY  $EDXINIT
   ENTRY  INITEXIT
CSECT     DISKINIT
   ENTRY  DISKBUFR
   ENTRY  DSKPREP2
   ENTRY  PREPIDCB
   ENTRY  DSKINIT1
   ENTRY  D66INIT
CSECT     RW4963ID
   ENTRY  ID4963IH
CSECT     TERMINIT
   ENTRY  TERMERRX
CSECT     INIT4978
CSECT     BSCINIT
CSECT     $BSCARAM
CSECT     $ACCARAM
CSECT     INIT4013
CSECT     LOADINIT
   ENTRY  $DSNFND
   ENTRY  $DSIOERR
   ENTRY  DSOPEN
CSECT     SBIOINIT
CSECT     TPINIT
CSECT     TIMRINIT
CSECT     EXIOINIT
CSECT     CLOKINIT
   ENTRY  TIMRINIT
```

APPENDIX C. V2.0 SUPERVISOR MODULE NAMES (CSECTS)

```
CSECT     EDXSYS
   ENTRY  $START
   ENTRY  RETURN
   ENTRY  $DISKDDB
   ENTRY  $IPLVOL
   ENTRY  $TIMRTBL
   ENTRY  $TESTADR
   ENTRY  $TPDDB
   ENTRY  $BSCADDR
   ENTRY  EDXFLAGS
   ENTRY  SVCFLAGS
   ENTRY  $SBPITAB
   ENTRY  LCBA
   ENTRY  SVCBFIN
   ENTRY  SVCBFOUT
   ENTRY  SUPRTRN
   ENTRY  SVCL1
   ENTRY  SVCLT
   ENTRY  SVCLT2
   ENTRY  SVCL3
   ENTRY  SVCLSB
   ENTRY  SVCIAR
   ENTRY  SVCAKR
   ENTRY  SVCLSR
   ENTRY  SVCRO
   ENTRY  SVCR1
   ENTRY  SVCR2
   ENTRY  SVCR3
   ENTRY  SVCR4
   ENTRY  SVCR5
   ENTRY  SVCR6
   ENTRY  SVCR7
   ENTRY  SVCIIAR
   ENTRY  SVCILSB
   ENTRY  SVCIAKR
   ENTRY  SVCILSR
   ENTRY  SVCIRO
   ENTRY  SVCIR1
   ENTRY  SVCIR2
   ENTRY  SVCIR3
   ENTRY  SVCIR4
   ENTRY  SVCIR5
   ENTRY  SVCIR6
   ENTRY  SVCIR7
   ENTRY  UNCHAKR1
   ENTRY  UNCHSAV6
   ENTRY  SYCPARMS
   ENTRY  CMDTABLE
CSECT     EDXSVCX
   ENTRY  SVC
   ENTRY  SVCA
```

```
    ENTRY  SETBUSY
    ENTRY  SVCI
    ENTRY  WAIT
    ENTRY  ENQ
    ENTRY  DEQ
    ENTRY  POST
    ENTRY  ATTACHX
    ENTRY  ATTACH
    ENTRY  DETACH
    ENTRY  SUPEXIT
    ENTRY  SUPEXTRL
    ENTRY  SUPLVLXO
    ENTRY  SATTACH
    ENTRY  SDETACH
    ENTRY  SCWAIT
    ENTRY  SWAIT
    ENTRY  SPOST
    ENTRY  SRESETEV
    ENTRY  SENQ
    ENTRY  SDEQ
    ENTRY  STPTASK1
    ENTRY  STPTASK2
    ENTRY  UNCHAIN
    ENTRY  $CP
    ENTRY  LPGMXP1
    ENTRY  LPGMXP2
CSECT      EDXALU
    ENTRY  #IFB
    ENTRY  #IFDW
    ENTRY  #IFW
    ENTRY  #IFTEST
    ENTRY  $COMPE
    ENTRY  $COMPNE
    ENTRY  $FINDE
    ENTRY  $FINDNE
    ENTRY  CGOTO
    ENTRY  BRANCH
    ENTRY  $EXEC
    ENTRY  #NOP
    ENTRY  CMDSETUP
    ENTRY  CMD$TEST
    ENTRY  SDOLOOP
    ENTRY  SCONTINU
    ENTRY  SAV222CR
    ENTRY  SAV424CF
    ENTRY  SAV444CR
    ENTRY  SAV224CR
    ENTRY  SAX222
    ENTY   SA222C
    ENTRY  SA222
    ENTRY  SSX222
    ENTRY  SS222C
    ENTRY  SS222
    ENTRY  SM222
    ENTRY  SM222C
```

```
ENTRY  SD222
ENTRY  SD222C
ENTRY  GETPAR3
ENTRY  GETCNT
ENTRY  SA424
ENTRY  SA424C
ENTRY  SS424
ENTRY  SS424C
ENTRY  SM424
ENTRY  SM424C
ENTRY  SD424
ENTRY  SD424C
ENTRY  SX444
ENTRY  SX444C
ENTRY  SX224R
ENTRY  SX224CR
ENTRY  SD422R
ENTRY  SD422CR
ENTRY  MOV1
ENTRY  MOV1C
ENTRY  AND1
ENTRY  AND1XX
ENTRY  IOR1
ENTRY  IOR1XX
ENTRY  EOR1
ENTRY  EOR1XX
ENTRY  SHR1
ENTRY  SHR1XX
ENTRY  SHL1
ENTRY  SHL1XX
ENTRY  MOV2
ENTRY  MOV2C
ENTRY  AND2
ENTRY  AND2XX
ENTRY  IOR2
ENTRY  IOR2XX
ENTRY  EOR2
ENTRY  EOR2XX
ENTRY  SHR2
ENTRY  SHR2XX
ENTRY  SHL2
ENTRY  SHL2XX
ENTRY  MOV4
ENTRY  MOV4C
ENTRY  AND4
ENTRY  AND4XX
ENTRY  IOR4
ENTRY  IOR4XX
ENTRY  EOR4
ENTRY  EOR4XX
ENTRY  SHR4
ENTRY  SHR4XX
ENTRY  SHL4
ENTRY  SHL4XX
ENTRY  MOVEXP
```

```
   ENTRY  SCALL
   ENTRY  SRETURN
   ENTRY  BFRCK
   ENTRY  USER
CSECT     EDXSTART
   ENTRY  INITTASK
   ENTRY  PCHKSIA
   ENTRY  PCHKLSB
   ENTRY  SFTKSIA
   ENTRY  STARTPGM
CSECT     DISKIO
   ENTRY  DISKRW
   ENTRY  DISKRW05
   ENTRY  TAPE060
   ENTRY  DISKIO00
   ENTRY  DISKER09
   ENTRY  DCBRETRN
   ENTRY  DISKFLIH
   ENTRY  DFLIH04
   ENTRY  DISKERR1
   ENTRY  DISKERR2
   ENTRY  DISKERR3
   ENTRY  DISKERR5
   ENTRY  DISKER6B
   ENTRY  DISKERR7
   ENTRY  DISKPOST
   ENTRY  VARYON
   ENTRY  VARYOFF
   ENTRY  VARYWORD
   ENTRY  VARYQCB
   ENTRY  VARYDSCB
CSECT     D49624
   ENTRY  D49624AT
   ENTRY  D4962IH1
   ENTRY  DFLIH50
   ENTRY  DFLIH54
   ENTRY  DISKATTN
   ENTRY  DATTN00
CSECT     D4963A
   ENTRY  D4963AT
   ENTRY  CNTLBUSY
   ENTRY  D4963IH1
   ENTRY  CNTLEND
   ENTRY  D4963ATN
   ENTRY  D4963AT1
CSECT     D4966A
   ENTRY  D4966AT
   ENTRY  D4966B
   ENTRY  D4966ATN
   ENTRY  VRY4966
CSECT     D4969A
   ENTRY  D69DHPT
   ENTRY  TAPEIO
   ENTRY  TAPEIO00
   ENTRY  TSOPEN
```

```
   ENTRY  ACLOSE
   ENTRY  OPNCLS
   ENTRY  VRY4969
CSECT     IOSEXIO
   ENTRY  EXOPEN
   ENTRY  EXIO
   ENTRY  EXFLIH
   ENTRY  EXIOCLEN
CSECT     EDXTIO
   ENTRY  $DPEND
   ENTRY  PRSKSP
   ENTRY  CURCTL
   ENTRY  CTLXFER
   ENTRY  PRTEXT
   ENTRY  NXTCOMD
   ENTRY  NXTCOMD1
   ENTRY  RDTEXTL
   ENTRY  RDTEXT
   ENTRY  QUESTION
   ENTRY  PRTNUM2S
   ENTRY  PRTNUM2
   ENTRY  PRTNUM4S
   ENTRY  PRTNUM4
   ENTRY  GETVAL2
   ENTRY  GETVAL4
   ENTRY  KBTASK
   ENTRY  ENDATTN
   ENTRY  TERMOUT
   ENTRY  TERMINT
   ENTRY  DECSCAN
   ENTRY  FLDCLEAR
   ENTRY  DBCWORD
   ENTRY  EBBICVT
CSECT     EDXTERMQ
   ENTRY  ENQDEQT
   ENTRY  QUTERMIN
   ENTRY  QUTERM
   ENTRY  DQTERM
   ENTRY  DQTERMIN
   ENTRY  DQTERMB
   ENTRY  DEQTERMS
CSECT     IOS4979
   ENTRY  IO4979
   ENTRY  IO4978
   ENTRY  IA4979
   ENTRY  IA4978
CSECT     IOS4974
   ENTRY  IO4974
   ENTRY  IO4973
   ENTRY  IA4974
   ENTRY  IA4973
CSECT     IOSTERM
   ENTRY  IOTERM
CSECT     IOSTTY
   ENTRY  WRTTY
```

```
   ENTRY  RDTTY
   ENTRY  IATTY
CSECT     IOSACCA
   ENTRY  WRACCA
   ENTRY  RDACCA
   ENTRY  IAACCA
   ENTRY  ACCALS
CSECT     IOS4013
   ENTRY  WR4013
   ENTRY  RD4013
   ENTRY  IA4013
CSECT     IOS2741
   ENTRY  WR2741
   ENTRY  RD2741
   ENTRY  IA2741
   ENTRY  IAPROC
CSECT     IOSVIRT
   ENTRY  IOVIRT
   ENTRY  IAVIRT
CSECT     ASCIITAB
   ENTRY  TRASCII
CSECT     EBASCII
   ENTRY  TREBASC
CSECT     EBCDTAB
   ENTRY  TREBCD
CSECT     CRSPTAB
   ENTRY  TRCRSP
CSECT     EDXTIMER
   ENTRY  TIMEROIA
   ENTRY  TIMER1IA
   ENTRY  SETIMER
   ENTRY  WAITIMER
   ENTRY  WAITIMER
   ENTRY  INTIME
   ENTRY  GTIMDATE
   ENTRY  PRINTIME
   ENTRY  WHATIME
   ENTRY  SETCLOCK
CSECT     EDXTIMR2
   ENTRY  TIMEROIA
   ENTRY  TIMRLSB
   ENTRY  SETIMER
   ENTRY  WAITIMER
   ENTRY  INTIMEX
   ENTRY  INTIME
   ENTRY  GTIMDATE
   ENTRY  PRINTIME
   ENTRY  WHATIME
   ENTRY  SETCLOCK
CSECT     BSCAM
   ENTRY  BSCENTRY
   ENTRY  BSCIA
   ENTRY  DEQBSC
CSECT     TPCOM
   ENTRY  STP
```

```
    ENTRY   $TPIA
    ENTRY   $TPDDB1
    ENTRY   TPSTATS
    ENTRY   IDCBSCSS
    ENTRY   IDCBRES
    ENTRY   IDCBSIO
CSECT       SBCOM
    ENTRY   SBERR
    ENTRY   GETDDB
    ENTRY   CKEXIT
CSECT       IOLOADER
    ENTRY   IOLOAD
    ENTRY   IOUNLOAD
CSECT       SBAI
    ENTRY   SAI
    ENTRY   SAIA
    ENTRY   SAIX
    ENTRY   SAIS
    ENTRY   AIIA
CSECT       SBAO
    ENTRY   SAO
    ENTRY   SAOA
    ENTRY   SAOX
CSECT       SBDIDO
    ENTRY   SDI
    ENTRY   SDAI
    ENTRY   SDIX
    ENTRY   SDIS
    ENTRY   SDO
    ENTRY   SDOA
    ENTRY   SDOX
    ENTRY   SDOS
    ENTRY   SDOP
    ENTRY   DIIA
    ENTRY   DOIA
CSECT       SBPI
    ENTRY   PIIALEX
    ENTRY   PIIAG17
    ENTRY   PIIABIT
CSECT       SYSLOG
    ENTRY   LCB
    ENTRY   $LOGIA
    ENTRY   $LOGTSK
    ENTRY   $SLOGIA
    ENTRY   $SLOGTSK
    ENTRY   $SLOGPRM
CSECT       NOSYSLOG
    ENTRY   $LOGTSK
    ENTRY   $LOGIA
    ENTRY   $SLOGTSK
    ENTRY   $SLOGIA
    ENTRY   $SLOGPRM
CSECT       CIRCBUFF
    ENTRY   CIRSTR
    ENTRY   CIRIN
```

```
   ENTRY  CIREND
   ENTRY  CIRCNT
   ENTRY  CIRESIZ
   ENTRY  CIRESTR
CSECT     RLOADER
   ENTRY  LOADPGM
   ENTRY  LPGMXPA
   ENTRY  LOADPGM0
   ENTRY  LOADEXIT
   ENTRY  LPGMXPB
   ENTRY  ENDCODE
   ENTRY  LOADQCB
   ENTRY  LOADORG
   ENTRY  LCMDKEY
   ENTRY  LCMDTGT
   ENTRY  GETMAIN
   ENTRY  FREEMAIN
   ENTRY  $ACTIVE
   ENTRY  GOTOTABL
   ENTRY  $CANCEL
   ENTRY  STOP
   ENTRY  STOPTASK
CSECT     EDXFLOAT
   ENTRY  FADD010
   ENTRY  FADD000
   ENTRY  FADD100
   ENTRY  FADD001
   ENTRY  FADD011
   ENTRY  FADD101
   ENTRY  FADD110
   ENTRY  FADD111
   ENTRY  FSUB000
   ENTRY  FSUB100
   ENTRY  FSUB001
   ENTRY  FSUB010
   ENTRY  FSUB011
   ENTRY  FSUB101
   ENTRY  FSUB110
   ENTRY  FSUB111
   ENTRY  FLOATERR
   ENTRY  FMPY000
   ENTRY  FMPY001
   ENTRY  FMPY010
   ENTRY  FMPY011
   ENTRY  FMPY110
   ENTRY  FMPY100
   ENTRY  FMPY111
   ENTRY  FMPY101
   ENTRY  FDIV000
   ENTRY  FDIV001
   ENTRY  FDIV010
   ENTRY  FDIV011
   ENTRY  FDIV110
   ENTRY  FDIV100
   ENTRY  FDIV111
```

