K. Kalinowsky X390 X2390

COPY NO. 019

TECHNICAL REPORT
1469–TR–4

N68-26292

FINAL REPORT

DATA MANAGEMENT SYSTEM STUDY

SUBMITTED TO

NASA/ERC
565 TECHNOLOGY SQUARE
CAMBRIDGE, MASSACHUSETTS 02139
ATTN: CARL KALINOWSKI, TECHNICAL MONITOR
KC/COMPUTER RESEARCH LAB.

IN ACCORDANCE WITH

PO NAS 12–562

APRIL 1, 1968

AUERBACH CORPORATION • 121 N. BROAD STREET
PHILADELPHIA, PENNSYLVANIA 19107

AUERBACH

NASA CR-86057

DATA MANAGEMENT SYSTEM STUDY

By Jerome D. Sable and James Cochrane

April 1968

Prepared under Contract No. NAS 12-562 by

AUERBACH Corporation
121 N. Broad Street
Philadelphia, Pennsylvania

Electronic Research Center

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

AUERBACH

NASA CR-

DATA MANAGEMENT SYSTEM STUDY

By Jerome D. Sable and James Cochrane

April 1968

Prepared under Contract No. NAS 12-562 by

AUERBACH Corporation
121 N. Broad Street
Philadelphia, Pennsylvania

Electronic Research Center

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

AUERBACH

PREFACE

The Final Technical Report of the Data Management System Study is submitted to NASA/ERC by AUERBACH Corporation in accordance with Purchase Order No. NAS 15-562. It covers the period of performance from May 15, 1967 to March 31, 1968 and summarizes the results of the four phases of this project:

Phase 1: Investigation of System Features

Phase 2: Evaluation of Systems

Phase 3: System Concepts

Phase 4: Prototype System Specification.

TABLE OF CONTENTS

PARAGRAPH	TITLE	PAGE
	SECTION I. INTRODUCTION	
1.1	SCOPE OF STUDY	1-1
1.2	CHARACTERIZATION OF THE SYSTEM	1-3
1.3	THE COHERENT SYSTEM STRUCTURE	1-5
	SECTION II. SYSTEM FEATURES	
2.1	INTERNAL FILE SYSTEM FEATURES	2-6
2.1.1	Design Objectives	2-6
2.1.2	Operations	2-7
2.1.3	Structures	2-8
2.1.4	Language Elements	2-9
2.2	EXTERNAL FILE SYSTEM FEATURES	2-10
2.2.1	Design Objectives	2-10
2.2.2	Operations	2-11
2.2.3	Structures	2-14
2.2.4	Language Elements	2-15
2.3	JOB MANAGEMENT FEATURES	2-15
2.3.1	Design Objectives	2-15
2.3.2	Operations	2-16
2.3.3	Structures	2-17
2.3.4	User Languages	2-17
	SECTION III. EVALUATION OF FILE SYSTEMS	
3.1	TDMS	3-2
3.2	GIS	3-4
3.3	ICS	3-5
3.4	DM-1	3-5
3.5	TSS/360	3-7
3.6	BTSS	3-7
3.7	MULTICS	3-9
3.8	INTIPS	3-10

TABLE OF CONTENTS (CONTINUED)

PARAGRAPH	TITLE	PAGE
	SECTION IV. THE COHERENT SYSTEM	
4.1	THE COHERENT SYSTEM CONCEPT	4-2
4.1.1	Coherence of Programs	4-2
4.1.2	Coherence of Control	4-3
4.1.3	Responsibility for Program and Control Coherence	4-3
4.2	JOB MANAGEMENT SYSTEM	4-3
4.2.1	Responsibility of the JMS	4-3
4.2.2	Jobs and Commands	4-4
4.2.3	Structure of the Job Management System	4-5
4.3	DATA AND DATA SERVICES	4-9
4.3.1	The Data Resource	4-9
4.3.2	Levels and Modes of Data Service	4-10
	SECTION V. THE EXTERNAL FILE SYSTEM	
5.1	THE CELL HIERARCHY	5-2
5.1.1	The Byte	5-2
5.1.2	The Page	5-3
5.1.3	The Train Cell	5-3
5.1.4	Copy Cells	5-3
5.2	TRAIN SERVICES	5-4
5.2.1	Virtual Store Trains and Implicit Services	5-4
5.2.2	Terminal Trains and Explicit Services	5-12
5.3	TRAIN PROTECTION AND ARCHIVES	5-15
5.4	MULTILEVEL STORAGE MANAGEMENT	5-16
	SECTION VI. INTERNAL FILE SYSTEM	
6.1	THE ITEM	6-1
6.2	ITEM STRUCTURE REPRESENTATION	6-3
6.3	ITEM LINKAGE	6-7

TABLE OF CONTENTS (CONTINUED)

PARAGRAPH	TITLE	PAGE
6.4	METHODS OF LINKAGE IMPLEMENTATION	6-12
6.5	OTHER ITEM STRUCTURES	6-14
6.5.1	Coded Valued Fields	6-15
6.5.2	Hierarchic Valued Fields	6-15
6.5.3	Variable Structures	6-16
6.5.4	Synonyms	6-18
6.6	THE DATA POOL	6-19
6.7	DATA RETRIEVAL	6-25
6.7.1	Primary Directories	6-25
6.7.2	Data Representation	6-28
6.7.3	Sample Retrieval	6-34
6.7.4	Indexing	6-39
6.7.5	Linkage	6-41
6.7.6	Data Integrity	6-44
6.8	ITEM SERVICES	6-46
6.8.1	Symbolic Coding and Implicit Services	6-46
6.8.2	The Mechanics of Implicit Item Services	6-48
6.8.3	Explicit Item Services	6-59

SECTION VII. USER LANGUAGES AND SYSTEM SUPPORT JOBS

7.1	LANGUAGE SPECIFICATION AND PROCESSING	7-1
7.2	JOB DEFINITION AND COMMAND LANGUAGES	7-2
7.2.1	Conditions	7-2
7.2.2	Job Constraints	7-5
7.3	ITEM DEFINITION LANGUAGE	7-12

SECTION VIII. CONCLUSIONS

APPENDIX A. FILE SYSTEM DESCRIPTIONS

APPENDIX B. INSCAN: A SYNTAX-DIRECTED LANGUAGE PROCESSOR

LIST OF ILLUSTRATIONS

FIGURE	TITLE	PAGE
1-1	The Coherent System	1-6
3-1	Summary of Capability	3-2
4-1	Interface Between Task, Job Management System, and File System	4-6
4-2	Job Management System	4-7
4-3	File System Services	4-12
5-1	External File System Train and Track	5-5
5-2	EFS Tables and Users	5-8
5-3	Addressing Within a Page in Memory	5-11
6-1	Item Structure Representations	6-4
6-2	Sequencing Linkage	6-9
6-3	Associative Linkage	6-10
6-4	Hierarchy Linkage	6-11
6-5	Adjoining Linkage	6-13
6-6	Switch Item Data Structure	6-17
6-7	Switch Item Record in Private Memory Table	6-18
6-8	Coherent System Data Pool	6-19
6-9	IFS Data	6-20
6-10	EFS Data	6-20
6-11	File System Service Levels	6-21
6-12	Data Base Example	6-23
6-13	Purchasing Item Structure	6-26
6-14	Sample Structure	6-32
6-15	Buss Tree Diagram and Data Stream	6-33
6-16	Directories for the Purchasing Item	6-36
6-17	Purchasing Item with Links	6-43
6-18	Quality Control File	6-53
6-19	Program Entry	6-55
6-20	Job Definition and Request	6-57
6-21	Page Tables and Task Train Table Creation	6-58

LIST OF ILLUSTRATIONS (CONTINUED)

FIGURE	TITLE	PAGE
7-1	Action Graph Symbols	7-3
7-2	Assignment Statement Compiler	7-4
7-3	A Typical Job Structure	7-8
7-4	A Restricted Job Structure	7-9
7-5	Action Graph for Command Syntax	7-10

LIST OF TABLES

TABLE	TITLE	PAGE
2-1	System Features	2-2
2-2	Data Integrity Protection	2-13
6-1	Field Conversions	6-5
6-2	Data Dependent Structure	6-16
6-3	Definition for the Purchasing Item	6-27
6-4	Term List and Item List	6-29
6-5	Name Encoding Table	6-30
7-1	Data Definition Languages	7-13

SUMMARY

The study of automatic computation is the study of information structures and their transformation. A recent book on current technology* has adopted that point of view in attempting to formulate an underlying theory or framework for the study of computer science. There are two parallel views of information structures which have evolved in the short history of computer science, and which will continue to play an important part of that field.

In the first view, the process of programming and computation is concerned with the manipulation of "rectangular" cell structures in an address space. The cells are passive registers for the storage of information, active machine registers for data transformation, containers for holding program variables, arrays, etc. They share the property that they are (or can be considered) locations in some address space. This view forms the computational framework of the assembly language programmer. He reduces an algorithm to the problems of allocation of information to cells, and the logistics of cell movement and transformation within a mosaic of address spaces. Although he deals in symbols rather than numerical addresses, he knows that the symbols merely stand for cell locations and he deals with them accordingly.

In the second view, programming and computation are concerned with the algebraic manipulation of symbols which stand for scalars, arrays, and other structured items. These items have properties (they are names, quantities in various codes, logical values, etc., and may be fixed or variable in size, etc.) and they have relationships to each other. Relationships such as ordering, inclusion, and association (class membership) may be defined. This second view then is that of the problem-oriented (e.g., compiler) language programmer. He is not concerned with space allocation or cell logistics but rather with relational or logical information structures and their transformation and algebraic manipulation.

The first view is more "machine oriented" than the second, and in some sense the relational structures of the second must be transformed into the cell domain

* P. Wegner, Programming Languages, Information Structures, and Machine Organization. McGraw-Hill, 1968.

before they can be operated upon by the machine. Modern computing machinery, through the use of address mapping hardware, makes it possible to create many independent address space (cell) structures which are bound to the real address space of the computer at the latest possible time (execution time).

The Coherent System developed in this report demonstrates that these two views can and should coexist in the File System and executive software of an advanced computing complex. The Coherent System maintains an on-going data base which is viewed as a common resource for all its users. This data base contains all system and user program and data structures and can be utilized at both the item structure and cell structure levels.

The problem or goal-oriented user will view the programs of the computing facility as a set of operators which can operate on items which he supplies interactively, or on items in the data base which he can name. The programmer, on the other hand, will use the system to create, manage, and manipulate symbolically named cell structures, both as operators (programs) and operands.

To accomplish this, the Coherent System is composed of three main implicitly active components plus a set of generalized operators or support functions. The implicitly active components are the Job Management System, the Internal File System, and the External File System. The support functions are:

(1) Those system jobs which allow the logical item structures, and the jobs that manipulate them, to be defined to the system.

(2) Those jobs which allow the structures to be indexed, searched, retrieved, restructured, etc.

The Job Management System is that component which:

(1) Interfaces with the interactive user

(2) Establishes his access rights

(3) Reacts to his commands

(4) Schedules the facilities of the system

(5) Binds the program variables according to the requirements of the job specification

(6) Interpolates implicit or explicit conditional item access and restructure operations

(7) Manages and sequences the tasks which comprise the job.

The Internal File System is that component which interfaces with the job task and Job Management System to provide services at the item level. The Internal File System has cognizance of , and management responsibility for, the data pool, a structured item which contains all subsystem directories and tables as well as the common data base of the users.

The External File System is that component which interfaces with the job task, time-sharing monitor and hardware, and Internal File System to provide services at the cell level. A symbolically named contiguous sequence of cells is called a train. Trains may be referenced symbolically, and train cells may be addressed relative to the head of a train. The External File System has cognizance of, and management responsibility for, the System Virtual Store, an address space for the Data Pool, those trains which comprise the program library, and data trains managed at the cell level.

This report develops these concepts after discussing system criteria and examining the features of several existing and proposed file systems. The structure and functional specification of a prototype Coherent System are developed. It is not the intent of this report to achieve a complete system specification, nor is such an objective claimed. In particular, considerable detail design must be carried out in the area of the Job Management System and the System Support Functions. It has been demonstrated, however, that a Coherent System which offers a new dimension of service to the problem or goal-oriented user in an interactive real-time environment can be developed.

SECTION I. INTRODUCTION

1.1 SCOPE OF STUDY

The Computer Research Laboratory of NASA's Electronic Research Center is responsible for developing the basic computer techniques and systems to be used in connection with NASA's post-Apollo space missions. These techniques, although not specific to particular space projects, are expected to influence or be adapted to space projects which will be operational from 1975 to 1985.

The Data Management System Study is the first phase of a program which will lead to the development of techniques for handling the vast amounts of digital data associated with the control and scientific experimental aspects of post-Apollo missions. The study consists of four phases, which are documented in this report:

(1) investigation of data management system features

(2) evaluation of current and proposed systems

(3) development of system concepts

(4) specification of a prototype system.

The problem environment of the data management system is assumed to be multiuser and multifunctional; it will include such functions as:

- Tracking and trajectory calculations
- Monitoring
- Recording and control of experiments
- Control of spacecraft maneuvers
- Data processing tasks, e.g., inventory control and information retrieval from stored libraries
- General scientific computing performed in conjunction with on-board experiments
- Liaison and data communication between ground-support and spaceborne computer systems.

Consequently, the data management system is assumed to be oriented towards a time-sharing mode of operation in which interactive and noninteractive tasks coexist.

The problem can be characterized by the following aspects:

(1) The need to maintain and access a large, on-going data base of scientific data, programs, and system tables

(2) The need to provide simultaneous access to the data by a number of scientists, technicians, programmers, and real-time processes

(3) The need to accommodate real-time tasks concerned with the control of spacecraft and experiments, the telemetry of experimental data, and the communication and message control of experimental and control information

(4) The availability of high-performance computers, memory systems, and input and display devices.

The scope of study encompasses the following areas:

(1) Job Management System (command language and job control)

(2) Internal File System (data definition language, directory structure, and logical data services)

(3) External File System (secondary storage management and physical data services).

The areas of procedural languages and translators, time-sharing and multiprogramming control, and machine organization and configuration are outside the scope of the study. Also outside the scope of this study are specific mission-dependent considerations.

The hardware environment of the data management system is assumed to have very broad capabilities. It can be expected that in the post-Apollo era, very high processor speeds (possibly with multiprocessing configurations) and very large capacity random-access storage devices will be available. The hardware system should, therefore, impose minimum limitations on the structure and functioning of the data management system.

Since the External File System performs many functions normally incorporated in a time-sharing executive, it seems reasonable to design the time-sharing executive in conjunction with the File System so that each can take advantage of the functions provided by the other. The design of programming language translators, both at the compiler and assembler levels, should also be influenced by the data management services available.

1.2 CHARACTERIZATION OF THE SYSTEM

The design perspective for the data management system can best be expressed by first describing some gross characteristics which have been taken as a model for the computation facility as a whole.

(1) Multiuse

The system to be developed is a multiuse system. At any one moment it may be used for many independent tasks of different character, such as interactive program development, scientific computation, data retrieval, and real-time control. This is in antithesis to a system which is dedicated to a single use, such as airline reservations or process control.

(2) On-Going Data Base

The system has the responsibility for maintaining the integrity of an on-going data base which, except during certain updating operations, is always accessible whether or not it is actively being used.

(3) Common Data Base

The data base should be capable of being viewed as a resource which is common to many users and processes which may or may not be externally coordinated. The common data base must be protected from collisions of simultaneous use, and access rights may be defined and checked so that data security is not violated. The data, in this sense, may "belong" to more than one user, process, or application, and may have a life which transcends any one of them.

(4) Job Library

The system should file and remember the definition of commands and jobs so that highly-parameterized, general-purpose programs can be used in different ways in different processes, yet be initiated simply by invoking prestored job descriptions.

(5) Program Coherence

It should be possible to expand the command repertoire of the system by simply cataloging new programs and job descriptions. All programs and subsystems should be "coherent" in that a common command directory, and directory search routines should be used, and standard methods of program call, parameter binding, and result communication should be employed. The effect of program coherence is to make all existing system programs the "tools" of any system user or tool maker.

(6) Control Coherence

It should be possible to develop lower strata of the command directory within the confines of a job so that, in effect, subsystems can be constructed as if they interface with a single user. Commands issued within the subsystem should be interpreted via the local directory. If no entry is found, the next higher level directory is searched. The motive for this control philosophy is to permit decentralized systems to be developed and used independently without loss of control at the highest level, and without the need for command uniqueness outside a local domain.

(7) Prerequisite Scheduling

In addition to a job definition, the user may define a condition which must be satisfied before the job is to be executed. The condition may involve independent external events, the value of data base items, an external message (e.g., job request), the value of a real-time clock or interval timer, or a combination of these. This results in a system which is far more flexible than one which is based on operator actions alone. It permits the coexistence of jobs which are interacting with the user at the console, real-time control functions triggered by a timer, real-time control functions triggered by asynchronous external events or the data base state, communication message triggered tasks, and background low-priority data processing or monitoring tasks.

(8) Real-Time Scheduling

Deadlines may be assigned to jobs, either at the time of definition, as for periodic real-time tasks, or at the time of job run request. Priorities may be assigned to resolve conflicts of competing jobs when the system is saturated and deadlines must be violated.

1.3 THE COHERENT SYSTEM STRUCTURE

Because the organization and functional design of the system described in this report have achieved a new degree of coherence between control functions, data management functions, system support functions, and mission-specific functions, the overall system will be referred to as the Coherent System. The overall structure of the Coherent System is shown in Figure 1-1. The system software, which is the main subject of this report, consists of the Job Management System, the File System, and a set of System Support Functions. The Job Management System and File System directly furnish executive control and data services, respectively, to all jobs.

The users also have available to them, in their repertoire of commands, a set of System Support Functions which perform such basic necessary services as item and job definition, query, data maintenance, and message processing. The Job Management System and the file systems, together with the System Support Functions, are called the Coherent System. The File System is composed of the Internal File System (IFS) and the External File System (EFS).

The division of responsibility between the Internal File System and the External File System is determined by whether the service is provided on the basis of the logical data entity, called the item, or the physical data entity, called the cell. The IFS manages the item, or composite logical data structure, and the EFS handles the composite physical data structure, called the cell. Each cell should be thought of as an address space which fits into a larger address space as a mosaic. The largest address space is called the "virtual store." The EFS also manages the allocation of tracks (fixed addressable space) on external devices to trains (relocatable strings of cells). These definitions are summarized in the following glossary.

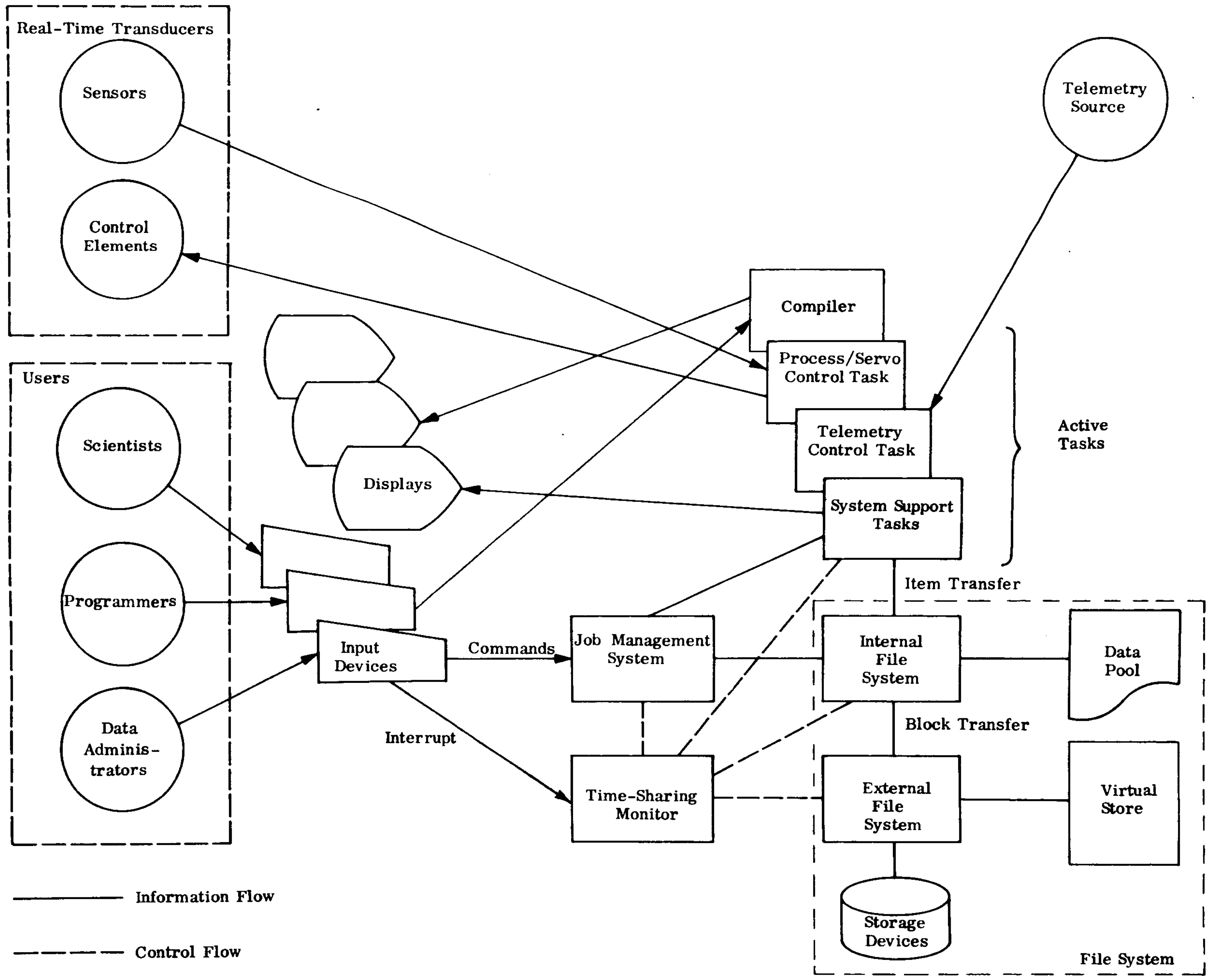


Figure 1-1. The Coherent System

Job Management System (JMS) — System component which recognizes commands, schedules jobs, provides task-to-task linking, and binds program parameters.

File System — System element which manages access to data and storage space. It is made up of the IFS and EFS.

External File System (EFS) — System component which manages the access to blocks and allocates trains to tracks on physical media.

Internal File System (IFS) — System component which manages the access to items. It allocates and formats blocks and interfaces with the EFS at the block or higher cell level.

Track — A contiguous addressable storage area on a physical medium, allocated to a train by the EFS.

Train — A string of one or more consecutive cells, subject to movement from track to track by the EFS. A train is an arbitrarily long contiguous data element which is symbolically named by the user and is undefined to the IFS. It is defined to the EFS, and accessed through the EFS.

Cell — One of a number of fixed-length data entities which is symbolically addressable by the system user. The lowest level cell (the bit) may be recognized only by the CPU; a higher level cell, such as the byte or word, may be recognized by the I/O channel. A still higher level cell, such as the block, may be recognized by the EFS. Thus, cells are a nested or a tree structure of storage compartments with the lowest level fixed. The entire virtual store of the system may be thought of as the highest level cell.

Item — A data entity whose name and structure are known to the IFS, and which is accessed only by the IFS. Several items may be allocated to the same block by the IFS, and a single item may extend over several blocks.

Task — A program which has been defined and bound as a job or job component.

Job — An entity whose name and formal parameters are assigned by the user, and defined in terms of previously defined programs and jobs.

SECTION II. SYSTEM FEATURES

This section discusses the system features which are pertinent to the NASA data management system. In effect, these features represent the criteria by which the design approaches to be studied will be evaluated. These features are presented from four points of view:

(1) Design Objectives — Discusses the overall goals of the system, without regard to the various ways in which these goals may be realized.

(2) Operations — Discusses the various system functions and capabilities for accomplishing the design objectives.

(3) Structures — Discusses the system components (i.e., tables, data structures, and program module structures, their composition and interrelationships) used to perform system operations.

(4) Language Elements — Discusses the system commands and service calls which may provide an appropriate interface with the system users and programmers.

The JMS, IFS, and EFS are discussed separately, but each follows the preceding breakdown. These system features are summarized in Table 2-1.

TABLE 2-1. SYSTEM FEATURES

A. DESIGN OBJECTIVES

JMS	IFS	EFS
Adaptability to Changes in • User language • User jobs Responsiveness • Ease of use • Speed of job request • Speed of job execution Efficiency • Utilization of equipment • Sharing of jobs and programs Reliability • Control of access • Error recovery	Responsiveness • Ease of use • Novice training • Quick response, search, and update Adaptability • Independence of logical data structure from EFS • Independence of logical data structure and program • Ability to combine data in unforeseen ways • Language and command definitional capability Efficiency in • Block utilization • Data representation for storage • Indexing arrangements • Retrieval strategy • Updating methods • Sharing of common data Reliability • Control of authorized access • Error recovery	Adaptivity to Changes in • Configuration } Hardware • Device types } Environment • Monitor } Control-software • Job executive } environment • IFS } • User programs • Data base size • Data usage patterns Responsiveness • Random- and serial-block access • Look-ahead • Minimum number of accesses • Block relocation that should change only one index Efficiency • Space utilization on storage device • Data utilization by sharing common data • Use and learning ease Reliability • Access control by classification or lockwords • Protection against usage collisions • Error recovery • Usage accountability

TABLE 2-1. SYSTEM FEATURES (Contd)

B. OPERATIONS

JMS	IFS	EFS
<u>User Functions</u> • Job execution • Job definition <u>System Functions</u> • User definition • Log-in and Log-out • Accounting • Scheduling • Job management • Parameter binding <u>Interface with</u> • User • Program • Monitor • File system	<u>User Functions</u> • Query • Editing • Updating • Report generation • Program entry • Program execution • Novice training <u>Interface with</u> • User • Program • External file system • Job management system <u>System Functions</u> • Translation of data values (input, output, and storage) • Data base updating • Directory updating and indexing • Data search • Data search look-ahead • Maintenance of data usage statistics • User accountability • Backup and failure recovery (job and data restart points)	<u>Data Access</u> • Block retrieval • Block storage • Buffer control <u>Storage Organization and Optimization</u> • Look ahead Page swapping (primary to secondary movement) Level changing Usage statistics Block grouping by process • Device allocation <u>Data Protection</u> • Edition control • Movement to archives • Lockword <u>Directory Maintenance</u> • Block address table • Cell name list • Usage tally (see storage optimization) • User table for storage accountability

TABLE 2-1. SYSTEM FEATURES (Contd)

C. STRUCTURES

JMS	IFS	EFS
• User list • Job definitions • Binding list	General Considerations • Data base structure and size • Variable versus fixed block length • Intra-block structure and format • Data linkage • Priority ordering of data segments in files System Structures • Logical data directories • Data file dictionary (item name dictionary) • Indexes • Access rights table Item Structure • Logical subdivision and relations among user items • Degree of nesting permitted System Program Modules • Modularity • Standard program interfaces • Generality of program functions	Track Structure as a Function of • Cell structure • Device type Cell Structure • • Block grouping by: Process User Segment Item (by IFS) Directories • Cell map • Segment dictionary System Program Modules • Modularity • Standard program interface

TABLE 2-1. SYSTEM FEATURES (Contd)

D. LANGUAGE ELEMENTS

JMS	IFS	EFS
<u>User Languages for</u> • Job run request • Job definition • Program specification • Parameter qualification	<u>User Languages for</u> • Program specification • Program execution • Data definition • Data entry • Query • Output format specification <u>Programmer Languages for</u> • Data updating • Data retrieval • Report generation • Task calling • Control transfers <u>System Languages</u> • Data coding schemes • Input/output formats • Interface with external file system • Interface with JMS	<u>User Program Interface (Job Task, IFS, or Monitor)</u> • Programmer service calls for symbolic cell or next block Open Close Read Write • <u>Message returns to caller</u> Normal return Undefined name Illegal read/write Temporary lock (try again) Write collision Active maintenance (lock on a table being updated)

2.1 INTERNAL FILE SYSTEM FEATURES

The IFS features summarized in Table 2-1 are discussed in the following paragraphs.

2.1.1 Design Objectives

2.1.1.1 Responsiveness. The primary design objective of the IFS should be system responsiveness to user needs. To the extent that the user deals directly with the IFS, it should be easy to use and learn. It must provide quick response to service requests, and rapid handling of search and update operations.

2.1.1.2 Adaptability. The IFS must be capable of adapting to a wide variety of user needs and environmental changes. In order to extend the useful life of various parts of the system and to minimize the implications of changes, the logical structure of data should be kept independent of both the EFS and the using programs. The system should also be able to combine and use data in unforeseen ways, so that the data structures and organization do not rigidly determine the ways in which data may be used. Finally, the user should be allowed to define his own languages and commands to the system, to accommodate special needs.

2.1.1.3 Efficiency. If the IFS is to meet effectively all the demands placed on it, operating efficiency is an important factor. To make maximum use of the available storage, methods of representing data for storage and methods of utilizing the space within data blocks should be carefully considered. Data of interest to more than one user should be capable of being shared, with proper attention paid to protecting the data and, where necessary, providing control over access to it. Indexing arrangements are probably the crucial factor in determining the speed and flexibility of accessing data. Data retrieval strategy and the methods of updating and maintaining the data base will also play key roles in determining system efficiency.

2.1.1.4 Reliability. In a real-time, spaceborne environment, reliable system operation is of the utmost importance. The chief concerns of the IFS in this area are the protection of data, by controlling who is allowed to access the data and change it, and the provisions for recovery and retry in case of system errors of any sort. (Table 2-2 contains several examples of potential data threats and protection devices.)

2.1.2 Operations

2.1.2.1 User Functions. The IFS user will ordinarily be thought of as a human with a problem to solve, but task programs may also be considered users, inasmuch as a task program may call on services provided by the IFS. For maximum flexibility, both kinds of users should be able to call all IFS services, although the appropriate languages for doing so need not be the same.

Since querying or obtaining information from the system is the primary user function, the facilities provided are extremely important. There should be a variety of ways of specifying conditions under which data is wanted. For the user who is not intimately familiar with the data base, it would be helpful to have a dialog query capability, in which the user would ask a series of increasingly specific questions, each based on the results of the previous ones, until the desired item was found. This dialog could also be the chief means of training novices in the use of the system.

Other user functions that may be called on by task programs as much as by human users include editing or arranging information for some specific purpose, updating the data base, and generating reports.

Finally, the human user requires facilities for entering programs into the system and calling for the execution of these programs. These functions would belong to the Job Management System, if it existed separately.

2.1.2.2 Interfaces. The IFS occupies a rather central position in the system since it interfaces with human users, task programs, the EFS, and any Job Management System. However, the IFS insulates these subsystems from one another as well as connects them, so changes to one subsystem should have a minimal effect on others.

2.1.2.3 System Functions. The system functions are the built-in, intrinsic functions of the IFS, and bear a large part of the responsibility for achieving the IFS's design objectives. First, there are the functions dealing directly with data, including translation of data values for input, output and storage, data base updating, and data retrieval. When data retrieval is done sequentially or in some other pattern, it should be possible to increase its efficiency by performing a look-ahead operation in conjunction with the EFS. Next, there are supervisory data functions, including directory updating and data indexing (which should be done automatically whenever the data base is changed), and the maintenance of data usage statistics. These statistics may be used to reorganize the data base in a more efficient manner, either automatically or manually. Finally, the IFS should play some role in keeping track of each user's use of the system with the EFS, by providing backup and failure recovery facilities, through job and data restart points or other means.

2.1.3 Structures

2.1.3.1 General Considerations. Various general structural considerations affect the design of an IFS. The following structural features will be considered:

(1) The organization of the data base

(2) The expected size of the data base, and its implication on the system design

(3) The length (fixed or variable) of the data block exchanged between IFS and EFS

(4) The logical structure and physical format of the data block

(5) The ordering principle used to arrange data segments within files

(6) The facilities which should be provided for data linkage.

2.1.3.2 System Structures. System structures are tables maintained by the system to describe and provide access to the data base. These structures include:

(1) Logical data directories, which describe the logical structure of the data items and their logical position in the data base.

(2) Data file or item name dictionaries, which provide a cross-reference between the symbolic item names used by the external user and the structural or other codes used to identify the items internally.

(3) Indexes, which tell where in the data base certain data values may be found, thereby enabling the system to perform searches without accessing the data itself, or accessing it minimally. The type and amount of indexing are critical in determining the system effectiveness since search time can be minimized through adequate indexing, while excessively detailed indexing requires much time and space for maintaining the indexes.

(4) Access rights tables, which enable the IFS to keep track of which users are authorized to access each portion of the data base, and for what purposes.

Other system structures, such as a list of active tasks and their data requirements, and a list of users and the extent to which they use system facilities, may also be required if the IFS is performing job management functions.

2.1.3.3 Item Structure. Item structure refers to the logical structure of individual data items. Items are divided into subitems, which may be further subdivided and ultimately divided into fields or values. Certain substructures may be repeated an arbitrary number of times; also certain substructures may be optional. Relations among items may be expressed implicitly by the logical nesting of items within other items, or explicitly by means of directories or various kinds of data linkage. The degree of complexity permitted in item structure is a matter of importance. Allowing arbitrary complexity will entail a certain overhead in system development costs and running time, but may be justified because the system will be much less subject to change arising from a need for data structures more complex than those originally envisioned.

2.1.3.4 System Program Modules. The organization of system programs comprising the IFS must also be considered. These programs should be as modular as possible in order to facilitate implementation, debugging, and documentation, and in order to minimize the effects of changes. The use of a standard method for program interface will also contribute to these ends. Finally, program functions should be as general as possible so that the programs may lend themselves to uses not originally foreseen and thereby extend their useful life.

2.1.4 Language Elements

2.1.4.1 User Languages. Program specification languages are used to define task programs. A special language may be provided for this purpose, or else the system may be built to accept the output of any standard procedural language processor. Program execution languages call for the execution of programs and supply them with necessary parameters. The user functions of defining data structures and entering data require appropriate languages. Language definitional facilities would be especially helpful for the data entry function, particularly where large quantities of data are involved.

A query language is necessary to enable the user to retrieve information from the data base. The principal considerations here should be flexibility and the user's ability to obtain information in spite of a limited familiarity with the data base. The user must also be able to specify the form in which he wants the results presented. Hence, an output formatting or report generation language is required.

2.1.4.2 Programmer Languages. Programmer languages are those used by the task programmer to call on IFS services. The most important of these are data updating, data retrieval, and report generation. In addition, a control language is needed so that tasks can call for the execution of other tasks and so that control may be passed from task to task and between tasks and the control system.

2.1.4.3 System Languages. System languages are those used by the IFS itself when it operates upon data and interfaces with other subsystems. Data coding schemes compress data in order to save space and also, possibly, to prevent unauthorized access. Also, data must be formatted appropriately for input/output operations. The primary interface of the IFS is with the EFS. Symbolic block names and data blocks themselves are exchanged in both directions across this interface. The IFS also interfaces with the Job Management System (JMS); the IFS can use the JMS as an intermediary in dealing with the user, and the JMS can call on the IFS for loading task programs.

2.2 EXTERNAL FILE SYSTEM FEATURES

A summary chart of External File System (EFS) features is given in Table 2-1. Each item is discussed in the following paragraphs.

2.2.1 Design Objectives

2.2.1.1 Adaptability. The EFS is a basic software module which, while undergoing a continual planned evolutionary development, must maintain an interface with its environment. This interface must be relatively insensitive to both internal and environmental changes; that is, it must adapt easily to changes in the configuration and the equipment types, both central processor and peripheral devices, in its hardware environment. Similarly, it must exist in an evolving control software environment, consisting of a time-sharing monitor, JMS, and IFS, and must adapt to changes in these components while presenting a compatible interface with user programs. Finally, it must adapt to changing user programs, a growing data base, and varied data usage patterns while maintaining high performance levels for the user.

2.2.1.2 Responsiveness. The EFS must be highly responsive to both random and serial block access request. This implies that a fast (possibly hardware implemented) mechanism must translate the symbolic block name to a hardware device address, and that blocks which are normally processed serially should be assigned to a single storage track, if possible. Where data needs can be predicted, there should be mechanisms for using this information. Thus, when an access request for this data occurs, it can be optimally executed. The number of external device accesses required to respond to a random block request should be minimized. The amount of "housekeeping" due to physical cell relocation should be minimized to take full advantage of look-ahead information.

2.2.1.3 Efficiency. Along with response time, one of the most important performance measures of the EFS will be the efficiency with which it utilizes space on storage devices. This efficiency will be a function of both the amount of storage required for directory data and the degree to which common data is shared by various users. Another important aspect, from the user's point of view, is the ease with which the system can be learned and used.

2.2.1.4 Reliability. Integrity of the data base is the responsibility of the IFS and the EFS. While permitting common data to be shared, the EFS must control unauthorized access to data by using methods such as lockwords and user classification. There must also be protection against collisions of legitimate usage resulting from simultaneous independent attempts to write the same block. In the event of error due to either system failure or user error, there should be the ability to recover an earlier state of the data base and a method to reconstruct the state prior to error. Accountability for data usage should be built into the system so that measures of use and service can be derived.

2.2.2 Operations

The operations which the EFS must perform to meet its goals fall into the functional areas of data access, storage organization and optimization, data protection, and directory maintenance.

2.2.2.1 Data Access. The block is the smallest unit of data the EFS handles. The block will be defined as a fixed-length module of data — usually equal in size to the quantum of data allocated to storage by the time-sharing mechanism — called the page. The basic function of the EFS is to service program requests for the retrieval and storage of blocks which are named symbolically or are identifiable as sequentially accessed members of some train or segment. The train and segment are contiguous modules of data. The EFS will also allocate core space for buffers and other temporary working storage for programs.

2.2.2.2 Storage Organization and Optimization. In carrying out the data access functions previously mentioned, the EFS will organize the system storage media for efficiency in space utilization, data movement, and process execution. Inherent in the problem of mass storage of a large data base and the technology of storage systems in the foreseeable future is the problem of handling data at different levels of access. The characteristics of storage media are such that fast-access storage has a higher cost per bit than slow-access storage, and that the device with the higher total capacity is associated with the slower access time. Although new storage techniques now in the research stage* may become operational during the life of the data management system to be developed, it is evident that the need for several access levels will continue, and that the foregoing physical and economic relations will continue to hold. A major function of the EFS will, therefore, be the movement of blocks and trains from a current level to a more appropriate level of storage. When information about data needs is known in advance, it can be strategic to use whatever free processor time and storage space are available to move this data to the highest (fastest access) level available. The process of anticipating data needs is called "look-ahead." Look-ahead information can be obtained implicitly from arguments in a user command or explicitly in the program as "open" and "close" statements. Optimal management of primary (core) storage implies primary to secondary block movement ("page swapping"). Some of the storage management strategies proposed use statistics on usage to determine which pages should be swapped from core and the appropriate level of storage for a given block or train. Other look-ahead strategies involve block grouping by process, either through block movement or chaining. Since the effectiveness of these strategies has yet to be verified, and it seems difficult (if not impractical) to simulate them, there is a strong incentive to build a flexible storage optimization module into the EFS so that several techniques can be tested in practice.

*New storage techniques, which promise higher capacity with reduced access time and volume requirements, include laser optical memories employing photochromic media and linear magnetic memories employing acoustic wave propagation of magnetic domain walls.