```
   ENTRY  FDIV101
   ENTRY  FLTCONV
   ENTRY  MOVFP4
   ENTRY  MOVFP8
   ENTRY  IFFLOAT
   ENTRY  IFFLOATL
   ENTRY  EDXFLEND
CSECT     EBFLCVY
   ENTRY  EBFLDBL
   ENTRY  EBFLSTDD
   ENTRY  FLEBDBL
   ENTRY  FLEBSTD
CSECT     QIO
CSECT     $DBUGNUC
   ENTRY  $TESTCOM
   ENTRY  STESTIN
   ENTRY  STESTOUT
   ENTRY  $TRCSIA
   ENTRY  $TRCLSB
CSECT     EDXINIT
   ENTRY  START
   ENTRY  $EDXINIT
   ENTRY  INITEXIT
CSECT     DISKINIT
   ENTRY  DISKBUFR
   ENTRY  DSKPREP2
   ENTRY  PREPIDCB
   ENTRY  DSKINIT1
   ENTRY  D66INIT
CSECT     D69INIT
CSECT     TAPEINIT
CSECT     LOADINIT
   ENTRY  $DSNFND
   ENTRY  $DSIOERR
   ENTRY  DSOPEN
CSECT     RW4963ID
   ENTRY  ID4963IH
CSECT     TERMINIT
   ENTRY  CCBFIXRT
   ENTRY  TERMERRX
CSECT     INIT4978
CSECT     INIT4013
CSECT     $ACCARAM
CSECT     BSCINIT
CSECT     $BSCARAM
CSECT     TPINIT
CSECT     TIMRINIT
CSECT     SBIOINIT
CSECT     EXIOINIT
CSECT     CLOKINIT
```

APPENDIX D. PROGRAM PREPARATION EXAMPLE

This four-part appendix contains a detailed example of how to code and prepare an interactive terminal program.

1. Part I shows the development of an Event Driven Language program. The program displays a terminal screen generated by the $IMAGE utility and accepts operator input.

2. Part II shows the full screen image being defined and stored in a data set using the $IMAGE utility program.

3. Part III shows the program being prepared for execution using the session manager to invoke the program preparation utilities. The program is compiled, listed, link-edited with the system-supplied subroutines, and converted to an executable load module.

4. Part IV shows a batch job stream procedure being used to prepare the program for execution. This step duplicates the processing done in part III for illustrative purposes.

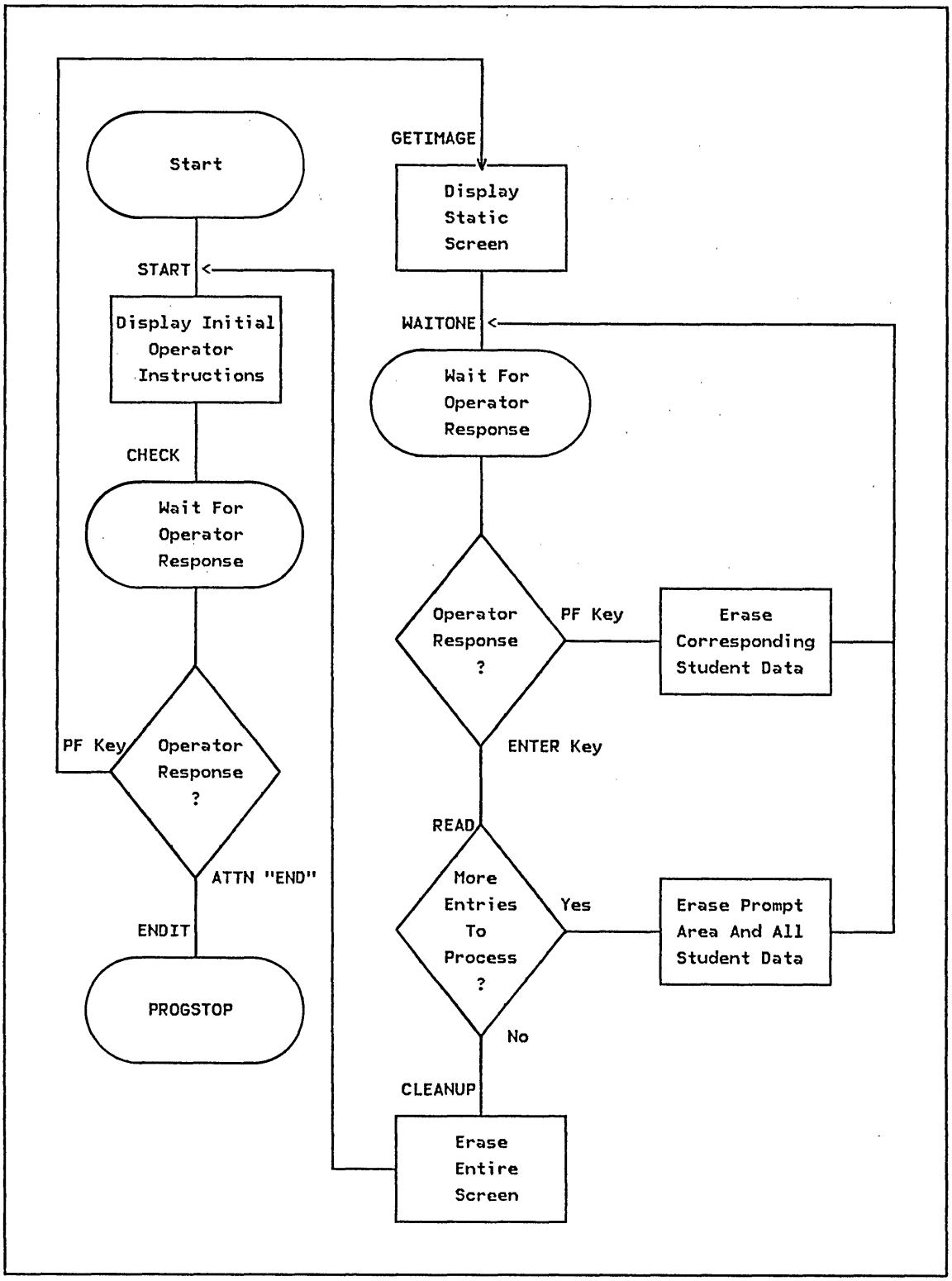


Figure 51. Flowchart of program operations

PART I. TERMINAL PROGRAM CODING EXAMPLE

In this part of the appendix, a sample program is developed. The program formats the display screen of the terminal used to load it and accepts data entered onto the screen by the operator. Figure 51 on page 368 depicts the operations performed by the program.

Processing the Initial Operator Instructions

The initial portion of the program displays instructions requiring the operator to (1) end the program, or (2) bring up the data entry screen (static-screen) and proceed. To obtain the operator's decision, the program uses the ATTNLIST facility, so an ATTNLIST statement is required.

This portion of the program operates the terminal in roll-screen mode, with no history lines defined (NHIST=0). The rest of the program uses the terminal in static-screen mode. A separate IOCB is required for each mode. Figure 52 shows the two IOCB statements, the ATTNLIST statement, and the associated attention routines.

```
XMPLSTAT PROGRAM   START
IOCB1    IOCB      NHIST=0
IOCB2    IOCB      SCREEN=STATIC
         ATTNLIST  (END,OUT,$PF,STATIC)
START    .
         .
OUT      POST      ATTNECB,1
         ENDATTN
STATIC   POST      ATTNECB,-1
         ENDATTN
         .
         .
         ENDPROG
         END
```

Figure 52. Code for IOCB's and attention handlers

Displaying the Initial Operator Instructions

The portion of the program that processes the initial operator instructions is shown in Figure 53: Execution begins at location START. The ENQT directed to IOCB1 changes NHIST=12 to NHIST=0 (the terminal is assumed to be a 4979 with NHIST=12 normally in effect).

Note: Because no terminal name is specified in the IOCB, the terminal enqueued defaults to the terminal used to load the program.

The five PRINTEXT statements following the ENQT statement display the program title and initial operator instructions on the screen. Because operator control has been defined through an ATTNLIST, and ATTNLIST is inhibited while the terminal is enqueued, the last PRINTEXT is followed by a DEQT, which reestablishes the ATTNLIST.

```
XMPLSTAT PROGRAM   START
IOCB1    IOCB      NHIST=0
IOCB2    IOCB      SCREEN=STATIC
         ATTNLIST  (END,OUT,$PF,STATIC)
START    ENQT      IOCB1
         PRINTEXT  'CLASS ROSTER PROGRAM',SPACES=15,LINE=0
         PRINTEXT  'HIT ''ATTN'' AND ENTER ''END'' TO END',SKIP=2
         PRINTEXT  ' THE PROGRAM'
         PRINTEXT  'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
         PRINTEXT  ' BRING UP THE ENTRY SCREEN'
         DEQT
CHECK    WAIT      ATTNECB,RESET
         IF        (ATTNECB,EQ,1),GOTO,ENDIT
         .
         .
ENDIT    PROGSTOP
         .
         .
OUT      POST      ATTNECB,1
         ENDATTN
STATIC   POST      ATTNECB,-1
         ENDATTN
ATTNECB  ECB
         .
         .
```

Figure 53. Code to process initial operator instructions

Waiting for an Operator Response

The ECB at location ATTNECB compiles with an initial value in the first word of -1, indicating "event complete". The WAIT at location CHECK is coded with a RESET operand, which resets the first word of the ECB at ATTNECB to zero before the WAIT is executed. A zero in the first word of an ECB indicates "event not occurred", so the WAIT at CHECK suspends task XMPLSTAT, waiting on event ATTNECB.

Note: If the WAIT had been coded without the RESET operand, it would have executed as a no operation.

After the program title and initial operator instructions have been written to the terminal (while the program is waiting at CHECK for the operator response), the screen looks like Figure 54.

```
LINE
NUMBER

  0 |               CLASS ROSTER PROGRAM
  1 |
  2 |HIT 'ATTN' AND ENTER 'END' TO END THE PROGRAM
  3 |
  4 |HIT ANY PROGRAM FUNCTION KEY TO BRING UP THE ENTRY SCREEN
  5 |_
  6 |
  7 |
  8 |
  9 |
 10 |
 11 |
 12 |
 13 |
 14 |
 15 |
 16 |
 17 |
 18 |
 19 |
 20 |
 21 |
 22 |
 23 |

CHAR  00000000011111111112222222222333333333344444444455555555556666666666777777777 8
POS   12345678901234567890123456789012345678901234567890123456789012345678901234567890
```

Figure 54. Screen showing initial operator instructions

Processing the Operator Response

If the operator presses the ATTN key, enters "END" and presses the ENTER key, the attention routine at OUT executes, posting the first word of the ECB at ATTNECB with a +1. Because a value other than zero in the first word of the ECB indicates "event complete", the WAIT operation terminates. Execution continues with the IF statement following the WAIT, which transfers control to location ENDIT.

If the operator wants to proceed with the CLASS ROSTER PROGRAM and presses a PF key, the attention routine at STATIC posts ATTNECB with a value of -1. The WAIT terminates, the IF that follows does not transfer control to ENDIT (because ATTNECB is not = +1), and execution continues with the static-screen portion of the program.

Formatting the Static Screen Image

Figure 55 shows the static-screen image that is used by the program.

```
LINE
NUMBER

  0
  1  ENTER KEY = PAGE COMPLETE     PF1 = DELETE ENTRY 1     PF2 = DELETE ENTRY 2
  2  PF3 = DELETE ENTRY 3          PF4 = DELETE ENTRY 4
  3  ------------------------------------------------------------------------------
  4  CLASS NAME: _                   INSTRUCTOR NAME:
  5  ------------------------------------------------------------------------------
  6  NAME:                         STREET:
  7                                CITY  :
  8                                STATE :
  9
 10
 11  NAME:                         STREET:
 12                                CITY  :
 13                                STATE :
 14
 15
 16  NAME:                         STREET:
 17                                CITY  :
 18                                STATE :
 19
 20
 21  NAME:                         STREET:
 22                                CITY  :
 23                                STATE :

CHAR  00000000011111111112222222222333333333344444444455555555556666666666777777777 8
POS   1234567890123456789012345678901234567890123456789012345678901234567890123456789 0
```

Figure 55. Static-screen image used by program

Note: The image is defined and saved in a data set in part II.

The statements required to access the stored screen image are explained in the following pages.

Reading the Stored Screen Image ($IMOPEN Subroutine)

Refer to Figure 56 for the following discussion.

The first step in using a stored screen image is to read the image data set into the application program. The $IMOPEN subroutine reads the data set into the buffer at IMAGEBUF. The name of the data set and volume is specified in a TEXT statement, and the label of the TEXT statement is passed to $IMOPEN as the first parameter in the CALL. The second parameter is the label of the buffer which is to receive the image. Both parameters must be enclosed in parentheses. The buffer is defined by a BUFFER statement. Data set VIDEO1 will be two records in length, so IMAGEBUF is defined as 512 bytes.

$IMOPEN returns a code in "taskname+2". The application program has the responsibility to check for proper completion (-1 return code). The example program includes a completion code check and error routine.

```
GETIMAGE CALL     $IMOPEN,(DSETNAME),(IMAGEBUF)
         IF       (XMPLSTAT+2,NE,-1)
           MOVE     ERRCODE,XMPLSTAT+2
           PRINTEXT 'ɑIMAGE OPEN ERROR, CODE ='
           PRINTNUM ERRCODE
           QUESTION 'ɑRETRY OPEN ? ',YES=GETIMAGE,NO=ENDIT
         ENDIF
         .
         .
ERRCODE  DATA     F'0'
IMAGEBUF BUFFER   512,BYTES
DSETNAME TEXT     'VIDEO1,EDX002'
```

Figure 56. Coding to read stored screen image

Setting IOCB Dimensions ($IMDEFN Subroutine)

Figure 57 on page 376 shows a CALL to subroutine $IMDEFN which fills in the specified IOCB with the dimensions of the screen image in the buffer. The CALL to $IMDEFN is not a required function; the IOCB may be enqueued without first calling the subroutine. By calling $IMDEFN, you are assured that the IOCB has the proper dimensions for the screen in the buffer. This enables you to change the dimensions in the stored screen image (using $IMAGE) without having to change the program.

Transferring the Stored Image to the Screen ($IMPROT/$IMDATA Subroutines)

Refer to Figure 57 on page 376 for the following discussion.

Before the screen can be displayed, the terminal must be enqueued as a static-screen device. The ENQT IOCB2 instruction accomplishes that.

Now that the terminal is enqueued, the screen image in the buffer can be displayed. The TERMCTRL BLANK following the ENQT blanks the screen, preventing flicker while the image is written. The CALL of subroutine $IMPROT transfers all the protected data from the image buffer to the screen, and the call to $IMDATA transfers the unprotected data.

Note: If a screen image consists of all protected or all unprotected data, only the appropriate subroutine need be called.

The PRINTEXT following the last CALL positions the cursor at the first data entry field, and TERMCTRL DISPLAY displays the screen.

```
         EXTRN     $IMOPEN,$IMDEFN,$IMPROT,$IMDATA
IOCB2    IOCB      SCREEN=STATIC
         .
         .
GETIMAGE CALL      $IMOPEN,(DSETNAME),(IMAGEBUF)
         IF        (XMPLSTAT+2,NE,-1)
           MOVE      ERRCODE,XMPLSTAT+2
           PRINTEXT  '@IMAGE OPEN ERROR, CODE ='
           PRINTNUM  ERRCODE
           QUESTION  '@RETRY OPEN ? ',YES=GETIMAGE,NO=ENDIT
         ENDIF
         CALL      $IMDEFN,(IOCB2),(IMAGEBUF)
         ENQT      IOCB2
         TERMCTRL  BLANK
         CALL      $IMPROT,(IMAGEBUF),0
         CALL      $IMDATA,(IMAGEBUF)
         PRINTEXT  LINE=4,SPACES=12
         TERMCTRL  DISPLAY
         .
         .
ERRCODE  DATA      F'0'
IMAGEBUF BUFFER    512,BYTES
DSETNAME TEXT      'VIDEO1,EDX002'
```

Figure 57. Code to transfer stored image to screen

The second parameter of the CALL $IMPROT statement (Figure 57) is coded as 0. This could be coded as the label of a BUFFER statement, in which case the $IMPROT subroutine builds a table of the location and sizes of all unprotected (data entry) fields on the screen. Each table entry is three words long. The first word contains the line number, and the second, the starting position of the field within the line (spaces from left margin of screen). The third word contains the length of the field. These entries can be used to read or write data entry fields on the screen.

For example, in Figure 58 on page 377, FIELDS contains the line, spaces, and size of the first data entry field. PRINTEXT positions the cursor, and TERMCTRL displays it at the first field, just as did the PRINTEXT/TERMCTRL pair in Figure 57. If the starting point of the first data entry field is changed ($IMAGE used to redefine the screen image), the program shown in Figure 57 would have to be changed, or the cursor would not be positioned properly. The program in Figure 58 on page 377 would pick up the new starting field location without any program modification required.

```
         CALL        $IMPROT,(IMAGEBUF),(FIELDS)
         PRINTEXT    LINE=FIELDS,SPACES=FIELDS+2
         TERMCTRL,   DISPLAY


FIELDS   BUFFER      3
```

Figure 58. Alternate coding technique

Using the Image Formatting Subroutines

The "$IM" subroutines are supplied as object modules on diskette XS1001 or XS2001. They are normally loaded into the volume ASMLIB.

Because they are object modules, they are combined with the main program during the link edit step, not during compilation. They must therefore be declared as external references in an EXTRN statement in the main program as shown in Figure 57 on page 376.

Processing Operator Input

Accepting Operator Input

The operator may position the cursor and enter data in any unprotected area of the screen. The program, by positioning the cursor at LINE=4, SPACES=12 (with the PRINTEXT following the CALL $IMDATA), provides a convenience to the operator, not a required function - the operator could have used the cursor positioning keys to move the cursor to the same position.

The tab right key is useful in controlling cursor movement. Assume that the operator enters "SERIES/1 HARDWARE" in the space following the protected "CLASS NAME:" message, and then presses the tab right key (-->|). The cursor automatically skips over the protected "INSTRUCTOR NAME:" field, and positions itself at the beginning of the unprotected area which follows.

If the operator presses the tab right key after entering the instructor name, the cursor moves to accept the first student name entry. Each time the operator presses the tab key, the cursor moves to the beginning of the next unprotected area on the screen. The cursor successively tabs to "NAME:", "STREET:", "CITY:", and "STATE:", and then down to the "NAME:" in the next data entry area.

Without program interaction, the operator can enter an entire screen of information and transfer it at one time. This is what is meant by static-screen operation, in contrast to the transactional prompt/reply dialogue typical of roll-screen operation.

A completed input screen is shown in Figure 59. The screen is now at the point where the program must be signalled to process the data entered.