2.2.2.3 Data Protection. Since many active processes share the computer system and many potential users share common program and data blocks, by implication the data management system accepts the responsibility for data protection. There are many events, deliberate and accidental, that pose potential threats to data integrity. A data protection device (or combination of devices) must exist to maintain data integrity in the face of these events. A discussion of these events and their pertinent protection devices appears next (see summary in Table 2-2).

TABLE 2-2. DATA INTEGRITY PROTECTION

Threatening Event	Protection Devices	Controlling Processes
Illegal Program Reference	Bounds Register/ Process Map	Time-Sharing Monitor
Block Write Collision	Block Edition (Generation) Control	EFS
Illegal Item Reference	Classification/Clearance Check Conditional Access Rights Check	IFS IFS
Illegal Block/ Segment Reference	Lockword/Password Check	EFS
Reading of Data Being Updated	Busy-Bit Lock on Item	IFS
Erroneous Program/ User/Device Action	Block Movement to Archive File	EFS

In a time-shared system, there may be two independent attempts to write a data block. For example, in an inventory system there may be two independent attempts to deplete the quantity of a resource held in insufficient quantity for both, say each depleting three units of a four-unit resource. If both processes read the initial quantity of four and then one after another write the net resource of one, there is an obvious error called a write collision. A possible protection device is to maintain an "edition number" for each block and to return the block with the edition number incremented by one if a change is made. The EFS will refuse to accept a block whose edition does not correspond to one higher than the current edition. This forces the process to reread the current edition of the block and again attempt the update process.

A process may attempt to read a symbolically named block to which it is not permitted access. If the block has a lockword that must be supplied with every read, then the EFS can detect an illegal read attempt.

Finally, an erroneous action on the part of the program, user, or storage device may have resulted in invalid data which is later detected. To allow recovery from this type of error, the EFS can write each edition of a cell in a separate storage area. By periodically retiring older editions to a low-level archival file, it should be possible to reconstruct the item as it existed at some earlier time.

2.2.2.4 Directory Maintenance. The directory maintenance function of the EFS will involve operations on several tables. Physical block movement for the purposes of level changing should involve only one table, the Block Address Table. If the symbolic cell name is other than a coded relative cell address in the user's virtual memory, then there must be a cell name list for converting the reference to a relative cell address. If usage statistics are maintained at the cell level, then a usage register must be tallied by the EFS for each cell reference. Finally, a user table should be maintained for storage space accountability, updated as space is acquired or released.

2.2.3 Structures

The EFS comprises a number of data structures and a program structure. The functions and properties of these structures are an important aspect in characterizing ane evaluating the system.

2.2.3.1 Track Structure. The track is a device-addressable string of physically contiguous storage areas ordinarily allocated to a cell or train of cells. The track structure will be a function of (1) the EFS allocation strategy, which attempts to optimize device utilization and responsiveness, (2) the cell structure employed in the EFS, and (3) the device type being employed and its address structure.

2.2.3.2 Cell Structure. In addition to the number of levels of cell structure employed by the EFS, and the size of the cell at each level, the EFS is characterized by the strategies of block grouping (e.g., by process, user, or segment) within the higher level cells. When managed through the IFS, blocks will ordinarily be grouped by item.

2.2.3.3 Directories. The two main directory structures within the EFS are the cell map, which maps the cells of the data pool and the process's virtual store onto tracks, and the Segment Dictionary, which maps symbolically named segments onto cells.

2.2.3.4 System Program Modules. The same features discussed with regard to system program modules in the IFS (Paragraph 2.1.3.4) apply equally to the EFS.

2.2.4 Language Elements

The EFS must provide language elements to interface with the storage devices and the user program. The device interface will be dependent upon device types used, but there may be some opportunity for device-independent design. Such commands as channel select, seek, read, and write must be available.

The program using the EFS services may be a job task, the IFS, or the time-sharing monitor. The EFS must respond to program service calls for a cell referenced randomly by symbolic name or sequentially with reference to some previously opened cell. Commands such as open, close, read, and write should be available.

Messages returned to the calling program should be identified as normal or specified according to type of error encountered, such as undefined name, illegal read or write, or the existence of a temporary lock that may be open on another attempt. The temporary lock may be due to a write collision or a busy bit.

2.3 JOB MANAGEMENT FEATURES

Job management features are summarized in Table 2-1.

2.3.1 Design Objectives

2.3.1.1 Adaptability. The structure of the Job Management System should be insensitive to changes in its hardware environment. From the user's point of view, it should adapt easily to changes in user language such as format of input, mode of interaction, and command repertoire.

2.3.1.2 Responsiveness. The Job Management System must be designed to be easy for the user to use; i.e., it must be convenient for the user to enter a job request. Once the job request has been entered into the system, the system should rapidly process the request to determine its deadline, priority, and urgency. The system should take whatever action is necessary to schedule the job and execute it so that deadlines are met.

2.3.1.3 Efficiency. The Job Management System should also be designed for efficient utilization of the equipment in the computing complex. Wherever possible, advantage should be taken of all possibilities of parallelism in computation, both in multiprocessing central computing units and parallel peripheral units. In order to accomplish this, jobs should be definable in terms of small separately processable modules with explicit interchange of information. Optimal advantage should be taken of sharable jobs and programs, especially by using re-enterable program modules.

2.3.1.4 Reliability. Access to restricted data should be effectively controlled by the system. This includes not only protection of the system module itself, but also control of access to private data and program modules. It should be possible to establish the access rights and need-to-know of individuals, as well as the security levels and access restrictions of data elements. Effective archiving and error recovery procedures should be built into the system to enable recovery from accidental errors and equipment malfunction.

2.3.2 Operations

2.3.2.1 User and System Functions. The system should be capable of accepting the definition of user and system jobs and executing them when called upon. It should be possible for an authorized individual to define the users of the system and their priority and access rights. Once the user list has been established, it should be possible for any authorized individual to "log in" with a simple action and execute any system or user job within known constraints. The system should provide adequate accounting of each individual's use of the equipment and the system. The Job Management System should schedule the execution of the job for efficient use of the system and the user's time. The system should determine the tasks to be executed, bind the program variables according to values specified, and control the task-to-task sequencing in the job.

2.3.2.2 Interfaces. The Job Management System should be designed to interface effectively with the elements of its environment. These consist of the user through a user language, the program through task management and executive services, the time-sharing monitor, and the file system.

2. 3. 3. Structures

The basic on-going data structures which the Job Management System will use are a user list containing names of the users and their access rights, and a job definition list which contains the definitions of the tasks and parameters of each system or user job. In addition to these structures, the Job Management System will also use dynamic structures such as the job queue and binding lists.

2.3.4 User Languages

The user language should allow predefined jobs to be executed and new jobs to be defined. The job execution request should permit job parameters to be specified in terms of constants or literals, names of data base items, and possibly names of data base items with a qualification which defines conditionally a subset of that item which is to be used as the argument of the job.

SECTION III. EVALUATION OF FILE SYSTEMS

Eight file systems (see Figure 3-1) were reviewed and, as far as information concerning them was available, an attempt was made to evaluate them according to the criteria discussed in Section II. The descriptions of each of these systems appear in Appendix A, but their major characteristics are evaluated in summary form in this section.

The study of the existing and proposed file systems has tended to underscore the fact that a system is usually developed to meet a given set of objectives within some well-understood constraints and, therefore, must be evaluated with those objectives and constraints clearly in mind. For example, none of the systems examined was purely an Internal File System (IFS) or an External File System (EFS). As a matter of fact, a good way to characterize and compare these systems is to chart the "coverage" they give to the main data management functional areas discussed in Section I: job management, internal data management, and external data management. Figure 3-1 shows this coverage in broad terms. Each horizontal bar in the figure is meant to convey the system's coverage in each of the three domains.

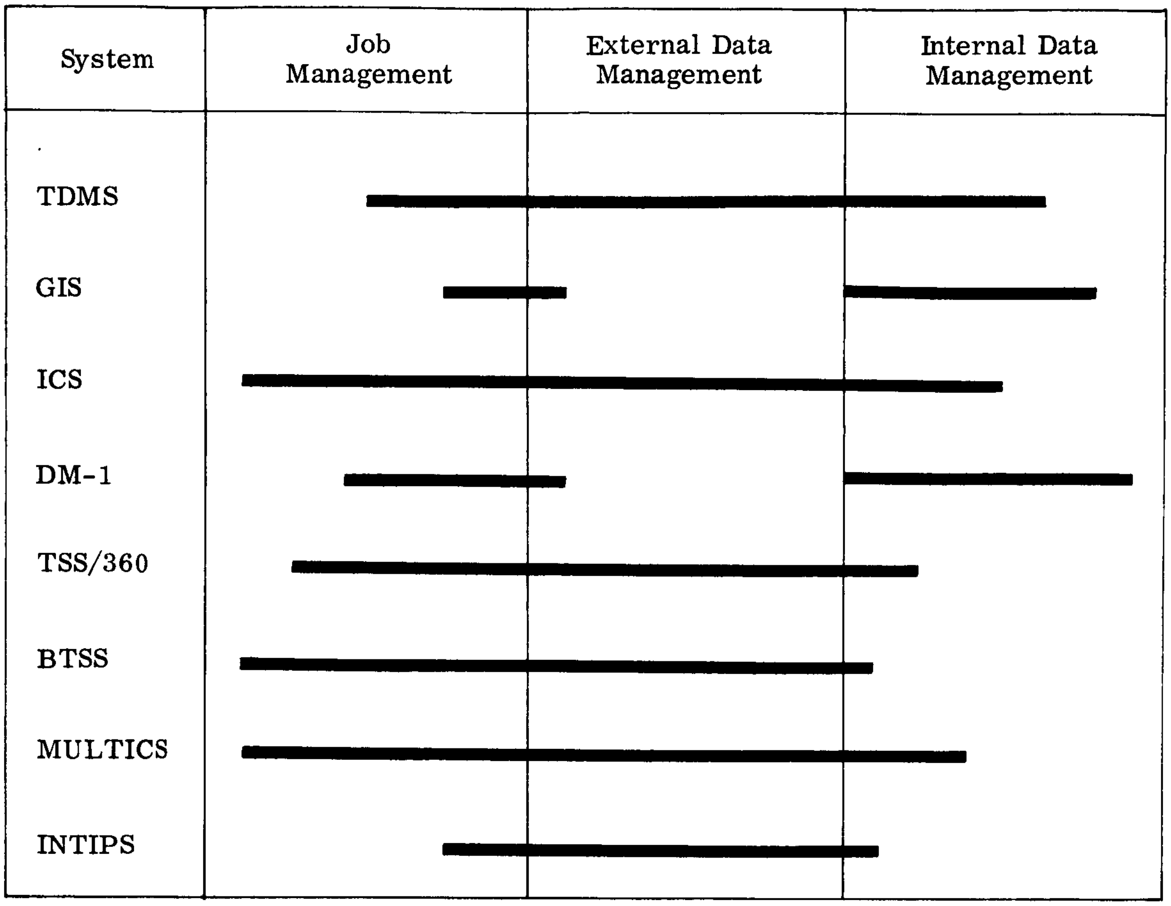


Figure 3-1. Summary of Capability

The major characteristics and an evaluation of the eight systems reviewed are summarized next.

3.1 TDMS

Time-Shared Data Management System (TDMS) is a closed system which encompasses a set of functions for file generation, maintenance,. query, and reporting. The TDMS files do not interface with programs other than the TDMS service functions mentioned. These functions are meant to encompass those services normally required by management level use of a structured file. TDMS interprets command language requests for these services, which are made via on-line display consoles.

TDMS appears to be a system that a nonprogrammer can easily use to define, update, and query a structured data file. A language is provided to evoke system-defined actions which define new data elements, load an initial file, maintain it, respond to queries, and produce reports. It appears, however, that there is no easy way for a programmer to interface with TDMS and provide for nonstandard actions in these areas; that is, there is no discussion of the programmer as a user of the system, and no direct data access services offered to the programmer as a user. As a consequence of this lack of interaction of TDMS with a programming system, it appears that batch-oriented or nonstandard data processing functions will have to be performed with a specially prepared version of the file.

The data base of TDMS is highly indexed; normally, there is a complete concordance of all fields. This means that the file is indexed by essentially each field value which occurs in the file, a strategy that minimizes search time in responding to queries. The results of making retrieval considerations dominant in the design are that a large amount of space is devoted to the index tables and a large amount of time is spent updating those tables when records are added or modified. These results are a manifestation of the space/time tradeoffs which are normally available to the programmer. If fast response to random queries is the most critical requirement for the query system, then TDMS indeed optimizes the right element. However, in a multifile data base there is usually a variation of response times which can be tolerated across the files. Some files have faster response requirements than others; and even within a single file, a predictable subset of properties will often enter into the conditional statement of queries while other properties are rarely qualified. It appears that a system which allows a tradeoff of these factors at the time a file is defined or loaded will be more efficient in the overall utilization of space and time. If the selection of which fields and field values are indexed can be made dynamically during the life of the system, then this tradeoff is adaptive to changing usage patterns or to experience with the use of the system.

The TDMS query language has one property which is extremely desirable in data management systems. It allows the interrogator to make effective use of the information he has concerning the structure of the data base which he is querying. The ability to qualify the scope of the condition through use of the "item name HAS" clause would be rendered more effective, however, if the item name were the name of an embedded file or record rather than the name of a field at the level of the embedded file within which the condition is directed. The inability of TDMS to take this approach seems to stem from a lack of recognition that file and record are two distinct entities and levels in the structure. In TDMS the phrase "repeated group" appears to be used indiscriminately to refer to either.

3.2 GIS

Generalized Information System (GIS) has a programming language and compiler in addition to a job request and query language. Only basic job management and query functions are provided in GIS. Other services must be obtained through programs written in the GIS language and compiled by the GIS compiler. The GIS language is oriented to the convenient specification of file manipulation procedures. A broad range of file structures and access strategies is specifiable in the GIS data definition language. GIS programs, however, run as jobs in the host Operating System, OS/360, which provides the bulk of job management and external data management services. The GIS internal data management services, unfortunately, are not available to the non-GIS language programmer.

GIS will offer a programming language, a user-oriented query and command language, and a set of basic data management functions such as data definition, update, and query. GIS programs are compiled by the GIS translator and become jobs which run within OS/360. Thus, although constrained to a particular operating system, GIS provides the user with a data management system of broad capability. There is a capability for incorporating user-supplied non-GIS tasks in the GIS job. These may be written in other procedural languages, such as FORTRAN or COBOL, but since they may make no use of GIS services and since they are bound to parameter and data specifications at compile time, they do not have the advantage of the data independence of GIS programs.

GIS has a query language with capability of expressing a logical condition of complete generality, and indexes which can be used for efficient retrieval of data which satisfies the conditional expression. In GIS, the indexes use physical, rather than logical, addresses to refer to data, thereby implying more direct access to data but additional maintenance due to physical data movement.

3.3 ICS

Information Control System (ICS) provides data management services to programmers using standard PL/1 and COBOL languages and compilers. This is accomplished by modifying the OS/360 to include executive level logical data services. (In this respect ICS is similar in concept to DM-1.) Although ICS is console driven, and conditional retrieval services are available for program parameter binding, there is no console user-oriented query language or on-going data base functions, such as storage, retrieval, and multiaccess protection.

DL/1, the conventions of data base description and processing in ICS, appears to be a system that will be reasonably easy to use since the standard programming languages of COBOL and PL/1 are continued under DL/1. The changes incorporated in the programmer's normal attack on a programming job appear only in his specification of the Program Control Block, his definition of the data base structure through the data base definition utility provided with the system, and his altered execution of I/O statements in his program. The terminal command language appears to be minimally structured to fit the specific requirements of the set of applications.

While many features of the general-purpose system have not been incorporated in the documentation reviewed on ICS, it is felt that these features could be incorporated as the system grows with use. The general-purpose Query Language, for example, could be added when the data base directory system is enlarged to accommodate a sufficiently large corpus of data.

The ICS-DL/1 system appears to respond to the requirements of the environment for which it was structured. The straightforward handling of data calls, described previously, does not change the job of the programmer to any major extent. The terminal system, with its associated command language, is probably sufficient for the environment for which it was designed.

3.4 DM-1

Data Manager-1 (DM-1) is a system with a broad range of job management and internal data management functions. DM-1 relies on an independent external data management system to allocate storage space and access symbolically addressed blocks of data.

The query function is embedded in DM-1's command language and command language interpreter. DM-1's internal data management services are available through executive service calls to programs compiled or assembled in the standard manner. Thus, there is no procedural level language or procedural language processor in DM-1. Programs using DM-1 services may be described and cataloged in the DM-1 directory. These programs, which may be general-purpose in nature, along with DM-1 support jobs (which are general-purpose) can then be combined into a set of composite jobs which are accessible to the (programmer and nonprogrammer) user through the command language. DM-1 support jobs, provided to define data structures, input data, maintain data, and print (display) data, are thought of as the nucleus of an expandable set of general-purpose data management functions. The conditional search function is a basic DM-1 service which is accessible to the command language interpreter and to the user. The DM-1 command language allows any command operand (job argument) to be qualified by a general logical condition so that jobs have a broad range of usefulness and a query is simply a display job with a qualified argument.

The separate management of data and programs with a mechanism for linking data to programs and jobs used in DM-1 will extend the life of the programs. The management of data, however, implies the ability for flexible description and processing of that data. The evolution of data from one format to another, the addition of fields to records, and the transformation of format are examples of characteristics of the environment for which DM-1 was developed.

The directory system, with its associated optional extensions to incorporate data field indexing and data item linkage, provides a flexible associative mechanism.

The concept of generalized programs is another design characteristic that should prove extremely useful in the development of software and application systems during the third and succeeding computer generations. One of the principal concerns of every programming or systems manager is the cost involved in the clerical aspects of programming. The same logical functions are repeated by application programmers utilizing varying algorithms. Generalization in DM-1 can provide both a means for reduction of the application system development effort and a mechanism for programmer standardization.

(Note: Except for TDMS, which operates only within the SDC time-sharing executive, the preceding systems may or may not be operated in a time-shared environment. The systems discussed next all provide external data management services within a time-sharing framework.)

3.5 TSS/360

Time-Sharing System/360 (TSS/360) is a multipurpose time-sharing system which is intended to provide a computing utility type of service to a large number of users. It attempts to disassociate its users from considerations of external data management by allowing programs to be compiled as if all data were in primary memory — a "virtual store" with an address space of 2^{24} eight-bit bytes. Since it will execute user programs stored as "data sets," it does not interfere with the operation of other user jobs. It provides a measure of logical item management by permitting data sets to be symbolically named, variable in length, and hierarchically structured.

In the tree structure of data sets within virtual storage, the ability to reference any branch symbolically is advantageous. The (partial) ordering within the structure being implied by the symbol (name) of the branch is useful, both from the point of view of the user working without the help of an Internal File System and from the standpoint of efficient operation, especially in changing the directories for additions and deletions.

Within a data set which is an indexed sequential file, logical records may be retrieved in the order stored, or in the order dictated by key data, as specified by the user.

The limitation to a tree structure is not as severe as it may seem on the surface, for other structures may be set up within that framework (although efficiency of operation is bound to suffer somewhat).

For the portion of the data structure which should be physically tree-structured, the access methods could be used to advantage by higher level file system programs. For other structures one would have to build his own External File System, which in turn uses the IOREQ facilities of TSS/360.

3.6 BTSS

The Berkeley Time-Sharing System (BTSS) is a general-purpose time-sharing system which was originally designed for experimental use by a rather small community of users. However, several versions of BTSS are being used in what amounts to a "computing utility" mode. In a sense, BTSS is more general-purpose than the other systems discussed because its users may interface with a system which looks very much like a medium-sized, general-purpose computer with a normal complement of peripheral equipment, with few restrictions on the type of programs it runs or software it develops and uses. The restrictions are that the programs use special input/output instructions (which look very much like the normal hardware input/output instructions). In this mode the system does little more than isolate this user from others and cause his programs to run slower because he is actually sharing the machine with other users, unknown to him. However, BTSS does provide other levels of software services and subsystems which are interactive with the user through an on-line console. Input/output services are provided for symbolically named items of arbitrary length. These items can be considered as either sequential or addressable (certain devices are inherently sequential and cannot be considered addressable), and the service is available at either the character, word, or block level.

The designers of BTSS started out with what they said was an experiment to test some ideas in time-sharing and man/machine interaction. What they accomplished must certainly be regarded as a highly successful operational system. The following can be listed as its accomplishments:

(1) It has introduced some important innovations in second-generation hardware for time-sharing; these compare favorably with what is being introduced in third-generation systems.

(2) It has an easily used interface with the user and has effectively supported the development of user-oriented subsystems such as an interactive editor, compiler, and computation system.

(3) It has been widely adopted by commercial time-sharing users.

The three-level structure of BTSS — monitor, executive, and subsystems — has contributed to the ease of developing user-mode software (including the executive and subsystems). The concept of an interface at the monitor/hardware level provides a general-purpose computer to the machine-language programmer and undoubtedly enhances the effectiveness of the system programmers who developed the higher levels of the system. This is an attractive principle which has a bootstrapping effect and perhaps should be more generally adopted in developing new time-shared systems.

The user interface (and system flexibility) provided by the combination of full-duplex operation and name recognition has much to recommend it. However, the structure of the system directories, particularly the use of randomized addressing in these tables, does not seem ideally suited to name recognition, which depends on the examination of partial argument strings and determination of whether they are unique. A table whose arguments were indexed (or chained) alphabetically would appear to be a better match to this function.

The structure of the sequential and random input/output operation is well conceived but there does not appear to be a need for, or an effective way to utilize, the very rudimentary variable record capability which is described. (None of the input/output instructions or file commands appear directed towards the access or processing of a variable length record.)

3.7 MULTICS

MULTICS provides a "computing-utility" type system to a large number of users. It can be compared directly to TSS/360 in its intent except that it provides a more flexible and adaptable executive (each user can "tailor" his executive interface somewhat) and allows more general data set and directory hierarchies in its external data management system.

The MULTICS file system design, however, is inclusive, and has overlooked few features of significance in the external data management area. Since it permits a very general hierarchy of file naming and structural membership, there is little need, if any, of file duplication for control purposes. Files and programs can be shared among specified users, with proper safeguards, and adequate provision is made for backup and recovery.

The only adverse criticism of the file system is that it is complicated by logical (internal) file management services which may have been easier to handle within the framework of an internal file management system.

3.8 INTIPS

Integrated Information Processing System (INTIPS), more specifically, the INTIPS external data management system, provides for management and retrieval of external data items which are at two levels, a block of 512 12-bit words or a defined structure made up of blocks. The user need not be concerned about which device will be allocated or even on what device his data is stored. The data management system will attempt to optimize storage utilization by moving data from one "level" of storage to another, depending on usage patterns.

The ability to manage data structured as a lattice provides the system with a significant degree of generality over systems which confine data structures to a tree. However, the more flexibility in the structure, the more critical become the design of the directories or indexes and the procedures for lookup. One of the challenges in designing file systems with nonsimple data structures is the discovery of ways to cut down the time needed for successive references to the indexes in locating an item. It is apparent that the INTIPS file system designer is at least conscious of this problem, as exemplified by the use of hash encoding.

A surprising omission is the apparent lack of a backup system. While this may be elaborate (automatically triggered backup or restoration of selected files) or simple (specially scheduled manually initiated utility program), some capability to protect against loss of a data base should be considered a requirement. Without doubt, at least a minimal capability is intended, as the design goals include immunity from catastrophies due to losing devices. This could be a significant cost area in a system which aspires to a data base of 10^9 blocks, as the number of tape reels needed to hold the backup data is in the neighborhood of half a million.

SECTION IV. THE COHERENT SYSTEM

It is clear, from examining the characteristics of the data management systems chosen for review, that there is a broad range of approaches to this problem, as well as a broad range of functional requirements which can be satisfied. The following can be identified as elements of an interactive system servicing an on-going data base for a community of users:

(1) An input string handler and "corrector," which can recognize commands and names, and which responds to control symbols in the input stream which "erase" characters, words, and lines.

(2) A basic time-slicing mechanism and monitor, which shares the system among several independent on-line users, without their knowledge, and with noninteractive scheduled and real-time jobs.

(3) A job management system (JMS), which interprets commands, assigns appropriate priority and urgency to tasks, schedules the hardware and software subsystems for effective utilization, and initiates the appropriate process to respond to commands.

(4) A file system, which provides access to either logical or physical data entities and also provides adequate safeguards to the data resources of the system.

(5) A set of language processing subsystems for interactive program development (assembly, compiling, editing, and debugging) and computation.

(6) A set of system services, callable by the user or programs, which allows the user to define new jobs and data structures, to enter data into the data base, to query the data base, and to use subsets of the data base (conditioned on content) as the values of job parameters.

4.1 THE COHERENT SYSTEM CONCEPT

4.1.1 Coherence of Programs

One of the most powerful tools, which can be used to create a coherent facility that can adapt to the needs of its users, is the capability to use and combine programs freely in different structures and contexts for performing specific functions. When both generalized and specific mission-oriented programs can be combined without restrictions (other than those determined by the functional logic) above the program level (that is, by the nonprogrammer user), then one can say the system possesses "program coherence."

The system described, in addition to possessing the program coherence attribute through the dynamic use of system commands, has the ability to define, name, and store program structures so that they may be easily and repeatedly invoked by the user. The structure, called a job, may contain programs or other jobs as its modules. The stored job definition may contain preassigned values for any of its component job or program parameters. Output parameters of any task may be bound to input parameters of any other task in the job.

Nonrepetitive jobs of a simpler structure can be invoked from an on-line terminal through a command or a string of commands. A command is a request to invoke a job whose function is to produce or display an instance of a single data item, such as a field, array, or statement. Because of this restriction, commands take the form of a single statement composed of an operator and string of operands. Operands may be nested, or qualified (or both); that is, the item produced by any command may be an operand of another command, or a part of the item which satisfies a condition may be an operand.

4.1.2 Coherence of Control

The principle of coherence of control implies that any program may function as a control organ which initiates and sequences the execution of other jobs. The mechanism for this is known as the job extension. A job extension is initiated by a message from a task to the Task Manager, the JMS module responsible for task-to-task linkage. The job extension specifies which job is to be executed and whether control is to be returned to the originating task at the termination of the new job.

One important use for a job extension is to permit a task to determine whether the prerequisites for running a job have been satisfied. The prerequisite condition evaluator may be a one-task job which is triggered solely by an interval timer overflow or some other specified interrupt, or by a normal job request or other message. Typically the prerequisite condition evaluator may check absolute time, data base state, or message queue to determine whether a job should be run or a status message generated. Prerequisite condition, and timer triggering are important aspects of real-time and control systems.

4.1.3 Responsibility for Program and Control Coherence

The responsibility for achieving a coherent system, as defined previously, is shared among the system elements, the program translators (compilers and assemblers), and the programmer. The Job Management System and the File Systems are necessary, but not sufficient components for system coherence. They must be supported by argument and result transmission standards used by the programmer or program translators. Basically, the Internal File System (IFS) should be used for all argument and result transmission. At task initiation time, temporary items containing the value of program parameters are created by the Job Manager. The compiler (or programmer) should read these items using the item service of the IFS. The names of the formal items are considered local to the program and are deleted at the time of task termination.

4.2 JOB MANAGEMENT SYSTEM

4.2.1 Responsibility of the JMS

The JMS supplies the mechanism for triggering and running jobs. A job is a unit of work which has been defined to the system by the user, triggered by an external request, and scheduled by the Job Management System. The mission-specific functions which the data processing facility is to serve demand the capability to execute a combination of real-time process control and user interactive functions. In addition to this the system must support program development and system checkout activities.

In order to accomplish this effectively, the JMS should be capable of interpreting a user-oriented command language with a repertoire of commands which are easily expandable by the user in terms of existing jobs and programs and newly created programs. When a job can be completely predefined (with no parameters), it should be possible to trigger its execution with a special interrupt line, a timer overflow, or a short job name keyed at an input keyboard. It should be possible to assign a deadline to a job so that jobs can be effectively scheduled. When the system is saturated, deadline conflicts should be resolved on the basis of user priority or job importance.

In addition to command interpretation and scheduling, the JMS will be responsible for job and program parameter binding and task-to-task sequencing of multitask jobs.

4.2.2 Jobs and Commands

A program is a module which is compiled as an entity and which is designed to satisfy some function. It is treated as a train of words and pages by the External File System (EFS) and uses the address space of a cell in the virtual store. Since a program is usable as a task, the program data must be submitted through the Program Entry service. Address space in the virtual store is assigned to contain the program, and the program description is entered into the Program Description List used by the Job Management System (JMS) and the File System. The program description, which establishes the information needed by the system to provide parameter binding and file services, consists of program name, parameter definitions, an external symbol dictionary, and formal file definitions. A service window is a formal program item or cell which is the source (or target) of explicit data service requests. Formal files are the formal data base items or trains which the programmer or the compiler assumes form the target (or source) of program data. The formal files are bound to system files (data base items or trains in the virtual store) by the task specification in the Binding List. A task is a program whose parameters have been bound in the context of some job or command. A binding list is prepared by the Job Management System and is used by the File System to establish the context (i.e., arguments) of the program.

A job is a structure, defined in terms of one or more subjobs, for accomplishing some system support or mission-specific function. It is defined to the system in a job definition request which places the job definition in the Job Definition List used by the Job Management System. The job definition consists of the job name, parameter list, and structure of the job in terms of tasks and jobs. The job structure is defined by binding subjob and program parameters, and indicating flow of control by using dummy variables and flow control statements when required.

A job structure with certain properties makes it possible for the system to offer additional services which improve the interface with interactive users. These jobs, which are called commands, have the restriction that they create or display exactly one data item as a result; and use data base items, substructures of data base items, or literals as arguments. As a result of this restriction, the system can allow commands to be nested and arguments qualified by conditional statements. Thus, the data item produced by a command can be used as an argument of a command "wrapped around it," such as "#Print (# Update (Mission File, New Experiments))." Also, a conditional search capability can be used by the command language interpreter to extract a portion of a data item which satisfies a logical condition and transmit that portion to the task as its argument. Thus, for example, a query function can be accomplished by using a generalized print command and qualifying the data item which is to be printed. For example, the command "Print (Propulsion System Status Where Spacecraft = Apollo and Launch Date > June 1970)" will cause the Print routine to receive from the Propulsion System file only those status records which pertain to the specified spacecraft and launch date.

4.2.3 Structure of the Job Management System

The data flow between the JMS and the task and File System (FS) is shown in Figure 4-1. The task uses the File System to obtain values of all formal items, including parameters and files. The equations which bind formal names to data pool items are given in the Binding List for the task. The FS uses the Binding List to respond to service calls.

Information flow within the JMS is shown in Figure 4-2. There are three broad functions performed in the JMS: job triggering, scheduling, and task binding.

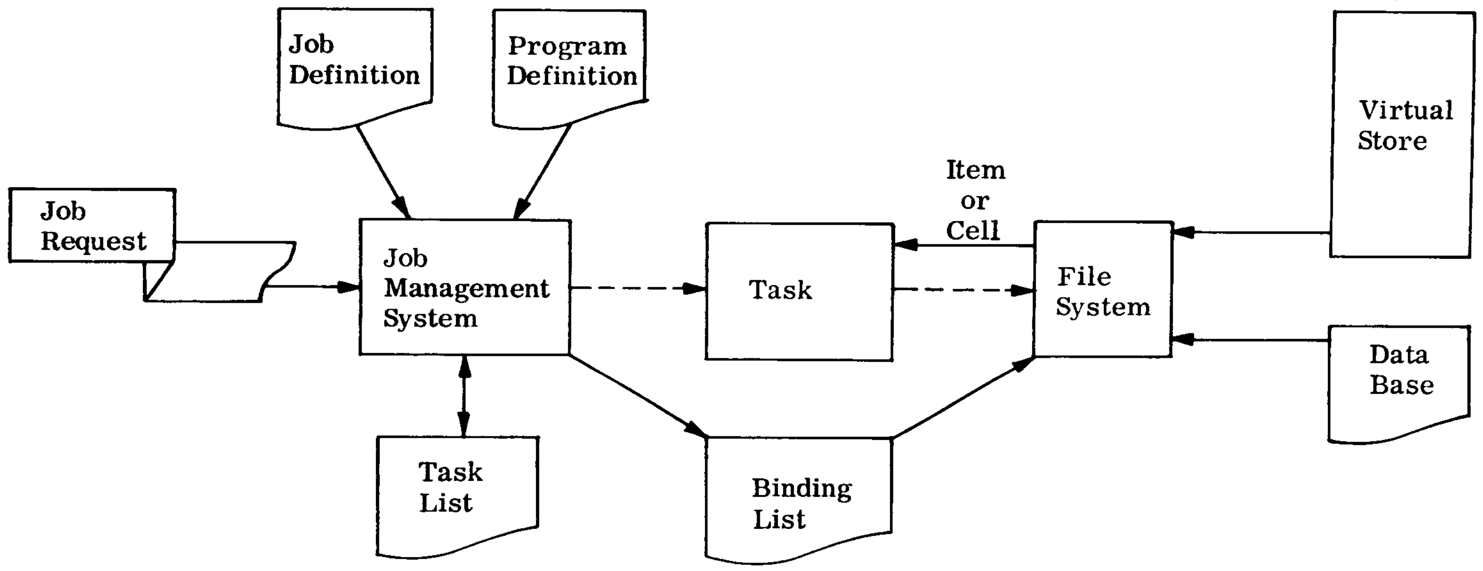


Figure 4-1. Interface Between Task, Job Management System, and File System

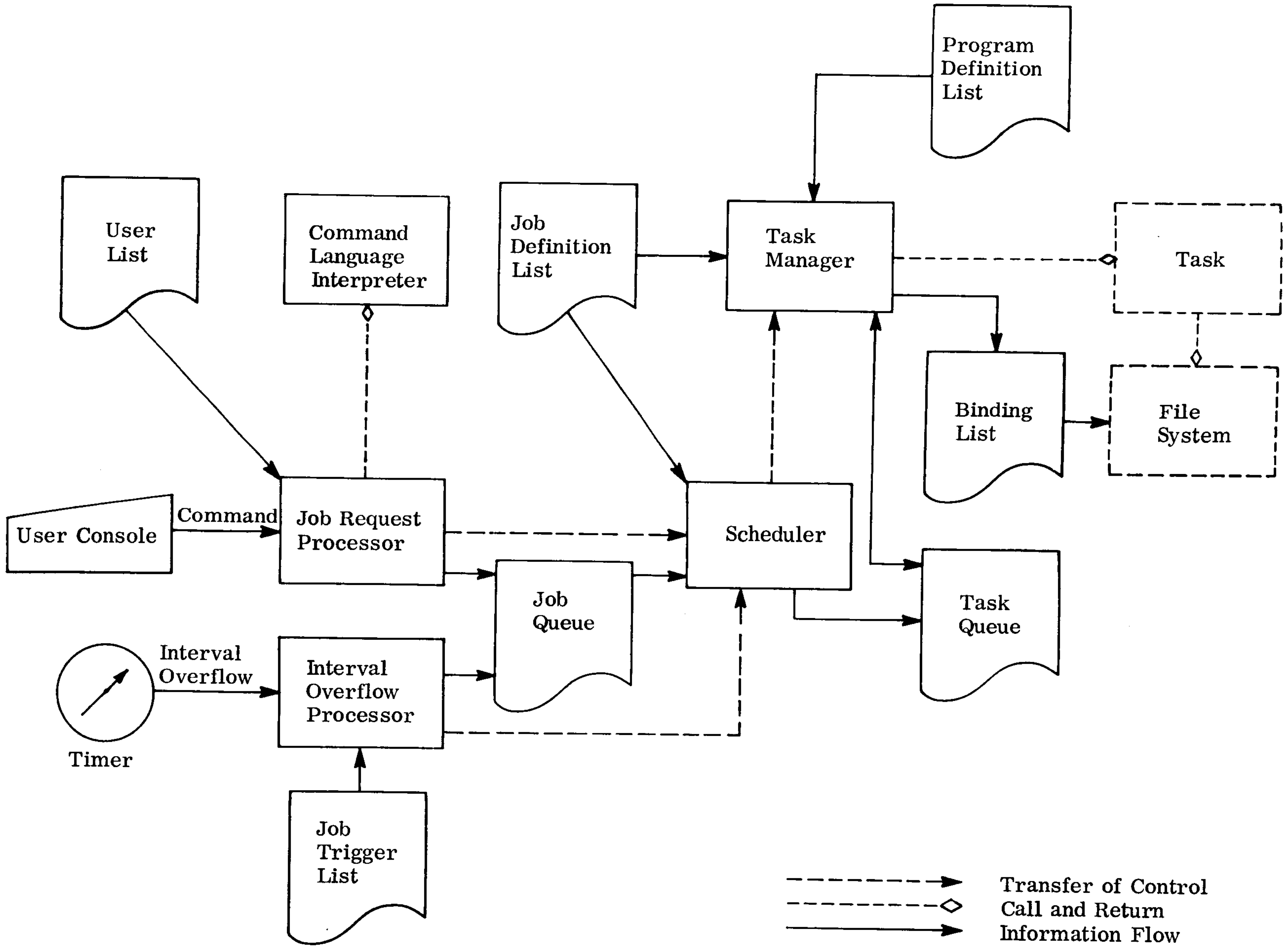


Figure 4-2. Job Management System

4.2.3.1 Job Triggering Functions. Jobs can be triggered in two different ways: either by the receipt of a job request, command, or other message, or by a timer overflow interrupt. The ability to trigger predefined jobs by means of interval timer overflow is important for real-time control applications. The Interval Overflow Processor uses a job trigger list to associate each timer overflow line with a job to be run. If the job depends on prerequisites other than time (such as the existence of a message or state of a data base item), a prerequisite condition evaluator job is triggered. If the prerequisite condition is satisfied, then a job extension to the desired job is issued.

Job requests, and messages in character stream form are processed by the Job Request Processor. By using a syntax table driven input scanner and parser, the Job Request Processor can accommodate a number of message, request, and command formats engineered to interface with the user.

4.2.3.2 Scheduling Functions. The scheduling strategy can play a large part in determining how well the system performs real-time processing. Since the job is the unit of work which is meaningful to the user, it is logical for the schedulable module to be a job. Since a job is a predefined entity in the system, running time, deadline, and priority estimates sometimes can be made before the job is triggered, and stored with the job definition.

The concepts of deadline, priority, and urgency require definition. Deadline will be defined as the latest time at which the job results should be available once the job is triggered. Priority is defined as the importance or relative weight of a job. Job priority is to be used to determine which job to favor when the system is saturated and deadlines must be violated. It may either be dependent on the inherent importance of the job or be assigned dynamically by the requestor (and depend on his "rank"). Urgency is the factor which determines the order in which jobs are initiated (the most urgent job is initiated next). The urgency of a job increases as time approaches deadline minus estimated running time. The Job Scheduler will read the job queue and the job definition and create a task queue ordered by job urgency.

4.2.3.3 Binding Functions. One of the precepts of the Coherent System is that the overall effectiveness of the system (and the programmers) is enhanced when many of the programs are highly parameterized and generalized, adaptable to many specific jobs in which they may be used, and largely independent of the on-going data structures to which they may be bound.

Binding takes place at several levels in the system. Programs must be defined to the system before jobs may be defined in terms of those programs. Jobs must be defined before requests may be made to run those jobs. These definitions and binding functions serve several purposes:

(1) The same program can be used in several contexts without changing the module.

(2) A program can be independent of the physical structure of the data base (locations of trains and files).

(3) A program can be independent of the format of an item maintained in the data base for IFS services.

(4) Tasks and supporting functions can be intelligently scheduled (e.g., file conflicts are known ahead of time, enabling task dispatching to be done with the status of all resources known; also, files may be opened before a task is actually begun, resulting in overall better responsiveness of the system).

(5) The system is made easy to use.

While it is desirable to have the flexibilities of constructing jobs from programs, and use programs (or jobs) in different data or procedural contexts, the design must take into account the fact that most jobs to be run in the system will not change in construction from run to run. The trap to avoid is a design which forces the user to follow a complicated procedure for those jobs which he rightly feels should be easy to run. This consideration alone demands that the job definition be a separate function from the job request.

4.3 DATA AND DATA SERVICES

4.3.1 The Data Resource

One of the basic premises upon which the Coherent System model is based is that the data of the center (or the mission) can and should be viewed as an on-going resource whose structure and useful life may transcend any one user, program, job, experiment, or project. (It is even reasonable to assume that just as a computer system and its support software may serve more than one space mission or space vehicle, a considerable amount of data will have a useful life which transcends the mission, e.g., data used for system checkout and test, benchmark problems, experiment control, and celestial navigation.) There is a strong incentive, therefore, to develop techniques which permit (and encourage) the design of data structures, and the maintenance of data aggregates which are decoupled, to a large extent, from the programs which may use the data. The system and techniques described accomplish these goals. Data structures may be defined, data aggregates built, maintained, searched, restructured, and displayed using system support jobs (commands). The same data, through logical data services at the programmer level, is also available to mission-specific programs for mission-oriented use.

As a consequence of providing this service, the File System must accept the responsibility for maintaining the integrity of data. It must provide for automatic recovery of data lost as a result of equipment, software, or user error. It is anticipated that there will be groups of users who are permitted access to certain data and denied access to others, and that given individuals may belong to more than one group.

At any one moment there will be a body of ephemeral data which will coexist with the on-going data base. This data will represent the working data of tasks, the data being communicated from task to task in multitask jobs, and the data being communicated between the user and the job. This data will be managed by the system as a service to the task, job, or user. Data pool items will be created by the Job Management System under the categories, Task Data and Job Data. Task Data is data created by the task for its own working use (scratch data), and Job Data is either data transcribed by the JMS from the job description (or job request) for use by the task, or that generated by the task for use by another task or the user. Task Data is deleted at the termination of the task, along with Job Data no longer required by the job or user. Job results are maintained until deleted by the user.

4.3.2 Levels and Modes of Data Service

Data services are provided in two modes on two levels. The level of service depends upon whether the commodity being serviced is an "object-oriented" entity, the item, or a "space-oriented" entity, the cell. (As mentioned previously, all item services are handled by the Internal File System (IFS), while cell services are handled by the External File System (EFS).) The external user normally deals with items, while system and user programs may require item or cell services, or both. Thus, a user program may use item services where the IFS will, in turn, use cell services. An analogy to this is found in programming, where the coder may deal in space-oriented entities and code at the machine instruction level, or he may deal in object-oriented entities with a higher, problem-oriented language. Yet, the compiler will, in turn, "code" at the machine language level.

An overall view of the levels of services performed by the File System is shown in Figure 4-3.

Character stream data in External Data Language is mapped to a train in Internal Data Language by the IFS mapping function (Data Entry), using the Item List, an IFS table which describes the structure of the data. The EFS mapping function, in turn, allocates device space and stores the train on a track. Data is transmitted to and from items and cells defined by the programmer.

The Coherent System will provide File System services in two modes, from the programmer's point of view:

(1) Explicit services, the traditional mode, where the programmer codes input and output statements along with processing statements.

(2) Implicit services, a mode made possible by the Coherent System approach, where the programmer codes only processing statements.

In the Coherent System, explicit services are the exception. The emphasis is heavily on implicit item services and implicit cell services. Typically, implicit item services are provided to programs coded in problem-oriented languages, where references are to item names rather than to locations or cells containing data. Implicit cell services are provided to assembly language programs, where references are to addresses in the program's virtual store. When either implicit item or cell services are provided, actual service requests are absent; the program is purely a processor, not a data transporter as well.

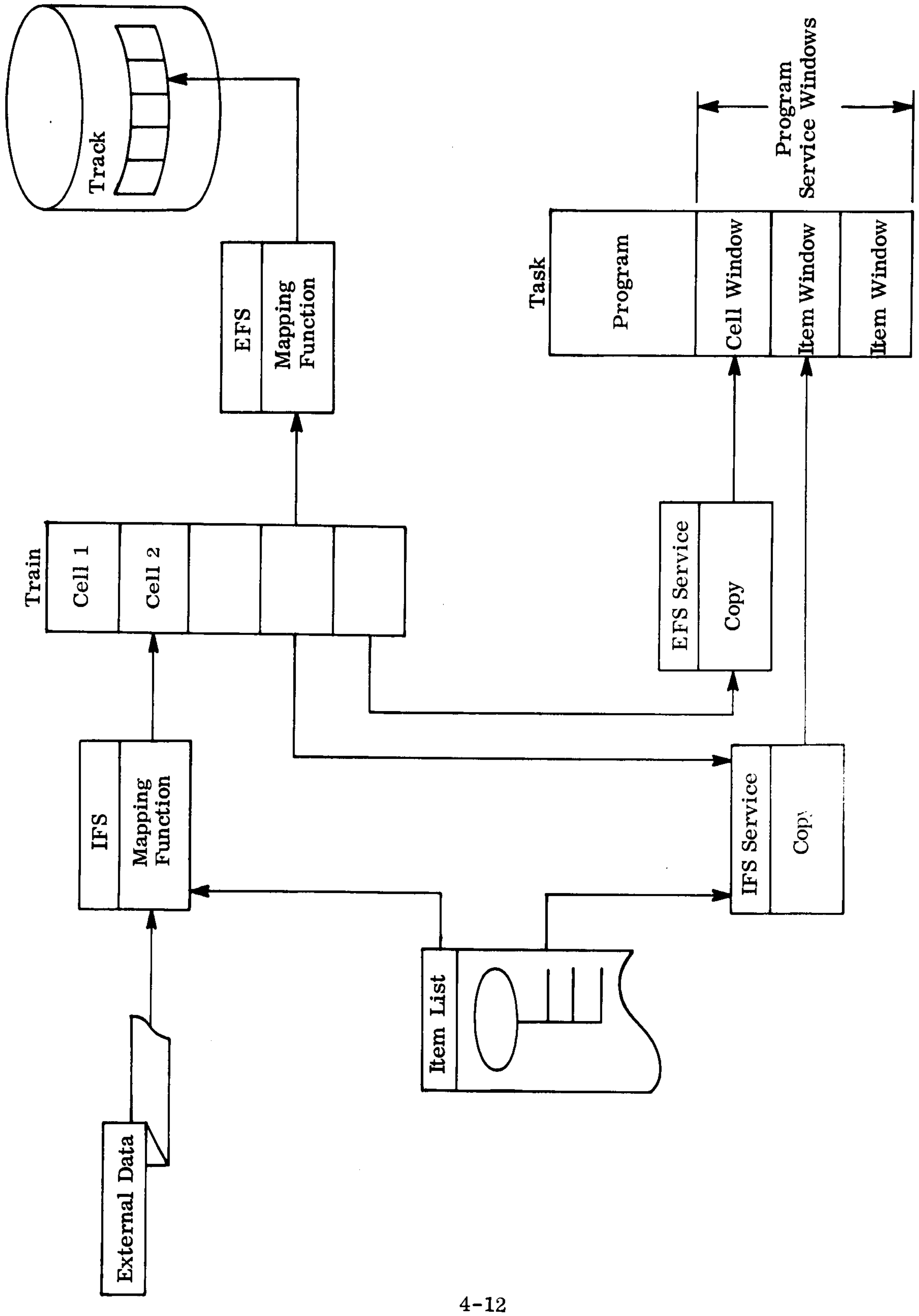


Figure 4-3. File System Services

SECTION V. THE EXTERNAL FILE SYSTEM

To set the stage for development of the EFS functional specifications for the Coherent System requires development of a set of premises which are in keeping with the design objectives. It is assumed to be desirable to strive for the full potential of the data management concepts developed in this study. Achieving this potential requires cognizance of the structure and tools of the whole system, and imposes a requirement that these be commensurate in capability to the File System in forming a unified whole. While a detailed treatment of the total system design is outside the scope of this study, it is necessary and appropriate to postulate other supporting software concepts and equipment features which are either within the current technology, or achievable in a practical sense within that technology.

(1) It is assumed that more than one task may be active at any one time. This situation may entail multiprogramming with one processor, or may be due to the use of more than one central processor jointly addressing a common memory in the execution of program instructions.

(2) Many of the programs may be simultaneously used by several tasks; therefore, the ability to construct and operate pure, or reentrant programs is a design objective.

(3) The burden of designing a program to manage a limited amount of primary memory should be lifted. The programmer should be able to design with the assumption that a virtual store is available at his command. The reconciliation of virtual addresses to real addresses, with the attendant "swapping" in and out of real memory during execution, will be the job of the monitor and File System.

(4) Intelligent management of the resources of the system demands a fluidity of structure, especially an independence of a program from physical storage assignments both with respect to itself and with respect to its data. Therefore, the system design should allow the construction and operation of programs which are insensitive to the physical location of its data and instructions, to a degree which is useful for total resource management. This imposes goals on the File System, the monitor, and the compilers or language translators.

(5) The compilers and the monitor should be designed to meet the system characteristics assumed in preceding items, and to achieve the File System's objectives, some of which aim for a level of programming and processing flexibility not heretofore attempted.

5.1 THE CELL HIERARCHY

The External File System deals only in transportation of strings of data. It is insensitive to the nature of the data, its logical structure, format, meaning or lack of meaning to the user, etc. It is interested in the size of the string, its source, and its destination either in storage or at a terminal device. A quantum of data addressable in some address space is called a cell. The EFS will transport data in "chunks," called copy cells, of various sizes dependent on the form of service and the terminals. It will be convenient to specify several levels of cells before developing the notions of transporting collections of these cells.

5.1.1 The Byte

The byte is the unit of size in which higher level cell sizes are expressed. It is defined as the smallest number of contiguous bits which may be addressed in primary memory by an appropriate machine instruction. Letting U denote this unit of size, the byte, then U represents some n bits according to the equipment chosen for the system.

Other cells will be specified by expressing their size as 2^n bytes or 2^nU. For example, one obvious copy cell should be of size 1U, the same as the byte; this cell would be used in connection with terminal keyboard devices. Another natural copy cell type might be the same size as the page. Perhaps these will suffice, but it could prove convenient to define an intermediate size which is a "natural" for containing display images.

5.1.2 The Page

At least one cell is defined as the unit of transfer for swapping purposes in and out of primary memory. There is at least one natural modulus in the CPU instruction set for the purpose of addressing the virtual store of a process. A real address in the CPU is formed by combining the location of the "page" (the "head" of the virtual address) with the relative address, i.e., the relative byte within the page. The latter is the "tail" of the virtual address, and most likely is specified within the instruction, as modified by indexing. It is possible that there is more than one natural cell of this type, but one will be sufficient to allow the development of the virtual-store management functions of the EFS.

5.1.3 The Train Cell

Before defining the train cell, it is important to recall the definition of a train: A string of one or more cell spaces (or simply "cells") identified by a symbol. The cells are ordered 1, 2, ..., C. For trains maintained in the virtual store, the cells are pages. The train cell (for a virtual store train) is the smallest power of two (number of pages) which will accommodate the storage of the train; that is, there exists an integer M such that $2^{M-1} < C \leq 2^M$. For a train of C pages, the train cell consists of 2^M ordered pages. The origin of the train is the same as the origin of the train cell containing it. The origin is located within the virtual store by having a prefix or head address assigned to it. The train cell is allocated to a track on physical store by means of a map in the EFS. Access modules from the track copied to core by the EFS consist of an integral number of pages. These relationships are depicted in Figure 5-1.

5.1.4 Copy Cells

The copy cell is the unit of data space to be used in holding images which are transferred between primary storage and terminal devices. Several levels of copy cells may be appropriate within the system. The types chosen are influenced by terminal equipment characteristics, the choice of cell size for the page (discussed previously), and the rule that its size is 2^kU.

5.2 TRAIN SERVICES

With the foregoing definitions, it can now be said that the EFS is concerned with the transportation and storage of the cells of trains. These services fall into two classes of service, implicit and explicit, rendered for different reasons and by somewhat different methods. Before investigating these, it may be noted that one requirement of the EFS, in order to accomplish any physical data movement, is that it must be able to construct and issue the hardware commands or instructions which cause the transfer of data to take place, and monitor their satisfactory execution. The design of this portion of the EFS would be heavily dependent upon the characteristics of the equipment. Also, because of the intimate interface with interrupt handling routines and the Process Manager, this module would be designed in conjunction with the monitor. Therefore, a Data Traffic Manager can be thought of as an EFS module which provides device services to the system processes, particularly to other EFS modules, although the specification for this module will not be developed further at this stage.

One class of services is concerned with intrasystem train service, that is, the movement of cells between primary memory and other storage media maintained by the EFS. This class of service is called "virtual store (train) management." The other class of EFS train service is concerned with the movement of cells between primary memory and terminal devices. This is called "terminal train management."

5.2.1 Virtual Store Trains and Implicit Services

The concept of a virtual store is a powerful one. Its realization in the form of a working system demands careful and integrated design of all system software (compilers, file system, the supervisor or monitor, etc.) and equipment features which make it feasible. It is a relatively recent development; its realization in today's systems is only moderately successful. Yet, here it is intended to advance to a new plateau of user power which stems from the combination of advanced data management concepts with those of the virtual store.

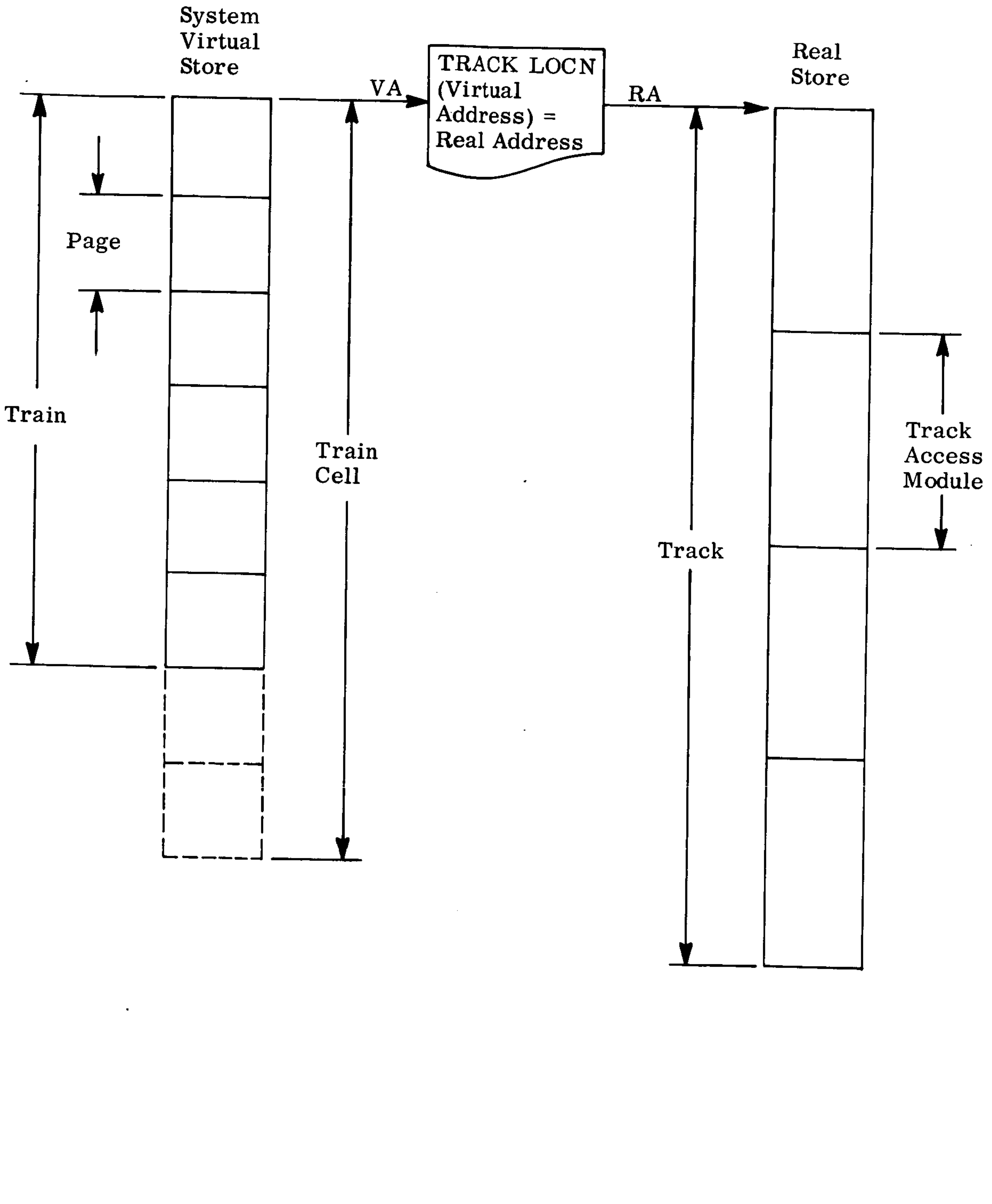


Figure 5-1. External File System Train and Track

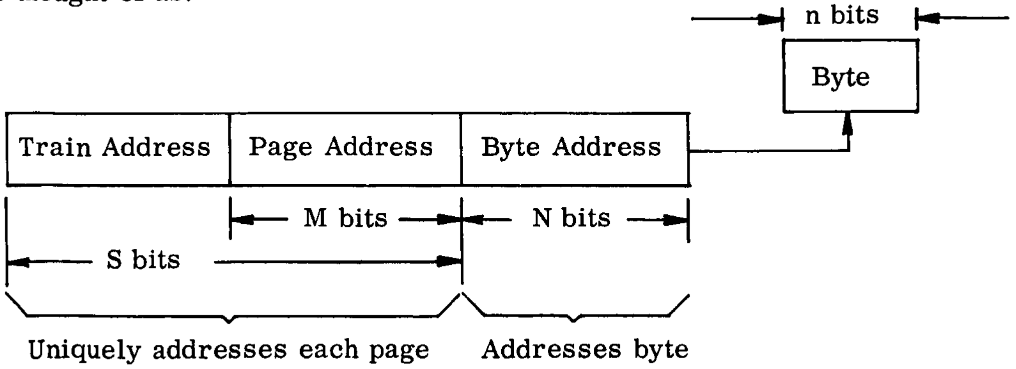
5.2.1.1 Coherent System Considerations. From the (assembler language) programmer's point of view, the virtual store is a symbolically referenced medium available to his processing statements. Other trains (besides his program) in the system, containing data or other programs, are immediately available for processing or linking. This powerful concept, especially in conjunction with other advanced notions such as the "purity" of the program (its ability to be executed as part of more than one process), demands an intelligent supervisor (an example of which is the MULTICS supervisor).

Beyond this, the Coherent System concept suggests that programs do not fix the virtual store trains, but rather reference a formal train. In this way the same program may be used in several contexts according to the actual virtual store trains which are bound to the formal trains, not within the program but within the job definition.

Further, because of our distinction between item services (IFS) and cell services (EFS), there should be no need ever for a program to explicitly read or write any portion of his formal train if it is bound to system trains which are part of the system's virtual store. The programmer merely regards it as part of his immediately accessible storage, and goes about coding his processing statements, unaware of the input/output aspect of computer programming.

5.2.1.2 The System's Virtual Store. The virtues of a symbolically referenced virtual store from a programmer's point of view were previously stated. The virtual store management functions also benefit from the notion of a system virtual store referenced by virtual addresses.

If the virtual store accommodates 2^S pages, virtual addresses run from 0 to $2^S - 1$. For a train cell consisting of 2^M pages, the full address of any bit in the train cell can be thought of as:



Uniquely addresses each page in the virtual store.

Addresses byte within a page.

A page brought into primary memory is located indirectly through registers used in the instructions referencing the formal train. These "base" registers effectively provide the head of the actual primary memory address of the page. The tail is, of course, the N-bit field shown above.

A virtual store map equates the symbolic train name to the virtual address of its train cell; TNLIST (TN) = VA.* Another table equates the VA to the track location; TRACKLOCN (VA) = RA (see Figure 5-1). Thus, once the train cell VA is assigned for a virtual store train, it does not vary with the physical location of the train cell, which may be altered by the multilevel storage management functions.

A diagram showing EFS tables and the system modules which use them is given in Figure 5-2. The Train Name List is entered only for initialization of the task, by the Task Manager. Subsequent requests for cell services by the program are interpreted via the Binding List and the Track Location Table. Important system data which is repeatedly referenced will be fixed in the virtual store or the real store (or both) and will be accessible without going through the binding process or the Train Name List.

5.2.1.3 <u>The Virtual Store Manager</u>. The Virtual Store Manager is that part of the EFS which resolves references to virtual address spaces. When reference is made using a virtual address, the Virtual Store Manager (VSM) must ensure two things:

(1) That the data involved is made physically available to the referrer.

(2) That a mapping is established between the virtual address and the real address.

There are two kinds of virtual addressing in the Coherent System's design concept. One is the reference made by a task to a part of its virtual address space, i.e., to a byte within a page of a train. Functions (1) and (2) in this case mean that the actual page must reside in primary memory, and that the mapping must be established in such a way that the reference may be carried out by the machine hardware in executing the instruction. This is discussed in more detail in Paragraph 5.2.1.4.

* The notation used in this report is in the form: f(x) = y, where f is the operator or table name, x is the input or "basis" argument, and y is the result.

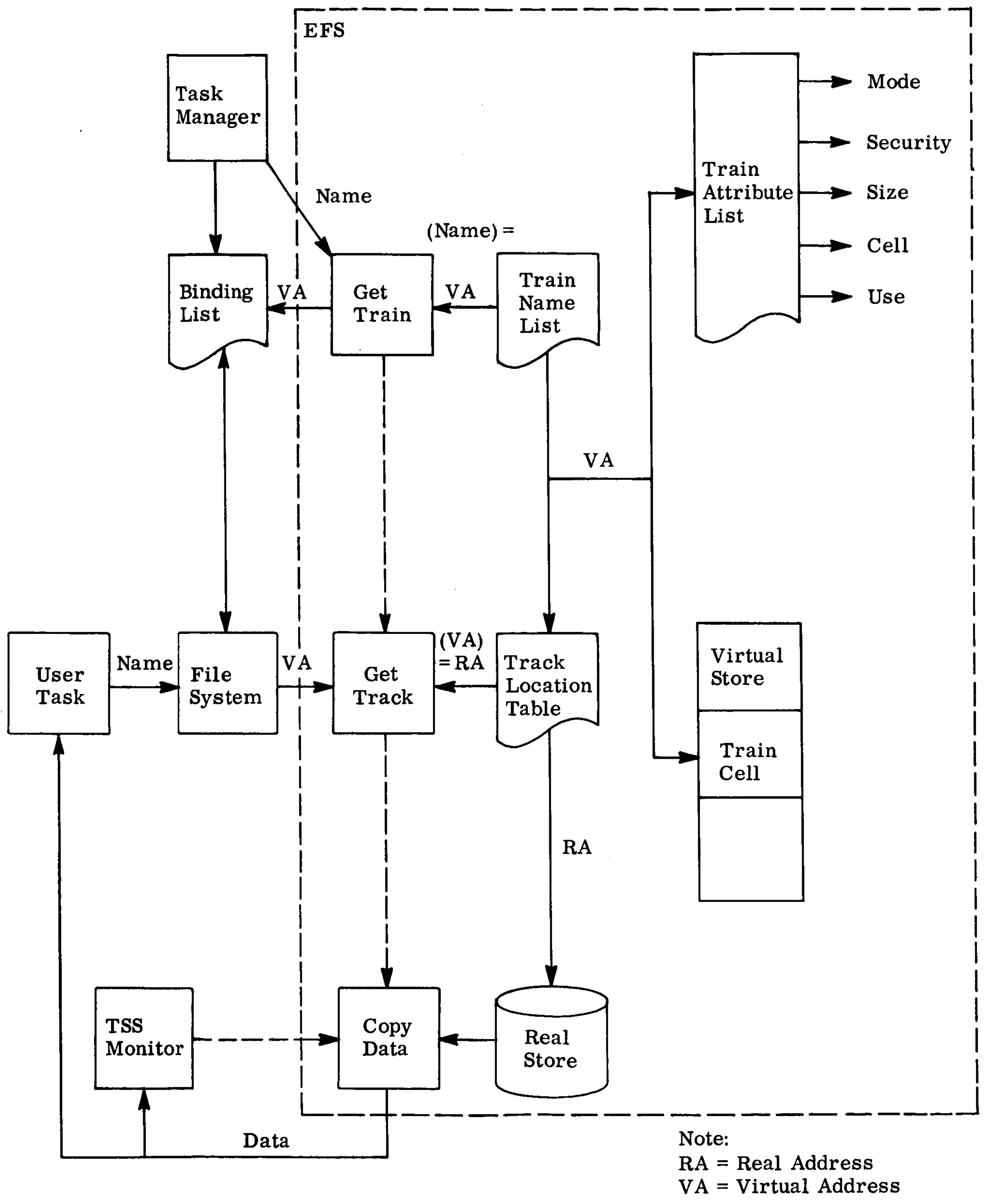


Figure 5-2. EFS Tables and Users

The second kind of virtual addressing supported by the VSM is used in referring to pages of addressable trains known to the EFS. In this case, a page p of train t is to be copied from one real location to another (e.g., secondary to primary memory). In order to do this, the head of the system's virtual address, t, is used to locate the real address of the train in the backing store, and page p is read from the backing store to a page in primary memory, using services of the Data Traffic Manager.

"Virtual Store Manager" denotes the set of functions in the Coherent System which support these two virtual addressing arrangements.

5.2.1.4 Program Execution with Virtual Addressing. Several computers support the execution of programs which use virtual addresses. The ultimate calculation, on a given instruction, of the real primary storage address from the virtual address requires special hardware features which operate through tables set up by system software. In the Coherent System, the design of the hardware operation must be such that it accommodates easily the concept of virtual address references to other trains besides that of the program, where the actual train to be referenced is not known at the time the program is assembled into machine instructions. The binding of the actual train to the formal train used by the program must actually be reflected in the process the hardware follows in carrying out the reference, keeping in mind such requirements as:

(1) The same page of an actual train may be referenced by more than one running task (assembled using different formal train names).

(2) The same program code may be operated as two or more tasks simultaneously (where one formal train name may thus be bound to different actual trains in the different tasks); that is, the same instruction may lead to different real addresses.

A method of translation of virtual address to real address, using tables constructed by the Task Manager after train binding, will now be described. This method is similar to that used in TSS/360 (see Appendix A).

Suppose the computer instruction's address specification consists of a byte address of N bits, an index register specification, and a base register specification. The content of the index register is an address consisting of M + N bits, where 2^M is the number of pages addressable in a train cell. A notational convention is: If m is the value in the M-bit field, and n is the value in the N-bit field, the value in the M+N bit field is expressed as $m \frown n$ (this concatenation is easier than writing $m \times 2^N + n$).

The FTN (Formal Train Name) is assigned a numeric value f by the assembler or compiler, and this is carried in the base register as $f \times 2^{M+N}$. Virtual address computation (ignoring indirect addressing) consists of summing the byte address, index register contents, and base register contents, as follows:

$$\begin{array}{lr}
\text{Byte Address} & b_1 \\
+\ \text{Index Register} & p_2 \frown b_2 \\
+\ \text{Base Register} & f \frown p_3 \frown b_3 \\
\hline
=\ \text{Virtual Address} & f \frown p \frown b
\end{array}$$

That is, $(p_2 + p_3) \times 2^N + (b_1 + b_2 + b_3)$ yield a new $p \frown b$. f is unchanged.

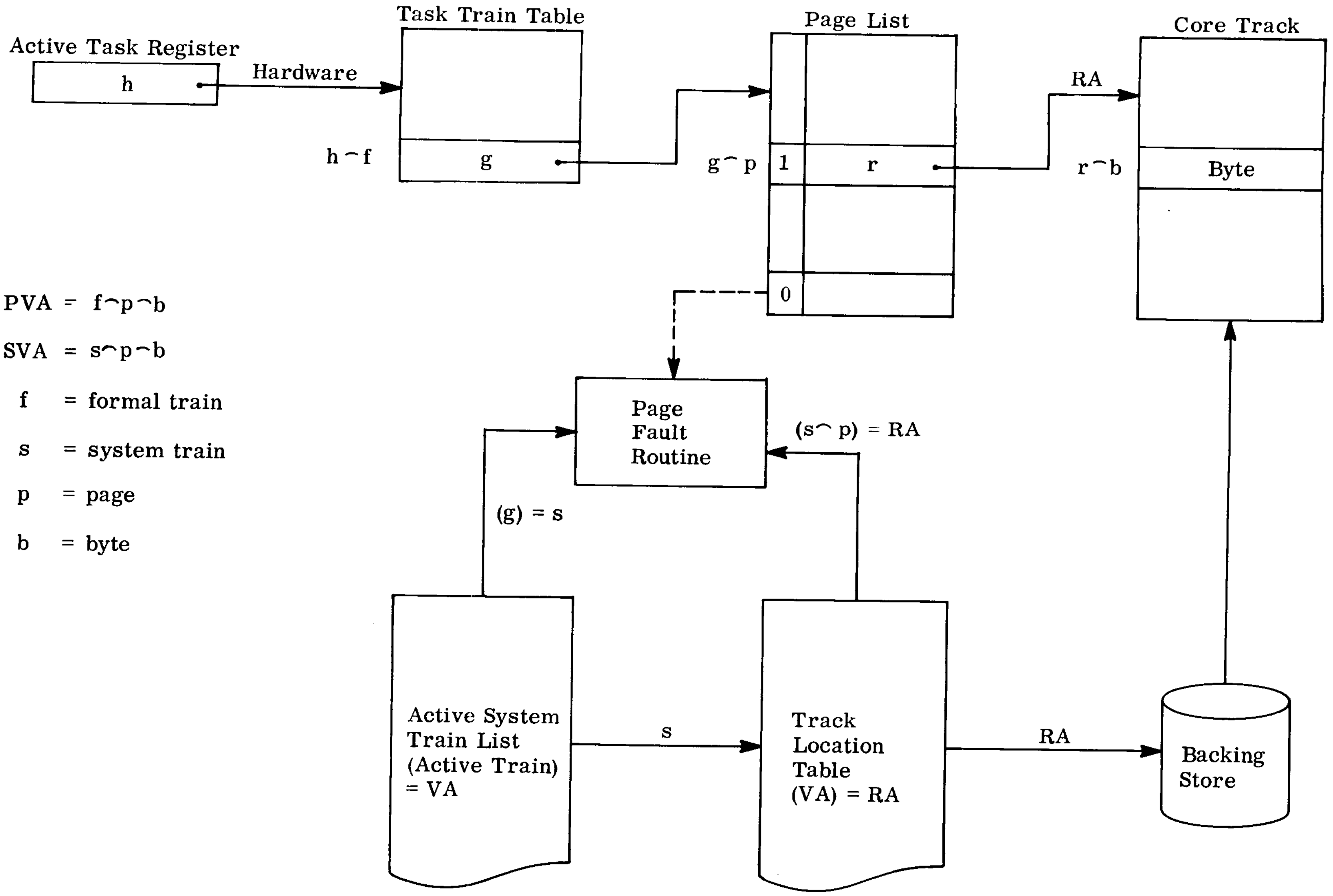
Besides what has been mentioned, one other register will play an important part in the address translation. In Figure 5-3 it is called the Task Train Table (TTT) Base Register; this register plays an important role in meeting design objectives of the Coherent System, as well as in address translating per se.

So far, the hardware has computed $f \frown p \frown b$. Now it must convert this to a real address. Refer to Figure 5-3.

For this task, a train table is established when loaded. This contains entries for all trains referenced by f. ($f = 0$ will be considered to refer to the train of the program itself. Thus, that value is never assigned for referenced FTN's.)

When executing (assigned a CPU), a hardware register for this CPU contains the high order part of the train table address for this task. Call this h. Then $h \frown f$ gives a reference within that table (performed by hardware). The entry in the train table locates a page table with a value to be called g.

The page table thus specified was built at load time when f and FTN were bound to an ATN (Actual Train Name). This page table represents the map of the ATN into primary memory. Now $g \frown p$ (still by hardware) locates the entry in the page table;



PVA = f⌒p⌒b

SVA = s⌒p⌒b

f = formal train

s = system train

p = page

b = byte

Figure 5-3. Addressing Within a Page in Memory

and either the high-order part of the real address (call it r) is combined with b, r—b, to give the final physical reference, or if the page is absent, a trap calls the Virtual Store Manager.

5.2.1.5 Scratch Trains. For a program which needs a large working storage for the duration of its execution, the Virtual Store Manager must be prepared to furnish scratch trains. From the programmer's point of view, this is instantaneously available as he is working with a virtual store. Yet the File System would like to distinguish such areas of the program's virtual store from other intraprogram references. Whereas a reference to another instruction (via a branch) might call for the retrieval of a page belonging to that program from a real train cell containing it, it should not be necessary to maintain in the system's virtual store the "blank" pages used as a scratch area by the program. In this light, a scratch area should take the form of a formal file; yet, the scratch area is never bound to an actual virtual store train representing permanent data. Therefore, to the programmer, the reference is to a scratch area; but to the EFS it must be handled and accounted for like a train (outside of the train which is the program).

These will be called scratch trains. They are known to the File System through the program definition in much the same way as are formal basis or result trains. The information needed for linking is built as part of the program definition just as for formal trains, by the language translators. But the symbol used by the program need never be known by the user constructing a job definition, since scratch trains are never bound to job arguments as are basis and result trains.

5.2.2 Terminal Trains and Explicit Services

Terminal train service is concerned with the transportation of copy cells between primary memory and terminal devices. Terminal trains are not addressable by program instructions making references to a virtual store, and hence, are not known to the Virtual Store Manager.

Undoubtedly, most user programs will confine their bases or results to formal items or trains which are known to the system, and thus take advantage of implicit IFS or EFS services. The data flowing between the system and its terminal devices would, for the most part, be handled by services of the Coherent System. Examples are:

- Inputs to the Job Management System, and Services
  (1) Program entries
  (2) Item definitions

(3) Job definitions

(4) Item entries

(5) Other commands and associated data

- Outputs by the File System, such as Archive Data
- Output by the JMS

(1) Interaction with user at terminal

(2) Messages to computer operators

- Outputs by Display Services.

For these system programs, and for user programs which must deal directly in transporting copy cells of terminal trains, a set of explicit services is provided.

5.2.2.1 The Program Interface. The amount of data prepared for a result terminal train, or the amount to be input and processed, is not known to the EFS. Programs concerned directly with such transportation functions must be in control of the real primary storage used in operation with terminal trains, so that by sequencing explicit requests for input or output, the same real storage area may be used repetitively to hold data being transported.

The program interfaces with the EFS on the basis of a train and the copy cell. A terminal train is symbolically named by the program and consists of a variable number of copy cells, which are numbered relative to the start of the train.

All copy cells should be composed of a string of a fixed number of cells (or, at the lowest level, bytes) at the next lower level. As a further restriction, it is proposed that this number (i.e., cell "level magnification factor") be a constant for all levels, and be an integral power of two. It would then follow that the subcells in each cell can be addressed with an address word of length appropriate to the cell level.

The programmer will define the formal terminal train to the language translator by supplying:

(1) A Formal Train Name

(2) The copy cell size (an integer k, indicating 2^k U)

(3) Number of copy cells needed for buffering (i.e., the service window)

(4) Branch for handling hard failures (optional).

Item (1) will be carried in the program definition for use by the Task Manager. The language translator will generate the following on behalf of the programmer:

(1) Buffer space within a scratch train

(2) The means for loading base registers for addressing the buffer area on the basis of the Formal Train Name

(3) A control block, carrying preceding items (1) through (4), and containing fields for error information; also the necessary controls for allowing the task to block itself while waiting for buffers to clear.

The program may call the EFS for reading, writing, or blocking. For read and write, the specified copy cell of the specified train will be connected to the terminal device with an I/O command issued by the Data Traffic Manager.

5.2.2.2 The User Interface. The formal terminal train is bound to a terminal device at job request time. The device is entered by the Job Management System into tables referenced by the Terminal Train Manager.

The Job Management System will activate, or "open" the train at this time, unless another task has seized the resources in question, in which case the scheduling of this job will be delayed. The train is deactivated either at the end of the task or at the end of the job, depending upon the train binding of the remaining tasks in the job.

5.2.2.3 Implicit Terminal Train Services. One might conjecture that something similar to the EFS Implicit Services (through the Virtual Storage Manager) can be provided for dealing with terminal trains. Although it is possible that something like this could be developed under certain conditions, study is needed as there are technical difficulties. For example, a program which produces display output as a terminal train is not limited in the amount produced. Nor is it possible for the EFS to predict (1) when the program has finished writing into a particular copy cell, or (2) the number of copy cells which will be filled when the task is complete.

One possible approach would be to provide implicit service to programs for which the following holds (for the example):

(1) Bytes of copy cells are stored into only once by the program.

(2) The mode is sequential.

Thus, a scratch train can be provided. As the program crosses page boundaries, previously referenced pages may safely be output to the terminal device bound to the formal terminal train. These limitations are not severe for the vast majority of programs dealing with terminal trains, and so this possibility may be pursued in the future.

5.3 TRAIN PROTECTION AND ARCHIVES

Trains in the virtual store are protected by the EFS from unauthorized access, accidental loss, and erroneous modification. Unauthorized access is prevented by checking the user's classification and group against entries in a cell access rights table. Entries in this table are made at train definition time on the basis of the security attributes in the command, leading to the definition of a train (e.g., Item Entry or Program Entry).

Protection from accidental loss and erroneous modification is offered by avoiding the overwriting of existing cells in a permanent train. To accomplish this, an edition number is assigned to each cell as it is written. The edition number is maintained in the track map, modulo 2^n (where n is some number, e.g., two), and is incremented each time the cell is written.

Each time a page of the virtual store is modified by a task, it is considered as a new edition. If a program's formal train is a result train, and was bound to a virtual store train through the job definition, the successful termination of the task calls for certain virtual storage management functions to be triggered by the Task Manager. Any pages of the result train which were made available to the task during its execution must now be replaced into the train cell as new editions. Also, these are written to the archives. Only when both of these functions are accomplished will the train cell be unlocked for use by other tasks as a basis train.

5.4 MULTILEVEL STORAGE MANAGEMENT

The technology of digital storage of data is undergoing rapid evolution. However, as advances in the technology improve the physical density, capacity, and access speed of data, there tends to persist a hierarchy of access times within the data base. There is a hierarchy certainly across storage hardware types. Registers, thin film, magnetic core, delay line, drum, disc, and magnetic card represent an ascending sequence in density, capacity, and access speed. Each of these device types can be thought of as being at a lower level of the storage hierarchy than that of the preceding type.