```
LINE
NUMBER

 0
 1  ENTER KEY = PAGE COMPLETE     PF1 = DELETE ENTRY 1     PF2 = DELETE ENTRY 2
 2  PF3 = DELETE ENTRY 3          PF4 = DELETE ENTRY 4
 3  ------------------------------------------------------------------------------
 4  CLASS NAME: SERIES/1 HARDWARE   INSTRUCTOR NAME: JOHN JONES
 5  ------------------------------------------------------------------------------
 6  NAME: AL BROWN                STREET: 111 GRANT AVENUE
 7                                CITY  : ENDICOTT
 8                                STATE : NEW YORK 13760
 9
10
11  NAME: BILL SMITH              STREET: 255 ALHAMBRA CIRCLE
12                                CITY  : CORAL GABLES
13                                STATE : FLORIDA 33135
14
15
16  NAME: JOE STANTON             STREET: 140 EAST TOWN STREET
17                                CITY  : COLUMBUS
18                                STATE : OHIO 43215
19
20
21  NAME: LINDA GREEN             STREET: 6216 WASHINTON AVENUE
22                                CITY  : RACINE
23                                STATE : WISCONSIN 53406_

CHAR  0000000001111111111222222222233333333334444444445555555556666666666777777778
POS   1234567890123456789012345678901234567890123456789012345678901234567890123456789
```

Figure 59. Screen with all data entered

In Figure 60, the WAIT KEY instruction at WAITONE terminates when the operator presses the ENTER key or a PF key. The computed GOTO following the WAIT KEY transfers control to various entry points, depending on the return code in "taskname+2". A return code of zero results from use of the ENTER key, causing a transfer to location READ. PF1 through PF4 return codes of 1 through 4, and result in transfers to E1 through E4, respectively (not shown). With the GOTO coded as shown, a PF key higher than PF4 causes a transfer to READ, because the the return code is outside the valid range of index values 1-4. The zero returned by the ENTER key is also outside that range, and results in a transfer to READ.

Assume that the operator presses the ENTER key, which signals the program that the page is complete and transfers control to READ. In an actual application program, the routine at location READ would contain the READTEXT instructions necessary to read all the data entered on the screen. The data would presumably be collected and used to print a class roster for the "SERIES/1 HARDWARE" course taught by "JOHN JONES".

```
XMPLSTAT PROGRAM   START
         .
         .
START    ENQT      IOCB1
         .
         .
WAITONE  WAIT      KEY
         GOTO      (READ,E1,E2,E3,E4),XMPLSTAT+2
         .
         .
READ     QUESTION  'MORE ENTRIES ?',LINE=2,SPACES=55,NO=CLEANUP
         ERASE     MODE=LINE,LINE=2,SPACES=55,TYPE=DATA
         ERASE     MODE=SCREEN,LINE=6
         PRINTEXT  LINE=6,SPACES=6
         TERMCTRL  DISPLAY
         GOTO      WAITONE
CLEANUP  ERASE     MODE=SCREEN,TYPE=ALL
         DEQT
         GOTO      START
         .
         .
         ENDPROG
         END
```

Figure 60. Code to process ENTER key

Determining if More Data is to be Entered

Assuming that the contents of the screen has been read, the QUESTION instruction at READ displays the prompt message "MORE ENTRIES ?" in the operator prompt area at the upper right of the screen, as shown in Figure 61.

```
LINE
NUMBER

  0
  1  ENTER KEY = PAGE COMPLETE     PF1 = DELETE ENTRY 1     PF2 = DELETE ENTRY 2
  2  PF3 = DELETE ENTRY 3          PF4 = DELETE ENTRY 4     MORE ENTRIES ?_
  3  ------------------------------------------------------------------------------
  4  CLASS NAME: SERIES/1 HARDWARE   INSTRUCTOR NAME: JOHN JONES
  5  ------------------------------------------------------------------------------
  6  NAME: AL BROWN                 STREET: 111 GRANT AVENUE
  7                                 CITY  : ENDICOTT
  8                                 STATE : NEW YORK 13760
  9
 10
 11  NAME: BILL SMITH               STREET: 255 ALHAMBRA CIRCLE
 12                                 CITY  : CORAL GABLES
 13                                 STATE : FLORIDA 33135
 14
 15
 16  NAME: JOE STANTON              STREET: 140 EAST TOWN STREET
 17                                 CITY  : COLUMBUS
 18                                 STATE : OHIO 43215
 19
 20
 21  NAME: LINDA GREEN              STREET: 6216 WASHINTON AVENUE
 22                                 CITY  : RACINE
 23                                 STATE : WISCONSIN 53406

CHAR  0000000001111111111222222222233333333344444444455555555556666666666777777777 8
POS   1234567890123456789012345678901234567890123456789012345678901234567890123456789 0
```

Figure 61. Screen contents after ENTER key is used

The "MORE ENTRIES ?" query is asking the operator, "Are there more students to add to this roster, or are the students just read from the the current screen the last ones at this time?"

Processing if More Data is to be Entered

Refer to Figure 60 on page 379 for the following discussion. Assume that more students are to be enrolled and "YES" is the response. Because YES= is not coded on the QUESTION statement, a response of "YES" results in execution of the ERASE instruction following the QUESTION. The first ERASE following the QUESTION clears the prompt and reply from the operator prompt area, and the second ERASE clears all unprotected data from the four data entry areas in lines 6 through 23. The "SERIES/1 HARDWARE" and "JOHN JONES" entries in the header area are left undisturbed, because the student names and addresses to be entered are still for the same class. The PRINTEXT following the second ERASE positions the cursor at the first unprotected entry field for the first data entry area. The TERMCTRL DISPLAY that follows displays the cursor, resulting in the screen shown in Figure 62.

```
LINE
NUMBER

  0
  1  ENTER KEY = PAGE COMPLETE     PF1 = DELETE ENTRY 1     PF2 = DELETE ENTRY 2
  2  PF3 = DELETE ENTRY 3          PF4 = DELETE ENTRY 4
  3  ------------------------------------------------------------------------------
  4  CLASS NAME: SERIES/1 HARDWARE   INSTRUCTOR NAME: JOHN JONES
  5  ------------------------------------------------------------------------------
  6  NAME: _                       STREET:
  7                                CITY  :
  8                                STATE :
  9
 10
 11  NAME:                         STREET:
 12                                CITY  :
 13                                STATE :
 14
 15
 16  NAME:                         STREET:
 17                                CITY  :
 18                                STATE :
 19
 20
 21  NAME:                         STREET:
 22                                CITY  :
 23                                STATE :

CHAR  00000000011111111112222222223333333334444444445555555556666666667777777778
POS   12345678901234567890123456789012345678901234567890123456789012345678901234567890
```

Figure 62. Screen contents after reply of YES to QUESTION

Processing if No More Data is to be Entered

If there are no more students to enter for this roster, and the response to the "MORE ENTRIES ?" prompt is "NO", the QUESTION statement (Figure 60 on page 379) transfers control to location CLEANUP. There, the program erases both protected and unprotected areas of the entire screen, dequeues the terminal, and goes back to the beginning of the program (START), bringing up the roll-screen with the initial operator instructions, as shown in Figure 63.

```
LINE
NUMBER
  0   |               CLASS ROSTER PROGRAM
  1   |
  2   |HIT 'ATTN' AND ENTER 'END' TO END THE PROGRAM
  3   |
  4   |HIT ANY PROGRAM FUNCTION KEY TO BRING UP THE ENTRY SCREEN
  5   |_
  6   |
  7   |
  8   |
  9   |
 10   |
 11   |
 12   |
 13   |
 14   |
 15   |
 16   |
 17   |
 18   |
 19   |
 20   |
 21   |
 22   |
 23   |

CHAR  0000000001111111111222222222233333333334444444444555555555566666666667777777778
POS   1234567890123456789012345678901234567890123456789012345678901234567890123456789
```

Figure 63. Screen contents after reply of NO to QUESTION

Processing the Program Function Keys

In Figure 64, assume the program is again suspended by the WAIT KEY at WAITONE, with the complete screen depicted in Figure 59 on page 378. The transfer to location READ and the "MORE ENTRIES ?" prompt from the QUESTION statement resulted from the operator pressing the ENTER key.

However, the WAIT KEY instruction may also be terminated by a PF key. No PF key functions are preassigned other than the hardcopy facility (PF6). Therefore, the purpose of a particular PF key in any program is defined by the instructions coded in the routine to which control is transferred when that PF key is pressed.

The PF1 through PF4 keys have been assigned by this program as delete functions for the four data entry areas, as shown by the operator prompts at the top of the screen (see Figure 59 on page 378).

```
XMPLSTAT PROGRAM   START
         .
         .
WAITONE  WAIT      KEY
         GOTO      (READ,E1,E2,E3,E4),XMPLSTAT+2
E1       MOVE      LINENBR,6
         GOTO      DELETE
E2       MOVE      LINENBR,11
         GOTO      DELETE
E3       MOVE      LINENBR,16
         GOTO      DELETE
E4       MOVE      LINENBR,21
DELETE   ERASE     MODE=LINE,TYPE=DATA,LINE=LINENBR
         ADD       LINENBR,1
         ERASE     MODE=LINE,TYPE=DATA,LINE=LINENBR
         ADD       LINENBR,1
         ERASE     MODE=LINE,TYPE=DATA,LINE=LINENBR
         SUBTRACT  LINENBR,2
         PRINTEXT  LINE=LINENBR,SPACES=6
         TERMCTRL  DISPLAY
         GOTO      WAITONE
         .
         .
LINENBR  DATA      F'0'
         ENDPROG
         END
```

Figure 64. Code to process the PF keys

Assume that for some reason, the student "JOE STANTON", the third entry on the screen, is not supposed to be on the class roster; the operator, therefore, presses PF3.

In Figure 64 on page 383, the PF key terminates the WAIT KEY instruction, and the computed GOTO transfers control to E3. The MOVE at E3 initializes the LINENBR variable to 16, which is the top line of the third data entry area. Control is then transferred to DELETE, where successive ERASE operations and adjustments of the LINENBR variable result in erasure of the unprotected portions of the third data entry area. Before returning to the WAIT KEY, the cursor is positioned and displayed at the first entry field of the erased data area, as shown in Figure 65.

```
LINE
NUMBER

  0
  1  ENTER KEY = PAGE COMPLETE     PF1 = DELETE ENTRY 1     PF2 = DELETE ENTRY 2
  2  PF3 = DELETE ENTRY 3          PF4 = DELETE ENTRY 4
  3  ------------------------------------------------------------------------------
  4  CLASS NAME: SERIES/1 HARDWARE   INSTRUCTOR NAME: JOHN JONES
  5  ------------------------------------------------------------------------------
  6  NAME: AL BROWN                STREET: 111 GRANT AVENUE
  7                                CITY  : ENDICOTT
  8                                STATE : NEW YORK 13760
  9
 10
 11  NAME: BILL SMITH              STREET: 255 ALHAMBRA CIRCLE
 12                                CITY  : CORAL GABLES
 13                                STATE : FLORIDA 33135
 14
 15
 16  NAME: _                       STREET:
 17                                CITY  :
 18                                STATE :
 19
 20
 21  NAME: LINDA GREEN             STREET: 6216 WASHINTON AVENUE
 22                                CITY  : RACINE
 23                                STATE : WISCONSIN 53406

CHAR  0000000001111111111222222222233333333334444444445555555556666666666777777777 8
POS   1234567890123456789012345678901234567890123456789012345678901234567890123456789
```

Figure 65. Screen contents after PF3 is used

The program preparation descriptions in part III and part IV assume that this program is stored in the data set STATSRC in volume EDX002.

Figure 66 and Figure 67 on page 386 are a listing of the complete program.

```
XMPLSTAT PROGRAM   START
         EXTRN     $IMOPEN,$IMDEFN,$IMPROT,$IMDATA
IOCB1    IOCB      NHIST=0
IOCB2    IOCB      SCREEN=STATIC
         ATTNLIST  (END,OUT,$PF,STATIC)
START    ENQT      IOCB1
         PRINTEXT  'CLASS ROSTER PROGRAM',SPACES=15,LINE=0
         PRINTEXT  'HIT ''ATTN'' AND ENTER ''END'' TO END',SKIP=2
         PRINTEXT  ' THE PROGRAM'
         PRINTEXT  'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
         PRINTEXT  ' BRING UP THE ENTRY SCREEN'
         DEQT
CHECK    WAIT      ATTNECB,RESET
         IF        (ATTNECB,EQ,1),GOTO,ENDIT
GETIMAGE CALL      $IMOPEN,(DSETNAME),(IMAGEBUF)
         IF        (XMPLSTAT+2,NE,-1)
           MOVE      ERRCODE,XMPLSTAT+2
           PRINTEXT  '@IMAGE OPEN ERROR,CODE ='
           PRINTNUM  ERRCODE
           QUESTION  '@RETRY OPEN ? ',YES=GETIMAGE,NO=ENDIT
         ENDIF
         CALL      $IMDEFN,(IOCB2),(IMAGEBUF)
         ENQT      IOCB2
         TERMCTRL  BLANK
         CALL      $IMPROT,(IMAGEBUF),0
         CALL      $IMDATA,(IMAGEBUF)
         PRINTEXT  LINE=4,SPACES=12
         TERMCTRL  DISPLAY
WAITONE  WAIT      KEY
         GOTO      (READ,E1,E2,E3,E4),XMPLSTAT+2
E1       MOVE      LINENBR,6
         GOTO      DELETE
E2       MOVE      LINENBR,11
         GOTO      DELETE
E3       MOVE      LINENBR,16
         GOTO      DELETE
E4       MOVE      LINENBR,21
DELETE   ERASE     MODE=LINE,TYPE=DATA,LINE=LINENBR
         ADD       LINENBR,1
         ERASE     MODE=LINE,TYPE=DATA,LINE=LINENBR
         ADD       LINENBR,1
         ERASE     MODE=LINE,TYPE=DATA,LINE=LINENBR
```

Figure 66. Complete program (Part 1 of 2)

```
         SUBTRACT  LINENBR,2
         PRINTEXT  LINE=LINENBR,SPACES=6
         TERMCTRL  DISPLAY
         GOTO      WAITONE
READ     QUESTION  'MORE ENTRIES ?',LINE=2,SPACES=55,NO=CLEANUP
         ERASE     MODE=LINE,LINE=2,SPACES=55,TYPE=DATA
         ERASE     MODE=SCREEN,LINE=6
         PRINTEXT  LINE=6,SPACES=6
         TERMCTRL  DISPLAY
         GOTO      WAITONE
CLEANUP  ERASE     MODE=SCREEN,TYPE=ALL
         DEQT
         GOTO      START
ENDIT    PROGSTOP
OUT      POST      ATTNECB,1
         ENDATTN
STATIC   POST      ATTNECB,-1
         ENDATTN
ATTNECB  ECB
LINENBR  DATA      F'0'
ERRCODE  DATA      F'0'
IMAGEBUF BUFFER    512,BYTES
DSETNAME TEXT      'VIDEO1,EDX002'
         ENDPROG
         END
```

Figure 67. Complete program (Part 2 of 2)

PART II. DEFINE FORMATTED SCREEN IMAGE USING $IMAGE

Part I of this appendix showed the development of a sample terminal program which formatted a static-screen using system-supplied subroutines to access the screen image defined using the $IMAGE utility.

This part of the appendix is a description of the $IMAGE utility session in which the static-screen image used by the example program is created and stored in a data set. The screen image that is created in this utility session is shown in Figure 55 on page 373 with the exception that the cursor does not appear on the screen defined using $IMAGE; it is displayed by the application program.

$IMAGE is used to create formatted screen images for use with terminals that support static screen functions. The image (formatted screens) is stored in a disk or diskette data set for later retrieval by application programs. $IMAGE can also retrieve stored images for modification.

Creating the Image Data Set

You must allocate a disk or diskette data set to store the formatted screen image created by $IMAGE. The formatting information and text are stored in a special packed format to conserve space. A stored screen may be any size from one character position up to an entire physical screen, and therefore the amount of space on disk or diskette required to store a given screen image varies. For most stored screens, a data set two records in length is adequate.

Because the screen image to be created encompasses an entire physical screen and contains several lines of text, a data set two records in length is required to store it.

Before beginning the $IMAGE utility session, a data set two records long, named VIDEO1 is created using $DISKUT1. Figure 68 shows the data set creation sequence.

```
> $L $DISKUT1
$DISKUT1       26P,00:32:06, LP= 5F00

USING VOLUME EDX002
COMMAND (?): AL VIDEO1 2
DEFAULT TYPE = DATA - OK? YES
VIDED01 CREATED

COMMAND  (?): END
DISKUT1 ENDED AT 00:32:33
```

Figure 68. Allocation of screen image data set (VIDEO1)

Loading $IMAGE and Entering Commands

Now the $IMAGE utility can be loaded, and the utility session begun. Figure 69 shows the commands used. Entering a "?" in response to the "COMMAND (?):" prompt results in a display of the available $IMAGE commands.

```
> $L $IMAGE
$IMAGE          37P,00:32:06, LP= 5F00

COMMAND  (?): ?

      DIMS -- DEFINE IMAGE DIMENSIONS
      HTAB -- DEFINE HORIZONTAL TAB SETTINGS
      VTAB -- DEFINE VERTICAL TAB SETTINGS
      NULL -- DEFINE NULL REPRESENTATIION
      EDIT -- ENTER EDIT MODE
      KEYS __ PROGRAM FUNCTION KEYS
      SAVE __ SAVE IMAGE ON DISK
      END  -- END PROGRAM

COMMAND  (?): DIMS 24 80
COMMAND  (?): HTAB 1 31
COMMAND  (?): NULL /
COMMAND  (?): EDIT
```

Figure 69. $IMAGE commands

All of the commands listed in Figure 69 may be entered in command mode only. They are not available in edit mode.

The DIMS command allows you to define the dimensions of the logical screen you are creating. The example shows a logical screen of 24 lines and 80 characters specified, which is equal to the entire physical screen.

HTAB is the horizontal tab settings you wish to have in effect while you are creating the screen. If not entered, HTAB defaults to 10,20,30,40,50,60,70. The example defines horizontal tab settings of 1 and 31. Those tab settings allow you to position the cursor to the corresponding display positions with the PF1 key in edit mode.

VTAB defines vertical tabs. The default is one vertical line for each vertical tab key usage. Since VTAB is not entered in this example, one-line vertical tabs will be in effect. The edit mode vertical tab key (PF2) moves the cursor down and to the column position of the last horizontal tab used.

The NULL command allows you to define the null character. When

in edit mode, you enter a null character in each character position in which you want to display unprotected data, or which is to accept data entered by the operator. The example defines the null character to be a slash (/).

The KEYS command lists the functions of PF1, PF2, and PF3 immediately after edit mode is entered. Refer to Figure 70.

```
PF1-define protected fields
PF2-define data fields (unprotected)
PF3-return to command mode
```

Figure 70. $IMAGE PF key functions upon edit mode entry

Creating the Image

Entering edit mode

The last command entered is EDIT, which places the $IMAGE utility in edit mode. If an existing screen image were to be edited, the data set name and volume containing that image would be entered with the EDIT command. Because this $IMAGE session is creating a new screen, EDIT is entered without reference to a data set. Before pressing any of the PF keys, the screen is entirely blank, and the cursor is in the lower left corner.

Protected and Unprotected Fields

The screen being created in this example contains both protected and unprotected data. The operator prompts on lines 1 and 2 are unprotected, and the rest of the fields are protected (see Figure 55 on page 373.)

Note: When the completed screen is displayed, the unprotected areas appear brighter than those that are protected, highlighting the prompts at the top of the screen.

Defining the Protected Fields

When both protected and unprotected text is to appear on a screen created by $IMAGE, you must enter the protected data first. Therefore press PF1 to signal the utility that protected fields are to be defined. The cursor moves to the first available character position, which is line 0, space 0, in this example.

As soon as either PF1 or PF2 is pressed, after entering edit mode, the function of PF1 and PF2 is redefined. PF1 is then used as the horizontal tab key and PF2 as the vertical tab key. Since no text appears on line 0, press the vertical tab key PF2 to move the cursor down to the first position of line 1.

When you define the protected areas of a screen image, all characters entered, other than the null character, are protected data. All areas of the screen not containing null characters will be protected when the screen is completed. The operator prompts on lines 1 and 2 are supposed to be unprotected. Therefore, the actual text of the prompts cannot be entered until the unprotected data definition portion of this utility session, which occurs after all protected fields have been defined. However, since these areas of the screen will contain unprotected text, null character fields must be entered; this allows the text entered to be accepted when the unprotected data definition is done.

Now format the rest of the screen. Note that any field meant to receive operator input when the screen is used must be defined using the null character.

Figure 71 shows the screen contents after all protected fields have been defined.

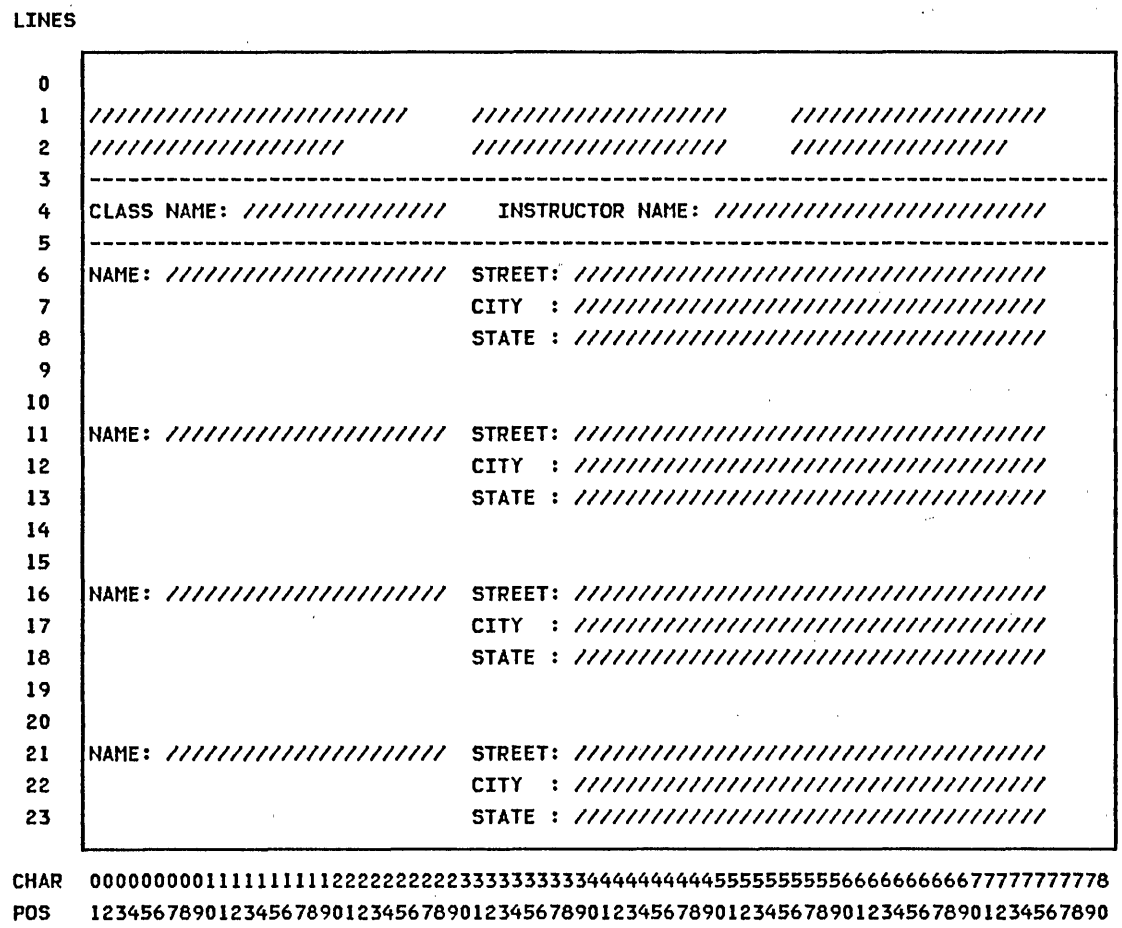


Figure 71. Screen with protected and null fields defined

Press the ENTER key to take the utility out of protected field definition, (back to the situation as it was before a define protected field or define unprotected field decision was made). PF1 and PF2 again have the meanings printed out by the KEYS command (refer to Figure 70 on page 390).

The ENTER key also causes the protected areas of the screen defined up to this point to be displayed as shown in Figure 72.

```
LINES
  0
  1
  2
  3  --------------------------------------------------------------------------------
  4  CLASS NAME:                    INSTRUCTOR NAME:
  5  --------------------------------------------------------------------------------
  6  NAME:                          STREET:
  7                                 CITY  :
  8                                 STATE :
  9
 10
 11  NAME:                          STREET:
 12                                 CITY  :
 13                                 STATE :
 14
 15
 16  NAME:                          STREET:
 17                                 CITY  :
 18                                 STATE :
 19
 20
 21  NAME:                          STREET:
 22                                 CITY  :
 23                                 STATE :

CHAR 00000000011111111112222222222333333333344444444455555555556666666666777777778
POS  12345678901234567890123456789012345678901234567890123456789012345678901234567890
```

Figure 72. Screen contents after ENTER key use

Entering the Unprotected Output Fields

If the desired screen image were now complete, you would press PF3 to reenter command mode, so that the image could be saved. In this example, however, there is still unprotected data to be defined (on lines one and two), so press PF2. Pressing PF2 allows unprotected fields to be entered and brings back the same screen image as in Figure 71 on page 392, with the unprotected fields defined as null characters.

Now fill in the unprotected null fields in the operator prompt area at the top of the screen. The other null fields are input fields that will be used by the application program, so they are left undisturbed.

After all unprotected text is defined, the screen looks like that shown in Figure 73.

```
LINES
  0
  1  ENTER KEY = PAGE COMPLETE     PF1 = DELETE ENTRY 1     PF2 = DELETE ENTRY 2
  2  PF3 = DELETE ENTRY 3          PF4 = DELETE ENTRY 4     ////////////////////
  3  ------------------------------------------------------------------------------
  4  CLASS NAME: /////////////////    INSTRUCTOR NAME: ////////////////////////////
  5  ------------------------------------------------------------------------------
  6  NAME: ////////////////////////  STREET: ////////////////////////////////////////
  7                                 CITY  : ////////////////////////////////////////
  8                                 STATE : ////////////////////////////////////////
  9
 10
 11  NAME: ////////////////////////  STREET: ////////////////////////////////////////
 12                                 CITY  : ////////////////////////////////////////
 13                                 STATE : ////////////////////////////////////////
 14
 15
 16  NAME: ////////////////////////  STREET: ////////////////////////////////////////
 17                                 CITY  : ////////////////////////////////////////
 18                                 STATE : ////////////////////////////////////////
 19
 20
 21  NAME: ////////////////////////  STREET: ////////////////////////////////////////
 22                                 CITY  : ////////////////////////////////////////
 23                                 STATE : ////////////////////////////////////////

CHAR  0000000001111111111222222222233333333334444444445555555556666666666777777778
POS   1234567890123456789012345678901234567890123456789012345678901234567890123456
```

Figure 73. Screen with unprotected fields defined

After you have entered the unprotected prompts on lines one and two, press the ENTER key. The completed screen is then displayed (see Figure 55 on page 373) If you want to make any changes to the screen, press PF1 to allow protected field entry or PF2 to allow unprotected field entry.

Saving the Image Created

Assuming that the image is correct, press PF3 to return to command mode. The screen is blanked and you are prompted for a command. Enter the SAVE command followed by the name of the data set that was allocated for this purpose. The $IMAGE utility session is ended.

```
COMMAND (?): SAVE VIDEO1
  SAVED (2 RECORDS)
COMMAND (?): END
```

Figure 74. Save screen image created and end $IMAGE

PART III. PREPARE PROGRAM USING SESSION MANAGER

In part I, a program was developed incorporating system-supplied subroutines to format the static-screen image used.

In part II, the $IMAGE screen formatting utility was used to create the screen, and save it in a screen image data set named VIDEO1.

In this part of the appendix, the program developed in part I is compiled, link-edited, and formatted. Each step is done using the session manager to invoke the various program preparation utilities.

Figure 75 shows a graphic overview of the program preparation process.

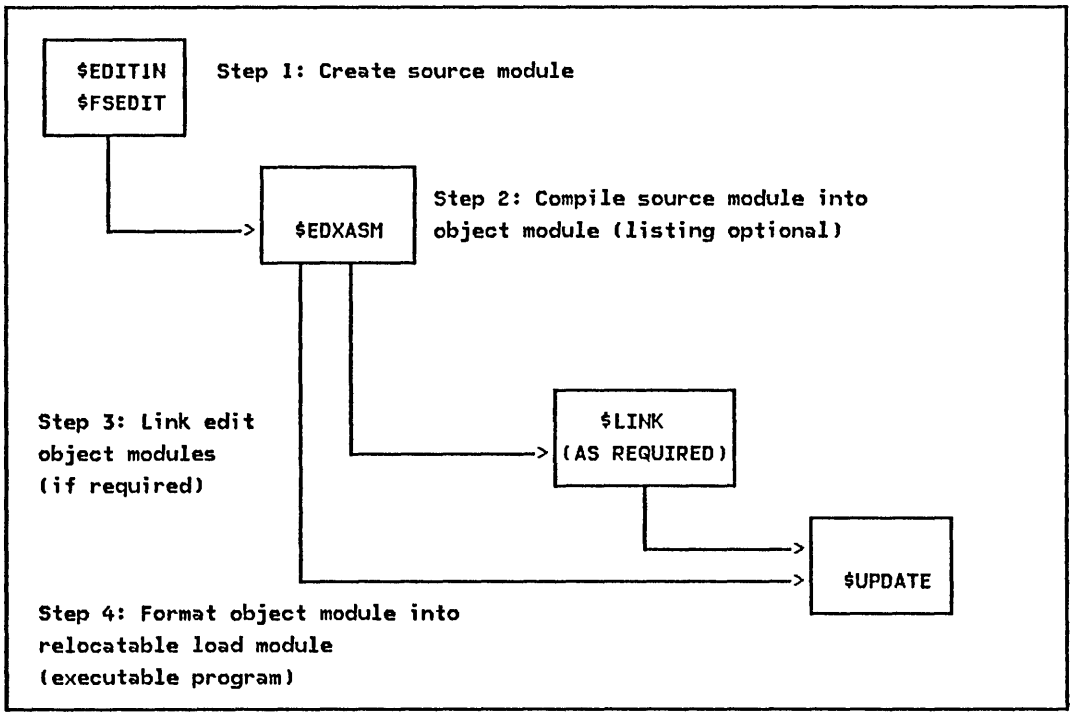


Figure 75. Program preparation steps

Step 1. Create Source Module Using $FSEDIT

The program to be prepared is shown in Figure 66 on page 385 and Figure 67 on page 386. The source program is assumed to be in a data set named STATSRC on volume EDX002.

Step 2. Compile Source Module Using $EDXASM

The names of the data sets and the volumes containing them that are used in the compilation step are:

	DATA SET NAME	VOLUME NAME
INPUT DATA SET	STATSRC	EDX002
OUTPUT DATA SET	ASMOBJ	EDX002

Note: You must allocate ASMOBJ if it does not already exist. It must be a data type member. A size of 100 records is adequate.

Figure 76 on page 398 shows the session manager display screen used to invoke the compiler.

Because no options are selected, $EDXASM produces a full compilation listing on the system printer. When the compiler finishes, it stores the resulting object module in ASMOBJ on volume EDX002. $EDXASM then loads $EDXLIST to produce the compilation listing.

```
$SMM0201: SESSION MANAGER $EDXASM PARAMETER INPUT MENU-------------------------
ENTER/SELECT PARAMETERS:                                  DEPRESS PF3 TO RETURN




    SOURCE INPUT  (NAME,VOLUME) ==> STATSRC,EDX002

    OBJECT OUTPUT (NAME,VOLUME) ==> ASMOBJ,EDX002

    ENTER OPTIONAL PARAMETERS BY POSITION ==>
                                            1--------2---------
                                            LIST     PRINTER NAME
                                            NOLIST
                                            ERRORS

                                            DEFAULTS ARE: LIST  $SYSPRTR
```

Figure 76. $EDXASM invocation

The compilation listing produced is shown in Figure 77 on page 399, Figure 78 on page 400, and Figure 79 on page 401.