In addition, within a given single memory device there is usually a "sequential component" which makes adjacent cells more easily accessible than cells at random. For example, the data on a given "cylinder" of a disc unit can be considered as having the same relative level of accessibility, while its absolute accessibility depends dynamically on the current position of the access arms and the current rotational position of the discs. Similar considerations apply also to most other storage devices.

Intelligent allocation of trains to storage devices at these various levels is needed in order to maximize system responsiveness. A train might be moved up the hierarchy for any one of a number of reasons:

(1) In general it is addressed frequently.

(2) The user has given it special status, perhaps because its use is connected with emergency measures.

(3) A job which has been requested is exceptionally active in addressing its formal train.

The criteria to be employed, and the method by which the EFS is commanded to employ them, would be developed in the context of a particular implementation.

Disregarding the form of the algorithms to be employed, the Coherent System would endeavor to reach a proper compromise between latency time and flexibility, as suggested in Appendix A, Figure 1 of the INTIPS system evaluation.

Also, the movement of a train from one real address to another will not require heavy restructuring of EFS tables, as the system's virtual address would remain constant. The tabular function TRACK LOCN(VA) = RA must be altered. Below this level, trains are known by their virtual address.

SECTION VI. INTERNAL FILE SYSTEM

6.1 THE ITEM

The data entities of the IFS are called items. All items are part of the most generic item of the IFS, called the data pool. Items have names and definitions which are stored in a directory which is itself a part of the data pool. Although both the definition of the structure of the item and the data values representing an instance of the item are in the data pool, it is possible to discriminate between them by using the modified forms, item structure and item instance. When unmodified, the word item will normally refer to both the structure and its instances. Instances of items are stored in the common data base, and structural definitions are stored in the directory. Both are part of the data pool.

An inclusion relation is defined among items, and a given item may be either simple (a "terminal" item) or composite (compound). Simple items are called fields (or scalars), and their instances are stored as values in the data base. (A field and its value correspond roughly to a scalar variable and its value in a programming system.)

Composite items are either statements or arrays. A statement is a composite item which subsumes some definite number of distinct subitems. A subitem of a statement may be a field, statement, or array. An array is a composite item whose members (array elements) are always instances of a single statement, called a record. The instances of a record are repeatable a fixed or variable number of times, and are identifiable by a set of indices of known dimension — such as element i of a one-dimensional array; element i, j of a two-dimensional array, etc. Any given array dimension may be either fixed or variable.

A vector would be a one-dimensional array of fixed size whose records contain a single scalar. A logical file in data processing terminology is a single-dimensional array of arbitrary length. An array in general may have one or more dimensions, and each dimension may have either a fixed or variable size. The record statement may consist of fields, other statements, and other arrays. In some data processing systems, the term "repeated group" or "repeated set" is sometimes used to refer to arrays that occur within records. The number of nested levels of items within items is logically unlimited and may reflect the hierarchic structure of objects in the outside world (for example, assemblies and subassemblies of components in equipment and systems).

Fields (sometimes called scalars or properties) are the terminal items in the data base and take specific values for each occurrence; e.g., the property propulsion system in an array space vehicles may take the values chemical, nuclear, ionic, etc., depending on the specific record involved.

Fields are simple (terminal) items in that they contain no subitems. The definition of a field is complete when its type (i.e., character set), size, and name are given. The definition of an array is complete when the size of each of its dimensions, its name, and the definition of the array element (record) are given.

These concepts are sufficient to represent data structures which can be shown topologically as a tree.

In a tree, each item can be represented as a node which belongs to at most one parent item (node). Tree structures can be mapped into cells and trains so that the fields encountered in a regular tree-coursing path which exhausts all branches systematically, left-to-right, produces fields which are contiguous in the train cell address space. For example, it is possible to have a sequential organization in which adjacent fields of each record are addressable contiguously in the train cell, and similarly for each record. If the record contains other embedded items, these too can be mapped contiguously in the address space.

Data hierarchies which are more general than trees (i.e., contain items which are part of more than one parent structure) require some mechanism other than adjacency in the cell, such as links or pointers in the data, to carry the processor from one item to a related item. A link is a reference from one item to another item which is physically remote. More generally, a link is a device for representing a logical relationship between one item (the source item) and another (the target item) which is independent of the need for physical adjacency of source and target items.

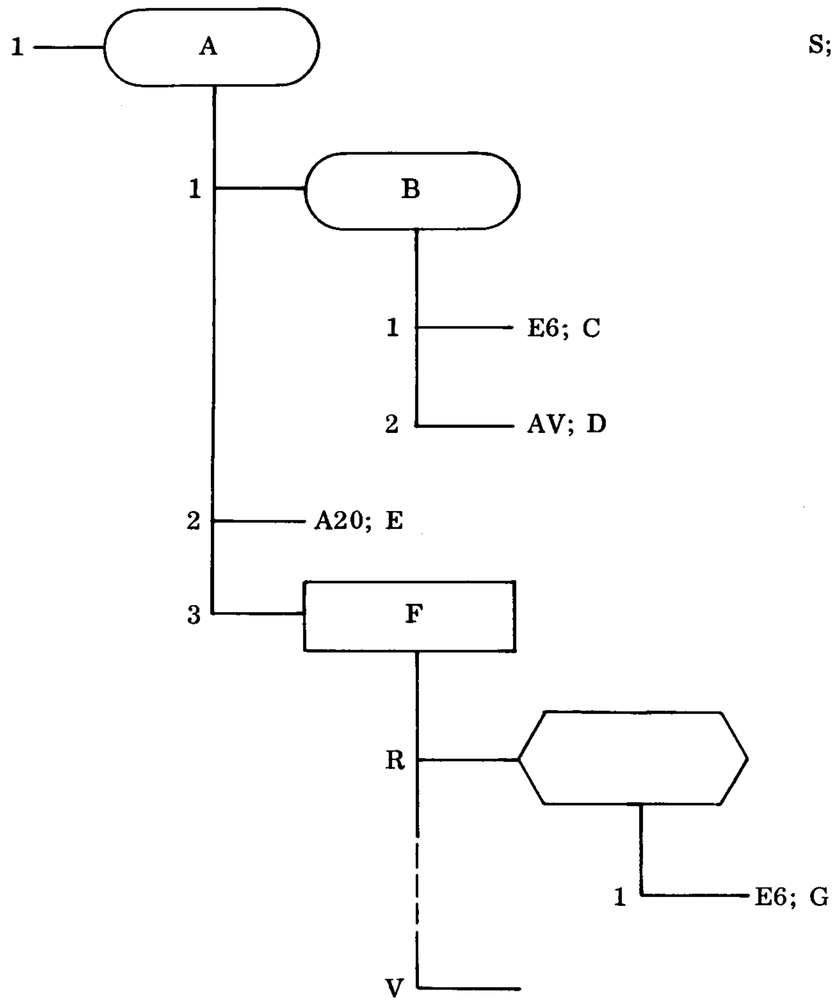
The ability to employ linkage in a data management system is important because the use of linkage as a pointing mechanism permits several different logical relationships between a source item and the target item to which it points. It transforms a tree into a more general network.

6.2 ITEM STRUCTURE REPRESENTATION

Data structures can be defined and represented in several ways. Figure 6-1 shows four methods for representing item structures. A structure of four levels is defined in this example, which corresponds topologically to a rooted tree.

Representation (a), called an indented tree, has the most intuitive appeal and would be used in the initial design of the structure. In the rooted tree, each increase in depth level is indicated by indenting to the right. Items connected to the same vertical line are at the same level. Structures are indicated by boxes of three different shapes: rectangular for arrays, hexagonal for records, and oblate for statements. The name of each structure appears in the box, and the name of simple items (scalars, fields, or leaves of the tree) appears after a slash joined to the vertical line representing its parent structure. A letter and number appearing in front of a field name indicate the character type and size of fields (V signifies variable length). Table 6-1 defines the input character set and conversion for each character type.

(a) Indented Tree



(b) Indented Outline

S; A
  S; B
    E6; C
    AV; D
  A20; E
  FV; F
    R
      E6; G

(c) Item List/Term List

S3	A
S2	B
E6	C
AV	D
A20	E
FV	F
R1	
E6	G

(d) Item Image

(A(B/E6;C/AV;D)/A20;E[V;F(/E6;G)])

Figure 6-1. Item Structure Representations

TABLE 6-1. FIELD CONVERSIONS

Field Type	Character Set on Input	Conversion
Binary	0, 1	binary unsigned
Octal	0, ..., 7	binary unsigned
Integer	+, -, 0, ..., 9	binary signed
Decimal	+, -, 0, ..., 9	BCD signed
Exponential	+, -, ., 0, ..., 9, E	floating point
Alphanumeric	any character	none
Text	any character (Note 1)	none
Coded	any character	binary (Note 2)
Hierarchy	any character	tree code (Note 3)

Note 1: For a text field, a hierarchy of delimiters can be defined such as space, line terminator, statement terminator, and page terminator.

Note 2: The value set for coded fields is given in the definition. The value is coded on input and decoded on output.

Note 3: The value structure for hierarchy fields is given in the definition. The value is coded on input and decoded on output.

The items of each structure are numbered in sequence, 1, 2, 3, etc., except that the record level under a file takes the variable "R" for each dimension, indicating that the structure is repeatable the number of times indicated at the end of the vertical line. Each item in the structure definition can be given a unique number by stringing together the item sequence numbers in the unique path from the root to the item. Thus, the unique number for Item G is 1.3.R.1.

The logic of this structure definition is not stratified as to level of storage, so it can be used to define internal system tables and program parameters, as well as large structures for secondary storage. There is no logical need to distinguish a file and records in secondary storage from a table and entries in core, so the same program logic is applied to both.

Representation (b), called the Indented Outline, follows closely the indented tree form, except that the box shapes are indicated by a character. This form can be keyed from the tree as input to the Item Definition Job. Representation (c), called the Item List/Term List combination, is similar to the internal representation of the structure in the system directory. In this form indentation is not used; instead, the size of statements and records is given (in terms of the number of subitems).

Representation (d), the Item Image, is a linear parenthesized string which can be used as input to define a structure. The scope of each statement is indicated by parentheses. Except for arrays (files), the name of each compound item follows the left parenthesis. In the case of arrays the scope is indicated by brackets and the size of each dimension precedes the name.

Because the logical structure of all data items is defined in this structure definition, which is stored internally as a system table (described, incidentally, with the same convention), considerable leeway can be tolerated in the format of input data. Data may be keyed in strict field sequence in agreement with the structure definition, with empty fields indicated by a slash, or without regard to field sequence if each out-of-sequence field is tagged with its name or identifying number. No blanks or leading zeros need be keyed. The system can automatically justify data and supply blanks or leading zeros. The system also rearranges the data in proper sequence, filling missing fields with blanks where necessary.

When record numbers are supplied to replace the R's, the structure definition number of an item becomes a number for a unique data item instance in the data base. This property is exploited to provide an efficient logical index to data items for efficient retrieval from random-access storage devices. The identifying number of an item in the structure definition is called an Item Class Code (ICC), and the number of an item in the data base (ICC with R-values supplied) is called an Item Position Code (IPC). The ICC and IPC of items are generated and used internally by the system, and the general user need only use the names of the items he is defining, requesting, or processing.

The structure definitions are collected into one comprehensive directory for all items in the data pool. An alphabetical listing by item name directs the system (by ICC and IPC) to all occurrences of the item as structure definition and data. Thus, information (both structure definitions and data) is available to the system's users without prior knowledge of the existence of the data or its detailed structure. The user need know only the generic names of the data to be investigated. There is no need to approach the data on a file-by-file basis, nor is the user limited in any way by the original structure definitions. Rather, an inquirer can be presented with a structure definition in a dialog mode query so that he may take advantage of the structure for more efficient use of the system and its data. This lack of constraint at both ends of the process — data definition and data retrieval — provides a new measure of flexibility in data management processes. Incidentally, it is this philosophy of centralization of structure definition in an overall system directory that permits all system data to be regarded as a "common pool."

This unusually non-constraining approach to data definition and retrieval is accessible to the system's administrators, programmers, data specialists, and operational users through a user-oriented job language and job-managing executive. Each of the preceding users must be given the appropriate language tools for effective execution of tasks in his domain.

6.3 ITEM LINKAGE

Item linkage falls into four general categories which will be discussed in the following paragraphs:

(1) Sequencing

(2) Associative

(3) Hierarchy

(4) Adjoining.

Each type of linkage will be discussed in terms of the item types involved, the relation represented by the link, and a diagrammatic representation of the linkage and its interpretation.

(1) Sequencing Linkage

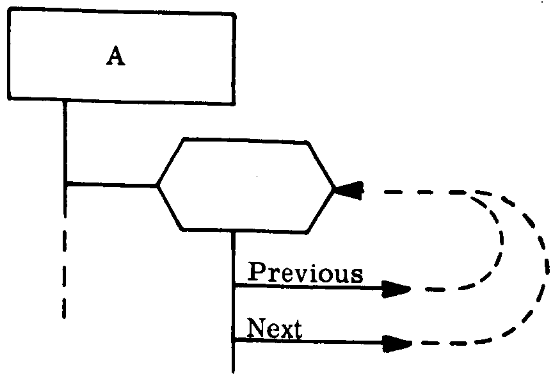
In Sequencing Linkage the link imposes an ordering relation between the source and the target items, which must be records of the same file (or, more generally, elements of the same array). Two Sequencing subtypes may be identified, since the link may be intended as a pointer to the previous record or to the next record of the chain. Both the previous and the next type of pointer can be used together if it is desirable to traverse the chain in either direction from any point. By means of Sequencing Linkage it is possible to order a file by more than one criterion, employing a linkage chain for each ordering desired. Thus, a given file may be sorted physically by employee number and sequenced logically by name (alphabetic ordering) and by salary (numeric ordering). In this case one ordering is given by physical sequence and two by Sequencing Linkage chains. An example of a diagrammatic representation of a Sequencing Linkage chain is given in Figure 6.2. Figure 6-2 (a) shows an item class definition in which the record class is represented by one structure, and Figure 6-2 (b) shows two specific records in the chain.

(2) Associative Linkage

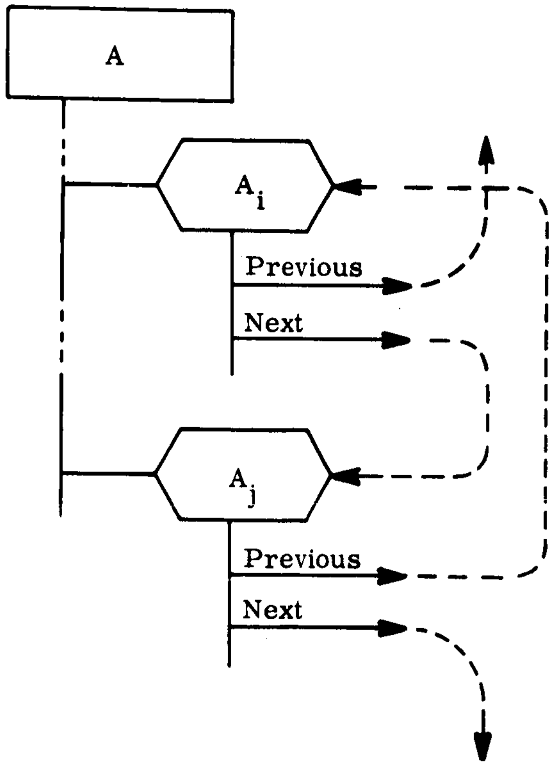
In Associative Linkage the link is used to establish a chain of items which share a particular value for a given property (field). This linkage can exist between items of any class as long as the specified property and value hold. Since the same value for the field exists for every item in the chain, there is no implication of logical ordering in the link. Rather it is a directive to see the target pointed to as an item associated with the source since it shares a given property and value. As in the Sequencing Linkage, an independent chain may be established for each property of the item. For example, in a file of automobiles, there may be a chain for all red cars, all Fords, all V8's, etc. A diagrammatic representation is given in Figure 6-3.

(3) Hierarchy Linkage

In Hierarchy Linkage the source and target must be items of different classes (e.g., records of different files). As in the case of Sequencing Linkage, two reciprocal subclasses may be defined since the linkage may represent the relation includes or the relation part-of. In either case, a hierarchy of items on different levels is established. In the case of includes, the source item is a parent of (i.e., master) or subsumes the target item. In the case of part-of, the source item is a child of (i.e., slave) or is subsumed by the target item. A diagrammatic representation is given in Figure 6-4. Note that this achieves a logical embedding of the target item in the source item without implications of physical adjacency; hence, if the target of an includes link were already part of another structure (either by physical embedding or independent linkage), it represents a non-tree (lattice point) connection in the structure. There is no limit to the number of target items in a Hierarchy Linkage from a single item.

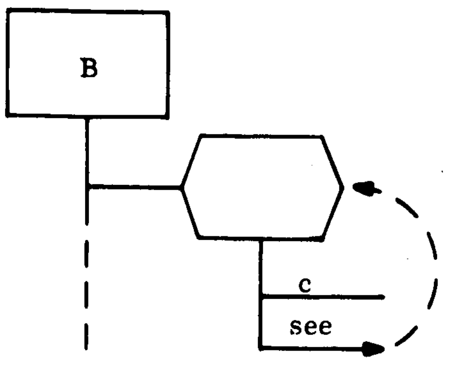


(a) Item Class Diagram

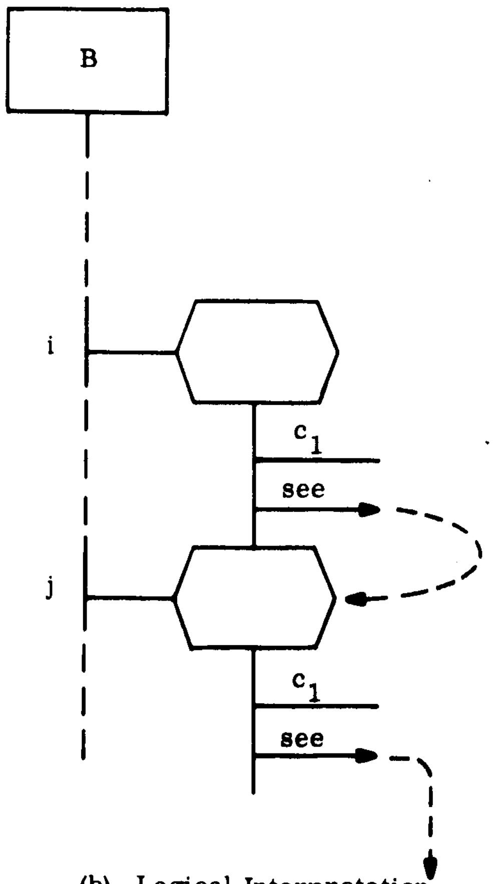


(b) Logical Interpretation

Figure 6-2. Sequencing Linkage

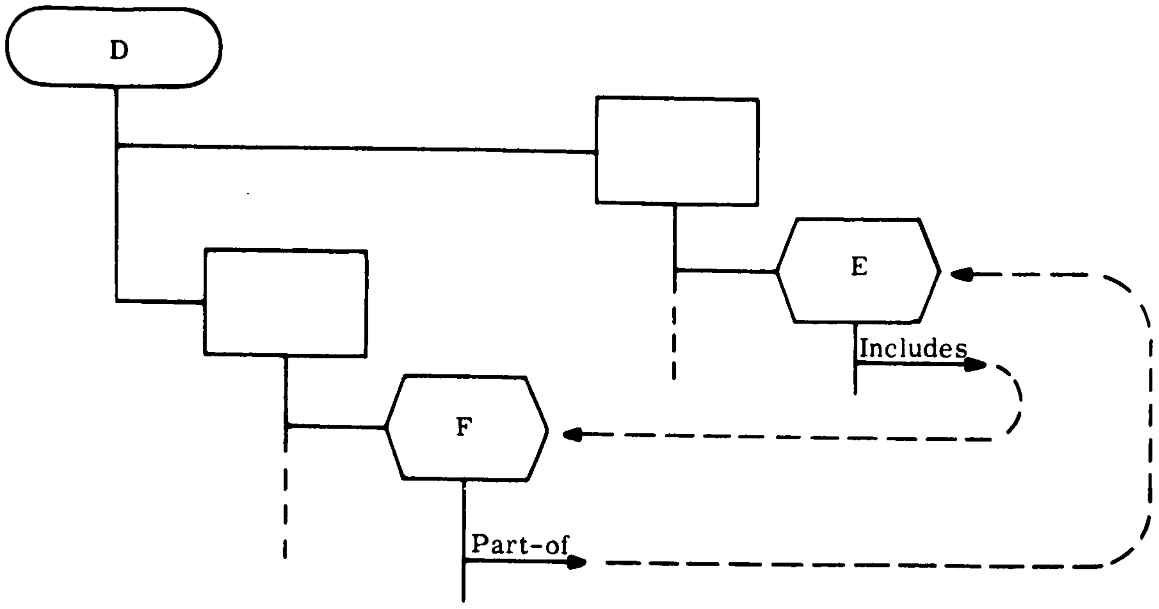


(a) Item Class Diagram

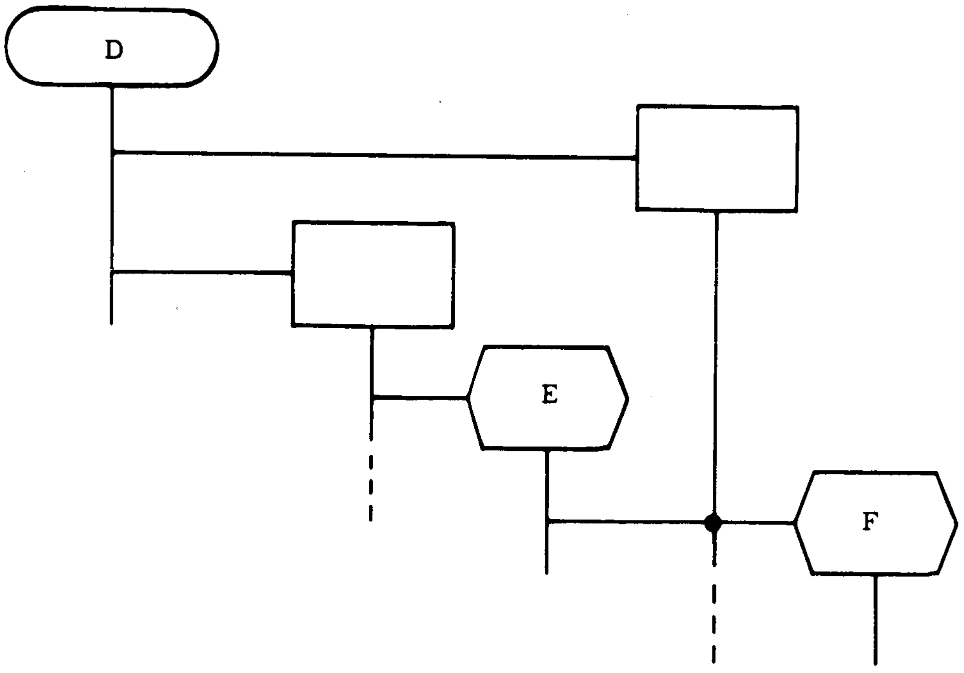


(b) Logical Interpretation

Figure 6-3. Associative Linkage



(a) Item Class Diagram



(b) Logical Interpretation

Figure 6-4. Hierarchy Linkage

(4) Adjoining Linkage

In Adjoining Linkage, the source and target must be items of different classes (e.g., records of different files). The link represents the relation of adjoined-with, which establishes the target item as an extension or continuation of the source item. Both items are considered to be on the same level. For example, the fixed information and the variable information in one logical item may be split into two files (for processing efficiency) with an adjoining link used to establish the variable record as a continuation of the fixed record. A diagrammatic representation is given in Figure 6-5.

6.4 METHODS OF LINKAGE IMPLEMENTATION

Data linkage can be implemented in several ways, depending on the type and directness of the connection between source and target item. In general, there is a tradeoff between speed of access and flexibility and speed of modification as the degree of directness in the link is changed. The implementation methods are discussed in the following paragraphs.

(1) Direct Addressing from Source Item

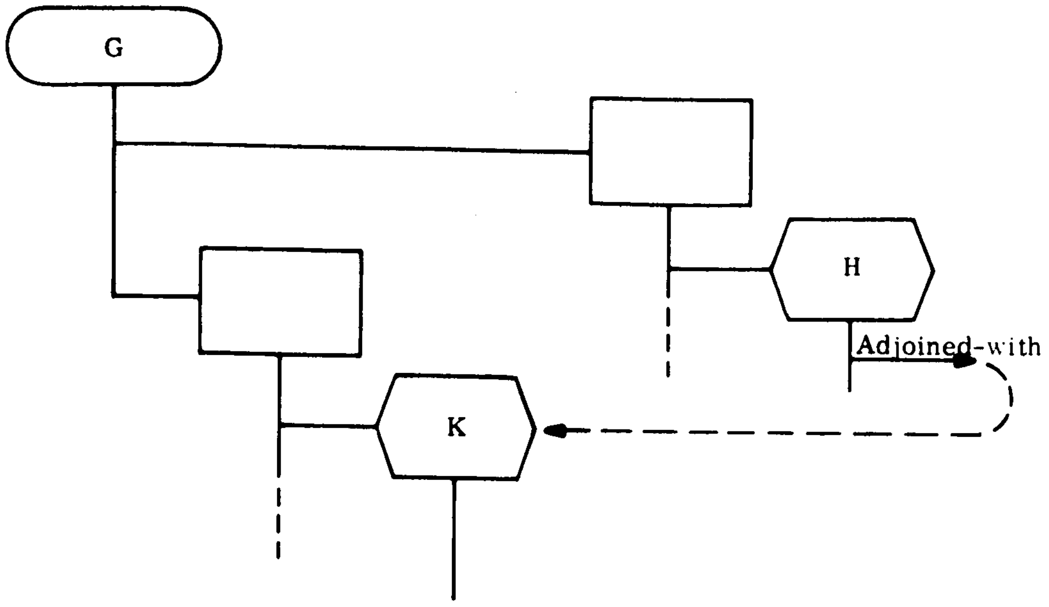
The most direct data linkage is achieved by a link field in the source item, which gives the direct physical address of the target item. This provides for direct accessibility of the target item but represents a maintenance problem if the target item is moved physically.

(2) Logical Item Position Code in Source Item

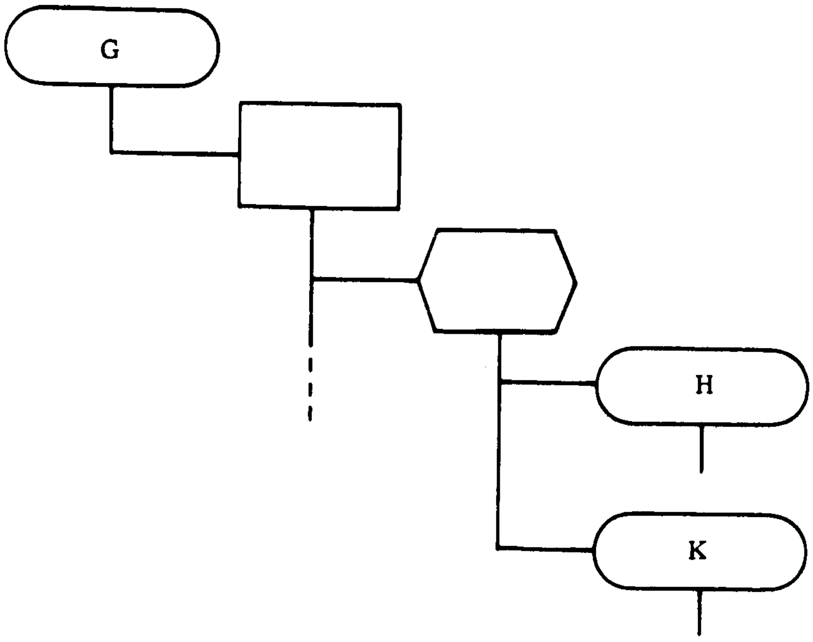
Independence from physical data movement can be achieved if a logical code is used in place of a direct physical address. The Item Position Code can be used to give a unique designation to the logical position of each item in the data base, and has the desirable properties of designating the entire logical scope of the target item and being independent of physical location.

(3) Criterial Items

The two methods discussed in Items (1) and (2) require a specific link field in the source item which supplies structural information but which is superfluous from a data content point of view. To establish linkage without this cost, it is often possible to utilize fields which carry data information values and would be present in the record whether or not linkage was desired. This is done by establishing a set of one or more fields which, when taken together, can be considered as a criterion for linkage to another record (or records, or other items). This field (if a single item) or statement (if a



(a) Item Class Diagram



(b) Logical Interpretation

Figure 6-5. Adjoining Linkage

collection of fields is required) is called the linkage criterial item of the item. By identifying such a criterial item in the item class definition of both the source and the target items, it is possible to establish a linkage between one or more records in the source file and one or more records in the target file. For example, in a mission administration data base (see Figure 6-12), mission may exist as a field in both the Flight Plan file and the Personnel files. By establishing mission as a linkage criterial source item in the Flight Plan file and a linkage criterial target item in the Personnel file, a set of target records in the Personnel file is designated by each record in the Flight Plan file. If mission is an indexed field, then the target records can be quickly established.

(4) Shadow Linkage

When the records of two ordered files of the same size have a one-to-one relationship, an implicit linkage can be established by considering one file as the Source file and the other as the Target file. In this way linkage is established between the i-th record of the Source file and the i-th record of the Target file, without any requirements for specific fields with linkage functions in either file. This technique is called Shadow Linkage and can be used to establish only Hierarchy or Adjoining Linkage types, and then only for files which are ordered and have a corresponding record in each file. (A real distinction between Shadow Linkage and Criterial Item Linkage exists only where the sequencing key does not appear as a field in both files.)

(5) Linkage Tables

Linkage can be established independently of data content or ordering by maintaining tables of linkage information in a directory which is separate from the files of data values. Either direct address or Item Position Code can be used with this technique. The link can be designed as a field in the source item, and recourse to a linkage table for the file can establish a target item (or items), if any, for each source item.

6.5 OTHER ITEM STRUCTURES

The data items previously defined (structures, arrays, links and fields) can be augmented with Code and Hierarchic Valued Fields and Variable Structures as discussed in the following paragraphs.

6.5.1 Coded Valued Fields

It should be possible for the user to define a field type called coded (C) which, in addition to a name, has a range of values it can take as specified in the item definition. For example:

C2; Sex {Male, Female}

C6; Color {Red, Orange, Yellow, Green, Blue, Violet}

The system would assign a binary integer to each value, code the data to binary on input, and decode the data for display.

6.5.2 Hierarchic Valued Fields

The concept of Hierarchic Valued Fields is related to Coded Valued Fields. Instead of an ordered set of values, however, the range of values is partially ordered in the item definition, for example:

H4, 3; Location { Mass (Boston, Springfield, Lexington),
New York (New York (Manhattan, Brooklyn, Queens),
Albany, Buffalo), New Jersey (Newark, Camden),
Pennsylvania (Philadelphia, Harrisburg) }

The "H4, 3" in the item definition specifies that the field is hierarchic with at most two levels and three values in any one "family." Each value would be encoded using a tree code which names each path in the structured value; e.g., "Brooklyn" would be encoded as "2. 1. 2." The advantage of hierarchic coding (in addition to saving space) is the ability to recognize one value as including another. Thus, a request for records satisfying the condition

Location = Mass (coded as 1)

is automatically satisfied by records containing

Location = Springfield (coded as 1. 2)

by immediate examination of the coded form of the value.

6.5.3 Variable Structures

A data structuring situation occasionally arises in which the interpretation, or range of values, of one field is dependent on the value of another field. For example, in a personnel file, the value of the fields, sex and marital status, determines whether the field, maiden name, is relevant. In order to communicate this logical dependence between items more clearly than simply permitting some items to be optional, a switch item can be defined. A switch item is a structure containing a known number of subitems. The value of the first subitem determines which of the subsequent subitems is to be employed in each occurrence of the item in the data base. For example, assume a personnel record has fields for sex, marital status, spouse's name, and maiden name. Clearly the last two fields are irrelevant for single persons, and maiden name is relevant only for married females.

This dependence may be shown in tabular form (see Table 6-2). It may also be shown as an indented tree data structure (see Figure 6-6). In this example, the fields marital status and sex, are the switch control items; these must be coded field value items whose values are given. There are 2 x 2 or 4 possible values for the switch control items so there must be four switch value items in the remainder of the structure. For any one record in the data base, one of the four items will be chosen by the values of marital status and sex. Marital status will determine the most significant bits of the path number, and sex will determine the least significant bits.

TABLE 6-2. DATA DEPENDENT STRUCTURE

		Sex	
		M	F
Marital Status	S	Λ	Λ
	M	Spouse's Name	Spouse's Name Maiden Name

Another example is shown in Figure 6-7. This example shows the structure of the Private Memory Table in the Berkeley Time-Sharing System.* Each record is a one-word item which shows the status of one of the 64 possible logical pages of a

* See Appendix A.

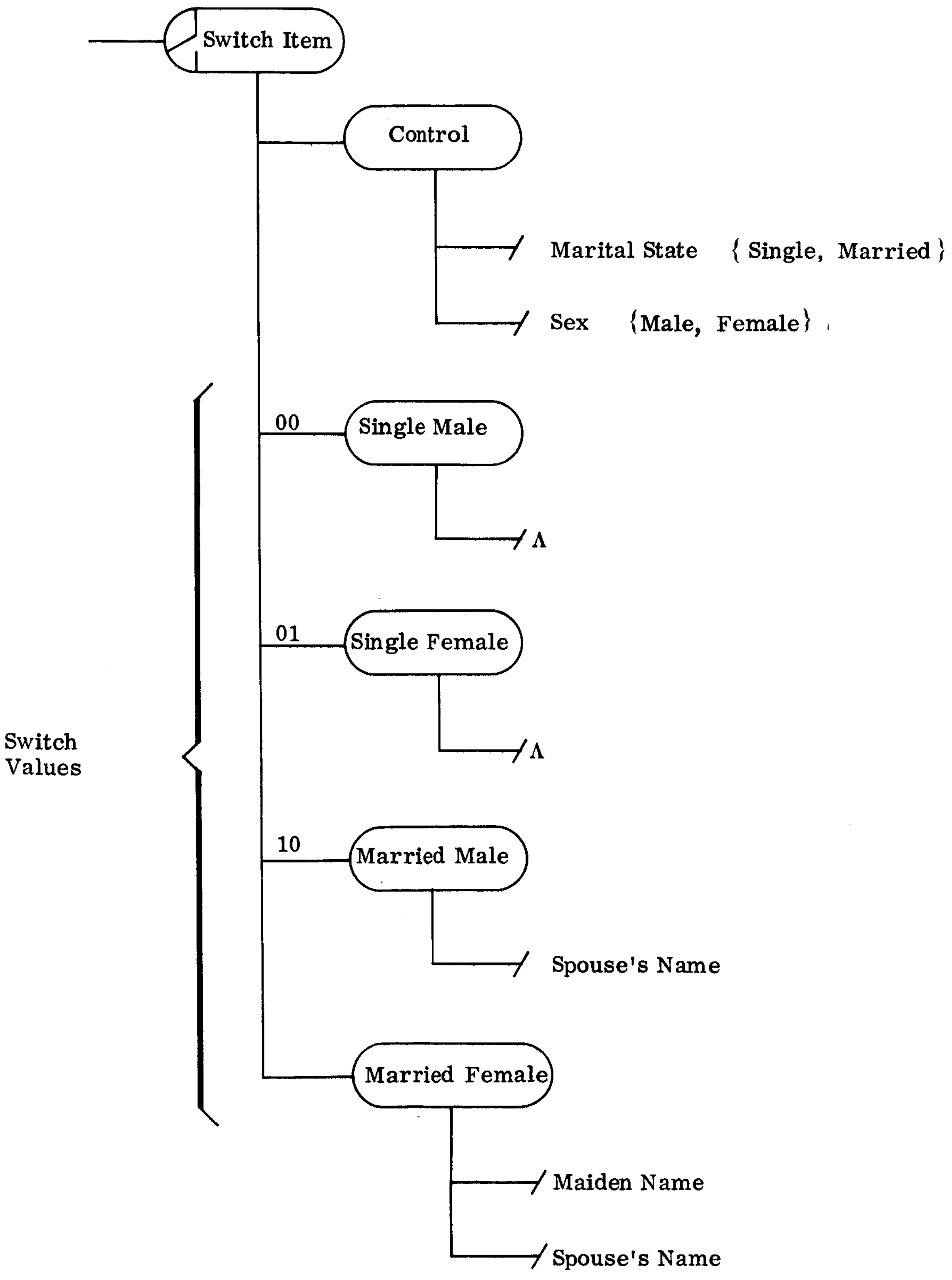


Figure 6-6. Switch Item Data Structure

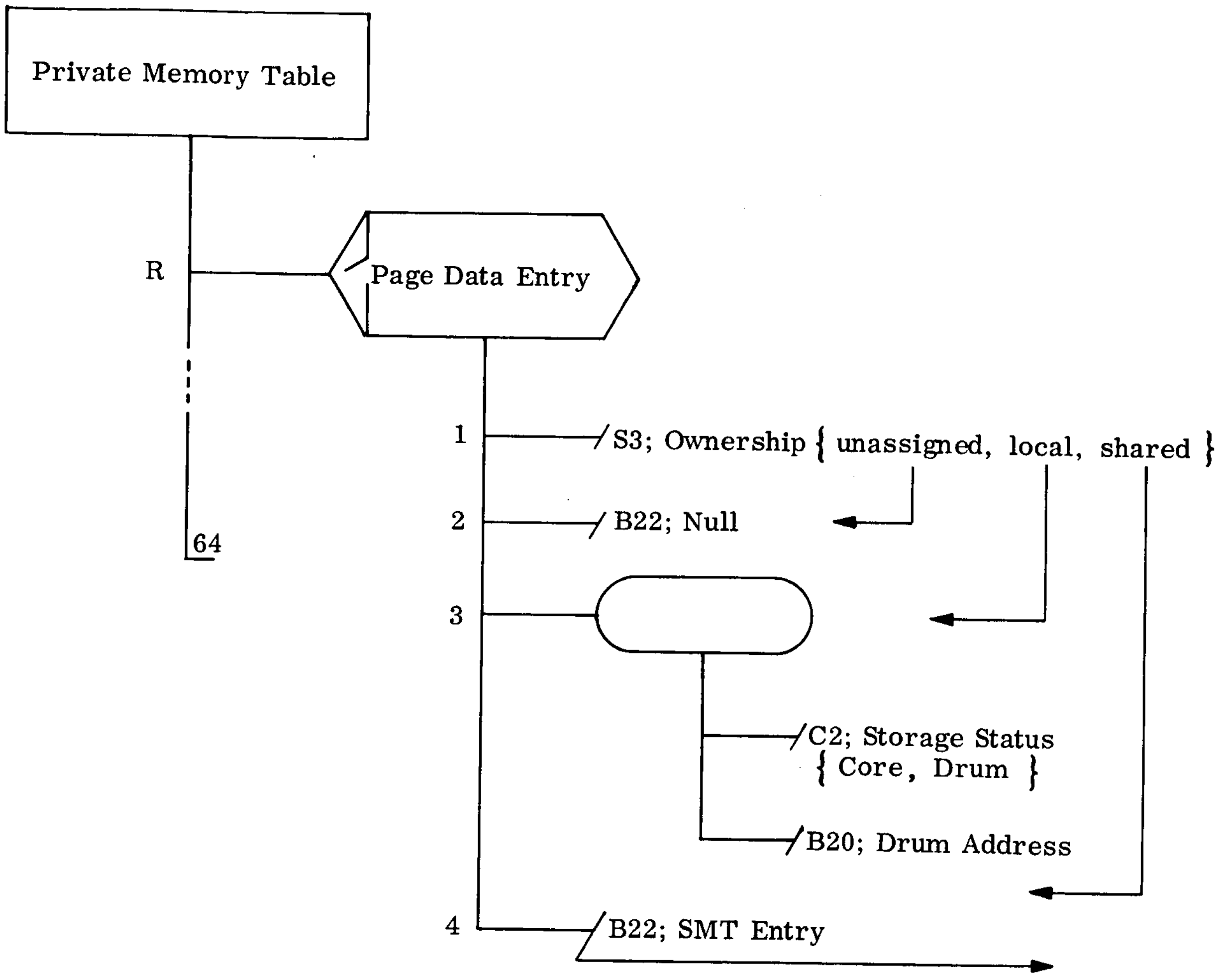


Figure 6-7. Switch Item Record in Private Memory Table

process. A two-bit field specifies page ownership. The interpretation of the remainder of the word depends on the page ownership field. If the page is unassigned, it is unused; if owned locally, it shows storage status and drum address; and if the page is shared, it points to an entry in the Shared Memory Table. All this information is shown concisely, and is graphically clear in the indented tree diagram.

6.5.4 Synonyms

When a data base is shared by several users, it may be desirable to permit each user to have a name structure defined over overlapping subsets of the data base. This permits each user to use names meaningful to him to preselect items of interest without requiring redundant storage of data in order to meet his requirements.

6.6 THE DATA POOL

All data items (that is, data accessible through IFS data services) are stored under a single generic statement item called the data pool. The data pool, in turn, is composed of four statements called IFS data, EFS data, JMS data, and common data, as shown in Figure 6-8. The IFS data consists of five tables used by the File System for recognition, definition, and retrieval of all items in the data pool. These tables, shown in Figure 6-9, are the Name Encoding Table, Item List, Name List, Train List, and Index. The EFS data consists of three tables as shown in Figure 6-10. The functions of these tables were discussed in connection with the EFS.

The File System provides several levels of service, depending upon the level of service requested and the stage of parameter binding at the time the service is provided. The sequence of levels, and how the file system tables are used, is shown in Figure 6-11.

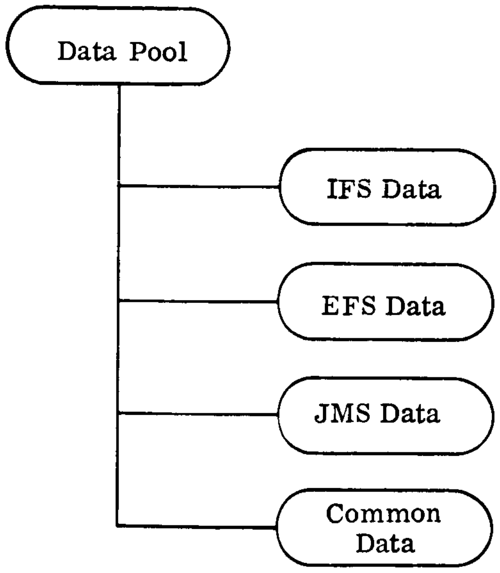


Figure 6-8. Coherent System Data Pool

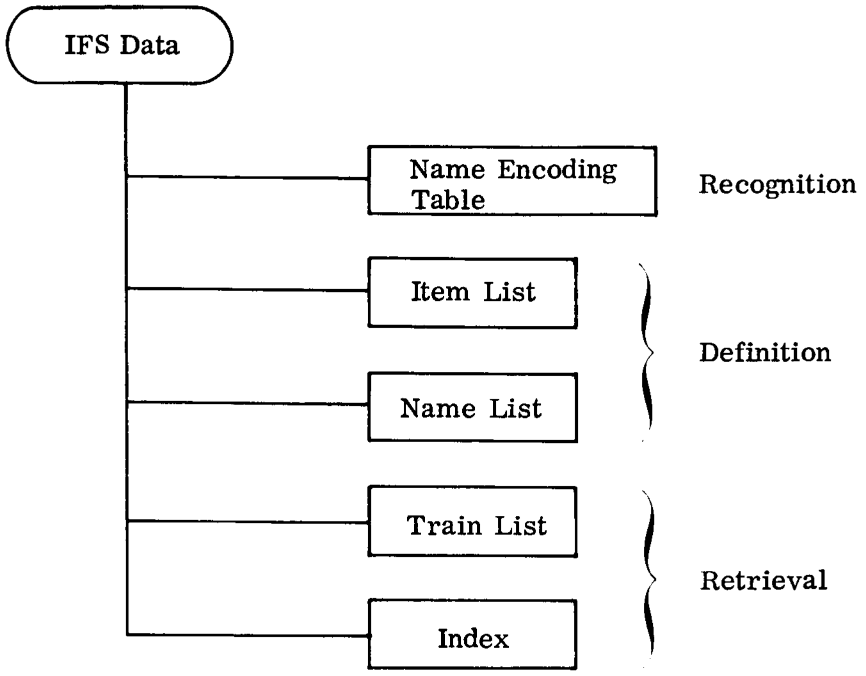


Figure 6-9. IFS Data

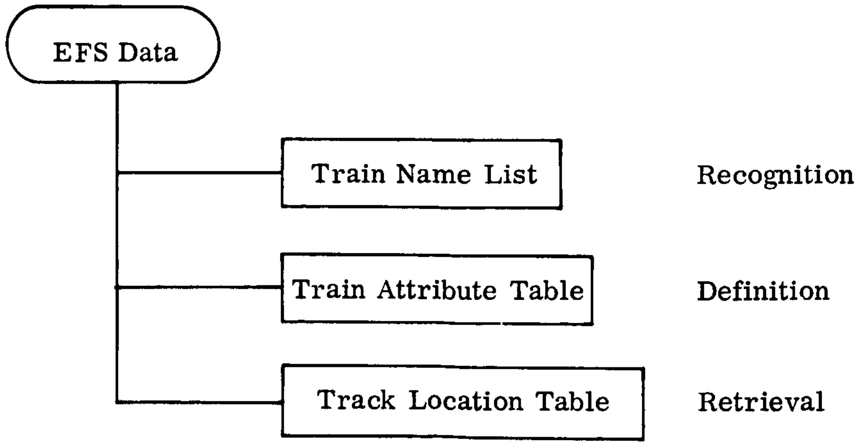


Figure 6-10. EFS Data

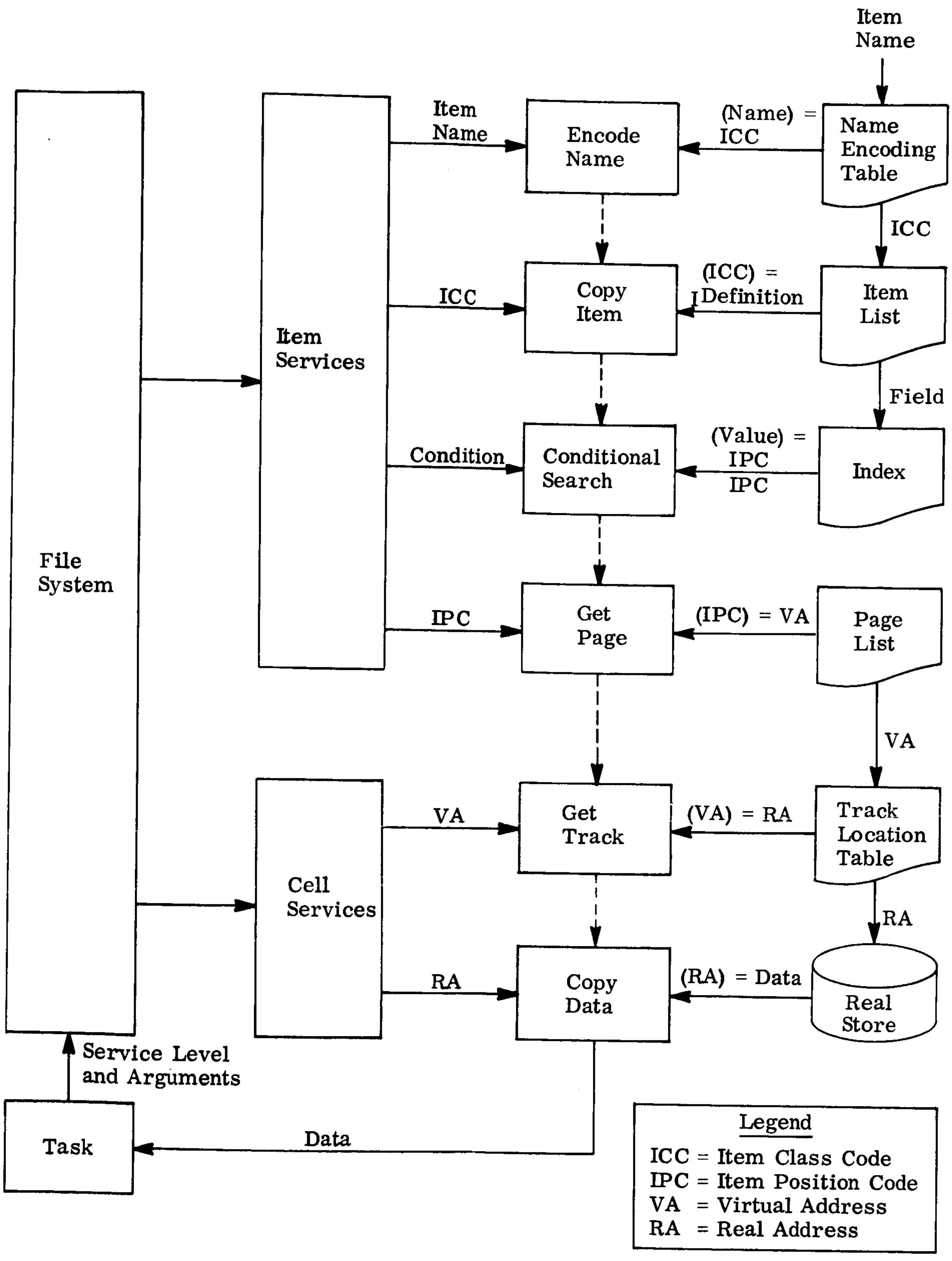


Figure 6-11. File System Service Levels

The JMS data contains all the tables required by the JMS as discussed in Paragraph 4.2 and illustrated in Figure 4-2. These are:

- User List (password, access, and modification rights)
- Job Definition List
- Program Definition List
- Job Queue
- Task Queue
- Binding List
- Job Data
- Task Data

The common data contains instances of data items defined by the user. Access and modification restrictions are defined along with the item structure. The data description language permits the user to specify a wide range of data structures. A definition of a hypothetical data base is given in Figure 6-12. The structural description of the item is transformed by the Item Definition Job into entries in the Name Encoding Table, Item List, and Name List of the IFS data.

Data can be stored and recalled to core memory service windows only by means of file system services which use the file system tables. The functions of these tables include translation of the names of data items into:

(1) Logical codes (ICC) which describe the logical position of the items in the common data structural definition

(2) Logical codes (IPC) which describe the position of the item instances in the common data

(3) The virtual address of the cells in which the data is located

(4) The real address of the track in which the data is located.

Figure 6-11 shows what each table requires as input and what each is designed to provide. The functions of the file system tables are as follows:

(1) Name Encoding Table

The function of the Name Encoding Table is to convert the name of an item from its alphanumeric form to a coded form which describes the logical position of the

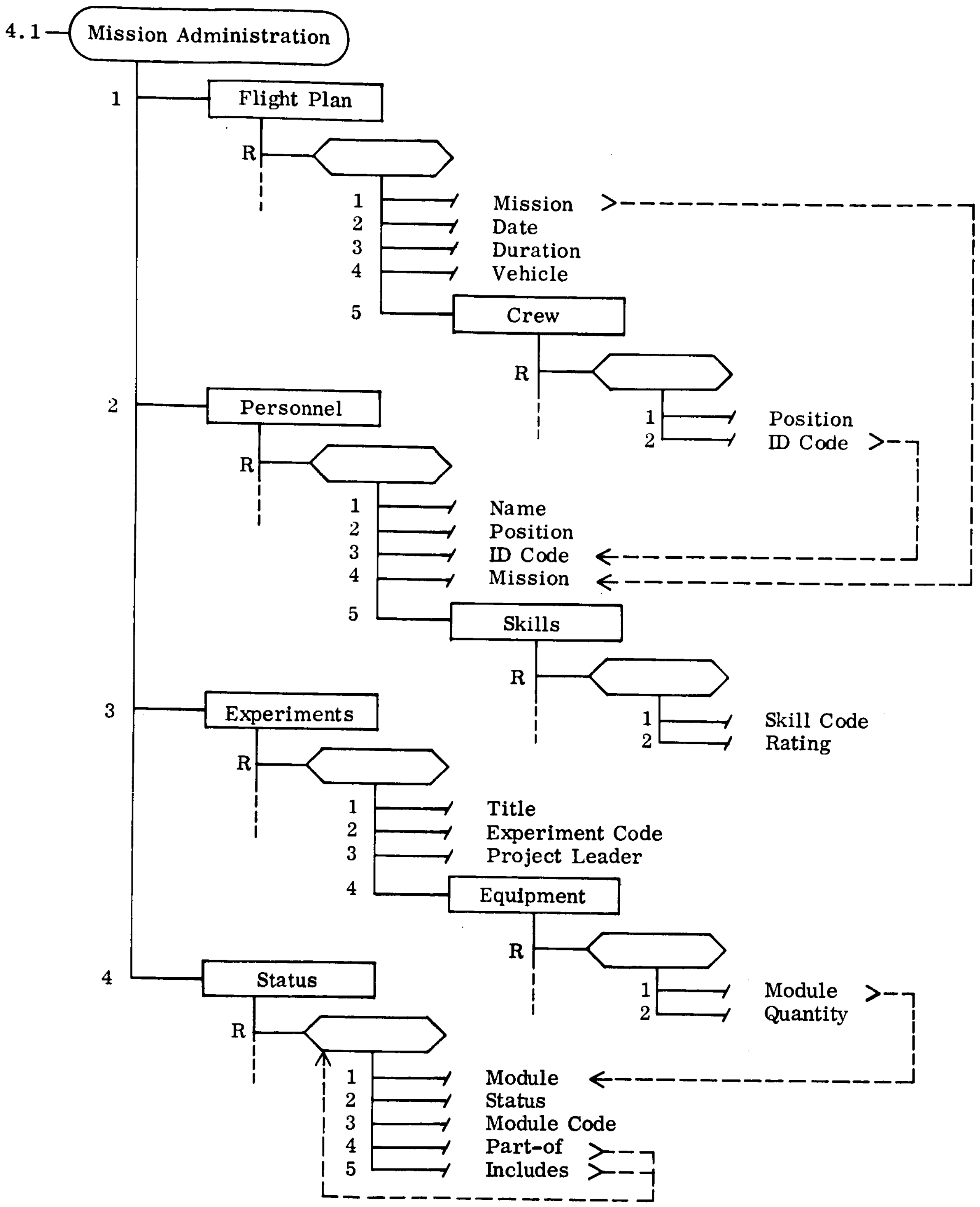


Figure 6-12. Data Base Example

item in the data structure. The coded form, called an Item Class Code (ICC), consists of integers which represent the nodes on the data pool tree structure which lie on the path from the data pool node to the node named. For example, the ICC of the skills array in Figure 6-12 would be 1. 2. R. 5. (A given name would have more than one ICC if it is used more than once in the data pool.) The letter "R" in the ICC indicates that at that point there is an array element (record) which is repeated in the data base as many times as instances of the item are recorded. Elements of a multidimensional array would have an R-value for each dimension.

(2) Item List

The Item Class Code must be converted to a code designating a unique instance of the item before the instance of the item can be retrieved from the real store. This is done by supplying values for the R's, creating the unique instance code called the Item Position Code (IPC). The Item List is used for this conversion, either alone or with the Index, depending on whether the request is conditional and the item is indexed. The entries of the Item List are ordered by ICC and supply a definition of the item in terms of its type and size. The Item List entry is linked to a Name List entry, which gives the name of the item, and to an Index Table if it is an indexed field.

(3) Index

For each indexed field, the Index is ordered by the values the field has taken (or can take). For each value, a list of R-numbers is given for the array elements which contain that value. Thus, a set of IPC's can be constructed for each value. Another function of the Index is to retain a count of item usage, by value. It is from these tallies that the need for indexing, or restructuring can be determined. Restructuring is the act of converting the instances of a data item to conform to a newly defined structure. It may be done to make the structure more efficient for a particular purpose.

(4) Page List

Since an IPC designates a unique item instance, it embodies sufficient information to enable the designation of its appropriate train and cell (or cells). Determining the virtual address (VA) of the page to be called is the function of the Page List. The Page List is ordered by the IPC of the first item of each page and supplies the virtual address of each page.

(5) Track Location Table

The Track Location Table maps the VA to real address (RA) for each track and access module on the real store.

6.7 DATA RETRIEVAL

The data retrieval strategy used by the system will be illustrated by an example. Consider a hypothetical data base of purchasing information shown in Figure 6-13 in indented tree form and in Table 6-3 in indented outline form. PURCHASING is a statement containing three files.

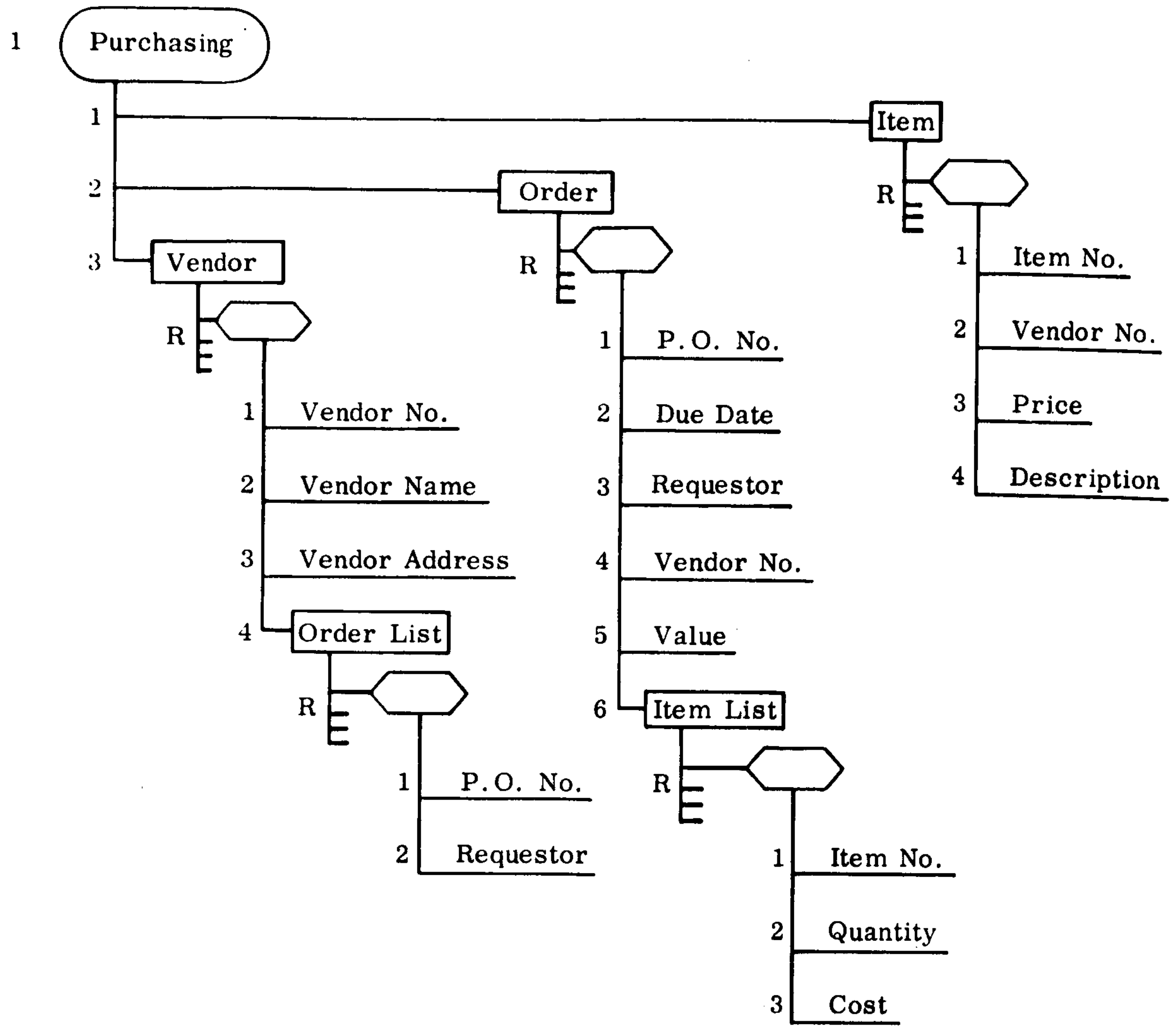
The ITEM file is a catalog containing a record for each item which might be ordered on a purchase order. Each record in the catalog contains three required fields. The ORDER file is a list of outstanding purchase orders. Each record contains identifying information for a purchase order and an ITEM LIST. The ITEM LIST contains a record for each item on the purchase order. This is an example of a file embedded in each record of a higher level file. The VENDOR file contains records describing vendors with a list of the active purchase orders for each vendor.

In the figure, the item PURCHASING has the ICC 1, since it is the parent node of the entire structure. Its subitems are numbered on the next level: 1.1 for the ITEM file, 1.2 for the ORDER file, and 1.3 for the VENDOR file. The records of the files occupy a level in the structure. The level is represented by an R in the ICC. The subitems of the records are numbered on the next level.

The Item Position Code (IPC) is used internally to identify units of data in the data pool. The ICC becomes an IPC when a record number replaces each R in the ICC. For example, the IPC 1.1.3.1 stands for the unique occurrence of the ITEM NO. field (1.1.R.1) in the third record of the ITEM file. The IPC 1.2.5.6 represents the ITEM LIST file (1.2.R.6) for the purchase order identified in the fifth record of the ORDER file.

6.7.1 Primary Directories

The structural description of the data is maintained in the system directories. The primary directories contain the information from the item definition. The primary directories are the Item List, the Name List, and the Name Encoding Table. These



Note: The letter "R" indicates that the item is a record which is repeated in the data as many times as the item is recorded.

Figure 6-13. Purchasing Item Structure

TABLE 6-3. DEFINITION FOR THE PURCHASING ITEM

ICC	ITEM DEFINITION
1	PURCHASING, S
1.1	ITEM, F
1.1.R.1	ITEM NO., I, V
1.1.R.2	VENDOR NO., I, 4
1.1.R.3	PRICE, E, V
1.1.R.4	DESCRIPTION, A, V
1.2	ORDER, F
1.2.R.1	P.O. NO., I, 6
1.2.R.2	DUE DATE, D, 6
1.2.R.3	REQUESTOR, A, V
1.2.R.4	VENDOR NO., I, 5
1.2.R.5	VALUE, E, V
1.2.R.6	ITEM LIST, F
1.2.R.6.R.1	ITEM NO., I, V
1.2.R.6.R.2	QUANTITY, I, 5
1.2.R.6.R.3	COST, E, V
1.3	VENDOR, F
1.3.R.1	VENDOR NO., I, 4
1.3.R.2	VENDOR NAME, A, V
1.3.R.3	VENDOR ADDRESS, A, V
1.3.R.4	ORDER LIST, F
1.3.R.4.R.1	P.O. NO., I, 6
1.3.R.4.R.2	REQUESTOR, A, V

are used to focus in on the data and to describe its structure. They function as a guide in interpreting the data so that it may be delivered to a program in a suitable form for processing.

6.7.1.1 Item List. The Item List is at the center of the directory system. It is a file with a record for each item (node) in the data pool structure. The records are in order by the Item Class Code of the item. Each Item List entry contains the item type and the size of the item. The size of records and statements is the number of subitems they subsume directly. The Item List also contains other information about the item. However, the primary structural information is the item type and size.

Table 6-4 shows the Item List for the PURCHASING item in the third column. The item type and the size are shown in each entry. The structure of an item is implied by the sizes given for nonterminal items.

6.7.1.2 Name List. The item names and units are maintained in a Name List file which is parallel to the Item List file. These elements are maintained separately because they are not needed to interpret the data structure and it is desirable to store the structural information as compactly as possible in the Item List. For each record in the Item List there is a record in the Name List, and the corresponding record numbers contain information about the same item.

6.7.1.3 Name Encoding Table. The item name is used as the identifier of the item by system users. The ICC is used as the identifier by the system. To enable the system to translate from an item's name to its ICC, the Name Encoding Table (NET) is maintained. The NET is a file containing a record for each unique item name in alphabetical order. The ordering permits rapid translation from an item's name to its ICC. Since there may be more than one item with a given name, each record of the NET contains a file of ICC's corresponding to a single name. Table 6-5 shows the Name Encoding Table for the purchasing statement. In practice, the names of other items in the data pool would be merged with the names of the items in the purchasing statement in a single NET.

6.7.2 Data Representation

The IFS treats an entire data pool as an unformatted stream of binary bits. This stream is segmented arbitrarily at any item boundary. Since no account is taken

TABLE 6-4. TERM LIST AND ITEM LIST

TERM LIST	ICC	ITEM LIST
PURCHASING	1	S, 3
ITEM	1.1	F, V
- - - -	1.1.R	R, 4
ITEM NO.	1.1.R.1	I, V
VENDOR NO.	1.1.R.2	I, 4
PRICE	1.1.R.3	E, 6
DESCRIPTION	1.1.R.4	A, V
ORDER	1.2	F, V
- - - -	1.2.R	R, 6
P.O. NO.	1.2.R.1	I, 6
DUE DATE	1.2.R.2	D, 6
REQUESTOR	1.2.R.3	A, V
VENDOR NO.	1.2.R.4	I, 5
VALUE	1.2.R.5	E, V
ITEM LIST	1.2.R.6	F, V
- - - -	1.2.R.6.R	R, 3
ITEM NO.	1.2.R.6.R.1	I, V
QUANTITY	1.2.R.6.R.2	I, 5
COST	1.2.R.6.R.3	E, 7
VENDOR	1.3	F, V
- - - -	1.3.R	R, 4
VENDOR NO.	1.3.R.1	I, 4
VENDOR NAME	1.3.R.2	A, V
VENDOR ADDRESS	1.3.R.3	A, V
ORDER LIST	1.3.R.4	F, V
- - - -	1.3.R.4.R	R, 2
P.O. NO.	1.3.R.4.R.1	I, 6
REQUESTOR	1.3.R.4.R.2	A, V

TABLE 6-5. NAME ENCODING TABLE

NAME	ICC FILE
COST	1.2.R.6.R.3
DESCRIPTION	1.1.R.4
DUE DATE	1.2.R.2
ITEM	1.1
ITEM LIST	1.2.R.6
ITEM NO.	1.1.R.1 1.2.R.6.R.1
ORDER	1.2
ORDER LIST	1.3.R.4
P.O. NO.	1.2.R.1 1.3.R.4.R.1
PRICE	1.1.R.3
PURCHASING	1
QUANTITY	1.2.R.6.R.2
REQUESTOR	1.2.R.3 1.3.R.4.R.2
VALUE	1.2.R.5
VENDOR	1.3
VENDOR ADDRESS	1.3.R.3
VENDOR NAME	1.3.R.2
VENDOR NO.	1.1.R.2 1.2.R.4 1.3.R.1

of word boundaries or the coding mechanisms of the devices, the data pool segments are independent of the characteristics of the computer and the storage devices. The service routines that interpret the data stream with the aid of the system directories are computer dependent. This approach focuses the computer dependence of the system in a small set of routines.

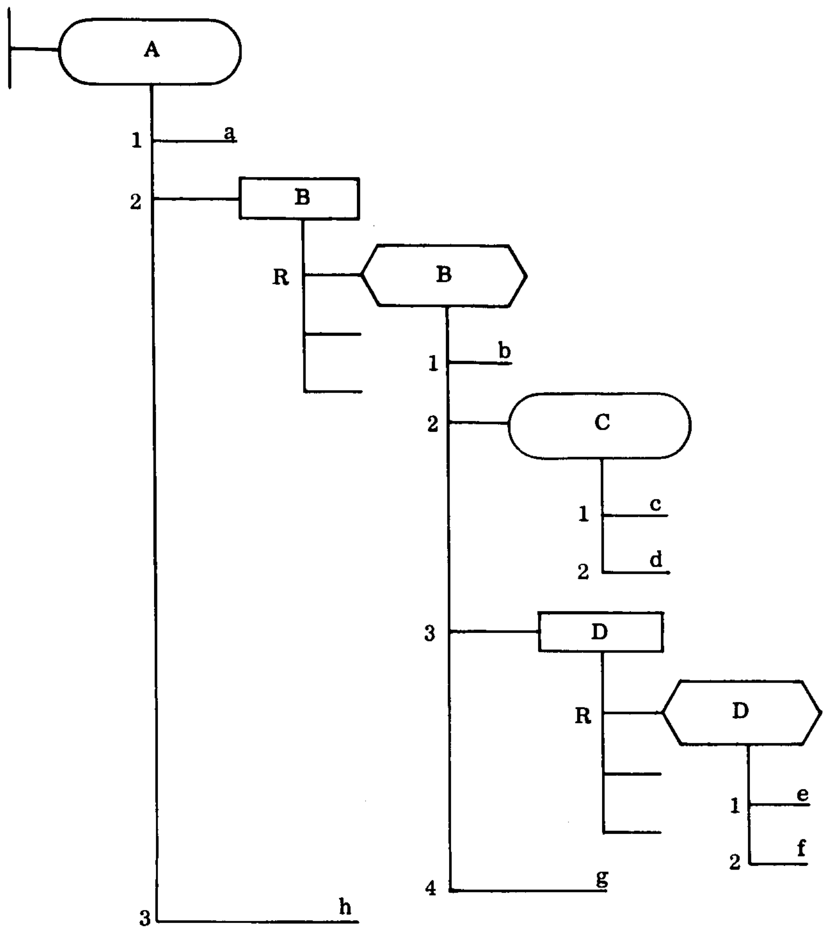
6.7.2.1 Data Stream. A hypothetical example will be used to explain the system's mechanism for representing data. Figure 6-14(a) contains a structure diagram of a statement named A. It consists of the field a, the file B, and the field h. The file B contains the field b, the statement C, and the file D, and the field g. The statement C contains two fields and each record of D contains two fields.

The files, records, and statements in a structure relate to their subnodes as a single entity, but only the fields take on values in the data. The "buss mesh" diagram in Figure 6-14(b) emphasizes this character of a structure. It shows the string of fields, a through h, emanating from vertical bars and representing the nonterminal items. The mesh arrows around the B and D records indicate the cyclic nature of these structures. To represent an actual data stream the buss diagram can be shown in expanded or "tree" form (see Figure 6-15).

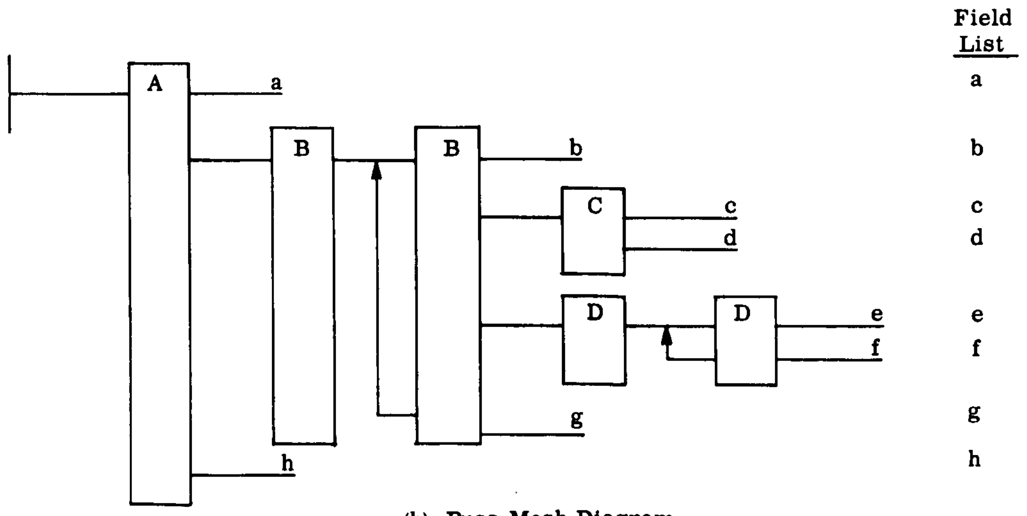
Only field values occur in the data stream. The data stream, represented in the figure by the column of subscripted letters at the right, consists of values for the fields. The interpretation of this data stream, with the Item List as a template, is symbolized by the buss tree diagram. The example assumes that there are three records in file B, with three records of file D in the first record of file B, four in the second, and two in the third.

6.7.2.2 Segmentation. The fields in the data stream must follow each other in the strict logical sequence dictated by the data pool structure. This does not mean that the data must be stored in a strict, physical sequence. The IFS segments the data stream and incorporates the ability to store the segments anywhere on the available devices. The logic of the system permits the segments to be of arbitrary size.

In writing data, the fields of the data stream are composed in memory cells of page size under the direction of the Item List. When a page is full, it may be stored in any available location of any storage device. The page is identified by the Item Position Code (IPC) of the first item it contains. This IPC is used as the key whenever the page is retrieved.



(a) Structure Diagram



(b) Buss Mesh Diagram

Figure 6-14. Sample Structure

IPC	Data Stream
1.1	a
1.2.1.1	b_1
1.2.1.2.1	c_1
1.2.1.2.2	d_1
1.2.1.3.1.1	e_{11}
1.2.1.3.1.2	f_{11}
1.2.1.3.2.1	e_{12}
1.2.1.3.2.2	f_{12}
1.2.1.3.3.1	e_{13}
1.2.1.3.3.2	f_{13}
1.2.1.4	g_1
1.2.2.1	b_2
1.2.2.2.1	c_2
1.2.2.2.2	d_2
1.2.2.3.1.1	e_{21}
1.2.2.3.1.2	f_{21}
1.2.2.3.2.1	e_{22}
1.2.2.3.2.2	f_{22}
1.2.2.3.3.1	e_{23}
1.2.2.3.3.2	f_{23}
1.2.2.3.4.1	e_{24}
1.2.2.3.4.2	f_{24}
1.2.2.4	g_2
1.2.3.1	b_3
1.2.3.2.1	c_3
1.2.3.2.2	d_3
1.2.3.3.1.1	e_{31}
1.2.3.3.1.2	f_{31}
1.2.3.3.2.1	e_{32}
1.2.3.3.2.2	f_{32}
1.2.3.4	g_3
1.3	h

Buss Tree Diagram

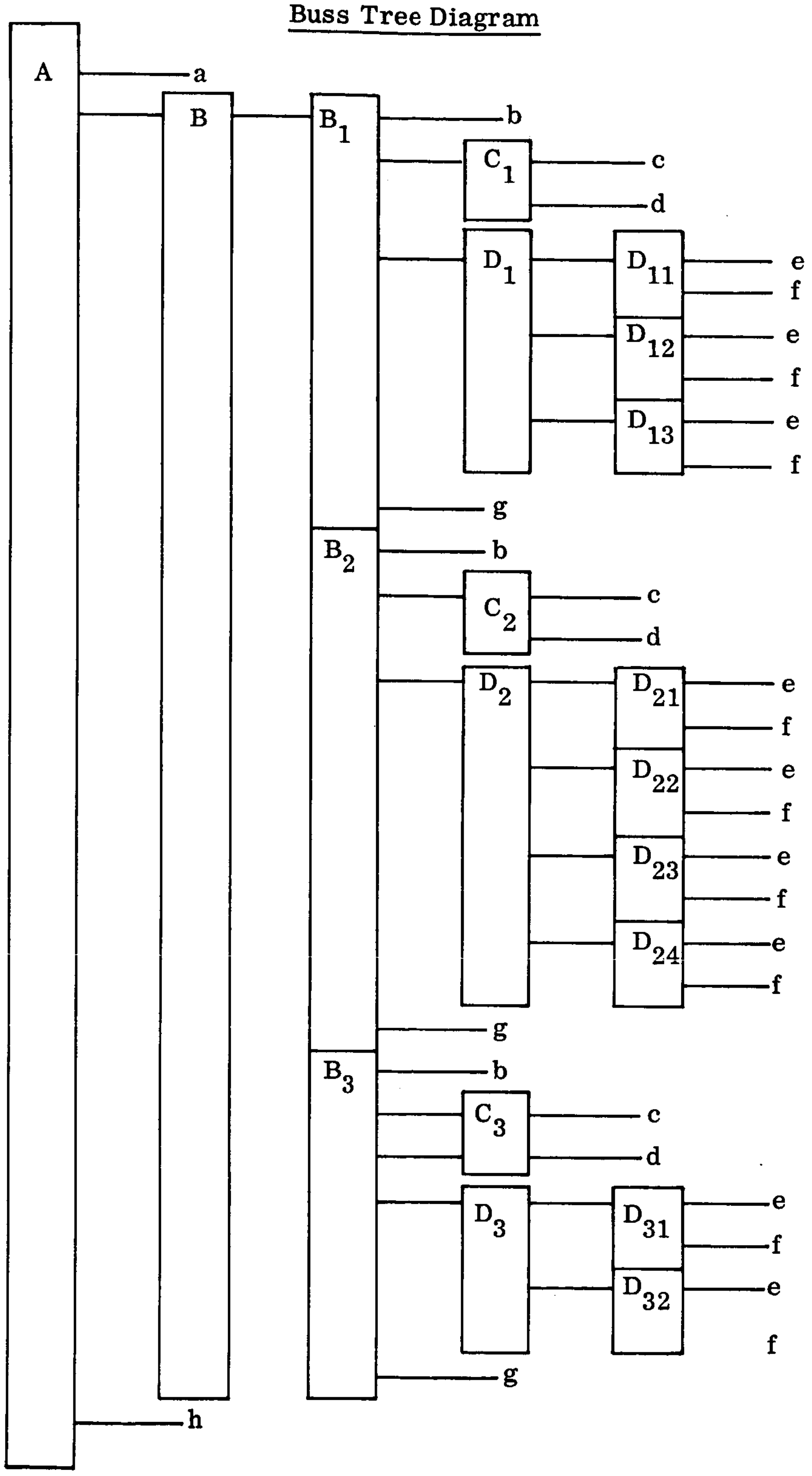


Figure 6-15. Buss Tree Diagram and Data Stream

Each segment is assigned a page whose virtual address (PVA) is used by the EFS to retrieve the page. The IFS maintains a Page List (PL) so that it may translate an IPC into the virtual address of the page containing the data identified by the IPC. The Page List is a file whose records contain the IPC of the first item in a page and the PVA. It is segmented like any other data, and the existence of PL entries for PL pages permits the system to focus rapidly on the desired page through a multi-level, variable depth, indirect addressing mechanism.