```
  EDX ASSEMBLER STATISTICS

  SOURCE INPUT  - STATSRC ,EDX002
  WORK DATA SET - $SM1USER,EDX003
  OBJECT MODULE - ASMOBJ  ,EDX002
  DATE: 02/21/80 AT 16:24:57
  ASSEMBLY TIME:     26 SECONDS
  STATEMENTS PROCESSED -     70

  NO STATEMENTS FLAGGED

0000   0008 D7D9 D6C7 D9C1 D440  XMPLSTAT PROGRAM   START
000A   0000 04D4 0052 0000 0000
0014   05D6 0000 0000 0000 0100
001E   05D4 0000 0000 0000 0554
0028   0000
                                          EXTRN     $IMOPEN,$IMDEFN,$IMPROT,$IMDATA
002A   4040 4040 4040 4040 8000  IOCB1     IOCB      NHIST=0
0034   00FF 0000 7FFF 0000 0000
003E   4040 4040 4040 4040 8800  IOCB2     IOCB      SCREEN=STATIC
0048   00FF 0000 7FFF 0000 0000
0052   0002 0403 C5D5 C440 02A6            ATTNLIST  (END,OUT,$PF,STATIC)
005C   0403 5BD7 C640 02AE
0064   1025 002A                  START     ENQT      IOCB1
0068   B02A 0000 000F 8026 1414            PRINTEXT  'CLASS ROSTER PROGRAM',SPACES=15,LINE=0
0072   C3D3 C1E2 E240 D9D6 E2E3
007C   C5D9 40D7 D9D6 C7D9 C1D4
0086   902A 0002 0000 8026 2221            PRINTEXT  'HIT ''ATTN'' AND ENTER ''END'' TO END',SKIP=2
0090   C8C9 E340 7DC1 E3E3 D57D
009A   40C1 D5C4 40C5 D5E3 C5D9
00A4   407D C5D5 C47D 40E3 D640
00AE   C5D5 C440
00B2   8026 0C0C 40E3 C8C5 40D7            PRINTEXT  ' THE PROGRAM'
00BC   D9D6 C7D9 C1D4
00C2   902A 0002 0000 8026 201F            PRINTEXT  'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00CC   C8C9 E340 C1D5 E840 D7D9
00D6   D6C7 D9C1 D440 C6E4 D5C3
00E0   E3C9 D6D5 40D2 C5E8 40E3
00EA   D640
00EC   8026 1A1A 40C2 D9C9 D5C7            PRINTEXT  ' BRING UP THE ENTRY SCREEN'
00F6   40E4 D740 E3C8 C540 C5D5
0100   E3D9 E840 E2C3 D9C5 C5D5
010A   8025                               DEQT
010C   0018 02B6                  CHECK     WAIT      ATTNECB,RESET
0110   A0A2 02B6 0001 0250                 IF        (ATTNECB,EQ,1),GOTO,ENDIT
0118   C29E 0000 04C6 02C4        GETIMAGE CALL       $IMOPEN,(DSETNAME),(IMAGEBUF)
0120   A0A2 04D6 FFFF 0168                 IF        (XMPLSTAT+2,NE,-1)
0128   005C 02BE 04D6                        MOVE      ERRCODE,XMPLSTAT+2
012E   8026 1818 7CC9 D4C1 C7C5            PRINTEXT  '@IMAGE OPEN ERROR,CODE ='
0138   40D6 D7C5 D540 C5D9 D9D6
0142   D96B C3D6 C4C5 407E
014A   0028 02BE 0001                      PRINTNUM  ERRCODE
0150   C026 0E0E 7CD9 C5E3 D9E8            QUESTION  '@RETRY OPEN ? ',YES=GETIMAGE,NO=ENDIT
015A   40D6 D7C5 D540 6F40 C02E
0164   0118 0250
```

Figure 77. Compilation listing (Part 1 of 3)

```
                                                    ENDIF
0168   C29E 0000 003E 02C4                          CALL      $IMDEFN,(IOCB2),(IMAGEBUF)
0170   1025 003E                                    ENQT      IOCB2
0174   1430                                         TERMCTRL  BLANK
0176   C29E 0000 02C4 0000                          CALL      $IMPROT,(IMAGEBUF),0
017E   819E 0000 02C4                               CALL      $IMDATA,(IMAGEBUF)
0184   B02A 0004 000C                               PRINTEXT  LINE=4,SPACES=12
018A   1C30                                         TERMCTRL  DISPLAY
018C   2030                              WAITONE   WAIT      KEY
018E   00A1 04D6 0004 0204 019E                     GOTO      (READ,E1,E2,E3,E4),XMPLSTAT+2
0198   01A8 01B2 01BC
019E   805C 02BC 0006                    E1        MOVE      LINENBR,6
01A4   00A0 01C2                                    GOTO      DELETE
01A8   805C 02BC 000B                    E2        MOVE      LINENBR,11
01AE   00A0 01C2                                    GOTO      DELETE
01B2   805C 02BC 0010                    E3        MOVE      LINENBR,16
01B8   00A0 01C2                                    GOTO      DELETE
01BC   805C 02BC 0015                    E4        MOVE      LINENBR,21
01C2   E02A 02BC 0000 F030 0004          DELETE    ERASE     MODE=LINE,TYPE=DATA,LINE=LINENBR
01CC   2000
01CE   8032 02BC 0001                              ADD       LINENBR,1
01D4   E02A 02BC 0000 F030 0004                    ERASE     MODE=LINE,TYPE=DATA,LINE=LINENBR
01DE   2000
01E0   8032 02BC 0001                              ADD       LINENBR,1
01E6   E02A 02BC 0000 F030 0004                    ERASE     MODE=LINE,TYPE=DATA,LINE=LINENBR
01F0   2000
01F2   8035 02BC 0002                              SUBTRACT  LINENBR,2
01F8   A02A 02BC 0006                              PRINTEXT  LINE=LINENBR,SPACES=6
01FE   1C30                                        TERMCTRL  DISPLAY
0200   00A0 018C                                    GOTO      WAITONE
0204   F02A 0002 0037 C026 0E0E          READ      QUESTION  'MORE ENTRIES ?',LINE=2,SPACES=55,NO=CLEANUP
020E   D4D6 D9C5 40C5 D5E3 D9C9
0218   C5E2 406F 802E 0244
0220   F02A 0002 0037 F030 0004                    ERASE     MODE=LINE,LINE=2,SPACES=55,TYPE=DATA
022A   2000
022C   F02A 0006 0000 F030 0000                    ERASE     MODE=SCREEN,LINE=6
0236   2000
0238   B02A 0006 0006                              PRINTEXT  LINE=6,SPACES=6
023E   1C30                                        TERMCTRL  DISPLAY
0240   00A0 018C                                    GOTO      WAITONE
0244   F030 0001 2000                    CLEANUP   ERASE     MODE=SCREEN,TYPE=ALL
024A   8025                                        DEQT
024C   00A0 0064                                    GOTO      START
0250   0022 FFFF                          ENDIT     PROGSTOP
0254   5050                                        DATA      X'5050'
0256   6060 6060 6060 6060 6060          DASHES    DATA      80C'-'
02A6   0019 02B6 0001                    OUT       POST      ATTNECB,1
02AC   001D                                         ENDATTN
02AE   0019 02B6 FFFF                    STATIC    POST      ATTNECB,-1
02B4   001D                                         ENDATTN
02B6   FFFF 0000 0000                    ATTNECB   ECB
02BC   0000                              LINENBR   DATA      F'0'
02BE   0000                              ERRCODE   DATA      F'0'
02C0   0000 0200 0000 0000 0000          IMAGEBUF  BUFFER    512,BYTES
02CA   0000 0000 0000 0000 0000
04BE   0000 0000 0000
04C4   0E0D E5C9 C4C5 D6F1 6BC5          DSETNAME  TEXT      'VIDEO1,EDX002'
04CE   C4E7 F0F0 F240
```

Figure 78. Compilation listing (Part 2 of 3)

```
04D4   0000 0000 0000 0234 0000          ENDPROG
04DE   0000 0000 0064 04D4 0000
04E8   0000 0000 0000 0000 0000
04F2   0002 0096 0000 0000 FFFF
04FC   0000 0000 0500 0000 0000
0506   0502 E7D4 D7D3 E2E3 C1E3
0510   0000 0000 0000 0000 0000
051A   0000 0000 FFFF 0000 0000
0524   0000 0000 0000 04D4 0000
052E   0000 0000 0000 0000 0000
054C   0000 0000 04D4 0080 0000
0556   0000 0000 0234 0000 0000
0560   0000 0000 0554 0000 0000
056A   0000 0000 0000 0000 0001
0574   000A 0000 0000 FFFF 0000
057E   0000 0580 0000 0000 0582
0588   5BC1 E3E3 C1E2 D240 0000
0592   0000 0000 0000 0000 0000
059C   0000 FFFF 0000 0000 0000
05A6   0000 0000 04D4 0000 0000
05B0   0000 0000 0000 0000 0000
05CE   0000 0554 0080 0000 0000
05D8   0000 0000 0000 0000 0000
05E2   0000 0000 0000 0000
                                         END

 SVC        WXTRN
 SUPEXIT    WXTRN
 SETBUSY    WXTRN
 $IMOPEN    EXTRN
 $IMDEFN    EXTRN
 $IMPROT    EXTRN
 $IMDATA    EXTRN
```

Figure 79. Compilation listing (Part 3 of 3)

Step 3. Link Edit Object Modules Using $LINK

The static-screen image formatting subroutines ($IMOPEN, $IMDEFN, $IMDATA, $IMPROT) used by the source program are distributed in the form of object modules, which normally become resident in ASMLIB.

To include these subroutines in the program, the object module output of the compilation (data set ASMOBJ) must be linked with the screen formatting support object modules using $LINK.

The names of the data sets and the volumes containing them that are used in the link-edit step are:

	DATA SET NAME	VOLUME NAME
INPUT DATA SETS	ASMOBJ $AUTO $IMGEN $IMOPEN	EDX002 ASMLIB ASMLIB ASMLIB
CONTROL DATA SET	LINKSTAT	EDX002
OUTPUT DATA SET	LINKOUT	EDX002

Note: You must allocate LINKOUT if it is not already allocated. It must be a data type member. A size of 100 records is adequate.

Using the Autocall Data Set

The INCLUDE control records for the screen formatting object modules are predefined in the system autocall data set $AUTO; they may be included using the autocall option. When you use the autocall option to include object modules, you need not specify those module names on INCLUDE control records.

Figure 80 is a listing of $AUTO, the system autocall data set. The screen formatting support modules are specified in autocall definition statements 220 and 230.

```
00010  $GPLIST,ASMLIB      $GPLIST
00020  $PUHC,ASMLIB        $PUHC
00030  $GEPM,ASMLIB        $GEPM
00040  $GEAC,ASMLIB        $GEAC
00050  $$GIN,ASMLIB        $$GIN
00060  $PUFC,ASMLIB        $PUFC
00070  $PUXC,ASMLIB        $PUXC
00080  $GEER,ASMLIB        $GEER
00090  $GEXC,ASMLIB        $GEXC
00100  $$SCREEN,ASMLIB     $$SCREEN
00110  $PUIC,ASMLIB        $PUIC
00120  $PUSC,ASMLIB        $PUSC
00130  $GESC,ASMLIB        $GESC
00140  $GEFC,ASMLIB        $GEFC
00150  $PUAC,ASMLIB        $PUAC
00160  $PUEC,ASMLIB        $PUEC
00170  $GEIC,ASMLIB        $GEIC
00180  $$PGIN,ASMLIB       $$PGIN
00190  $$CONCAT,ASMLIB     $$CONCAT
00200  $$XYPLOT,ASMLIB     $$XYPLOT
00210  $MFSL,ASMLIB        $MFSL
00220  $IMGEN,ASMLIB       $IMDEFN  $IMPROT  $IMDATA  $PACK  $UNPACK
00230  $IMOPEN,ASMLIB      $IMOPEN  DSOPEN
00240  $$RETURN,ASMLIB     RETURN
00250  $$SVC,ASMLIB        SVC
00260  $$EDXATSR,ASMLIB    SETBUSY  SUPEXIT  **END
```

Figure 80. $AUTO data set listing

If you wish to have your own autocall definitions, you can add them to this data set, and continue to use the system autocall data set $AUTO, or build your own autocall data set. In either case, the last statement in the data set must contain the "**END" text, indicating the end of the autocall data set.

Using the Link Control Data Set

The names of the output object module data set, the autocall data set (if required), and the object module data sets to be linked are passed to the linkage editor in the link control data set. The link control data set used for this example is named LINKSTAT. In Figure 81, the link control statements required for this link-edit are listed, along with some preceding comment lines explaining their function.

```
00010 * THIS LINK EDIT CONTROL DATA SET SPECIFIES:
00020 *      1) THE LINKED OUTPUT OBJECT MODULE WILL
00030 *         BE STORED IN 'LINKOUT' ON EDX002
00040 *      2) THE AUTOCALL DATA SET IS '$AUTO' ON
00050 *         VOLUME ASMLIB (SYSTEM SUPPLIED)
00060 *      3) 'ASMOBJ' ON EDX002 IS THE ONLY INPUT
00070 *         OBJECT MODULE TO BE INCLUDED EXPLICITLY
00080 *
00090 OUTPUT LINKOUT AUTO=$AUTO,ASMLIB
00100 INCLUDE ASMOBJ
00110 END
```

Figure 81. Link edit control statements (LINKSTAT)

Use $FSEDIT to create this control statement data set and store it in LINKSTAT using the WRITE function at the end of the text edit session.

Invoking $LINK

At $LINK load time, supply the name of the link control data set and the name of the device to which linkage-editor messages are to be directed. The linkage editor, using the LINKSTAT link control data set, links the compiled object module in ASMOBJ (specified on the INCLUDE control statement) with the screen formatting object modules in ASMLIB, found through autocall definitions in $AUTO; the linked object module is stored in LINKOUT (specified on the OUTPUT control statement). Required error or information messages are read from the system link message data set, $LEMSG.

Figure 82 shows the session manager display screen used to invoke $LINK.

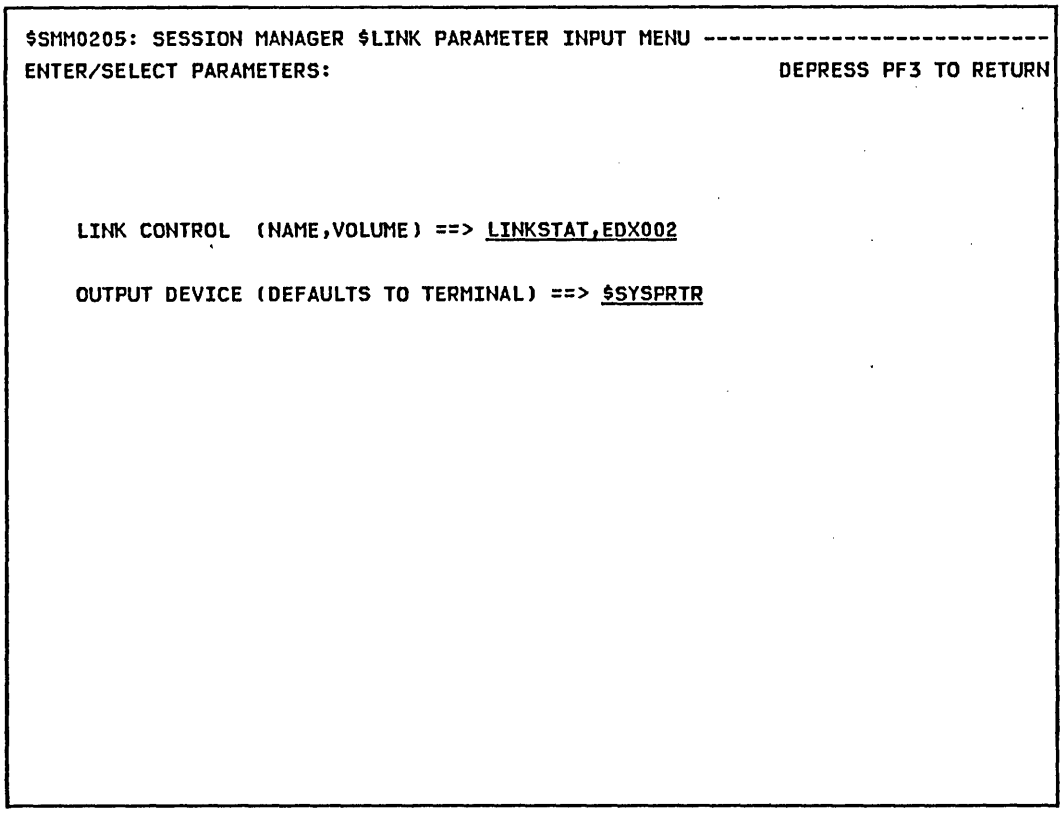


Figure 82. $LINK invocation

Figure 83 shows the $SYSPRTR output resulting from this link-edit.

```
$LINK EXECUTION CONTROL RECORDS
440000    FROM  LINKSTAT,EDX002
      OUTPUT LINKOUT AUTO=$AUTO,ASMLIB
000210INCLUDE ASMOBJ
000220  INCLUDE $IMOPEN,ASMLIB       VIA AUTOCALL
        INCLUDE $IMGEN,ASMLIB        VIA AUTOCALL
        INCLUDE $$RETURN,ASMLIB      VIA AUTOCALL
        END
      *****  UNRESOLVED EXTERNAL REFERENCES
410000    WXTRN   SVC
          WXTRN   SUPEXIT
          WXTRN   SETBUSY
      OUTPUT NAME=  LINKOUT
          ESD TYPE   LABEL    ADDR    LENGTH

430000    CSECT               0000      05EA
          CSECT               05EA      076A
             ENTRY  $IMOPEN   05EC
             ENTRY  DSOPEN    090E
          CSECT               0D54      03E0
             ENTRY  $IMDEFN   0D56
             ENTRY  $IMPROT   0DF2
             ENTRY  $IMDATA   0F38
             ENTRY  $PACK     1018
             ENTRY  $UNPACK   1088
          CSECT               1134      0026
             ENTRY  RETURN    1134
      MODULE TEXT LENGTH= 115A,  RLD COUNT=  424
      LINKOUT   ADDED TO  EDX002


$LINK COMPLETION CODE=     -1
AT 16:28:55  ON 02/21/80

$LINK    ENDED AT 16:28:55
```

Figure 83. Link edit listing

Step 4. Format Object Modules Using $UPDATE

Before a linked (or compiled) object module can be executed, it must be processed by $UPDATE to format the object module into a relocatable load module acceptable to the system loader.

The names of the data sets and the volumes containing them that are used in the $UPDATE step are:

	DATA SET NAME	VOLUME NAME
INPUT DATA SET	LINKOUT	EDX002
OUTPUT DATA SET	STATPROG	EDX002

Figure 84 shows the session manager display screen used to invoke $UPDATE.