6.7.2.3 Page Index. The Item List does not contain enough information to allow the IFS to interpret the data stream. It gives the fixed size (number of subnodes) of records and statements and the size for each fixed length field; however, there are two levels of variability which must be taken into account. The sizes of variable length of fields may differ from one value to the next. The Item List contains only one entry for a field which may take many values. Similarly, the size of a file, i.e., the number of records it contains, varies from one occurrence of the file to the next. A file embedded in a higher file occurs in each record of the higher file, but it has only one entry in the Item List.

The page index, a string of bytes in the data segment, gives the size for variable length fields and the number of records for files in the order of the occurrence of the variable items in the segment. The segment index for the data stream of Figure 6-15 might be:

3	18	3	7	5	4	20	12	2	8
B	c_1	D_1	g_1	c_2	D_2	g_2	c_3	D_3	g_3

if the fields c and g are variable length. Since the first variable item encountered in the stream is the file B, its size is given first in the segment index. The file has three records. The next variable length item in the stream is the field c. Its value in the first record of file B is 18 units (bits or characters) long. The field g has a value whose length is seven units. The remaining numbers of the segment index give the sizes of the field C, the file D, and the field g for their occurrences in records 2 and 3 of the file B.

6.7.3 Sample Retrieval

An example will be used to demonstrate the use of the system directories in the retrieval of an item of data. The example is somewhat artificial because it is taken out of context, and it must be simplified to highlight the relationships without burdening the reader with undue detail.

The PURCHASING item, introduced in Table 6-6, Figure 6-13, and Tables 6-7 and 6-8, will be used as a basis for the example. The directories for the item are shown in Figure 6-16.

Suppose that a user wants to retrieve the purchase order numbers for purchase orders issued by J. Jones against the vendor whose number is 32Ø4. The request might look like:

```
RETRIEVE:       P. O. NO.
          IF VENDOR NO. = 32Ø4 AND
                   REQUESTOR = J. JONES
```

This request would be handled by the query job; however, the steps explained in this section are common to many retrieval situations. The details are greatly simplified and the condition is selected so that an orderly retrieval results. The general conditional search capability of the system is more comprehensive than this example implies.

6.7.3.1 Name Translation. The item names in the request must be translated to the system identifiers, the ICC's. This is done through the Name Encoding Table (NET). The NET is a data file like any other data in the system and it is segmented. The names are translated to ICC's by retrieving the appropriate page of the NET and searching its entries until a match is found on the names. This is handled by a system service routine.

The routine first uses the Page List (PL) to discover the page of the NET to retrieve. The PL entries for the NET are prefixed with a special identifier. The PL of Figure 6-16 shows some entries prefixed with (N). These are NET entries. To translate the name P.O. NO., the routine takes these steps:

(1) Match the name against the identifiers in the PL until an identifier less than or equal to P.O. NO. is found, where the next entry has an identifier greater than P.O. NO. Since the entry sought falls between the identifiers, it is in the segment identified by the first of the two identifiers.

NAME ENCODING TABLE

Name	ICC
Cost	1.2.R.6.R.3
Description	1.2.R.4
Due Date	1.2.R.2
Item	1.1
Item List	1.2.R.6
Item No.	1.1.R.1
	1.2.R.6.R.1
Order	1.2
Order List	1.3.R.4
P.O. No.	1.2.R.1
	1.3.R.4.R.1
Price	1.1.R.3
Purchasing	1.
Quantity	1.2.R.6.R.2
Requestor	1.2.R.3
	1.3.R.4.R.2
Value	1.2.R.5
Vendor	1.3
Vendor Addr.	1.3.R.3
Vendor Name	1.3.R.2
Vendor No.	1.1.R.2
	1.2.R.4
	1.3.R.1

ITEM LIST

ICC	Type	Size
1.	S	3
1.1	F	V
1.1.R	R	4
1.1.R.1	I	V
1.1.R.2	I	4
1.1.R.3	E	6
1.1.R.4	A	V
1.2	F	V
1.2.R	R	6
1.2.R.1	I	6
1.2.R.2	D	6
1.2.R.3	A	V
1.2.R.4	I	5
1.2.R.5	E	V
1.2.R.6	F	V
1.2.R.6.R	R	3
1.2.R.6.R.1	I	V
1.2.R.6.R.2	I	5
1.2.R.6.R.3	F	7
1.3	F	V
1.3.R	R	4
1.3.R.1	I	4
1.3.R.2	A	V
1.3.R.3	A	V
1.3.R.4	F	V
1.3.R.4.R	R	2
1.3.R.4.R.1	I	6
1.3.R.4.R.2	A	V

PAGE LIST

Identifier	Page Address
(I) 1	58469
(I) 1.2.R.2	87466
(I) 1.3	42879
(N) Cost	74346
(N) Order	49632
(N) Quantity	89248
(P) 1	73298
(P) 1.1.68.4	25321
(P) 1.1.132.2	64843
(P) 1.2.5.3	46932
(P) 1.2.23.5	65187
.	.
.	.
.	.
(P) 1.3.10.2	87933
(P) 1.3.48.3	34658
(P) 1.3.64.1	24863
(P) 1.3.82.1	52178

Figure 6-16. Directories for the Purchasing Item

In the PL shown in Figure 6-16, the name P.O. NO. is found to fall between ORDER and QUANTITY. The NET page containing the entry for P. O. NO. is, therefore, the one which begins with the entry for the name ORDER; the page address is 49632.

(2) Retrieve the page of the NET containing the desired entry, and search the NET entries until a match is found. In the NET in Figure 6-16, the entry for P.O. NO. is found in the second page. There are two items with that name: 1.2.R.1 and 1.3.R.4.R.1. Only one of these is needed. The choice must be made by the use of qualifiers or by the context of the problem. For example, the qualifier VENDOR could have been used in the problem statement to indicate that all pertinent items are subsumed by the VENDOR file. This would dictate the selection of the ICC 1.3.R.4.R.1 for P.O. NO. since VENDOR has the ICC 1.3 and is a parent of the pertinent item.

(3) Follow similar steps for the names REQUESTOR and VENDOR NO. The consistent set of ICC's discovered for the three names is:

P. O. NO.	=	1.3.R.4.R.1
VENDOR NO.	=	1.3.R.1
REQUESTOR	=	1.3.R.4.R.2

6.7.3.2 Search Strategy. The structural relationships among the items in the sample retrieval request dictate the strategy of searching for the pertinent purchase order numbers. The ICC of the P.O. NO. field, 1.3.R.4.R.1, contains two record numbers. The condition on VENDOR NO. and REQUESTOR is used to set these record numbers to the values which meet the conditions. The search strategy used in an actual query job is more comprehensive than the one to be discussed here.

The best strategy is to establish the record numbers which meet the conditions at the higher level first. This narrows the number of files which must be searched at the lower level. The condition on VENDOR NO. is the key to establishing the record numbers at the higher level. Only those records which contain the value 32Ø4 for VENDOR NO. need be considered.

There are several ways of determining which records contain the key value. If the field is indexed, the system maintains a subsidiary directory table that relates each value the field assumes to the set of record numbers containing the value. Indexing is discussed in Paragraph 6.7.4. If the field is not indexed, the file whose record numbers are to be established must be searched to determine the records which contain the key value. For the purposes of this example, assume that record number 51 is found to contain the value 32Ø4 for the field VENDOR NO.

The second record number need be established only within record number 51 at the higher level. If the higher file contains 100 records, there are 100 files at the lower level. Establishing the record number at the higher level first eliminates 99 files from consideration. Effectively, the ICC of the pertinent purchase order numbers is translated from 1.3.R.4.R.1 to 1.3.51.4.R.1, with only the record number of one file to be established by further operations. The condition on the field REQUESTOR establishes the record numbers in the lower level file. Again, the subsidiary directory is used if the field is indexed, or the file is searched if it is not. In this example, assume that record number 12 is discovered to contain the key value J. JONES for REQUESTOR.

On each level, more than one record number might meet the condition. In general a multidimensional array of record numbers is developed from a condition. The array provides the appropriate record numbers for any set of desired items to be retrieved under the condition. The retrieval steps discussed in the following paragraphs are performed for each set of related items and for each of the record number groups which meet the condition.

The retrieval steps will be discussed for the retrieval of the purchase order number in the twelfth record of the Order List file which is in the fifty-first record of the VENDOR file. The condition establishes those record numbers which convert the logical identifier of the desired field, P.O. NO., from the ICC 1.3.R.4.R.1 to the IPC 1.3.51.4.12.1.

6.7.3.3 Data Page Retrieval. When the IPC of the pertinent item is known, the data page containing that item can be retrieved through the PL. The steps discussed are used for random retrieval of any item in the data pool or for initializing an item for serial processing or random processing within the bounds of the item.

The first step is to obtain the page virtual address and the range of its data contents from the PL. The desired item has the IPC 1.3.51.4.12.1. As shown in Figure 6-16, this IPC falls between the page identifiers 1.3.48.3 and 1.3.64.1. The desired item is within the page addressed 34658 with other data ranging between the bounding identifiers.

The next step is to prepare an Item List Table to act as the structure template of the part of the data stream contained in the page. Since both boundaries of the page are within the file whose ICC is 1.3 (the VENDOR file), the definition for that file is sufficient to interpret the entire data stream within the data page. The page of the Item List which contains the definition for the VENDOR file is retrieved through the SNL. The page needed is 42879 as shown in the PL of Figure 6-16. This Item List page is retrieved, and the definition for the VENDOR file is mapped into the Item List Table.

The data page (34658) is retrieved and the Item List Table is initialized so that the system can step through the data stream on the page to the desired item. This is accomplished by stepping down the Item List Table to the entry corresponding to the first item in the data page (1.3.R.3 corresponding to 1.3.48.3), thereby setting parameters which direct the system in further stepping.

6.7.3.4 Data Stream Interpretation. At this point in the example, a page of the data stream is available with the part of the Item List needed to interpret it. The system steps from the item 1.3.48.3 to the item 1.3.51.4.12.1, which is the desired purchase order number. The stepping is accomplished by summing the sizes of each item preceding the desired item to develop a pointer to the precise bit location of the desired item within the page. The sizes of fixed length fields, records, and statements are obtained from the Item List Table. The sizes of variable length fields and files are obtained from the page index. Sizes are accumulated and the IPC of each item is developed until the IPC of the desired item is reached. The value for this item may be extracted for display or processing.

6.7.4 Indexing

An important characteristic of the IFS is the ability to index selected fields. When a field is indexed, the system maintains a subsidiary directory table relating the values assumed by the field to the numbers of the records in which those values occur.

The indexing feature provides a tradeoff between the speed of retrieval and the size of storage required. When a field used in a condition is indexed, the system can focus very rapidly on the pertinent items without searching the data stream. This is accomplished by maintaining a Field Value Table (FVT) and an R-Value Index Table (RVIT) which occupy additional storage space and must be updated each time a change is made to the set of values for the field.

The payoff for indexing a field is readily apparent when the alternatives available to the system in interpreting a condition are investigated. In the preceding section, the retrieval example included the condition:

VENDOR NO. = 32Ø4

The attribute, VENDOR NO., occurs in each record of the VENDOR file. There may be hundreds of such records. Since each record contains a subsumed file, the occurrences of the VENDOR NO. field are widely dispersed through the data stream. If the data stream must be searched to determine the records which meet the condition, there is a high likelihood that a different segment retrieval will be required for each record to be checked. This amounts to several hundred segment retrievals.

If the VENDOR NO. field is indexed, the values it assumes are stored in a compact file with a link to the list of record numbers which contain that value. There is a high likelihood that the first segment of the FVT retrieved will contain the desired value. If there is only one record meeting the condition, its value is stored in the FVT and the search is finished with one page retrieval. If there are a number of records with the key value, the list of record numbers is maintained in the RVIT. A link in the FVT entry for the key value points directly to the record number list. In this case, the search is accomplished with two page retrievals. Either way, the time saving is great. The payoff is greatest for records of files at a high level which contain embedded files, and it improves as the number of records in the file increases.

An indexed field is tagged in the Item List with a code specifying the type of indexing: all values, ranges, or selected values. The Item List entry for an indexed field contains a record number which identifies the specific Field Value Table for that field. When the system needs to determine the record numbers for the occurrences of an indexed field which contains a given value, the record number in the Item List entry converts the ICC of the FVT, 1.2.5.R.4, to an IPC. The correct FVT can be retrieved directly, and the entry containing the given value can be found. This record contains the record number sought, if there is only one occurrence of the field with the given value. Otherwise, the FVT record contains a record number identifying a specific RVIT file which contains the list of record numbers for the occurrences of the field which contains the given value.

6.7.5 Linkage

A data pool is basically a tree structure. Each node has a single parent node and may subsume a number of subnodes. In order to relate separate items in a tree structure, these items must branch off from a common stem of the tree at the point they have in common.

Figure 6-13 was a diagram of the pure tree structure for the PURCHASING item. The fields in the VENDOR file, VENDOR NO., VENDOR NAME, and VENDOR ADDRESS are related because they are all attributes of a vendor. The list of outstanding purchase orders against the vendor, ORDER LIST, is also an attribute of the vendor. The relationship among these four items is shown in the tree structure by placing them as direct subitems of the record of the VENDOR file. The fields of the purchase order file, ORDER, and the list of items ordered have a similar relationship. They are all attributes that describe a purchase order, so they are defined as direct subitems of the record of the ORDER file.

The VENDOR file and the ORDER file are two elements of purchasing information. This relationship is shown in the tree structure by subsuming both files directly under the statement PURCHASING. However, there is another relationship between the two files which is not shown in the structure. All the attributes of a purchase order are pertinent descriptors for the purchase orders in the list for a given vendor. This could be shown by placing the entire set of purchase order attributes in each record of the file ORDER LIST. This results in a gross redundancy if the existing purchase order file is retained. If that file is eliminated, the purchase order information is available in the VENDOR file, but it must be grouped by vendor in that part of the tree structure.

It can be assumed that the high activity use for purchase order information is accomplished more conveniently if the ORDER file is retained in purchase order number sequence as a direct subitem of the PURCHASING statement. Also, the detailed attributes of a purchase order are needed only rarely when processing the VENDOR file, and such information is superfluous most of the time. In such cases, a logical link can be established to cut across separate branches of the tree structure. The link shows the relationship between its source and target items and allows it to be exploited while eliminating the redundancy of duplicate items and permitting the high activity data to be stored compactly. The high activity data can be processed much more efficiently when superfluous data is removed to a logically independent node, yet rarely used data can be associated with the source item when it is needed.

Figure 6-17 shows the same purchasing information in a structure which makes liberal use of links. More of the relationships among the items are shown with less redundancy. The VENDOR file contains only the fields which describe a vendor. The list of purchase orders against the vendor has been replaced by a link to the records of the ORDER file. Each record of the VENDOR file is linked to the set of records in the ORDER file which have a VENDOR NO. field equal to the VENDOR NO. field of the VENDOR record. Logically, each VENDOR record subsumes a purchase order file for one vendor, with the full set of descriptors for each purchase order. Yet the VENDOR file remains a compact list of vendors' attributes and no duplication of data is required. The ORDER file may be maintained without reference to the VENDOR file. The set of records associated with a given vendor is automatically redefined by changes in the VENDOR NO. field (the link criterion) in the ORDER file.

Similar links have been established in Figure 6-17 to relate other items. The ORDER file contains a link to a record of the LISTS file where the list of items on that purchase order is maintained. The link criterion is the purchase order number. Also, each item in the ITEM LIST file is linked, by item number, to a record in the ITEM file. This makes the full item description logically available with each item on the purchase order without cluttering the ITEM LIST file for the majority of uses.

A link connects two items, i.e., a source and a target. The source link is an item with some characteristics of a statement. It subsumes the link criterion, a field whose value is the key to closing the link. The target link subsumes the criterion field in the target branch of the tree. When the values of the criterion field are equal in the source and target links, the target link's parent item is logically subsumed by

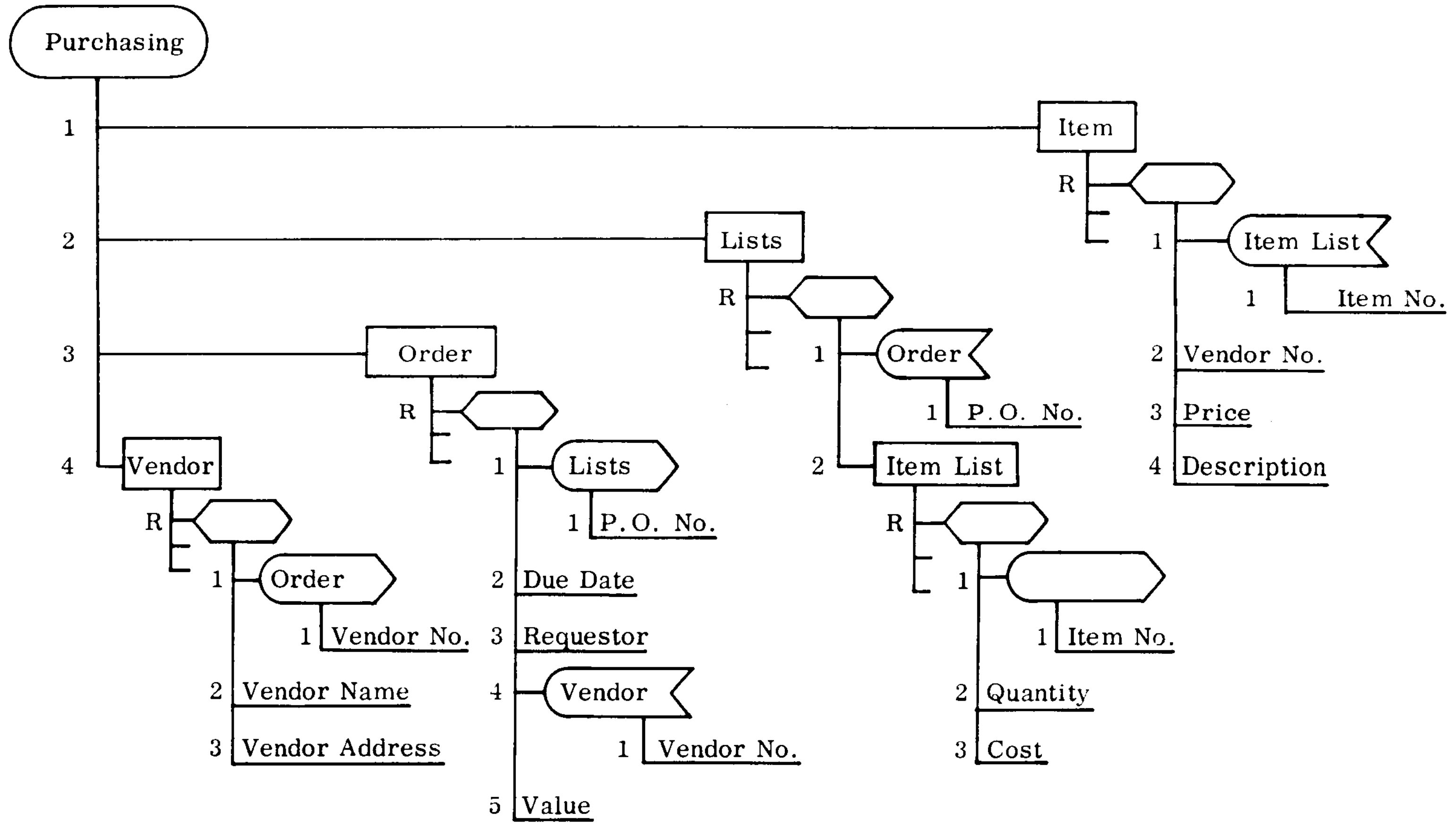


Figure 6-17. Purchasing Item with Links

the source link. The source link operates like a file whose records contain the items of the target structure for each occurrence of a match between the source and target criterion values.

The system follows a link through a subsidiary directory, the Linkage Table. A link item contains a record number in its Item List entry. This leads to an entry in the Linkage Table, which contains the ICC of the matching link. Both source and target links have entries in the Linkage Table, permitting the system to follow a link in either direction. The target's substructure is subsumed by the source link, and the source's superstructure subsumes the target link.

6.7.6 Data Integrity

This objective can be stated in terms of the following subgoals:

(1) Data Security Checks must be provided to protect the user against invasion of privacy by protecting his data against unauthorized read and write operations.

(2) Data Validity Checks must be provided to protect authorized users against collisions of data usage in a time-shared common data base and to protect the data base against illegitimate modification by authorized users.

The data protection mechanism is built into the resident reentrant Service Package which responds to all user data access and storage requests. Since these routines perform all data access and storage operations for all users, the constraints with regard to data usage for these routines must be inherited from their parent jobs and not be inherent in the routines themselves.

6.7.6.1 Security Safeguard. The system of data protection employs two separate but interacting mechanisms: security level and access/modification rights. Each data item class is assigned a security level for access and another level for modification from one of eight classifications.* Likewise, each user receives a clearance level which gives him access and modification rights to all items below his level. His rights to items classified at his level or above depend upon whether or not the item requested is on his access-rights or modification-rights list. A table of such rights, negotiated with

* In practice, security levels will range from 0 (unrestricted) to 6 (highest restriction), and clearance levels will range from 1 (lowest clearance) to 7 (no constraints).

a Data Administrator, is maintained for each user. A message to a Data Administrator is prepared for each unauthorized access or modification attempt.

Two degrees of access/modification rights will be recognized. The simpler is a right to a class of data given by item name, such as a file. A more discriminating right is to a particular subset of records in a file, where the subset is made conditional on a data check. An example of this is the right to specific raw data such as test results only if the data satisfies a condition, such as a given value in an identity field.

6.7.6.2 Validity Safeguard. The approach to access or modification rights solves the problem of data integrity with regard to controlling access against unauthorized users. However, by itself, it does not protect a data base against destruction of data by system failure or by authorized users acting on the basis of invalid information. This aspect of the integrity-ensurance problem is aided by an ability to recover older editions of each page from an "archive file."

6.7.6.3 Item Lockout (Busy Bit). Comprehensive data maintenance operations which perform structural modifications over multisegment data sets present additional conflict and protection requirements. In such situations, multiusers' read-write safeguards may not be sufficient to ensure valid operations if the data set being modified is used independently during the maintenance operation. To provide for such protection, temporary data lockout is provided by a "busy bit" in the SNL entries for the data being modified. This bit, set and reset at the request of a maintenance job, effectively locks out use of any data in a class during the time the privileged maintenance job is running, thus ensuring that all data delivered to users is consistent with the Item List.

6.7.6.4 General Procedure. There are two three-bit fields in each entry of the Item List assigning security restriction levels to the item, one for access and one for modification. Level 0 will designate unrestricted data and level 6 will designate the most highly restricted data. Restriction levels are assigned so as to be nondescending when moving from an item to its parent item. Each user is assigned a clearance level which gives him unconditional access to all data whose restriction level is below his clearance level. Access or modification to data classified at or above the user's clearance level is conditional upon whether the data belongs to the class of data for which the user has specific rights (Open Class) or meets a conditional check for unique items (Field Condition).

A user can be assigned a class of data items or specific data items expressed as a field-value condition, for which he has explicit rights. A rights check will be made only if a data request fails the clearance check. The usual rules of inclusion of items in an item class hold for the rights check. In the case of a field-value condition, the user is permitted access only to those records containing a field whose value is specified in the Field Condition List in his entry.

6.8 ITEM SERVICES

The concept of the virtual store of a program, properly supported by the system, has the effect of relieving the assembly language program designer of input/output considerations if the program is dealing with trains and cells. In the Coherent System, however, the majority of application programs will deal with items in a higher level language rather than with cells and trains. Users and programmers enjoy a great advantage in dealing with logical structures of data in terms of generic item names, so that they are independent of the problems of the layout of the data in an address space. This not only greatly simplifies interaction with the data base, but also allows the user or program procedure to remain invariant even though the actual item structures and their physical mapping may change.

It would be desirable if these application programs, though dealing in items through IFS services, could also achieve independence from explicit input or output of items, somewhat akin to the implicit cell service available to programs dealing in address spaces through the EFS. This capability can be realized in the Coherent System, through the unified design and development of the modules which support it.

6.8.1 Symbolic Coding and Implicit Services

In order to develop an insight into the concept of implicit item services, and to understand what is entailed in the construction and operation of programs using them, it is appropriate to recall an important distinction (and parallelism) in the typical use of symbolic references between programs coded in assembly language and those coded in higher level languages such as FORTRAN. Keeping this in mind will aid in drawing analogies between implicit item services, a new concept, and implicit train services, a concept already understood in systems where the program addresses a virtual store. It will also lead to the identification of demands which implicit item services will make on the design of translators for the Coherent System.

In an assembly language program, data is referenced for processing by knowing its location. A symbol is usually employed to denote the base location of a set of data to be processed, and numeric references are made to locations relative to the symbolic one. Numeric equivalents to the symbol are assigned by the assembly program.

In a problem-oriented (e.g., FORTRAN) language program, data is referenced for processing by giving it a name. The symbol for this name denotes a list of occurrences which may be indexed, so that successive references may be made using the symbol. Machine instructions are generated and numeric references to the data locations area are supplied by the compiler.

In an assembly language program, the coder instructs the assembler to reserve the space needed to hold the data. The programmer must then code explicit statements which cause the input of data before his processing statements reference the address space holding it. Also, the programmer must code explicit statements which cause output of data from such address spaces following processing.

In the problem-oriented language program, the coder instructs the compiler on rules for the dimensions of the data, and the compiler reserves the address space. Through rules and explicit statements given to the compiler regarding the input and output of the data, explicit instructions are generated to cause these operations before and after processing, respectively.

The use of symbols in making data references through the EFS is akin to the way they are used in an assembly language program; i.e., they refer to the <u>location</u> of the data. Thus, implicit train service (using a virtual store which includes all actual system trains) is natural to processing statements of the assembly language.

The use of symbols in referencing data through the IFS is akin to the way they are used in a higher level language program; i.e., they refer to the <u>value</u> of the item (<u>not</u> the location). Thus, implicit item service (using a virtual item array which includes all actual data base occurrences) is natural to the processing statements of the higher language.

Suppose in a FORTRAN program, the following statements are coded:

```
    5      SUM = 0
   10      DO 20 I = 1, 100
   20      SUM = SUM + SAMPL(I)**2
   25      RMS = SQRT(SUM)
```

Suppose that SAMPL represents data base occurrences which must be input before this processing takes place, and RMS is a result which is to be entered into the data base. Conventionally, the coder must provide explicit statements to perform these I/O operations. But since the coder does not deal with storage locations (even symbolic oncs) in the processing statements, it seems natural to remove storage considerations altogether in a language where his references denote items, not their locations. He should be able to consider that all items in the system are instantaneously available to his processing statements. This goal is the parallel of the EFS implicit train service, where the coder regards all train storage as instantaneously available to statements referencing symbolic locations.

To realize this objective, calls to the IFS item services would not be coded by the programmer. Nevertheless, the calls must be made (just as the Virtual Store Manager is called by a trap unbeknown to the program).

Having received the formal item definitions supplied by the coder and having inspected the processing statements, the compiler must be able to generate the mechanisms for reading and storing through IFS services. The Job Management System will know which actual items have been bound to formal items in the job definition, so it can then couple the system trains involved to the task and, therefore, indirectly to the IFS.

6.8.2 The Mechanics of Implicit Item Services

The concept of implicit item services is developed in the context of programs whose role is to process system items maintained in the on-going data base. Of course, there will be programs in the system which must input from terminal devices, or output to terminal devices, but these are mostly system services or other modules which are a part of the Coherent System. It is assumed that the great body of user programs will be confining their attention to items already in the system. For these, the notion of a data base containing items which may be directly referenced in the processing statements of a higher language is an important new idea. Thus, the emphasis will be on developing this notion for use by such programs. For the other class of programs, concerned more with logistics than computation, it is not yet clear whether implicit item services are appropriate, or if so, what form they should take to be feasible, due to limitations imposed by having to deal with terminal files. This will require further study. In any case, nothing is taken away by this new concept; explicit item services may be designed, and all traditional forms of I/O capability are still available.

For the purposes of developing this concept, therefore, the term "program" in the ensuing presentation will include only members of that large class of programs which are designed to process data pool items. (While, in fact, compilers will have to recognize, in general, other kinds of I/O services, the presentation of this new demand on the compiler will gain clarity by omitting all but the pure case of implicit item service.)

6.8.2.1 The Compiler's Role. Just as a compiler may recognize a function name in a statement as a requirement to set up a linkage to a subroutine, it also recognizes formal item names and sets up the proper parameters, and coupling to the the IFS, for the IFS may be looked upon as a subroutine, or a collection of subroutines. A requirement of the item services is that they must be able to serve (or "be a part of") more than one task. Consequently, the IFS is reentrant, and never stores into its own train. This dictates certain design requirements:

(1) A subroutine which becomes a part of a task may refer only to the trains known to that task; i.e., the task train table of the parent task is the one used by the processor. Thus, a task using IFS services must supply all data areas within some train to which it is bound.

(2) One copy of the IFS services (call it the IFS train) is shared by many tasks. When control is passed to the IFS, the register used by the IFS to make references within itself must be loaded by the parent task prior to the branch. (For different parent tasks, the formal file used by the EFS will be different.)

Considerations (1) and (2) require that the compiler build certain records of the coupling between the program train and the other trains which will be needed.

- If a subroutine function is needed, the train containing it must be assigned a value of f, corresponding to a task train table entry. This value of f would be part of the register contents used to reference the subroutine. The compiler makes the register assignment according to the established standard subroutine linkage conventions.

- The program itself must be identified as one which can be made into a task (as opposed to a subroutine). Such programs will eventually require a task train table, and are themselves compiled as formal train zero. Other programs do not directly reference other trains, and are not considered except in the context of the parent task.

- For a processing statement X = F (Y), the compiler would do the following in the Coherent System:

  — For the Symbol Y. If Y does not appear on the left-hand side of any other statement, e.g., Y = G(Z), it must be an input (basis) needed by the program. The symbol is then a Formal Item Name (abbreviated FIN) and is identified as a basis.

  — For the Symbol X. If X does not appear on the right-hand side of any other statement, it must be a result produced by the program. The symbol is then a FIN and is identified as a result.

  If either X or Y appears on both sides of assignment statements, it may be an intermediate variable, but could conceivably be a result item also. For example, if

  $$Y = f(X)$$

  $$Z = f(Y)$$

  the role of Y as a candidate for implicit item services is not predictable by the compiler. The compiler could list Y as a "potential" FIN. The job definition will make it clear to the Job Management System if Y is to be a result item, since a binding equation will exist for it.

- If there is at least one basis or result FIN, a formal train is assigned in order to provide coupling to the train of the IFS; i.e., IFS implicit item services will be needed.

- If the symbol is indexed, it represents an array of item occurrences. In these cases the compiler, implicit I/O notwithstanding, must know the dimensions of the array, if fixed, or is aware of the case where it is variable just in order to generate the proper instructions for stepping through the locations containing the item values. Also it must assign the storage to hold the values.

To generate the proper processing instructions and again for storage assignment purposes, the compiler must know the format of the item values (e. g. , floating point, fixed point of a certain length, etc.).

Such information is also needed to set up the mechanisms for implicit item services. The point is that nothing new is needed in the way of language specification to achieve this, as current languages already provide for implicit or explicit specification of these particular item attributes.

- For each variable item array, a formal train is assigned to hold the data. Its size is limited only by the virtual addressing capability of the hardware within a train. The compiler will associate the formal train with the Formal Item Name (FIN) symbol and will indicate whether it is basis, scratch, or result. This will be needed by the Task Initiator and/or Task Terminator modules of the Task Manager.

- For all fixed size arrays, one formal scratch train is assigned within which virtual addresses are assigned by the compiler to hold the data. One such virtual address assignment is needed for each symbol representing data, whether basis, scratch, or result. A scratch area is also provided for use by the IFS. This can probably be assigned as the remaining virtual address space in this formal scratch train.

- For each basis FIN and result FIN, a formal basis or result train is assigned.

For the sample program presented earlier ...

```
 5      SUM = 0
10      DO 20 I = 1,100
20      SUM = SUM + SAMPL(I)**2
25      RMS = SQRT(SUM)
```

... the compiler generates the program, and a program definition. The program definition contains a dictionary of formal item names, in this case:

FIN	Sense	Format	Array Size	Base Address in Scratch Train
SAMPL	Basis	E(Exponential)	100	L(SAMPL)
RMS	Result	E	1	L(RMS)

It also carries virtual store requirements such as the highest location referenced within itself; also the skeleton of the task train table, reflecting formal train assignments, e. g. ,

FTN(f)	Sense	(Comments)
0	Execution	(For train to hold this program)
1	Execution	(For linkage to IFS)
2	Execution	(For linkage to external SQRT program)
3	Scratch	(For arrays and use by IFS)
4	Basis	(Will be bound to actual system train)
5	Result	(Will be bound to actual system train)

The basis train 4 and result train 5 will not be referenced directly by this program, but by the IFS, which will use the task train table created for this program at execution time.

Other information is carried in the program definition which enables the binding to other programs to be established when this program is entered into the Coherent System's repertoire of programs which may be used as tasks, that is, the linkage information needed to use SQRT and the IFS.

6.8.2.2 From Program Compilation to Running Task. In order to properly expose the mechanics of implicit item services, the steps leading up to the execution of a program in a particular job and data context should be reviewed. To do this, a particular data context and job context will be postulated for the program sample used previously, and the Coherent System will be viewed from the inside as the presentation is developed.

Postulate that a data base has been built up through System Support Data Services. Each user item has been assigned to a train by the IFS. The structure of each user item is set by an Item Definition Job, and data is entered through the Item Entry Job. All of these items are addressable as trains by programs using EFS implicit train services.

Suppose one array contains records holding collections of statistical error samples. Each record holds an imbedded file of 100 samples, and two simple items which are the root-mean-squares of the 100 values of each of the fields in each sample. This structure is illustrated in Figure 6-18. Notice that XERROR and YERROR are carried as integers, and that XRMS and YRMS are in exponential format. Let us postulate that the programs which enter the error samples into these arrays are already designed, but the function of producing the values from the samples and storing them into the Quality Summary has yet to be programmed.

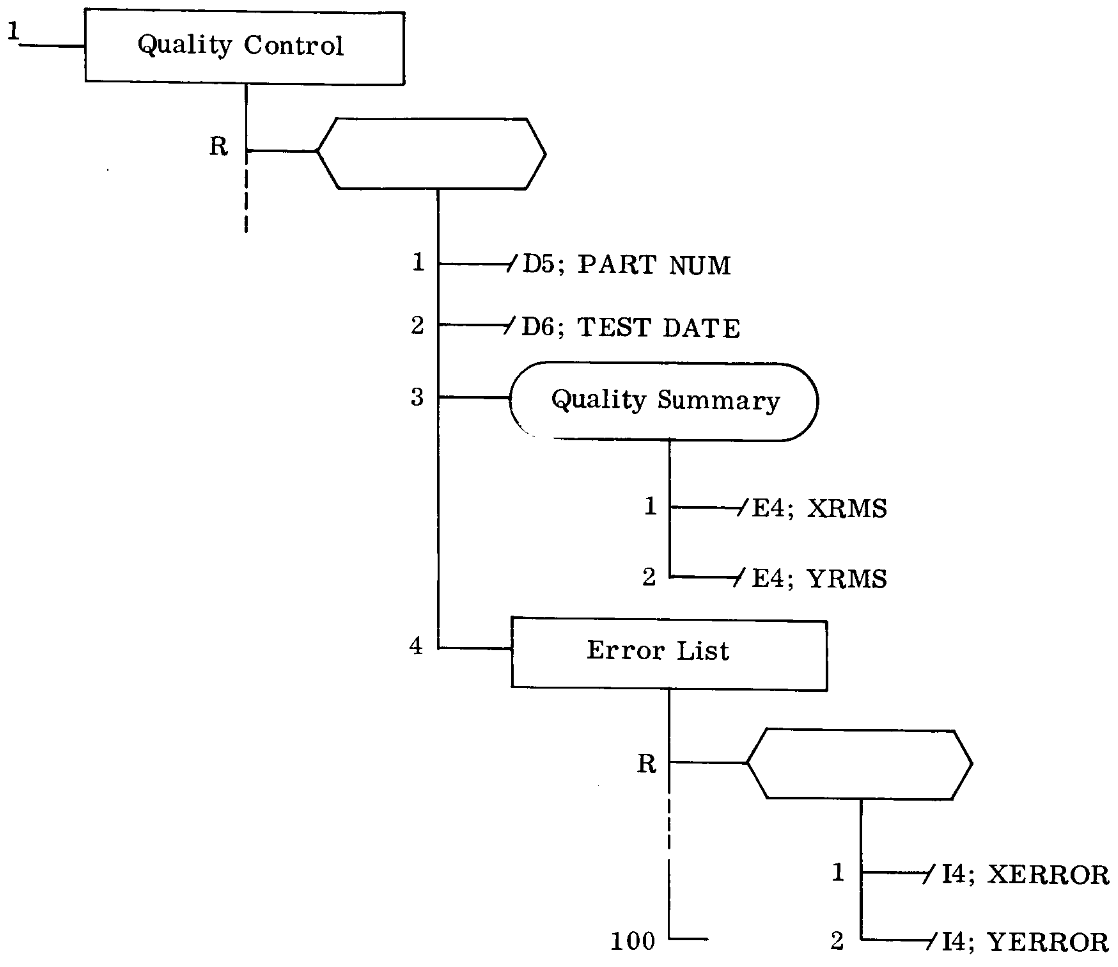


Figure 6-18. Quality Control File

Now this code is written and compiled, as described in the previous section.

The object data of the compilation may now be submitted as input data to a Program Entry Job, one of the Repertoire Expansion System Services. In this job, the program is defined to the system by a user-assigned name. The user also specifies the train name of the addressable train to contain the program. If it is a main program (capable of being named as a task in a job) rather than a subroutine, a new train must be defined. In this case the default option will be to assign it the same name as the program.

In the example, the name of the program (and of its train) will be RMSER. The input to the Program Entry job is the object program code and the program definition (collectively called the "program data"), and the name "RMSER." In the example, this step is illustrated by the flow in Figure 6-19. The user at this point can think of the system as containing the function

RMSER(SAMPL) = RMS.

Now RMSER is known to the Job Management System as a program, and it may be used as part of a job. One job needed by the user is one which can provide the RMS values for newly acquired sets of XERROR and YERROR values.

If this were a "one-shot" operation, our next step could be the Job Request itself. But most jobs, including this one, will be functions which are desirable to repeat as new sets of data are available. The next step in building the user's repertoire is to define a job, in which RMSER is identified as a task bound to the actual items in the data base, as postulated above. The program RMSER will be used to update both XRMS and YRMS in specified records of the Quality Control array.

The input to the Job Definition System Service Job consists of the Job Name, UPDATE XYRMS, the programs to be executed in the job, and for each program, the binding equations which make the programs into tasks.

```
#DF JOB UPDATE XYRMS
          RMSER (XERROR where Part Num = 12345) = XRMS where Part Num = 12345
          RMSER (YERROR where Part Num = 12345) = YRMS where Part Num = 12345
```

This definition is entered into the Coherent System's repertoire of jobs, to be recalled when a Job Request is issued which calls for UPDATE XYRMS (Figure 6-20). Other job definitions may call for the same program to operate with other data base files of similar structure, or other records in this file. In this example, the system is now prepared to respond to a user's Job Request to run UPDATE XYRMS.

The Job Request is a statement to the Job Management System to execute such-and-such a job in the repertoire. The foregoing System Services each had job names and were called with a Job Request. Now UPDATE XYRMS has achieved the

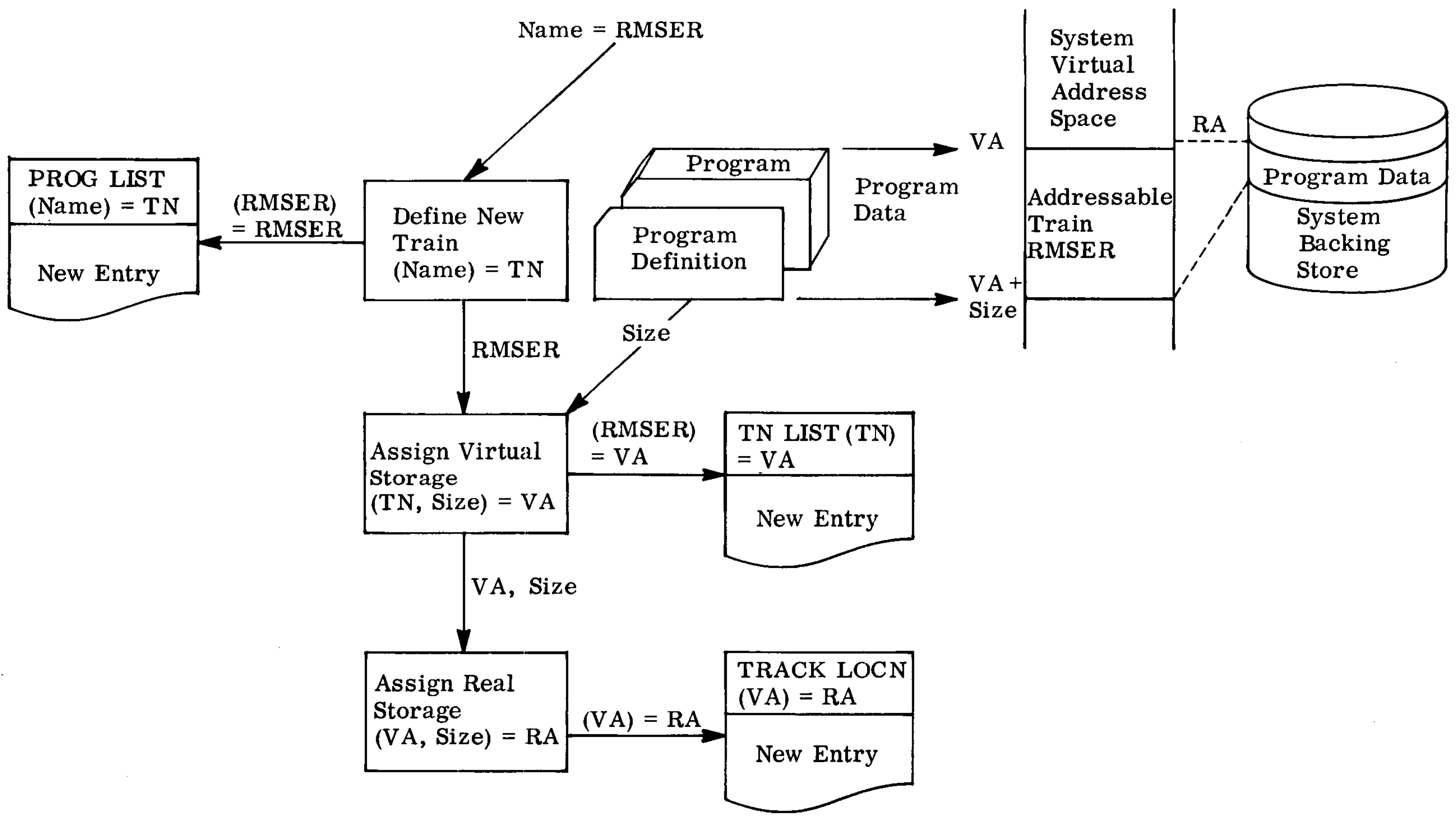


Figure 6-19. Program Entry

same status. The request EXECUTE UPDATE XYRMS now causes the Job Management System to take many steps:

(1) UPDATE XYRMS is found to be in the Coherent System's job repertoire, and the job definition is placed in the Task List for the Scheduler. (To keep the presentation simple, such considerations as validating the user's identity, showing the priority or other scheduling criteria, are omitted as not central to the point here.)

(2) When the time comes for Task 1 of the job to be started, the program name is used to recall the program definition which includes the Task Train Table (TTT) skeleton, a dictionary, and symbols used for external linkage (see Figure 6-20).

(3) Now this, combined with the binding equations, allows the Task Manager to construct the Page Tables and the Task Train Table needed to execute this task (Figure 6-21). The Task Specification will be retained by the Task Manager until the end of the job.

(4) Now task initiation and initialization may take place. (The best implementation approach for this function requires further analysis. For this example, the code performing this function is called the Task Initiator, although it will perhaps be generated by the compiler as the first executable portion of the program.) In creating the Task Train Table, it was located in primary memory at $h \frown 0$. The Task Manager now loads h into the TTT Base Register, and branches to the Task Initiator. The Task Initiator links to the IFS module which will build the basis array for the task.

(5) From the dictionary for RMSER, the basis array SAMPL is the one to be constructed. The binding equation for this task says that SAMPL is really XERROR where PART NUM = 12345. From the dictionary, it is seen that the array will be provided at relative address L (SAMPL) within the train f = 3, and that 100 field values required are to be in E = exponential format.

The IFS, through its Name Encoding Table, identifies item XERROR with the structural position (ICC) 1. R. 4. R. 1 in the Quality Control file (see Figure 6-19). PART NUM, in position 1. R. 1 will be examined for each R until the value 12345 is found, say in record r. Now the value for XERROR is retrieved from 1. R. 4. R. 1 for R=1, 2, . . . , 100 and placed in a scratch area in the scratch train f = 3 provided for IFS use. The virtual addresses needed for these retrievals are maintained in the Page List of the IFS which holds the Quality

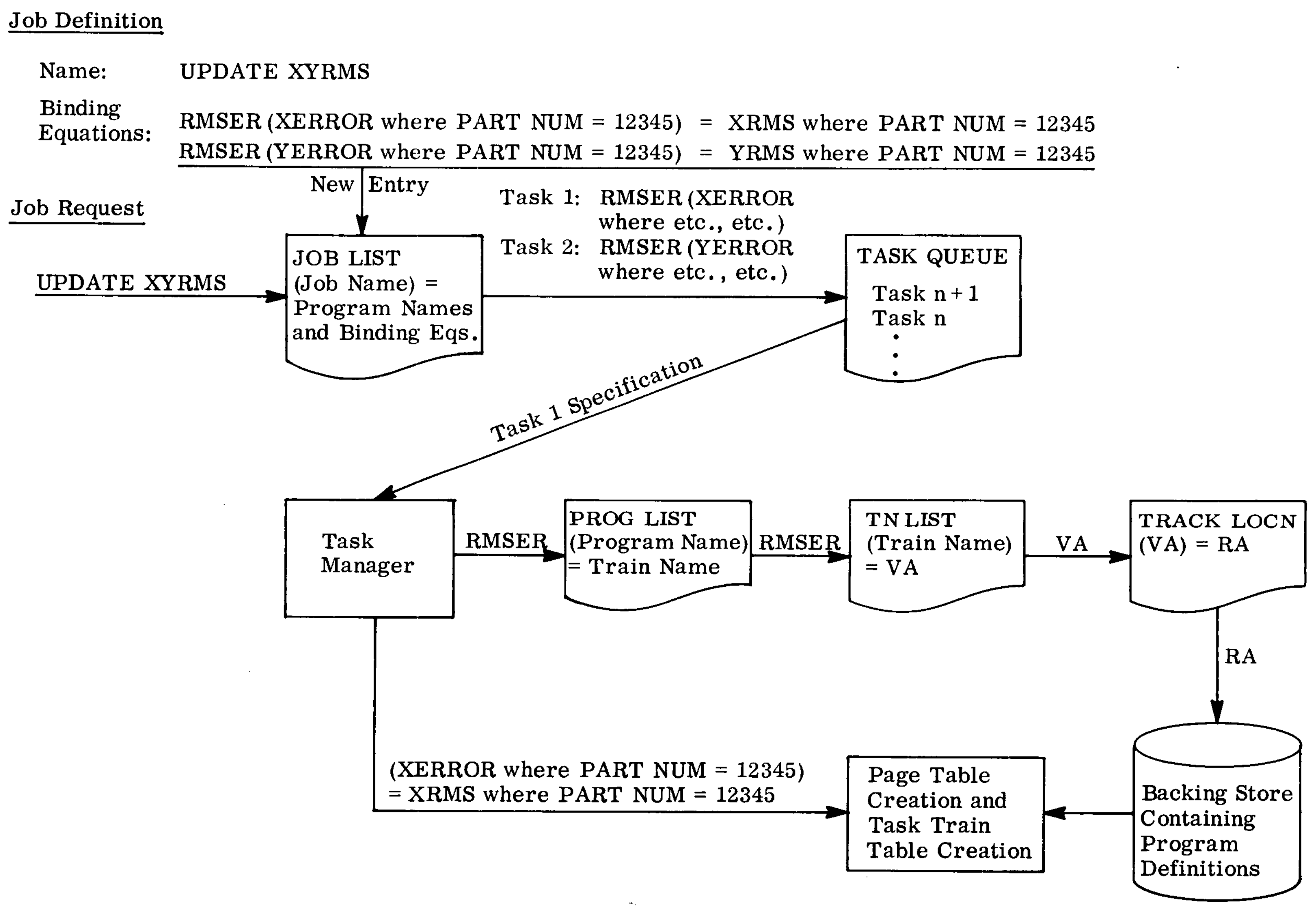
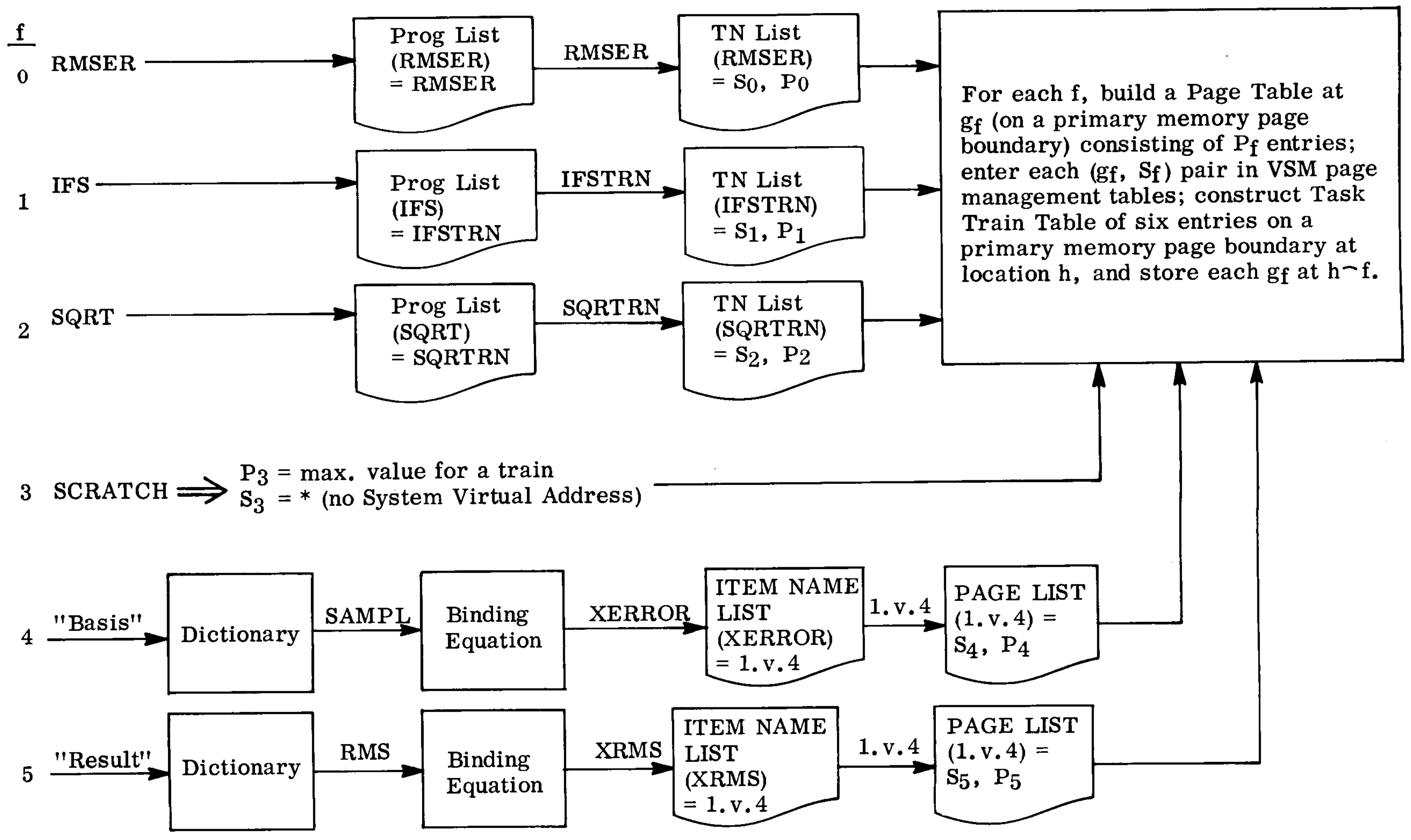

Job Definition
Name: UPDATE XYRMS
Binding Equations: RMSER (XERROR where PART NUM = 12345) = XRMS where PART NUM = 12345
RMSER (YERROR where PART NUM = 12345) = YRMS where PART NUM = 12345
New Entry
Job Request
UPDATE XYRMS
JOB LIST (Job Name) = Program Names and Binding Eqs.
Task 1: RMSER (XERROR where etc., etc.)
Task 2: RMSER (YERROR where etc., etc.)
TASK QUEUE
Task n+1
Task n
Task 1 Specification
Task Manager
RMSER
PROG LIST (Program Name) = Train Name
RMSER
TN LIST (Train Name) = VA
VA
TRACK LOCN (VA) = RA
RA
Backing Store Containing Program Definitions
Page Table Creation and Task Train Table Creation
(XERROR where PART NUM = 12345) = XRMS where PART NUM = 12345

Figure 6-20. Job Definition and Request


f
0 RMSER
1 IFS
2 SQRT
3 SCRATCH
4 "Basis"
5 "Result"
Prog List (RMSER) = RMSER
Prog List (IFS) = IFSTRN
Prog List (SQRT) = SQRTRN
RMSER
IFSTRN
SQRTRN
TN List (RMSER) = S_0, P_0
TN List (IFSTRN) = S_1, P_1
TN List (SQRTRN) = S_2, P_2
For each f, build a Page Table at g_f (on a primary memory page boundary) consisting of P_f entries; enter each (g_f, S_f) pair in VSM page management tables; construct Task Train Table of six entries on a primary memory page boundary at location h, and store each g_f at h⌢f.
P_3 = max. value for a train
S_3 = * (no System Virtual Address)
Dictionary
SAMPL
RMS
Binding Equation
XERROR
XRMS
ITEM NAME LIST (XERROR) = 1.v.4
ITEM NAME LIST (XRMS) = 1.v.4
1.v.4
PAGE LIST (1.v.4) = S_4, P_4
PAGE LIST (1.v.4) = S_5, P_5

S_f = System Virtual Address of train bound to f except for scratch trains.

P_f = Number of pages in the train bound to f. If scratch, then P_f is the maximum allowable.

Figure 6-21. Page Tables and Task Train Table Creation

Control file. The IFS, using these virtual addresses, was able to make direct references to the train because it was bound to the task, and is entry f = 4 in the Task Train Table; that is, the Page List for the required Quality Control record was built at the start of the task by the Task Manager.

Now, the values retrieved are integers and are required to be in exponential format. These are now converted and stored beginning at relative location L (SAMPL) in the scratch train supplied.

(6) The RMSER program was constructed by the compiler to make references to its basis array at L (SAMPL) in scratch train f = 3. Now the code may be executed. It calculates the root-mean-square of the 100 values just supplied, and stores the result (in exponential format) at L(RMS) in scratch train f = 3.

6.8.3 Explicit Item Services

Data services can also be offered to the programmer through a Data Service Language (DSL). The requesting program may be written in any language that can provide a service call with literal parameters. The parameters of the service request will typically be the following:

(1) Service command (including condition, if any)

(2) Service window or formal file

(3) Error control word.

These explicit item services fall in two general categories: positioning and copying. Two modes, sequential and random, apply to either category. The positioning commands position a pointer in the formal file (and item bound to it); the copying command reads data into (or writes data from) the service window. The sequential mode operates relative to the current pointer position, and the random mode positions the pointer to an absolute element in an array (given as a record number).

The item service commands are listed below:

(1) Step < file > (< n >);

This command steps the pointer on the formal file relative to its current position. Direction of step is indicated by sign of n (with a default value of +1).

(2) Step Absolute <file> (<n >);

Moves pointer to record n of file. Default value of n is 1. The value of n may be end.

(3) Write < item > ;

Writes the formal item to the actual item bound to it. Pointer is moved one step.

(4) Read <item>;

Reads the actual item bound to the formal item into the window. The pointer is moved one step.

SECTION VII. USER LANGUAGES AND SYSTEM SUPPORT JOBS

7.1 LANGUAGE SPECIFICATION AND PROCESSING

The specification and processing of system languages appear in many contexts within the Coherent System (or any system which manages a large on-going data base for interactive users). Within such a system there will be at least the following types of system languages:

(1) Job request, or command language

(2) Data item definition language

(3) Data item input language

(4) Job description language

(5) Data service request language

(6) On-line (interpretive) computational languages

(7) Compiler languages

(8) Macro-assembler language.

It should be possible to perform a large part of the input stream scanning, analysis, and interpretation of these languages with basically the same, or similar, syntax-directed processors. In this section, a designer-oriented language capable of specifying the syntax and semantics of system languages will be discussed. In the remainder of the section it will be used to specify the syntax of languages, in Items (1) through (4), for the Coherent System. The form of language specification to be discussed is called action graphs.

Action graphs are similar to a transition diagram of the syntax of a language. The symbols used in action graphs are shown in Figure 7-1. As an example, an action graph for a simplified "assignment statement" compiler is shown in Figure 7-2. A detailed description of action graphs and a processor for action graphs is given in Appendix B, a paper entitled "Inscan: A syntax-Directed Language Processor."

7.2 JOB DEFINITION AND COMMAND LANGUAGES

7.2.1 Conditions

The basic objective of the job language in the Coherent System is to enable the user to run a predefined job with a minimal amount of information required at job request time. The approach taken is to permit a prestored job description to contain a specification of the structure of a job in terms of subjobs and tasks.

There are two basic approaches to job input parameter specification in a system with an on-going data base and an IFS. The choice hinges on the question of how qualification of data base items should be accomplished. The first approach requires that the user create a data base item for each input parameter by naming an existing item or explicitly running a Conditional Search and Extract Job which creates the item in the data base. The second approach, and the one adopted here, is to permit an input argument to be a data base name which may be qualified by a conditional statement. This approach implies that Conditional Search and Extract is not a user job but rather a system act which is performed whenever a qualifier (WHERE-clause) is encountered in an argument list. The advantage of this approach is that jobs that are to operate on a subset of an item (a frequent need) can be requested in a single statement, and the awkward situation of being required to handle every condition as a parameter is avoided completely. A "query," in this context, is simply a display or print job with a qualified input argument. Another major advantage of this approach is that the conditional search operation need not be considered as a separate task which must run to completion before a subsequent task is initiated, but rather as a function which may be executed dynamically when the task program references its input parameter.

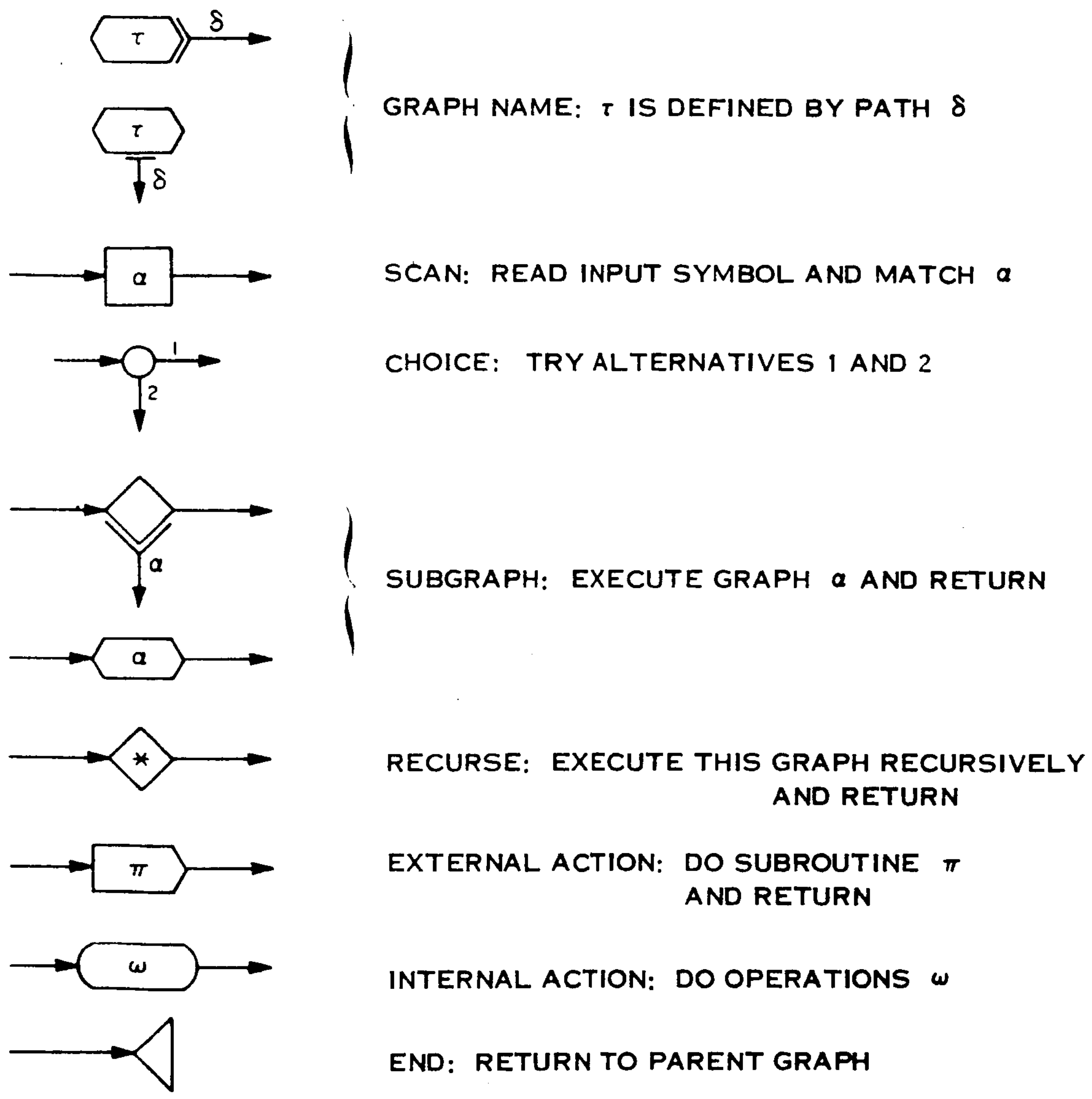
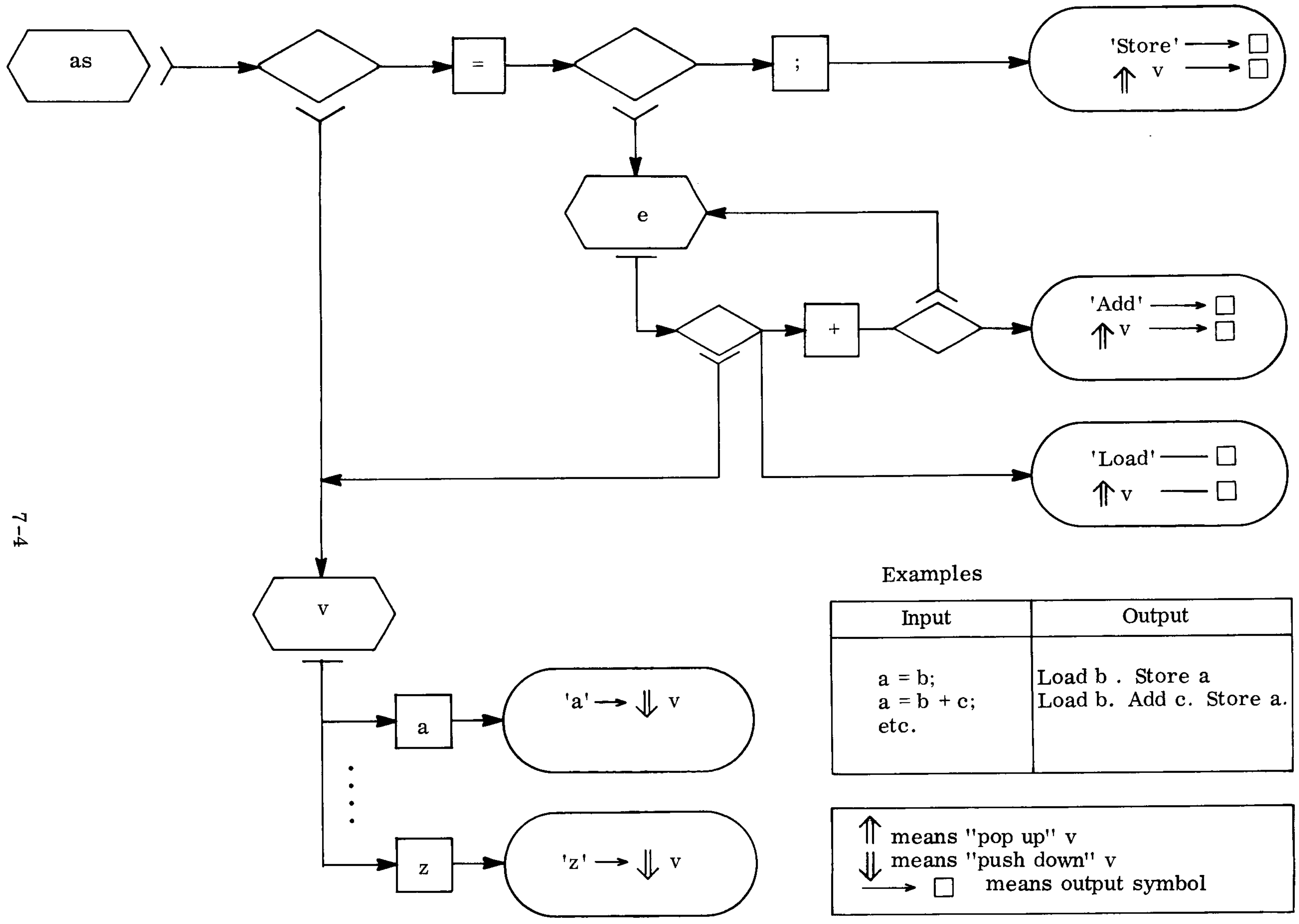

τ
δ
GRAPH NAME: τ IS DEFINED BY PATH δ
α
SCAN: READ INPUT SYMBOL AND MATCH α
1
2
CHOICE: TRY ALTERNATIVES 1 AND 2
α
SUBGRAPH: EXECUTE GRAPH α AND RETURN
*
RECURSE: EXECUTE THIS GRAPH RECURSIVELY AND RETURN
π
EXTERNAL ACTION: DO SUBROUTINE π AND RETURN
ω
INTERNAL ACTION: DO OPERATIONS ω
END: RETURN TO PARENT GRAPH

Figure 7-1. Action Graph Symbols



Examples

Input	Output

a = b;
a = b + c;
etc.
	Load b . Store a
	Load b. Add c. Store a.

⇑ means "pop up" v
⇓ means "push down" v
→ □ means output symbol

Figure 7-2. Assignment Statement Compiler

Examples of requests using this scheme might be:

(1) # Print (Resources Where Experiment = Surveyor)

or (2) # Delete (Mission Record Where Date <1968)

Using the first scheme, job (1) would require the sequence:

# Extract (Resources, Experiment = Surveyor) = Temp

# Print (Temp)

and job (2) would require

# Conditional Delete (Mission Record, Date < 1968).

It is also clear that the first approach is a special case of the second (the adopted approach), in that jobs with explicit specification of conditional tasks in lieu of qualified arguments can also be executed. The second approach is also more flexible in that all possible points that may potentially require Extract need not be prespecified in the job description. This has the effect of reducing the inventory of unique job descriptions.

7.2.2 Job Constraints

The question of scope or power of the job request language must be answered independently of the preceding question of argument qualification. To answer this question the motives and requirements of the job request language must be reviewed.

The job request (or command) language can conceivably occupy any part of the spectrum from simply naming a single program (and arguments) to be executed, to specifying a general procedure in some macro-oriented language such as TRAC* or GPM.** The "single program" end of the spectrum provides the advantage of simplicity in the binding and job management functions but the disadvantage of limited flexibility from the user's point of view — he requires a program change in the total job program (or a different program) for each change in function. Programs tend to be large (although they still may be modular) and oriented to a single job goal. The "general macro-language" end of the spectrum offers the advantage of flexibility from the user's point of view. Without "programming" in a procedural language he can freely manipulate

*C. N. Mooers: TRAC — A Procedure Describing Language for the Reactive Typewriter, Comm ACM March, 1966.

**C. Strachey: A General Purpose Macrogenerator, Computer Journal, October, 1965.

goal-oriented macros which perform operations at any desired level of complexity, including conditional operations, nested operations, iterative operations, and loop control. The disadvantage of this degree of generality is that it introduces a high degree of interpretive operation and the difficulty of binding parameters of macros which may appear in loops.

The approach taken is to permit multitask jobs but no iteration (or loop control) over tasks (at the job level). This permits smaller task modules which perform simpler job steps to be programmed and used flexibly in job structures but avoids the complexities of loop control in the Job Manager. It does provide the possibility of parallel processing at the job level (that is, having more than one task of a job active at one time), since the data prerequisites of each task are known at job request time.

One additional point may be made. To permit nesting of jobs, it is necessary that the result of every nested job be a single data item. For example, in order that the argument u of a job A (u, v) = w be specified as a result z of job B (x, y) = z by using the notation

$$(1) \qquad A\ (B\ (x, y),\ v) = w,$$

it is necessary that the result of B(x, y) be a single item. If it is more (or less) than a single item, then the expression (1) is ambiguous. Although this natural, and compact, notation should be permitted for the large class of jobs that may meet this constraint, there is an important class of programs that does not. In order to permit these programs to be used in jobs (or as jobs), a structure more general than the familiar form (1) will be defined for jobs. Any partial ordered (non-looping) structure of jobs and tasks will be permitted to be defined as a job. (Of course, constraints due to compatibility of parameter structures and item types must be maintained.)

The user should be able to define new programs to the system and to define new jobs (commands) in terms of previously existing jobs and programs. These definitions could be formal structures stored by the JMS in the data pool using IFS services. They then would be available to the JMS at job run time to direct parameter binding and task linkage functions.

The definition of new programs and jobs (and new data items) is accomplished by using standard system jobs and suitable descriptive languages provided for those purposes. A program description consists of the name of the program and the name and data definition of its input and output parameters. A job definition, in addition, contains a structural description in terms of a list of component tasks and their parameters. An example of the type of job structure that can be defined is given in Figure 7-3. The formal description of the job J shown in the figure may be written as follows:

```
# DF      J(1, 2) = (3, 4, 5)
              A (1, 'p') = (3, *1, *2)
              B (*2, #, 2) = (*3, *4)
              C (*1, *3) = (4, *5)
              D (*3, *4, q) = (*6, #)
              E (*5, *6) = 5
```

where # represents a null input or unused output; *1, *2, *3, *4, *5, and *6 are dummy (bound) variables; and A through E are previously defined jobs. Of course, in an actual situation, symbolic names which serve as semantic cues should be used for job names and parameters.

It is important to note that since the data management system contains structural descriptions of program and job parameters as well as data base items, extensive checking and automatic restructuring operations can be performed by the system. This is a key capability in extending the useful life of programs while data structures are modified and improved.

The job run request for the above job would be:

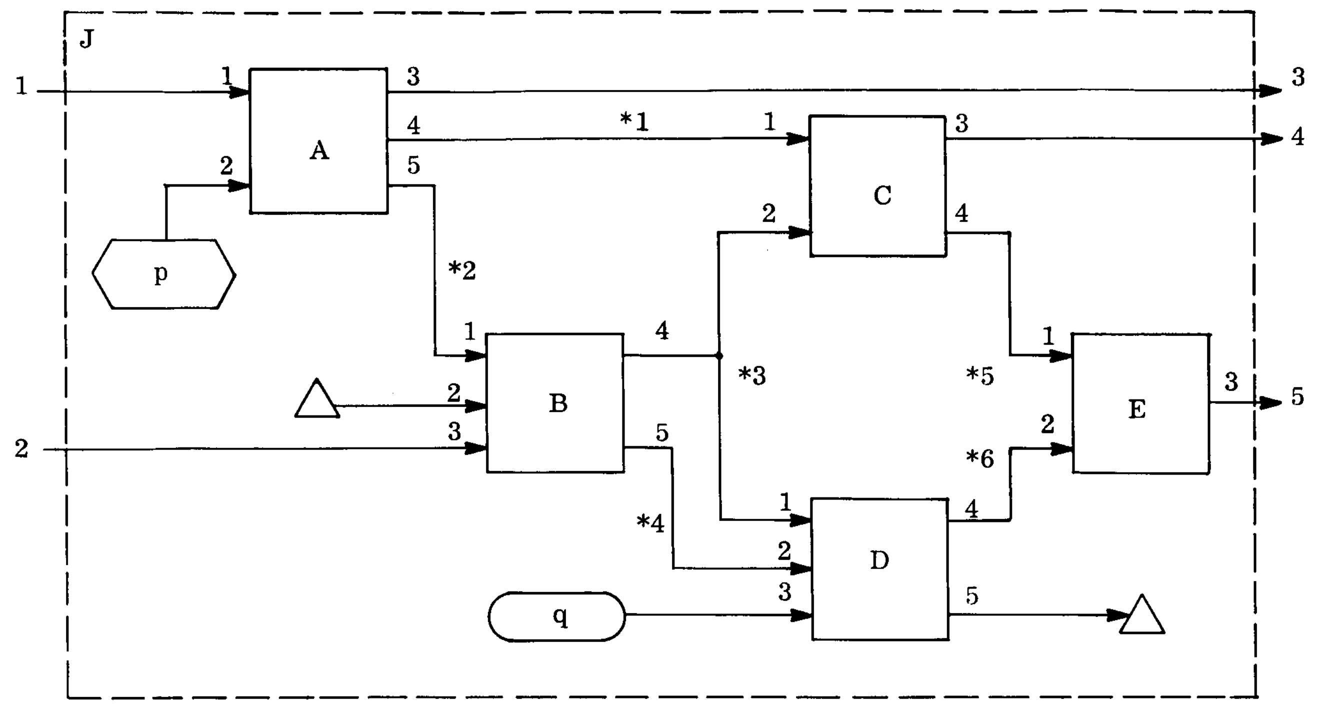
```
# J (a, b) = (c, d, e);
```

where a to e are actual values for the formal job parameters.