```
$SMM0206: SESSION MANAGER $UPDATE PARAMETER INPUT MENU ------------------------
ENTER/SELECT PARAMETERS:                                DEPRESS PF3 TO RETURN



    OBJECT INPUT (NAME,VOLUME) ==============> LINKOUT,EDX002

    PROGRAM OUTPUT (NAME,VOLUME) =============> STATPROG,EDX002

    REPLACE (ENTER YES IF PROGRAM EXISTS) ===> YES

    LISTING (TERMINAL NAME / *) ==============> $SYSPRTR




    NOTE: THE OBJECT INPUT, PROGRAM OUTPUT AND LISTING TERMINAL NAME ARE
          REQUIRED PARAMETERS AND MUST BE ENTERED.
          AN '*' MAY BE USED TO SPECIFY THIS TERMINAL AS THE LISTING TERMINAL
```

Figure 84. $UPDATE invocation

If data set STATPROG does not exist, $UPDATE creates it. The program STATPROG can be loaded and executed when this step is completed.

PART IV. PREPARE PROGRAM USING $JOBUTIL

An alternative way to prepare a program for execution is to run the steps involved as a batch job under control of the batch job processor utility ($JOBUTIL). This part of the appendix demonstrates the use of this method to prepare the example program for execution.

The procedure to be used requires the use of work data sets to pass to $EDXASM and $LINK. The procedure also uses the same input, control, and output data sets used in part III.

The required work data sets are as follows:

PROGRAM	DATA SET NAME	VOLUME NAME	SIZE
$EDXASM	ASMWORK	EDX002	250
$LINK	LEWORK1 LEWORK2	EDX002 EDX002	400 150

Note: You must allocate the work data sets if they are not available. They are normally allocated during system installation.

Figure 85 on page 409 is a listing of the $JOBUTIL procedure data set used to prepare the example program. The statements in a procedure data set are created using $FSEDIT, and saved in a data set.

In this example, the procedure data set is STATPROC on EDX002.

```
00010 JOB      STATIC
00020 LOG      $SYSPRTR
00030 *
00040 PROGRAM  $EDXASM,ASMLIB
00050 REMARK   COMPILE OF 'STATSRC' STARTED
00060 DS       STATSRC
00070 DS       ASMWORK
00080 DS       ASMOBJ
00090 PARM     LIST      $SYSPRTR
00100 NOMSG
00110 EXEC
00120 JUMP     BADASM,NE,-1
00130 *
00140 * THIS LINK INCLUDES THE 'IM' SUBROUTUINE SUPPORT BY
00150 * USE OF THE AUTOCALL OPTION. THE AUTOCALL DEFINITION
00160 * STATEMENTS FOR THE 'IM' SUPPORT ARE IN THE SYSTEM
00170 * SUPPLIED AUTOCALL DATA SET '$AUTO' ON ASMLIB.
00180 *
00190 PROGRAM  $LINK,ASMLIB
00200 REMARK   LINK EDIT OF 'ASMOBJ' OBJECT MODULE STARTED
00210 REMARK   NAME OF LINK CONTROL DATA SET?
00220 PAUSE
00230 DS       LEWORK1
00240 DS       LEWORK2
00250 PARM     $SYSPRTR
00260 NOMSG
00270 EXEC
00280 JUMP     BADLINK,NE,-1
00290 *
00300 PROC     FORMPROC,EDX002
00310 *
00320 JUMP     END,EQ,-1
00330 REMARK   FORMAT STEP FAILED
00340 JUMP     END
00350 LABEL    BADASM
00360 REMARK   COMPILE STEP FAILED
00370 JUMP     END
00380 LABEL    BADLINK
00390 REMARK   LINK EDIT STEP FAILED
00400 LABEL    END
00410 EOJ
```

Figure 85. Batch Job Processor procedure data set

Invoking $JOBUTIL

When you load $JOBUTIL, you are prompted for the name of the procedure data set to be used to direct the batch job. Enter the data set name - STATPROC.

In Figure 85 on page 409, the JOB command at statement 10 causes the display of a "job started" message on the loading terminal as follows:

```
> $L $JOBUTIL
$JOBUTIL        3P,00:05:32, LP= 5F00
 DS1 (NAME,VOLUME): STATPROC
*** JOB - STATIC  - STARTED AT 00:05:55  00/00/00  ***

JOB       STATIC
```

$EDXASM Step Invocation Under $JOBUTIL

The LOG command at statement 20 in Figure 85 on page 409 causes the procedure data set statements (other than internal comments) to print on the system printer. Statements 40 through 110 load and execute the compiler. The source, work, and output data sets are specified in the DS commands. The PARM command at statement 90 directs the compiler listing to the system printer. The NOMSG command following the PARM prevents the $EDXASM load message from being displayed on the loading terminal, but the REMARK at statement 50 does appear:

```
> $L $JOBUTIL
$JOBUTIL        3P,00:05:32, LP= 5F00
 DS1 (NAME,VOLUME): STATPROC
*** JOB - STATIC  - STARTED AT 00:05:55  00/00/00  ***

JOB       STATIC
REMARK    COMPILE OF 'STATSRC' STARTED
```

The normal completion code for an error-free compilation is -1. The JUMP command (statement 120) tests the compiler completion code. If it is not equal to -1, the JUMP transfers control to the label BADASM, which is defined by the LABEL command at statement 350. The REMARK at 360 would be displayed on the loading terminal, and the JUMP at 370 would transfer to label END, ending the job.

$LINK Step Invocation Under $JOBUTIL

Assuming normal compiler operation, $JOBUTIL continues with the link-edit step.

Through the PAUSE command, $JOBUTIL allows the operator to enter job control commands. To illustrate this capability, the link control data set is not specified in a DS command. Instead, the PAUSE at statement 220 allows the operator to enter the link control data set name. When the link procedure is entered, the two REMARK statements preceding the PAUSE are displayed, along with the PAUSE operator instructions, and $JOBUTIL waits for the operator to press the ATTN key and enter a command:

```
> $L $JOBUTIL
$JOBUTIL        3P,00:05:32, LP= 5F00
 DS1 (NAME,VOLUME): STATPROC
*** JOB - STATIC  - STARTED AT 00:05:55  00/00/00  ***

JOB       STATIC
REMARK    COMPILE OF 'STATSRC' STARTED
REMARK    LINK EDIT OF 'ASMOBJ' OBJECT MODULE STARTED
REMARK    NAME OF LINK CONTROL DATA SET ?

PAUSE-*-ATTN:GO/ENTER/ABORT

PAUSE
```

You can continue (GO), enter a job control command (ENTER), or abort the job stream processor and end the job (ABORT). In this example, the link control data set is to be specified, so enter "ENTER".

You are prompted for the command to be entered. Enter "DS" to indicate that a data set is to be specified.

You are prompted for the command operand to be entered. Enter "LINKSTAT" to specify the link control data set.

You are prompted for the next command to be entered. Enter "GO" to direct $JOBUTIL to continue.

The $JOBUTIL prompts and operator responses entered are shown as follows:

```
> $L $JOBUTIL
$JOBUTIL        3P,00:47:17, LP= 5F00
 DS1 (NAME,VOLUME): STATPROC
*** JOB - STATIC  - STARTED AT 00:47:26  00/00/00  ***

JOB       STATIC
REMARK    COMPILE OF 'STATSRC' STARTED
REMARK    LINK EDIT OF 'ASMOBJ' OBJECT MODULE STARTED
REMARK    NAME OF LINK CONTROL DATA SET ?

PAUSE-*-ATTN:GO/ENTER/ABORT

PAUSE
> ENTER
ENTER COMMAND DS
ENTER OPERAND LINKSTAT
ENTER COMMAND GO
```

$UPDATE Step Invocation Under $JOBUTIL (Nested Procedure)

$JOBUTIL allows secondary (nested) procedures to be invoked from a primary procedure. To illustrate, the following $UPDATE (formatting) step $JOBUTIL control statements have been defined as a nested procedure, stored in data set FORMPROC.

```
00010  ********************************************************************
00020  * THIS IS A "NESTED" PROCEDURE, INVOKED FROM
00030  * 'STATPROC' BY THE 'PROC' COMMAND. $JOBUTIL
00040  * SUPPORTS ONE LEVEL OF NESTING.
00050  *
00060  REMARK   FORMATTING OF 'LINKOUT' STARTED
00070  PROGRAM  $UPDATE
00080  PARM     $SYSPRTR   LINKOUT    STATPROG YES
00090  NOMSG
00100  EXEC
00110  EOP
```

After testing for a successful link-edit (JUMP command at statement 280) the primary procedure used in Figure 85 on page 409 invokes the nested procedure FORMPROC by the PROC command at statement 300. At the conclusion of the formatting step, control is returned to the primary procedure at statement 320. If $UPDATE executed properly, the job is ended without displaying the error message (REMARK at 330).

The $JOBUTIL display at the end of the job is shown as follows:

```
> $L $JOBUTIL
 DS1 (NAME,VOLUME): STATPROC
*** JOB - STATIC  - STARTED AT 00:05:55  00/00/00  ***

JOB       STATIC
REMARK    COMPILE OF 'STATSRC' STARTED
REMARK    LINK EDIT OF 'ASMOBJ' OBJECT MODULE STARTED
REMARK    NAME OF LINK CONTROL DATA SET ?

PAUSE-*-ATTN:GO/ENTER/ABORT

PAUSE
> ENTER
ENTER COMMAND DS
ENTER OPERAND LINKSTAT
ENTER COMMAND GO
REMARK    FORMATTING OF 'LINKOUT' STARTED

$JOBUTIL ENDED AT 00:10:18
```

Figure 86 on page 414, Figure 87 on page 415, Figure 88 on page 416, and Figure 89 on page 417 are the $SYSPRTR output resulting from execution of the $JOBUTIL procedure data set STATPROC.

The program (STATPROG on volume EDX002) may now be loaded and executed by either the $L operator command or the session manager primary option menu.

```
LOG       $SYSPRTR
PROGRAM   $EDXASM,ASMLIB
DS        STATSRC
DS        ASMWORK
DS        ASMOBJ
PARM      LIST       $SYSPRTR
NOMSG
EXEC
```

```
  EDX ASSEMBLER STATISTICS

  SOURCE INPUT  - STATSRC ,EDX002
  WORK DATA SET - ASMWORK ,EDX002
  OBJECT MODULE - ASMOBJ  ,EDX002
  DATE: 02/21/80 AT 16:19:17
  ASSEMBLY TIME:      25 SECONDS
  STATEMENTS PROCESSED -      70

  NO STATEMENTS FLAGGED
```

```
0000   0008 D7D9 D6C7 D9C1 D440  XMPLSTAT PROGRAM   START
000A   0000 04D4 0052 0000 0000
0014   05D6 0000 0000 0000 0100
001E   05D4 0000 0000 0000 0554
0028   0000
                                           EXTRN     $IMOPEN,$IMDEFN,$IMPROT,$IMDATA
002A   4040 4040 4040 4040 8000  IOCB1     IOCB      NHIST=0
0034   00FF 0000 7FFF 0000 0000
003E   4040 4040 4040 4040 8800  IOCB2     IOCB      SCREEN=STATIC
0048   00FF 0000 7FFF 0000 0000
0052   0002 0403 C5D5 C440 02A6            ATTNLIST  (END,OUT,$PF,STATIC)
005C   0403 5BD7 C640 02AE
0064   1025 002A                  START     ENQT      IOCB1
0068   B02A 0000 000F 8026 1414            PRINTEXT  'CLASS ROSTER PROGRAM',SPACES=15,LINE=0
0072   C3D3 C1E2 E240 D9D6 E2E3
007C   C5D9 40D7 D9D6 C7D9 C1D4
0086   902A 0002 0000 8026 2221            PRINTEXT  'HIT ''ATTN'' AND ENTER ''END'' TO END',SKIP=2
0090   C8C9 E340 7DC1 E3E3 D57D
009A   40C1 D5C4 40C5 D5E3 C5D9
00A4   407D C5D5 C47D 40E3 D640
00AE   C5D5 C440
00B2   8026 0C0C 40E3 C8C5 40D7            PRINTEXT  ' THE PROGRAM'
00BC   D9D6 C7D9 C1D4
00C2   902A 0002 0000 8026 201F            PRINTEXT  'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00CC   C8C9 E340 C1D5 E840 D7D9
00D6   D6C7 D9C1 D440 C6E4 D5C3
00E0   E3C9 D6D5 40D2 C5E8 40E3
00EA   D640
00EC   8026 1A1A 40C2 D9C9 D5C7            PRINTEXT  ' BRING UP THE ENTRY SCREEN'
00F6   40E4 D740 E3C8 C540 C5D5
0100   E3D9 E840 E2C3 D9C5 C5D5
010A   8025                               DEQT
010C   0018 02B6                  CHECK     WAIT      ATTNECB,RESET
0110   A0A2 02B6 0001 0250                  IF        (ATTNECB,EQ,1),GOTO,ENDIT
0118   C29E 0000 04C6 02C4        GETIMAGE  CALL      $IMOPEN,(DSETNAME),(IMAGEBUF)
0120   A0A2 04D6 FFFF 0168                  IF        (XMPLSTAT+2,NE,-1)
0128   005C 02BE 04D6                         MOVE      ERRCODE,XMPLSTAT+2
012E   8026 1818 7CC9 D4C1 C7C5              PRINTEXT  '@IMAGE OPEN ERROR,CODE ='
0138   40D6 D7C5 D540 C5D9 D9D6
0142   D96B C3D6 C4C5 407E
014A   0028 02BE 0001                        PRINTNUM  ERRCODE
0150   C026 0E0E 7CD9 C5E3 D9E8             QUESTION  '@RETRY OPEN ? ',YES=GETIMAGE,NO=ENDIT
015A   40D6 D7C5 D540 6F40 C02E
0164   0118 0250
```

Figure 86. $JOBUTIL Execution Listing (Part 1 of 4)

```
                                                ENDIF
0168   C29E 0000 003E 02C4                      CALL        $IMDEFN,(IOCB2),(IMAGEBUF)
0170   1025 003E                                ENQT        IOCB2
0174   1430                                     TERMCTRL    BLANK
0176   C29E 0000 02C4 0000                      CALL        $IMPROT,(IMAGEBUF),0
017E   819E 0000 02C4                           CALL        $IMDATA,(IMAGEBUF)
0184   B02A 0004 000C                           PRINTEXT    LINE=4,SPACES=12
018A   1C30                                     TERMCTRL    DISPLAY
018C   2030                           WAITONE   WAIT        KEY
018E   00A1 04D6 0004 0204 019E                 GOTO        (READ,E1,E2,E3,E4),XMPLSTAT+2
0198   01A8 01B2 01BC
019E   805C 02BC 0006                 E1        MOVE        LINENBR,6
01A4   00A0 01C2                               GOTO        DELETE
01A8   805C 02BC 000B                 E2        MOVE        LINENBR,11
01AE   00A0 01C2                               GOTO        DELETE
01B2   805C 02BC 0010                 E3        MOVE        LINENBR,16
01B8   00A0 01C2                               GOTO        DELETE
01BC   805C 02BC 0015                 E4        MOVE        LINENBR,21
01C2   E02A 02BC 0000 F030 0004       DELETE    ERASE       MODE=LINE,TYPE=DATA,LINE=LINENBR
01CC   2000
01CE   8032 02BC 0001                           ADD         LINENBR,1
01D4   E02A 02BC 0000 F030 0004                 ERASE       MODE=LINE,TYPE=DATA,LINE=LINENBR
01DE   2000
01E0   8032 02BC 0001                           ADD         LINENBR,1
01E6   E02A 02BC 0000 F030 0004                 ERASE       MODE=LINE,TYPE=DATA,LINE=LINENBR
01F0   2000
01F2   8035 02BC 0002                           SUBTRACT    LINENBR,2
01F8   A02A 02BC 0006                           PRINTEXT    LINE=LINENBR,SPACES=6
01FE   1C30                                     TERMCTRL    DISPLAY
0200   00A0 018C                                GOTO        WAITONE
0204   F02A 0002 0037 C026 0E0E       READ      QUESTION    'MORE ENTRIES ?',LINE=2,SPACES=55,NO=CLEANUP
020E   D4D6 D9C5 40C5 D5E3 D9C9
0218   C5E2 406F 802E 0244
0220   F02A 0002 0037 F030 0004                 ERASE       MODE=LINE,LINE=2,SPACES=55,TYPE=DATA
022A   2000
022C   F02A 0006 0000 F030 0000                 ERASE       MODE=SCREEN,LINE=6
0236   2000
0238   B02A 0006 0006                           PRINTEXT    LINE=6,SPACES=6
023E   1C30                                     TERMCTRL    DISPLAY
0240   00A0 018C                                GOTO        WAITONE
0244   F030 0001 2000                 CLEANUP   ERASE       MODE=SCREEN,TYPE=ALL
024A   8025                                     DEQT
024C   00A0 0064                                GOTO        START
0250   0022 FFFF                      ENDIT     PROGSTOP
0254   5050                                     DATA        X'5050'
0256   6060 6060 6060 6060 6060      DASHES    DATA        80C'-'
02A6   0019 02B6 0001                 OUT       POST        ATTNECB,1
02AC   001D                                     ENDATTN
02AE   0019 02B6 FFFF                 STATIC    POST        ATTNECB,-1
02B4   001D                                     ENDATTN
02B6   FFFF 0000 0000                 ATTNECB   ECB
02BC   0000                           LINENBR   DATA        F'0'
02BE   0000                           ERRCODE   DATA        F'0'
02C0   0000 0200 0000 0000 0000       IMAGEBUF  BUFFER      512,BYTES
02CA   0000 0000 0000 0000 0000
04BE   0000 0000 0000
04C4   0E0D E5C9 C4C5 D6F1 6BC5       DSETNAME  TEXT        'VIDEO1,EDX002'
04CE   C4E7 F0F0 F240
```

Figure 87. $JOBUTIL Execution Listing (Part 2 of 4)

```
04D4    0000 0000 0000 0234 0000            ENDPROG
04DE    00D0 0000 0064 04D4 0000
04E8    0000 0000 0000 0000 0000
04F2    0002 0096 0000 0000 FFFF
04FC    0000 0000 0500 0000 0000
0506    0502 E7D4 D7D3 E2E3 C1E3
0510    0000 0000 0000 0000 0000
051A    0000 0000 FFFF 0000 0000
0524    0000 0000 0000 04D4 0000
052E    0000 0000 0000 0000 0000
054C    0000 0000 04D4 0080 0000
0556    0000 0000 0234 0000 00D0
0560    0000 0000 0554 0000 0000
056A    0000 0000 0000 0000 0001
0574    000A 0000 0000 FFFF 0000
057E    0000 0580 0000 0000 0582
0588    5BC1 E3E3 C1E2 0240 0000
0592    0000 0000 0000 0000 0000
059C    0000 FFFF 0000 0000 0000
05A6    0000 0000 04D4 0000 0000
05B0    0000 0000 0000 0000 0000
05CE    0000 0554 0080 0000 0000
05D8    0000 0000 0000 0000 0000
05E2    0000 0000 0000 0000
                                            END

 SVC       WXTRN
 SUPEXIT   WXTRN
 SETBUSY   WXTRN
 $IMOPEN   EXTRN
 $IMDEFN   EXTRN
 $IMPROT   EXTRN
 $IMDATA   EXTRN
```

```
COMPLETION CODE =     -1

$EDXASM  ENDED AT 16:19:50
JUMP     BADASM,NE,-1
PROGRAM  $LINK,ASMLIB
DS       LINKSTAT
DS       LEWORK1
DS       LEWORK2
PARM     $SYSPRTR
NOMSG
EXEC
```

Figure 88. $JOBUTIL Execution Listing (Part 3 of 4)

```
$LINK EXECUTION CONTROL RECORDS
440000    FROM  LINKSTAT,EDX002
      OUTPUT LINKOUT AUTO=$AUTO,ASMLIB
000210INCLUDE ASMOBJ
000220  INCLUDE $IMOPEN,ASMLIB       VIA AUTOCALL
        INCLUDE $IMGEN,ASMLIB        VIA AUTOCALL
        INCLUDE $$RETURN,ASMLIB      VIA AUTOCALL
        END
      *****  UNRESOLVED EXTERNAL REFERENCES
410000    WXTRN    SVC
          WXTRN    SUPEXIT
          WXTRN    SETBUSY
      OUTPUT NAME=  LINKOUT
          ESD TYPE    LABEL     ADDR     LENGTH

430000    CSECT                 0000       05EA
          CSECT                 05EA       076A
             ENTRY  $IMOPEN     05EC
             ENTRY  DSOPEN      090E
          CSECT                 0D54       03E0
             ENTRY  $IMDEFN     0D56
             ENTRY  $IMPROT     0DF2
             ENTRY  $IMDATA     0F38
             ENTRY  $PACK       1018
             ENTRY  $UNPACK     1088
          CSECT                 1134       0026
             ENTRY  RETURN      1134
      MODULE TEXT LENGTH= 115A,  RLD COUNT=  424
      LINKOUT   ADDED TO  EDX002


$LINK COMPLETION CODE=     -1
AT 16:22:07  ON 02/21/80

$LINK     ENDED AT 16:22:08
JUMP      BADLINK,NE,-1
PROC      FORMPROC,EDX002
PROGRAM   $UPDATE
PARM      $SYSPRTR  LINKOUT   STATPROG  YES
NOMSG
EXEC
STATPROG STORED

$UPDATE   ENDED AT 16:22:18
JUMP      END,EQ,-1
LABEL     END
```

Figure 89. $JOBUTIL Execution Listing (Part 4 of 4)

EVENT DRIVEN EXECUTIVE LIBRARY SUMMARY

The library summary is a guide to the Event Driven Executive library. By briefly listing the content of each book and providing a suggested reading sequence for the library, it should assist you in using the library as a whole as well as direct you to the individual books you require.

Event Driven Executive Library

The IBM Series/1 Event Driven Executive library materials consist of five full-sized books, a quick reference pocket book, and a set of tabs:

- IBM Series/1 Event Driven Executive System Guide (or System Guide), SC34-0312
- IBM Series/1 Event Driven Executive Utilities, Operator Commands, Program Preparation, Messages and Codes (or Utilities), SC34-0313
- IBM Series/1 Event Driven Executive Language Reference (or Language Reference), SC34-0314
- IBM Series/1 Event Driven Executive Communications and Terminal Application Guide (or Communications Guide), SC34-0316
- IBM Series/1 Event Driven Executive Internal Design (or Internal Design), LY34-0168
- IBM Series/1 Event Driven Executive Multiple Terminal Manager Internal Design (or Multiple Terminal Manager Internal Design), LY34-0190
- IBM Series/1 Event Driven Executive Indexed Access Method Internal Design (or Indexed Access Method Internal Design), LY34-0189
- IBM Series/1 Event Driven Executive Reference Summary (or Reference Summary), SX34-0101
- IBM Series/1 Event Driven Executive Tabs (or Tabs), SX34-0030

System Guide

The System Guide introduces the concepts and capabilities of the Event Driven Executive system. It discusses multi-tasking, program and task structure, program overlays, storage management, and data management.

Planning aids include hardware and software requirements, along with guidelines for storage estimating.

The System Guide also presents step-by-step procedures for generating a supervisor tailored to your Series/1 hardware configuration and software needs.

The description of the Indexed Access Method contains the information on how to write applications that use indexed data sets.

The description of the session manager includes a procedure for modifying the session manager to include application programs in the primary option menu so that you can execute them under the session manager. You can also add a procedure to compile, link, and update programs.

Information is also provided concerning partitioned data sets, tape data organization, diagnostic aids, inter-program communication, logical screens, and dynamic data set allocation.

Utilities

Utilities describes:

- Event Driven Executive utility programs
- Operator commands
- Procedures to prepare and execute system and application programs
- The session manager -- a menu-driven interface program that will invoke the programs required for program development
- Messages and codes issued by the Event Driven Executive system

The operator commands, program preparation facilities, and session manager are grouped by function and discussions include detailed syntax and explanations. The utilities are presented in alphabetical order.

Language Reference

The Language Reference familiarizes you with the Event Driven Language by first grouping the instructions into functional categories. Then the instructions are listed alphabetically, with complete syntax and an explanation of each operand.

The final section of the Language Reference contains examples of using the Event Driven language for applications such as:

- Program loading
- User exit routine
- Graphics
- I/O level control program
- Indexing and hardware register usage

Communications Guide

The Communications Guide introduces the Event Driven Executive communications support -- binary synchronous communications, asynchronous communications, and the Host Communications Facility.

The Communications Guide contains coding details for all utilities and Event Driven language instructions needed for communications support and advanced terminal applications.

Internal Design

Internal Design describes the internal logic flow and specifications of the Event Driven Executive system so that you can understand how the system interfaces with application programs. It familiarizes you with the design and implementation by describing the purpose, function, and operation of the various Event Driven Executive system programs.

Multiple Terminal Manager Internal Design and Indexed Access Method Internal Design describe the internal logic flow and specifications of these programs.

Unlike the other manuals in the library, the Internal Design books contain material that is the licensed property of IBM and they are available only to licensed users of the Event Driven Executive system.

Reference Summary

The Reference Summary is a pocket-sized booklet to be used for quick reference. It lists the Event Driven language instructions with their syntax, the utility and program preparation commands, and the completion codes.

Tabs

The tabs package must be ordered separately. The package contains 33 index tabs by subject, with additional blank tabs. These extended tabular pages can be inserted at the front of various sections of the library. The tabs are color coded according to the major library topics.

Reading Sequence

All readers of the Event Driven Executive library should begin with the first three chapters of the System Guide ("Introduction," "The Supervisor and Emulator," and "Data Management") for an overview of the Event Driven Executive concepts and facilities.

Readers responsible for installing and preparing the system should then continue in the System Guide with "System Configuration" and "System Generation."

All readers should review the Utilities "Introduction" to become familiar with the utility functions available for the Event Driven Executive system. Then you can read more specific sections for particular utilities, operator commands, and program preparation facilities.

After you have a basic understanding of the Event Driven Executive system and how you can best use the system for your applications, you should read the Language Reference "Introduction." This will familiarize you with the potential

of the Event Driven Language and prepare you to start coding application programs.

If you have communications support for your Event Driven Executive system, you should read the Communications Guide, which is an extension of the System Guide, Utilities, and the Language Reference.

After you know the functions of the various Event Driven Language instructions, utilities, and program preparation facilities, you may wish to refer only to the Reference Summary for correct syntax while coding your applications.

Only readers responsible for the support or modification of the Event Driven Executive system need to read Internal Design.

OTHER EVENT DRIVEN EXECUTIVE PROGRAMMING PUBLICATIONS

- IBM Series/1 Event Driven Executive FORTRAN IV User's Guide, SC34-0315.
- IBM Series/1 Event Driven Executive PL/I Language Reference, GC34-0147.
- IBM Series/1 Event Driven Executive PL/I User's Guide, GC34-0148.
- IBM Series/1 Event Driven Executive COBOL Programmer's Guide, SL23-0014.
- IBM Series/1 Event Driven Executive Sort/Merge Programmer's Guide, SL23-0016
- IBM Series/1 Event Driven Executive Macro Assembler Reference,GC34-0317.
- IBM Series/1 Event Driven Executive Study Guide, SR30-0436.

OTHER SERIES/1 PROGRAMMING PUBLICATIONS

- IBM Series/1 Programming System Summary, GC34-0285.
- IBM Series/1 COBOL Language Reference, GC34-0234.
- IBM Series/1 FORTRAN IV Language Reference, GC34-0133.

- IBM Series/1 Host Communications Facility Program Description Manual, SH20-1819.
- IBM Series/1 Mathematical and Functional Subroutine Library User's Guide, SC34-0139.
- IBM Series/1 Macro Assembler Reference Summary, SX34-0128
- IBM Series/1 Data Collection Interactive Programming RPQ P82600 User's Guide, SC34-1654.

OTHER PROGRAMMING PUBLICATIONS

- IBM Data Processing Glossary, GC20-1699.
- IBM Series/1 Graphic Bibliography, GA34-0055.
- IBM OS/VS Basic Telecommunications Access Method (BTAM), GC27-6980.
- General Information — Binary Synchronous Communications, GA27-3004.
- IBM System/370 Program Preparation Facility, SB30-1072.

SERIES/1 SYSTEM LIBRARY PUBLICATIONS

- IBM Series/1 4952 Processor and Processor Features Description, GA34-0084.
- IBM Series/1 4953 Processor and Processor Features Description, GA34-0022.
- IBM Series/1 4955 Processor and Processor Features Description, GA34-0021.
- IBM Series/1 Communications Features Description, GA34-0028.
- IBM Series/1 3101 Display Terminal Description, GA34-2034.
- IBM Series/1 4962 Disk Storage Unit and 4964 Diskette Unit Description, GA34-0024.
- IBM Series/1 4963 Disk Subsystem Description, GA34-0051.
- IBM Series/1 4966 Diskette Magazine Unit Description, GA34-0052.

- IBM Series/1 4969 Magnetic Tape Subsystem Description, GA34-0087.
- IBM Series/1 4973 Line Printer Description, GA34-0044.
- IBM Series/1 4974 Printer Description, GA34-0025.
- IBM Series/1 4978-1 Display Station (RPQ D02055) and Attachment (RPQ D02038) General Information, GA34-1550
- IBM Series/1 4978-1 Display Station, Keyboard (RPQ D02056) General Information, GA34-1551
- IBM Series/1 4978-1 Display Station, Keyboard (RPQ D02057) General Information, GA34-1552
- IBM Series/1 4978-1 Display Station Keyboards (RPQ D02064 and D02065) General Information, GA34-1553
- IBM Series/1 4979 Display Station Description, GA34-0026
- IBM Series/1 4982 Sensor Input/Output Unit Description, GA34-0027
- IBM Series/1 Data Collection Interactive RPQs D02312, D02313, and D02314 Custom Feature, GA34-1567

This glossary contains terms that are used in the Series/1 Event Driven Executive software publications. All software and hardware terms are Series/1 oriented. This glossary defines terms used in this library and serves as a supplement to the IBM Data Processing Glossary (GC20-1699).

$SYSLOGA. The name of the alternate system logging device. This device is optional but, if defined, should be a terminal with keyboard capability, not just a printer.

$SYSLOG. The name of the system logging device or operator station; must be defined for every system. It should be a terminal with keyboard capability, not just a printer.

$SYSPRTR. The name of the system printer.

ACCA. See asynchronous communications control adapter.

address key. Identifies a set of Series/1 segmentation registers and represents an address space. It is one less than the partition number.

address space. The logical storage identified by an address key. An address space is the storage for a partition.

application program manager. The component of the Multiple Terminal Manager that provides the program management facilities required to process user requests. It controls the contents of a program area and the execution of programs within the area.

application program stub. A collection of subroutines that are appended to a program by the linkage editor to provide the link from the application program to the Multiple Terminal Manager facilities.

asynchronous communications control adapter. An ASCII terminal attached via #1610, #2091 with #2092, or #2095 with #2096 adapters.

attention list. A series of pairs of 1 to 8 byte EBCDIC strings and addresses pointing to EDL instructions. When the attention key is pressed on the terminal, the operator can enter one of the strings to cause the associated EDL instructions to be executed.

backup. A copy of data to be used in the event the original data is lost or damaged.

base records. Records that have been placed into an indexed data set while in load mode.

basic exchange format. A standard format for exchanging data on diskettes between systems or devices.

binary synchronous device data block (BSCDDB). A control block that provides the information to control one Series/1 Binary Synchronous Adapter. It determines the line characteristics and provides dedicated storage for that line.

block. (1) See data block or index block. (2) In the Indexed Method, the unit of space used by the access method to contain indexes and data.

BSCDDB. See binary synchronous device data block.

buffer. An area of storage that is temporarily reserved for use in performing an input/output operation, into which data is read or from which data is written. See input buffer and output buffer.

bypass label processing. Access of a tape without any label processing support.

CCB. See terminal control block.

character image. An alphabetic, numeric, or special character defined for an IBM 4978 Display Station. Each character image is defined by a dot matrix that is coded into eight bytes.

character image table. An area containing the 256 character images that can be defined for an IBM 4978 Display Station. Each character image is coded into eight bytes, the entire table of codes requiring 2048 bytes of storage.

cluster. In an indexed file, a group of data blocks that is pointed to from the same primary-level index block, and includes the primary-level index block. The data records and blocks contained in a cluster are logically contiguous, but are not necessarily physically contiguous.

COD (change of direction). A character used with ACCA terminal to indicate a reverse in the direction of data movement.

command. A character string from a source external to the system that represents a request for action by the system.

common area. A user-defined data area that is mapped into every partition at the same address. It can be used to contain control blocks or data that will be accessed by more than one program.

completion code. An indicator that reflects the status of the execution of a program. The completion code is displayed or printed on the program's output device.

conversion. See update.

cross partition service. A function that accesses data in two partitions.

data block. In an indexed file, an area that contains control information and data records. These blocks are a multiple of 256 bytes.

data set. A group of contiguous records within a volume pointed to by a directory member entry in the directory for the volume.

data set control block (DSCB). A control block that provides the information required to access a data set, volume or directory using READ and WRITE.

data set shut down. An indexed data set that has been marked (in main storage only) as unusable due to an error.

DCE. See directory control entry.

DDB. See disk data block.

direct access. (1) The access method used to READ or WRITE records on a disk or diskette device by specifying their location relative the beginning of the data set or volume. (2) In the Indexed Access Method, locating any record via its key without respect to the previous operation.

directory. A series of contiguous records in a volume that describe the contents in terms of allocated data sets and free spaces.

directory control entry (DCE). The first 32 bytes of the first record of a directory in which a description of the directory is stored.

directory member entry (DME). A 32-byte directory entry describing an allocated data set.

disk data block (DDB). A control block that describes a direct access volume.

display station. An IBM 4978 or 4979 display terminal or similar terminal with a keyboard and a video display.

DME. See directory member entry.

DSCB. See data set control block.

dynamic storage. An increment of storage that is appended to a program when it is loaded.

end-of-data indicator. A code that signals that the last record of a data set has been read or written. End-of-data is determined by an end-of-data pointer in the DME or by the physical end of the data set.

ECB. See event control block.

EDL. See Event Driven Language.

emulator. The portion of the Event Driven Executive supervisor that interprets EDL instructions and performs the function specified by each EDL statement.

end-of-tape (EOT). A reflective marker placed near the end of a tape and sensed during output. The marker signals that the tape is nearly full.

event control block (ECB). A control block used to record the status (occurred or not occurred) of an event; often used to synchronize the execution of tasks. ECBs are used in conjunction with the WAIT and POST instructions.

event driven language (EDL). The language for input to the Event Driven Executive compiler ($EDXASM), or the Macro and Host assemblers in conjunction with the Event Driven Executive macro libraries. The output is interpreted by the Event Driven Executive emulator.

EXIO (execute input or output). An EDL facility that provides user controlled access to Series/1 input/output devices.

external label. A label attached to the outside of a tape that identifies the tape visually. It usually contains items of identification such as file name and number, creation data, number of volumes, department number, and so on.

external name (EXTRN). The 1- to 8-character symbolic EBCDIC name for an entry point or data field that is not defined within the module that references the name.

FCA. See file control area.

FCB. See file control block.

file control area (FCA). A Multiple Terminal Manager data area that describes a file access request.

file control block (FCB). In an indexed data set, the first block of the data set. It contains descriptive information about the data contained in the data set.

file manager. A collection of subroutines contained within the program manager of the Multiple Terminal Manager that provides common support for all disk data transfer operations as needed for transaction-oriented application programs. It supports indexed and direct files under the control of a single callable function.

formatted screen image. A collection of display elements or display groups (such as operator prompts and field input names and areas) that are presented together at one time on a display device.

free pool. In an indexed data set, a group of blocks that can be used as either a data block or an index block. These differ from other free blocks in that these are not initially assigned to specific logical positions in the data set.

free space. In the Indexed Access Method, record spaces or blocks that do not currently contain data, and are available for use.

free space entry (FSE). A 4-byte directory entry defining an area of free space within a volume.

FSE. See free space entry.

hardware timer. The timer features available with the Series/1 processors. Specifically, the 7840 Timer Feature card or the native timer (4952 only). Only one or the other is supported by the Event Driven Executive.

host assembler. The assembler licensed program that executes in a 370 (host) system and produces object output for the Series/1. The source input to the host assembler is coded in Event Driven Language or Series/1 assembler language. The host assembler refers to the System/370 Program Preparation Facility (5798-NNQ).

host system. Any system whose resources are used to perform services such as program preparation for a Series/1. It can be connected to a Series/1 by a communications link.

IACB. See indexed access control block.

IAR. See instruction address register.

ICB. See indexed access control block.

IIB. See interrupt information byte.

image store. The area in a 4978 that contains the character image table.

index. In the Indexed Access Method, an ordered collection of pairs, each consisting of a key and a pointer, used to sequence and locate the records in an Indexed Access Method data set.

index block. In an indexed file, an area that contains control information and index entries. These blocks are a multiple of 256 bytes.

indexed access control block (IACB/ICB). The control block that relates an application program to an indexed data set.

indexed access method. An access method for direct or sequential processing of fixed-length records by use of a record's key.

indexed data set. A data set specifically created, formatted and used by the Indexed Access Method. An indexed data set may also be called an indexed file.

indexed file. Synonym for indexed data set.

index entry. In an indexed file, a key-pointer pair, where the pointer is be used to locate a lower-level index block or a data block.

index register (#1, #2). Two words defined in EDL and contained in the task control block for each task. They are used to contain data or for address computation.

input buffer. (1) See buffer. (2) In the Multiple Terminal Manager, an area for terminal input and output.

input output control block (IOCB). A control block containing information about a terminal such as the symbolic name, size and shape of screen, the size of the forms in a printer.

instruction address register (IAR). The pointer that identifies the instruction currently being executed. The Series/1 maintains a hardware IAR to determine the Series/1 assembler instruction being executed. It is located in the level status block (LSB).

interactive. The mode in which a program conducts a continuous dialogue between the user and the system.

internal label. An area on tape used to record identifying information (similar to the identifying information placed on an external label). Internal labels are checked by the system to ensure that the correct volume is mounted.

interrupt information byte (IIB). In the Multiple Terminal Manager, a word containing the status of a previous input/output request to or from a terminal.

job. A collection of related program execution requests presented in the form of job control statements, identified to the jobstream processor by a JOB statement.

job control statement. A statement in a job that specifies requests for program execution, program parameters, data set definitions, sequence of execution, and, in general, describes the environment required to execute the program.

job stream processor. The job processing facility that reads job control statements and processes the requests made by these statements. The Event Driven Executive job stream processor is $JOBUTIL.

key. In the Indexed Access Method, one or more consecutive characters in a data record, used to identify the record and establish its order with respect to other records. See also key field.

key field. A field, located in the same position in each record of an Indexed Access Method data set, whose content is used for the key of a record.

level status block (LSB). A Series/1 hardware data area that contains processor status.

library. A set of contiguous records within a volume. It contains a directory, data sets and/or available space.

line. A string of characters accepted by the system as a single input from a terminal; for example, all characters entered before the carriage return on the teletypewriter or the ENTER key on the display station is pressed.

link edit. The process of resolving symbols in one or more object modules to produce another single module that is the input to the update process.

load mode. In the Indexed Access Method, the mode in which records are initially placed in an indexed file.

load module. A single module having cross references resolved and prepared for loading into storage for execution. The module is the output of the $UPDATE or $UPDATEH utility.

load point. A reflective marker placed near the beginning of a tape to indicate where the first record is written.

lock. In the Indexed Access Method, a method of indicating that a record or block is in use and is not available for another request.

LSB. See level status block.

member. A term used to identify a named portion of a partitioned data set (PDS). Sometimes member is also used as a synonym for a data set. See data set.

menu. A formatted screen image containing a list of options. The user selects an option to invoke a program.

menu-driven. The mode of processing in which input consists of the responses to prompting from an option menu.

multifile volume. A unit of recording media, such as tape reel or disk pack, that contains more than one data file.

multiple terminal manager. An Event Driven Executive licensed program that provides support for transaction-oriented applications on a Series/1. It provides the capability to define transactions and manage the programs that support those transactions. It also manages multiple terminals as needed to support these transactions.

multivolume file. A data file that, due to its size, requires more than one unit of recording media (such as tape reel or disk pack) to contain the entire file.

non-labeled tapes. Tapes that do not contain identifying labels (as in standard labeled tapes) and contain only files separated by tapemarks.

null character. A user-defined character used to define the unprotected fields of a formatted screen.

option selection menu. A full screen display used by the Session Manager to point to other menus or system functions, one of which is to be selected by the operator. (See primary option menu and secondary option menu.)

output buffer. (1) See buffer. (2) In the Multiple Terminal Manager, an area used for screen output and to pass data to subsequent transaction programs.

overlay. The technique of reusing a single storage area allocated to a program during execution. The storage area can be reused by loading it with overlay programs that have been specified in the PROGRAM statement of the program.

overlay area. A storage area within a program reserved for overlay programs specified in the PROGRAM statement.

parameter selection menu. A full screen display used by the Session Manager to indicate the parameters to be passed to a program.

partition. A contiguous fixed-sized area of storage. Each partition is a separate address space.

physical timer. Synonym for hardware timer.

prefind. To locate the data sets or overlay programs to be used by a program and to store the necessary information so that the time required to load the prefound items is reduced.

primary-level index block. In an indexed data set, the lowest level index block. It contains the relative block numbers (RBNs) and high keys of several data blocks. See cluster.

primary menu. The program selection screen displayed by the Multiple Terminal Manager.

primary option menu. The first full screen display provided by the Session Manager.

primary task. The first task executed by the supervisor when a program is loaded into storage. It is identified by the PROGRAM statement.

priority. A combination of hardware interrupt level priority and a software ranking within a level. Both primary and secondary tasks will execute asynchronously within the system according to the priority assigned to them.

process mode. In the Indexed Access Method, the mode in which records may be retrieved, updated, inserted or deleted.

processor status word (PSW). A 16-bit register used to (1) record error or exception conditions that may prevent further processing and (2) hold certain flags that aid in error recovery.

program. A disk- or diskette-resident collection of one or more tasks defined by a PROGRAM statement; the unit that is loaded into storage. (See primary task and secondary task.)

program header. The control block found at the beginning of a program that identifies the primary task, data sets, storage requirements and other resources required by a program.

program/storage manager. A component of the Multiple Terminal Manager that controls the execution and flow of application programs within a single program area and contains the support needed to allow multiple operations and sharing of the program area.

protected field. On a display device, a field in which the operator cannot enter, modify, or erase data from the keyboard. It can contain text that the user can read.

PSW. See processor status word.

QCB. See queue control block.

QD. See queue descriptor.

QE. See queue element.

queue control block (QCB). A data area used to serialize access to resources that cannot be shared. See serially reusable resource.

queue descriptor (QD). A control block describing a queue built by the DEFINEQ instruction.

queue element (QE). An entry in the queue defined by the queue descriptor.

record. (1) The smallest unit of direct access storage that can be accessed by an application program on a disk or diskette using READ and WRITE. Records are 256 bytes in length. (2) In the Indexed Access Method, the logical unit that is transferred between $IAM and the user's buffer. The length of the buffer is defined by the user.

recovery. The use of backup data to recreate data that has been lost or damaged.

reflective marker. A small adhesive marker attached to the reverse (nonrecording) surface of a reel of magnetic tape. Normally, two reflective markers are used on each reel of tape. One indicates the beginning of the recording area on the tape (load point), and the other indicates the proximity to the end of the recording area (EOT) on the reel.

relative record number. An integer value identifying the position of a record in a data set relative to the beginning of the data set. The first record of a data set is record one, the second is record two, the third is record three.

reorganize. For an indexed data set, the copying of the data to a new indexed data set in a manner that rearranges the data for more optimum processing and free space distribution.

return code. An indicator that reflects the results of the execution of an instruction or subroutine. The return code is placed in the task code word (at the beginning of the task control block).

roll screen. A display screen on which data is displayed 24 lines at a time or data is entered line by line, beginning with line 0 at the top of the screen and continuing through line 23 at the bottom of the screen. When a roll screen device's screen is full (all 24 lines used), an attempt to display the next line results in removal of the old screen (screen is erased) and the new line on line 0 is displayed at the top of the screen.

SBIOCB. See sensor based I/O control block.

second-level index block. In an indexed data set, the second-lowest level index block. It contains the addresses and high keys of several primary-level index blocks.

secondary option menu. In the Session Manager, the second in a series of predefined procedures grouped together in a hierarchical structure of menus. Secondary option menus provide a breakdown of the functions available under the session manager as specified on the primary option menu.

secondary task. Any task other than the primary task. A secondary task must be attached by a primary task or another secondary task.

sector. The smallest addressable unit of storage on a disk or diskette. A sector on a 4962 or 4963 disk is equivalent to an Event Driven Executive record. On a 4964 or 4966 diskette, two sectors are equivalent to an Event Driven Executive record.

sensor based I/O control block (SBIOCB). A control block containing information related to sensor I/O operations.

sequential access. The processing of a data set in order of occurrence of the records in the data set. (1) In the Indexed Access Method, the processing of records in ascending collating sequence order of the keys. (2) When using READ/WRITE, the processing of records in ascending relative record number sequence.

serially reusable resource (SRR). A resource that can only be accessed by one task at a time. Serially reusable resources are usually managed via (1) a QCB and ENQ/DEQ statements or (2) an ECB and WAIT/POST statements.

session manager. A series of predefined procedures grouped together as a hierarchical structure of menus from which you select the utility functions, program preparation facilities, and language processors needed to prepare and execute application programs. The menus consist of a primary option menu that displays functional groupings and secondary option menus that display a breakdown of these functional groupings.

shared resource. A resource that can be used by more than one task at the same time.

shut down. See data set shut down.

source module/program. A collection of instructions and statements that constitute the input to a compiler or assembler. Statements may be created or modified using one of the text editing facilities.

standard labels. Fixed length 80-character records on tape containing specific fields of information (a volume label identifying the tape volume, a header label preceding the data records, and a trailer label following the data records).

static screen. A display screen formatted with predetermined protected and unprotected areas. Areas defined as operator prompts or input field names are protected to prevent accidental overlay by input data. Areas defined as input areas are not protected and are usually filled in by an operator. The entire screen is treated as a page of information.

subroutine. A sequence of instructions that may be accessed from one or more points in a program.

supervisor. The component of the Event Driven Executive capable of controlling execution of both system and application programs.

system configuration. The process of defining devices and features attached to the Series/1.

SYSGEN. See system generation.

system generation. The processing of user selected options to create a supervisor tailored to the needs of a specific Series/1 configuration.

system partition. The partition that contains the supervisor (partition number 1, address space 0).

tapemark. A control character recorded on tape used to separate files.

task. The basic executable unit of work for the supervisor. Each task is assigned its own priority and processor time is allocated according to this priority. Tasks run independently of each other and compete for the system resources. The first task of a program is the primary task. All tasks attached by the primary task

are secondary tasks.

task code word. The first two words (32 bits) of a task's TCB; used by the emulator to pass information from system to task regarding the outcome of various operations, such as event completion or arithmetic operations.

task control block (TCB). A control block that contains information for a task. The information consists of pointers, save areas, work areas, and indicators required by the supervisor for controlling execution of a task.

task supervisor. The portion of the Event Driven Executive that manages the dispatching and switching of tasks.

TCB. See task control block.

terminal. A display station, teletypewriter or printer.

terminal control block (CCB). A control block that defines the device characteristics, provides temporary storage, and contains links to other system control blocks for a particular terminal.

terminal environment block (TEB). A control block that contains information on a terminal's attributes and the program manager operating under the Multiple Terminal Manager. It is used for processing requests between the terminal servers and the program manager.

terminal screen manager. The component of the Multiple Terminal Manager that controls the presentation of screens and communications between terminals and transaction programs.

terminal server. A group of programs that perform all the input/output and interrupt handling functions for terminal devices under control of the Multiple Terminal Manager.

trace range. A specified number of instruction addresses within which the flow of execution can be traced.

transaction oriented applications. Program execution driven by operator actions, such as responses to prompts from the system. Specifically, applications executed under control of the Multiple Terminal Manager.

transaction program. See transaction-oriented applications.

transaction selection menu. A Multiple Terminal Manager display screen (menu) offering the user a choice of functions, such as reading from a data file, displaying data on a terminal, or waiting for a response. Based upon the choice of option, the application program performs the requested processing operation.

unprotected field. On a display device, a field in which the user can enter, modify, or erase data using the keyboard. Unprotected fields on a static screen are defined by the null character.

update. (1) To alter the contents of storage or a data set. (2) To convert object modules, produced as the output of an assembly or compilation, or the output of the linkage editor, into a form that can be loaded into storage for program execution and to update the directory of the volume on which the loadable program is stored.

user exit. (1) Assembly language instructions included as part of an EDL program and invoked via the USER instruction. (2) A point in an IBM-supplied program where a

user written routine can be given control.

vary offline. (1) To change the status of a device from online to offline. When a device is offline, no data set can be accessed on that device. (2) To place a disk or diskette in a state where it is not available for use by the system; however, it will still be available for executing I/O at the basic access level (EXIO).

vary online. To restore a device to a state where it is available for use by the system.

volume. A disk or diskette subdivision defined during system configuration. A volume may contain up to 32,767 records. As many volumes may be defined for a disk as will physically fit. A diskette is limited to one volume.

volume label. A label that uniquely identifies a single unit of storage media.

This index is common to the Event Driven Executive library. The index includes entries from the seven publications listed below. (The Glossary is not indexed.) Each publication has a copy of the index, which provides a cross-reference between the publications.

Each page number entry contains a single letter prefix which identifies the publication where the listed subject can be found. The letter prefixes have the following meanings:

- C = Communications and Terminal Application Guide
- I = Internal Design
- L = Language Reference
- S = System Guide
- U = Utilities, Operator Commands, Program Preparation, Messages and Codes
- M = Multiple Terminal Manager Internal Design
- A = Indexed Access Method Internal Design

Special Characters

A

B

C

D

E

F

G

J

K

L

M

N

O

P

Q

S

T

U

V

W

XYZ

Numeric Subjects

READER'S COMMENT FORM

SC34-0312-2

IBM Series/1 Event Driven Executive
System Guide

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or the IBM branch office serving your locality.

Corrections or clarifications needed:

Page Comment

Cut or Fold Along Line

Please indicate your name and address in the space below if you wish a reply.

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A.
(Elsewhere, an IBM office or representative will be happy to forward your comments.)

Reader's Comment Form

Cut Along Line

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Systems Publications, Dept 27T
P.O. Box 1328
Boca Raton, Florida 33432

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

SC34-0312-2
Printed in U.S.A.

SC34-0312-2
Printed in U.S.A.