The special case in which all but (possibly) the last job of a structure has a single item output is illustrated in Figure 7-4. In this case the job definition need not have been made before the run request. In that case the run request would be

```
# J (#H(#F(1, 'p'), #G (, 2)), #I (#G(, 2), q)) = (3, 4, );
```

The preceding form is called a command. Formal specification of a command is given in Figure 7-5.



```
DF     J (1, 2) = (3, 4, 5) =
       A (1,  'p)       =  (3,  *1,  *2)
       B (*2,  , 2)     =  (*3,  *4)
       C (*1,  *3)      =  (4,  *5)
       D (*3,  *4,  q)  =  (*6, )
       E (*5,  *6)      =  5;
```

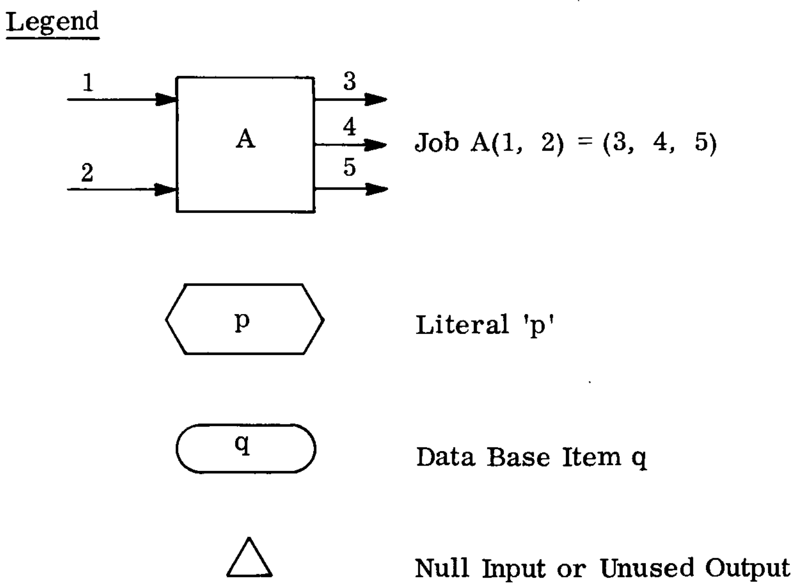
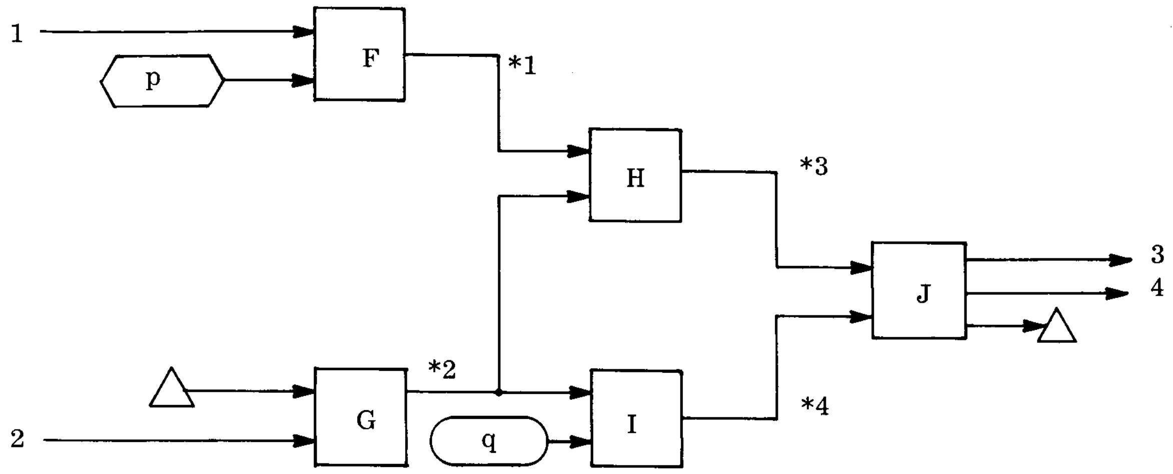


Figure 7-3. A Typical Job Structure



```
DF      K (1, 2)  =  (3, 4):
            F (1, 'p')   =  *1
            G (, 2)      =  *2
            H (*1, *2)   =  *3
            I  (*2, q)   =  *4
            J (*3, *4)   =  (3, 4, );
```

Figure 7-4. A Restricted Job Structure

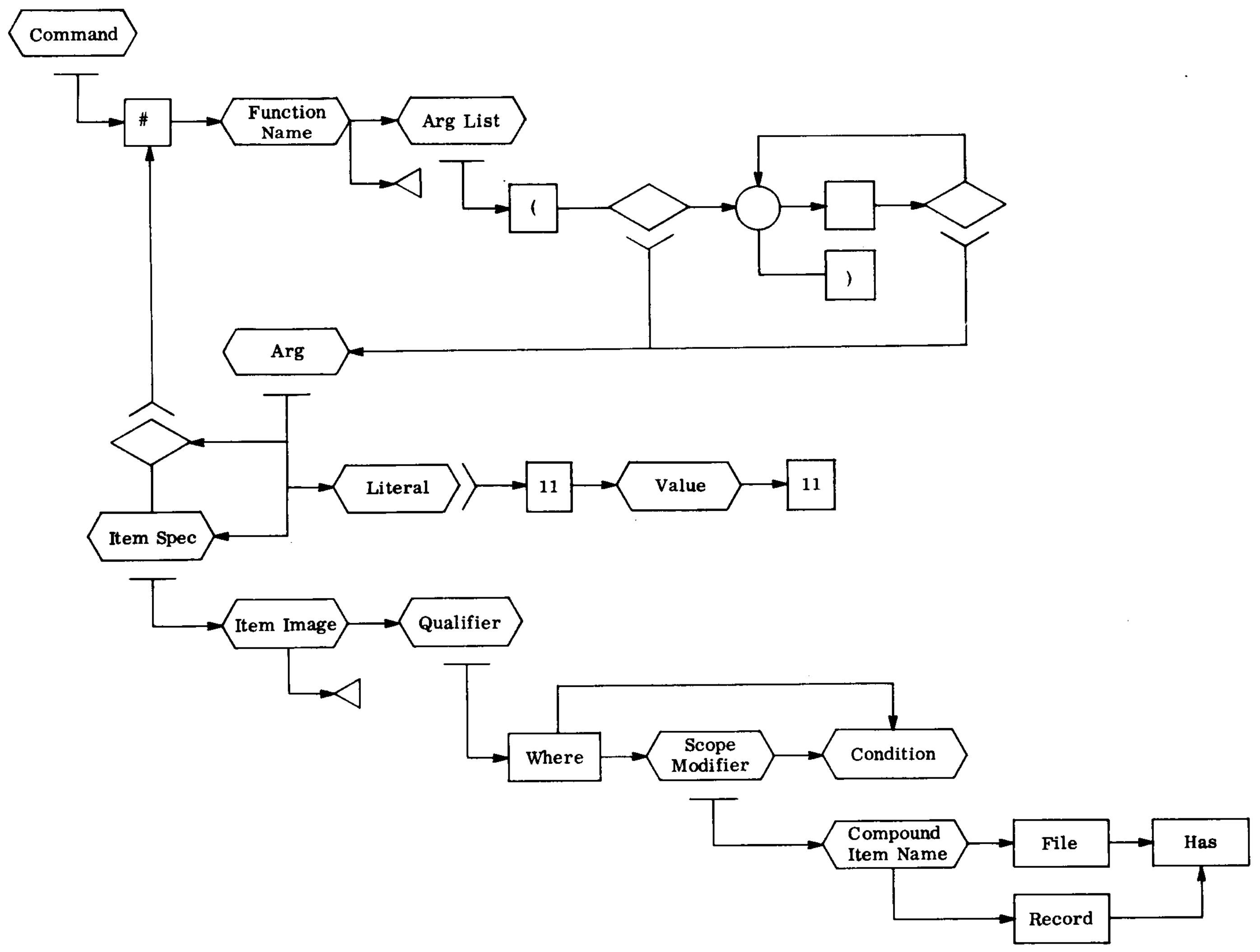


Figure 7-5. Action Graph for Command Syntax (Sheet 1 of 2)

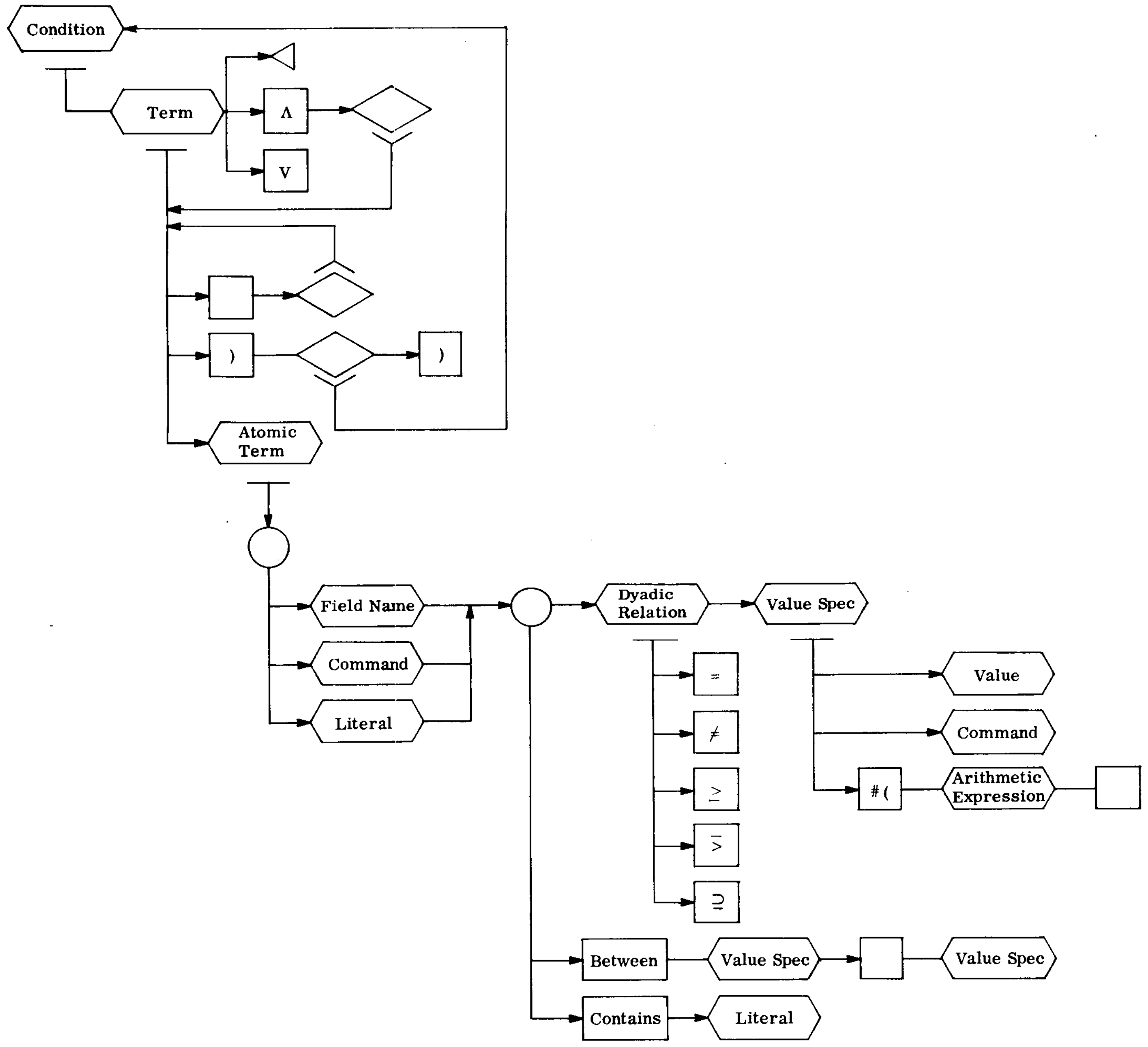


Figure 7-5. Action Graph for Command Syntax (Sheet 2 of 2)

7.3 ITEM DEFINITION LANGUAGE

The Item Definition Language is used by a console job initiator (such as the data administrator) for entering a new data item definition into the data base directory, or modifying an existing definition. It is also used by the programmer in defining the logical structure of formal items in his program, which will be the subject of Internal File System services. The formal item definitions are part of the program description. The proper entries are made in system tables by the Item Definition and Program Entry system support jobs.

The Item Definition Language must be able to conveniently define the data structures which the IFS is able to manage. These structures, which can be represented by indented tree diagrams as discussed in Section VI, consist of arrays, statements, and fields. As discussed in Paragraph 6.2, the tree diagram can be represented either in indented outline form or as a linear parenthesized string. The convention shown in Figure 6-1 can be expanded to include the Variable Structure, Coded Field, and Hierarchic Field as discussed in Paragraph 6.5. The Item Definition Language is summarized in Table 7-1. For simple structures, such as formal program parameters, the item image form will be most convenient (and probably preferred by the programmer). For large complex structures, however, the linear parenthetic form may not be easily read and may lead to errors. The indented outline form will probably be preferred by most nonprogramming users who deal with common data base structures.

TABLE 7-1. DATA DEFINITION LANGUAGES

Item Type	Indented Outline	Item Image
Statement	S; <name>	(<name> ...)
Array (File)	F <size> ; <name>	[<size> ; <name> (...)
Array Element (Record)	R; <name>	(<name> ...)
Variable Structure	V; <name>	(V; <name>)
Link Statement	L; <name> <target name>	(L; <name> ...) ⟨target name⟩
Binary Field	B <size> ; <name>	/B <size> ; <name>
Octal Field	O <size> ; <name>	/O <size> ; <name>
Integer Field	I <size> ; <name>	/I <size> ; <name>
Decimal Field	D <size> ; <name>	/D <size> ; <name>
Exponential Field	E <size> ; <name>	/E <size> ; <name>
Alphanumeric Field	A <size> ; <name>	/A <size> ; <name>
Text Field	T <size> ; <name>	/T <size> ; <name>
Coded Field	C <size> ; <name> {value list}	/C <size> ; <name> {value list}
Hierarchy Field	H <levels> , <values>; <name> {<value structure>}	/H <levels> , <values>; <name> {value structure}

SECTION VIII. CONCLUSIONS

This report has examined the data processing needs of NASA post-Apollo missions and related those needs to features and capabilities required in system software. It has concentrated on the job and data management aspects of the software and has identified and developed three areas of concern:

(1) The Job Management System and its interface with the interactive user through a goal-oriented command language

(2) The Internal File System and its interface with the procedural language programmer with implicit item services

(3) The External File System and its interface with the machine-oriented language programmer with implicit cell services.

The orientation (and possible bias) of this work has been toward the concept that a dominant data processing problem of future space missions will be the management of a large centralized data base concerned with many aspects of the space mission and shared, perhaps for different purposes, by many different users. This view results from the likelihood that storage equipment permitting rapid access (milliseconds) to a large amount of data (10^9 bits) will be available in the next decade, and that techniques for effectively time-sharing a large computing complex among independent users are rapidly being developed.

Examination of a number of current and proposed systems disclosed that many of the desirable features of such a system could be found in embryonic form in several time-sharing computers, monitors, file systems, and generalized data management systems. It was also clear, however, that they had not been brought together to form a coherent user-oriented system in any one instance. Starting with this basis, a Coherent System concept has been developed. Using techniques within reach of today's technology, this system provides a new measure of data management capability, program effectiveness, and programming ease.

The Job Management System specified gives the goal-oriented user the ability to combine existing programs and jobs, prebind any of their parameters, and use that structure as a single operator, executable when the appropriate command is given, control action taken, or data and temporal prerequisites are satisfied. Any job parameters may be specified as data base values which meet a given logical condition. In effect, this embeds a general query capability within the command language.

The Internal File System allows the user to define, build, and maintain logical data item structures of wide generality in an on-going system data pool. Access to this data can be effectively controlled by means of user lists, security classification, and "need-to-know" conditions. Item structures exist independently of the programs which use them and may be modified in structure and content without making the programs which use them obsolete. Implicit item services allow the programmer to reference items in formal structures as if the actual items to which they are bound are in the immediate processing environment of the program at execute time. That is, one can write procedural language programs which declare formal items but are not concerned with the binding process, data location, or the logistics of data movement.

The External File System maps the items of the system data pool into a large virtual address space and thence to tracks in the real multilevel storage system. It allows machine-oriented programmers to write programs in a symbolic assembly language which treats the program and any formal files which it references as symbolically addressable trains of cells in independent address spaces. These may be bound to permanent data base trains or, in the case of sequentially accessed terminal trains, to input/output devices.

The result of this work is a functional description of a prototype Coherent System. The three major system elements mentioned previously are described, along with some System Support Functions for defining new jobs and data structures. There are several areas (notably, System Support Functions for indexing and conditional item retrieval, terminal train services, and implicit item services) which deserve further development. Since the topics described are, in almost all aspects, innovative and the technology itself is young, it is strongly recommended that a system design be detailed and that at least a partial experimental capability be implemented within a current hardware/software environment.

TABLE OF CONTENTS

Paragraph	Title	Page

APPENDIX A. FILE SYSTEM DESCRIPTIONS

TIME-SHARED DATA MANAGEMENT SYSTEM (TDMS)

GENERALIZED INFORMATION SYSTEM (GIS)

INFORMATION CONTROL SYSTEM (ICS)

DATA MANAGER – I (DM-1)

TIME-SHARING SYSTEM/360 (TSS/360)

BERKELEY TSS (BTSS)

MULTICS

INTEGRATED INFORMATION PROCESSING SYSTEM (INTIPS)

APPENDIX A. FILE SYSTEM DESCRIPTIONS

AUERBACH

TIME–SHARED DATA MANAGEMENT SYSTEM (TDMS)

1. INTRODUCTION

1.1 General Background and Implementation Status

TDMS (Time-Shared Data Management System) is a user-oriented internal file system which is being developed by SDC. Designed to operate under the control of the SDC time-sharing executive on the IBM System/360, Models 50 and 65, it consists of a set of file command interpreters that permit a non-programmer to define, store, process, and output data according to his requirements. The system allows the user to define and process a broad range of data structures. It does not interface with programming languages or other systems.

TDMS is an outgrowth of TSS-LUCID, an experimental system that has been operational at SDC on the Q-32 computer since 1964. The first model of TDMS is expected to be operational in 1968.

1.2 Objectives

The overall design goals of TDMS, as stated by Bleier, [1]* are as follows:

(1) a reasonably complete set of general-purpose data management capabilities

(2) a language that is oriented toward the non-programming user

(3) a capability to define and handle hierarchically structured data

(4) system tables and directories that will provide rapid response to queries

(5) an ability to obtain rapid solutions to a wide variety of users' problems.

1.3 Definitions

The concepts of TDMS (and all systems to be evaluated in this study) will be discussed in terms of a common language. Although use of a common language will facilitate comparison of these systems, one danger associated with this approach is that the concepts of the system designers may not be conveyed with complete accuracy in the standard language used. When this danger is greater than usual, an effort will be made to point out this difficulty; however, all responsibility for warped connotation or inaccurate interpretation will, of course, be the responsibility of the writer.

In the definitions which follow, the TDMS term is underlined.

- A <u>data base</u> is a file which is not embedded in any higher user-defined structure.
- A <u>logical entry</u> is a record of the data base.
- A <u>data base description</u> is a file description.
- A <u>component</u> is an item.
- An <u>element</u> is a field.

*Superscript numbers indicate references in the bibliography.

- A repeating group is a record of an embedded file. (The fact that there is no term for referring to the embedded file as an aggregate leads to conceptual and logical difficulties, which will be discussed.)
- A set is a statement with numbered occurrences but no generic name in DMS.

A file in TDMS is made up of records which are composed of a single set and some number of embedded files. Since each embedded file, in turn, is composed of records with a single set plus possibly other embedded files, a record at the highest (data base) level can be viewed as a sequence of sets, the first at level 0 and the following ones at greater depths.

2. SYSTEM DESCRIPTION

2.1 Operations

TDMS users, who are management oriented, can define data items and their relationships (within constraints that will be discussed) and perform operations with the system in order to load, update, query, and display data. The TDMS system components, together with the way they relate to system data, are shown in the boxes of Figure 1. The functions of these components are given in the following list:

(1) DEFINE. Allows the user to create a new file description or alter an existing one with inputs prestored or provided interactively, on-line.

(2) LOAD. Accepts all types of numeric and non-numeric data, either batched or interactively, from a console; performs legality checks; permits error correction on-line; and includes the capability to convert pre-existing data values to the required TDMS format.

(3) QUERY. Allows the user to request single values, multiple values, or entire entries, printed on-line in a standard tabular format; and provides full arithmetic capabilities, plus operators such as sum, minimum, maximum, count, average, and standard deviation.

(4) UPDATE. Allows changing of values and adding or deleting either single values or entire entries.

(5) DISPLAY. Allows the user to define, generate, manipulate, save, and recall a variety of display formats.

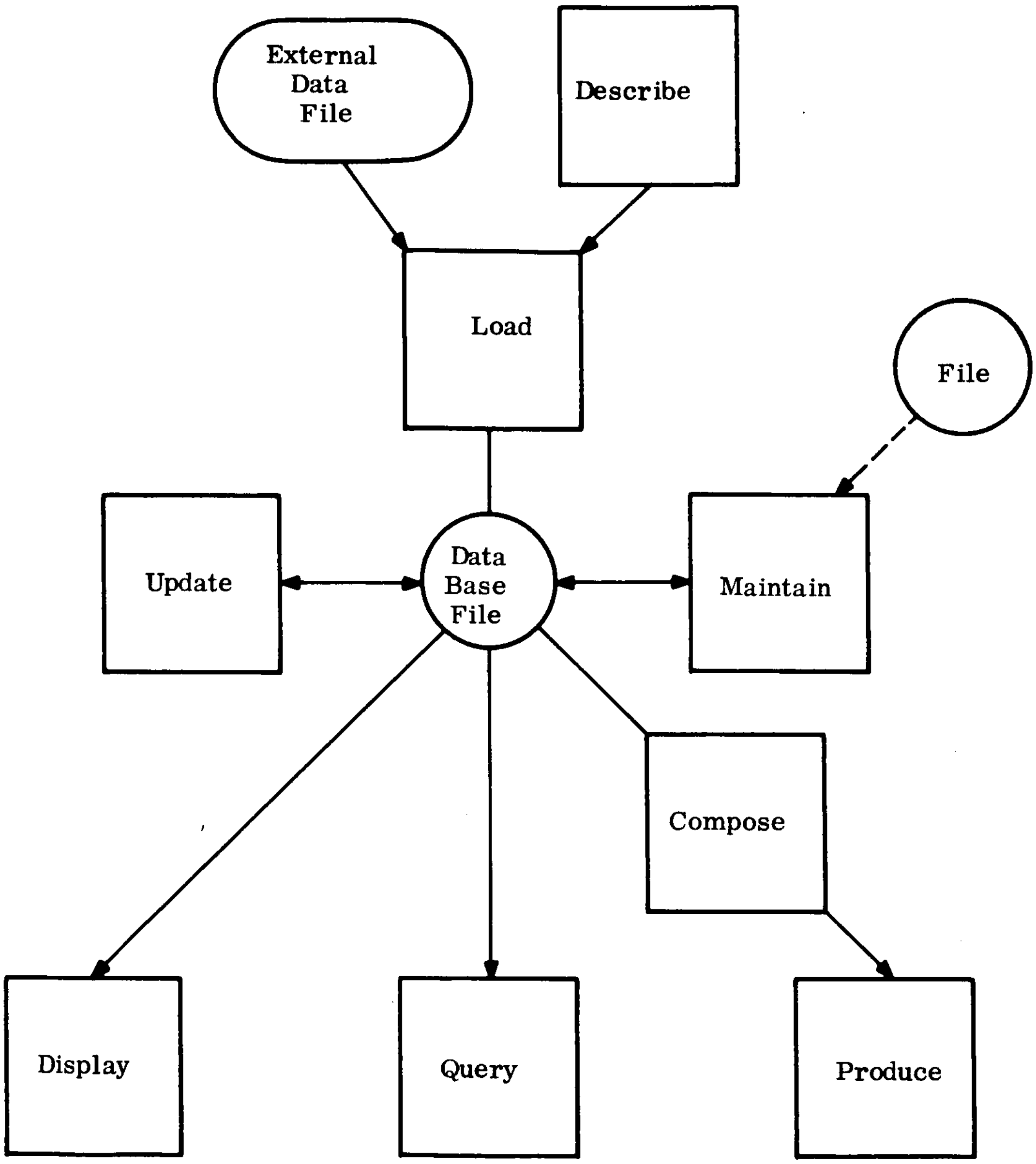


Figure 1. TDMS System Components

(6) COMPOSE/PRODUCE. Allows the user to specify report format and content in a user-oriented language; provides arithmetic, ordering, formatting, and tutorial capabilities; and produces reports on the interactive console or line printer.

(7) MAINTAIN. Provides for merging, subsetting, extracting, ordering, restructuring, and updating files; and accepts one or two different files, an on-line description of the desired output data, rules for selection of data, and the transformations that are required.

2.2 Structures

Two types of data structures are pertinent when describing TDMS (or any other internal file system). The first is the item structure, the user-determined logical structure that defines an item in some system-determined descriptive language. The second is the system-determined data structure for mapping the user item definitions into system definition tables, and the data into the data base and directories.

2.2.1 User-Determined Item Structure

2.2.1.1 General Structural Scheme and Constraints. The highest level item which a TDMS user may define is a file; in TDMS the file is called the user's data base. The structure required for each file is shown in Figure 2. The records of the TDMS data base are called logical entries and consist of a single statement, called a set, which must contain all the fields (called elements) at this level of the structure. The statement is not named in TDMS but is referred to by referencing the ordinal item number (component ID) of the record/file to which it belongs. Following the statement in the TDMS record may be some fixed number of embedded files called repeating groups. The structure of each repeating group follows the pattern of a data base file; that is, it is made up of records having a single statement (set) and some number of embedded files (repeating groups), recursively. The lowest level record in any structure does not contain embedded files.

2.2.1.2 User Data Definition Language. The user data definition language is based on a list of items (called components in TDMS) in which the record and statement level are omitted. The list may be written in an indented format if desired, but this is not necessary since the parent file of each item must be identified.

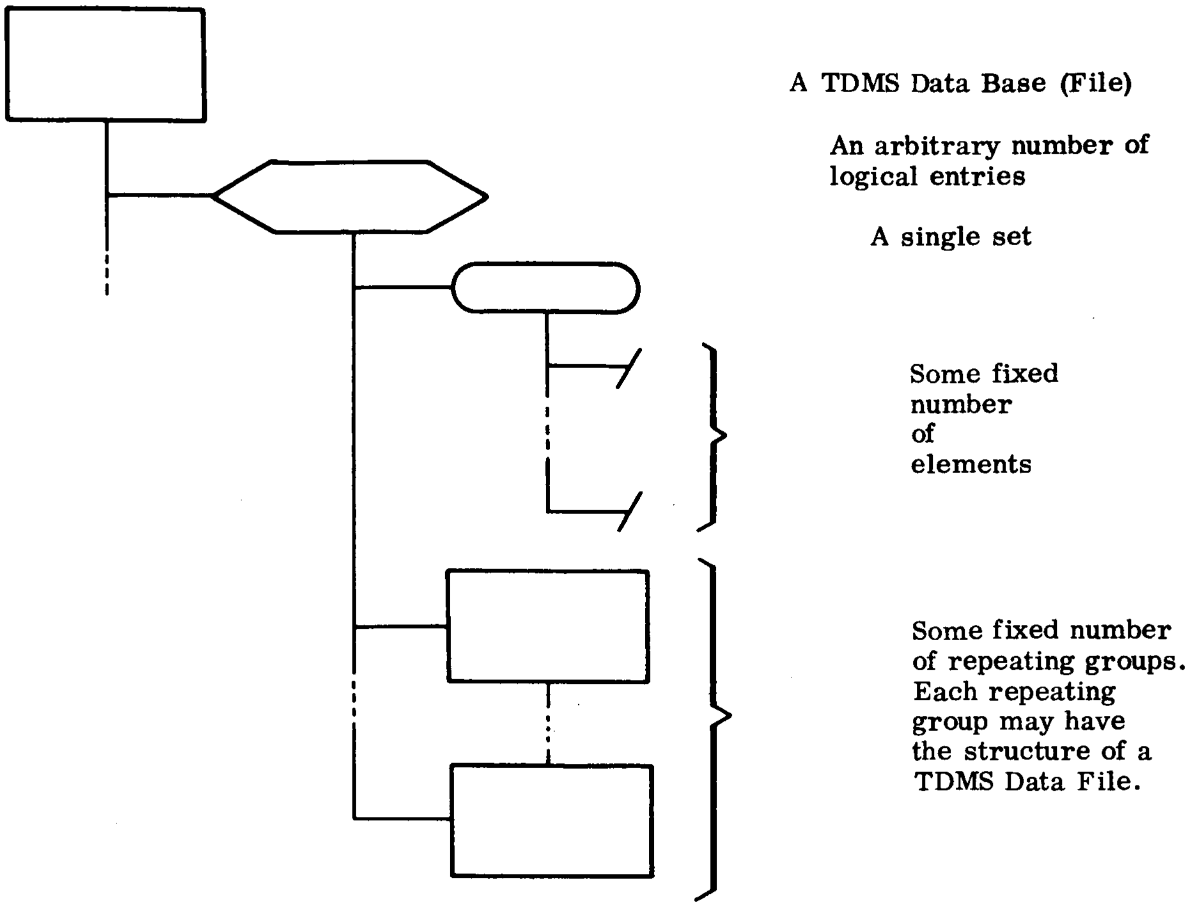

A TDMS Data Base (File)
An arbitrary number of logical entries
A single set
Some fixed number of elements
Some fixed number of repeating groups. Each repeating group may have the structure of a TDMS Data File.

Figure 2. A TDMS Data Base Structure

In order to illustrate best the characteristics of the data definition (and other TDMS operations), a typical example will be given and developed.*

The example chosen is a personnel file. Each record contains the employee's name, and data about his jobs and children. A structural diagram of this data is given in Figure 3. The mode representing the statement (set) in each record has been omitted since it is not given in the data definition.

*This example is used by SDC in many of its reports and system documentation, e.g., RE Bleier: Treating Hierarchical Data Structures in the SDC Data Management System (TDMS), Proc ACM Conference, 1967.

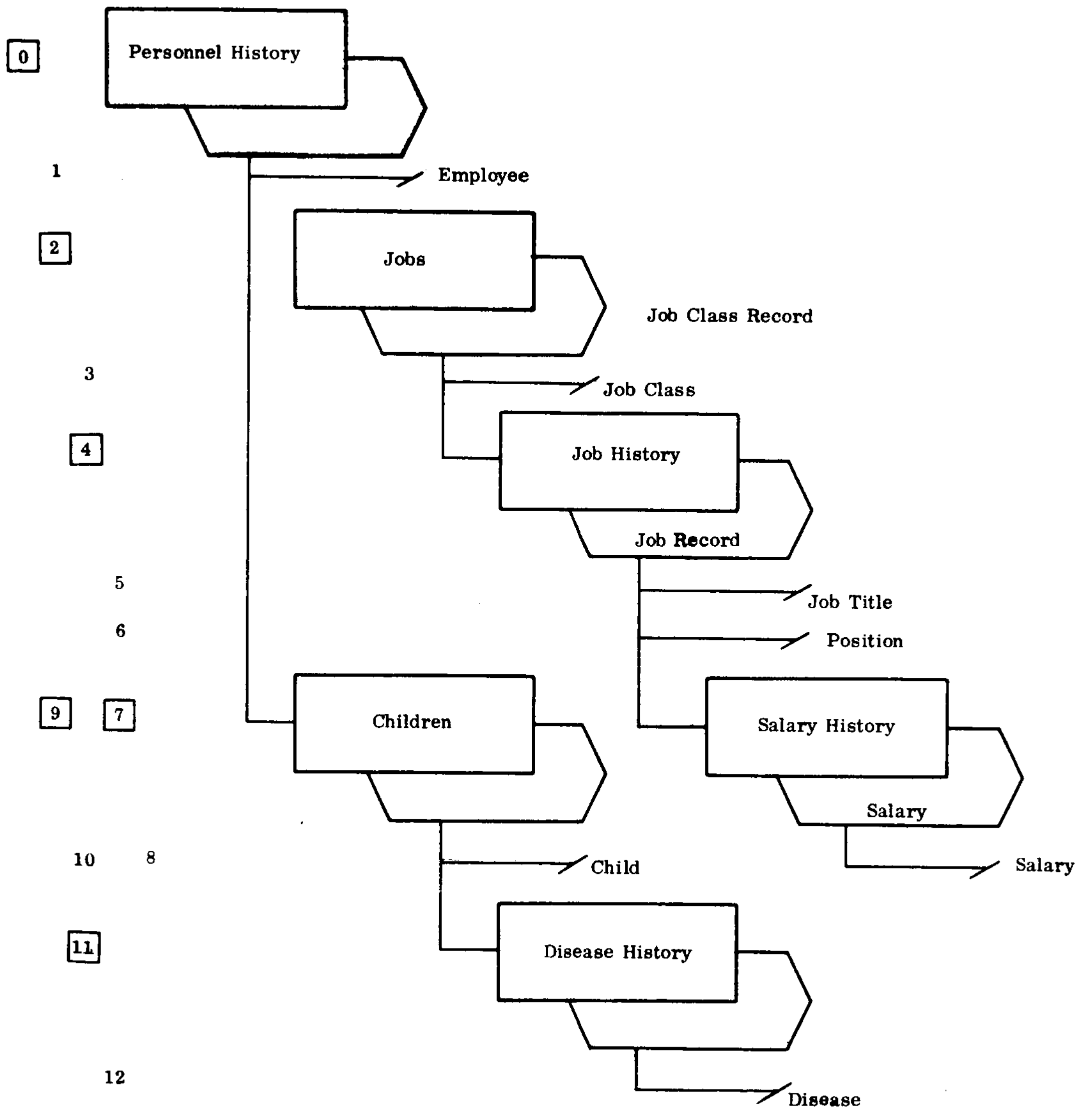


Figure 3. Personnel History File

The data structure shown in Figure 3 is defined using the TDMS DEFINE program. The user enters a description (typically on-line), as shown in Figure 4. This description defines the components and interrelations of components in the data base. Note that there is only one EMPLOYEE in each entry, but he may have had one or more jobs and one or more children.

```
                DATA BASE NAME IS: PERSONNEL HISTORY

                      TERMINATOR IS: END

1 EMPLOYEE (NAME)

2 JOBS (REPEATING GROUP)

  3 JOB CLASS (NAME IN JOBS) VALUES ARE EXEMPT, NON EXEMPT

  4 JOB HISTORY (RG IN 2)

    5 JOB TITLE (NAME IN 4)

    6 POSITION (NAME IN 4)

    7 SALARY HISTORY (RG IN 4)

      8 SALARY (NUMBER IN 7) VALUES ARE 400 ... 2000

9 CHILDREN (RG)

  10 CHILD (NAME IN 9)

  11 DISEASE HISTORY (RG IN 9)

    12 DISEASE (NAME IN 11)

TERM
```

Figure 4. TDMS Define Example

Indentation is used in this example to bring out the TDMS concept of level. Indentation is not syntactic in the language as defined, but is used in the example as a logical signal for transition from one level to another. (Users may indent or not as they see fit.) In the example, components 1, 2, and 9 are said to be at level zero; the order in which their values are input is of no consequence, nor do their values depend upon the existence of other data in the logical entry. Components 10 and 11 at level 1, and 12 at level 2, are in the repeating group "CHILDREN"; the order of input to the data base for these elements is very important. For example, in the logical entry which has data about three children named Mary, Jim, and Bill, the name Bill must not be associated with Mary's diseases.

Specifications of the hierarchical nature of these data is accomplished by naming the beginning of a hierarchical structure (repeating group) and then specifying any components that belong to that group. It is stated, for example, that component 4 (JOB HISTORY) is a repeating group in component 2 (JOBS) and component 5 (JOB TITLE) is a name in component 4 (JOB HISTORY).

2.2.1.3 Loading the Data Base. Examples of data configured for input to the TDMS LOAD program are shown in Figure 5. It is assumed that each line is a card image or a line on the teletype. Again, indentation is not syntactic; it is merely logical.

The order of component values must be maintained, but the user may pack as much information on a line as he wants. Data may be omitted. For example, in the first occurrence of component 4 (JOB HISTORY), if JOB TITLE were missing, the line would read:

```
        4) , PROGRAMMER
```

The component numbers chosen at define time identify the data fields that follow. The next occurrence of the form "space, number, close parenthesis, space" signifies the end of the previous field and the beginning of the next field. Within repeating groups, the input fields may be sequenced and separated by commas, or given field numbers.

2.2.2 System-Determined Data Structures. TDMS processes the file definition to produce entries in a set of system tables which define the item names and structure of the file. When the file is loaded, entries with field values will be established in

```
1) JONES A. V.
    2) NON EXEMPT
        4) PROGRAMMER TRAINEE, PROGRAMMER
            7) 450 7) 500
        4) PROGRAMMER, PROJECT HEAD
            7) 550 7) 600 7) 650
    2) EXEMPT
        4) PROGRAMMER ANALYST, PROJECT HEAD
            7) 650 7) 700 7) 750
        4) PROGRAMMER ANALYST SR, SECTION HEAD
            7) 800 7) 850 7) 900
        4) COMPUTER SYSTEMS SPEC, GROUP HEAD
            7) 900 7) 950 7) 1000
9) MARY
    11) MUMPS 11) MEASLES
9) BILL
    11) WHOOPING COUGH 11) CHICKEN POX
9) JIM
    11) MUMPS 11) MEASLES 11) CHICKEN POX
END
```

Figure 5. Example of Numbered and Sequenced Fields Data Set

a data table. Index type data which allows rapid response to queries and retrieval of the data will also be established. The basic TDMS data retrieval strategy depends upon a complete indexing of all data values in the file so that each data value is cross-indexed to all statements (set occurrences) which contain that value. This conversion goes through several stages (value to set number to location) that will be described.

The logical structure of the internal tables of TDMS is shown in Figure 6. The TDMS data retrieval strategy follows from the structure of these tables and their relationships.

The Item Definition table has an entry for each component (item) in component number order. The data stored in this table for each item gives code type (for fields), Repeating Group Identifier (RGID) (for embedded items), level number, and value of data in the data file. For each field there is a link to a table of legality data; an index (concordance) of each value in the field lists the set number (based on ordering all statements in a tree-scan order) for all sets in which the values occur. This set number is used as a link to two tables: (1) the Data File (Data Base) itself, which gives values for all fields in the set, and (2) the table called CFIND, which gives the RGID and relative "UP" and "DOWN" links to related sets.

The definition and interpretation of UP and DOWN links in TDMS are given in Figure 7. This interpretation is based on the report by Bleier[1] and is somewhat at variance with that given by Ziehe[7] and earlier TDMS documentation.

The UP and DOWN pointers in CFIND are relative pointers from each set entry to a related (not necessarily distinct) set. The DOWN pointer from the highest level set in each record points to the corresponding set in the next record. The DOWN pointer in each embedded set, however, points to the immediate set of which it is part (immediate parent).

The UP pointer from the highest level set in each record points to itself. The UP pointer from each embedded set points to the next highest numbered set at its own level. The UP pointer from the highest numbered set at each level (below the top) points to itself.

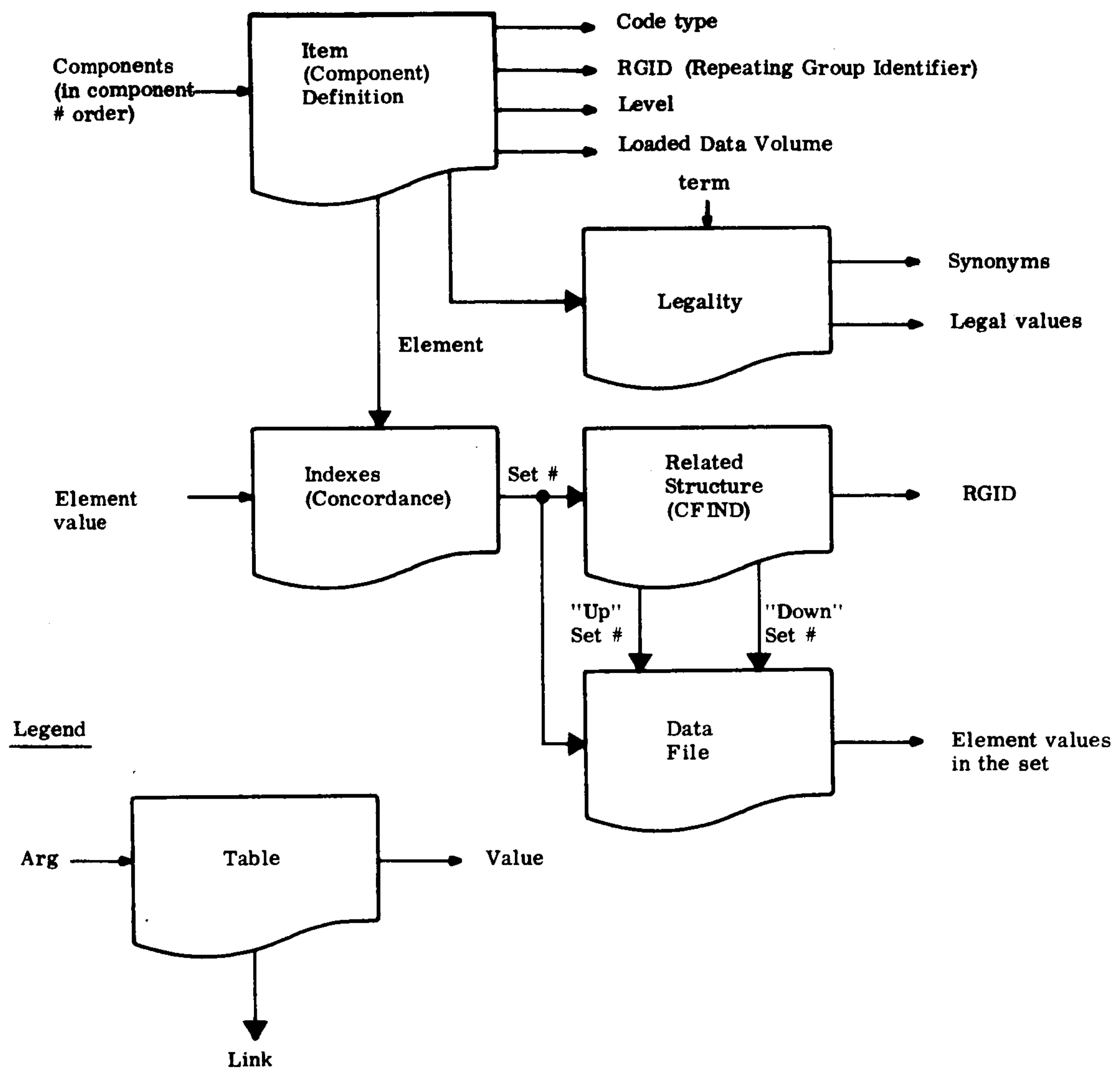


Figure 6. TDMS Table Structure

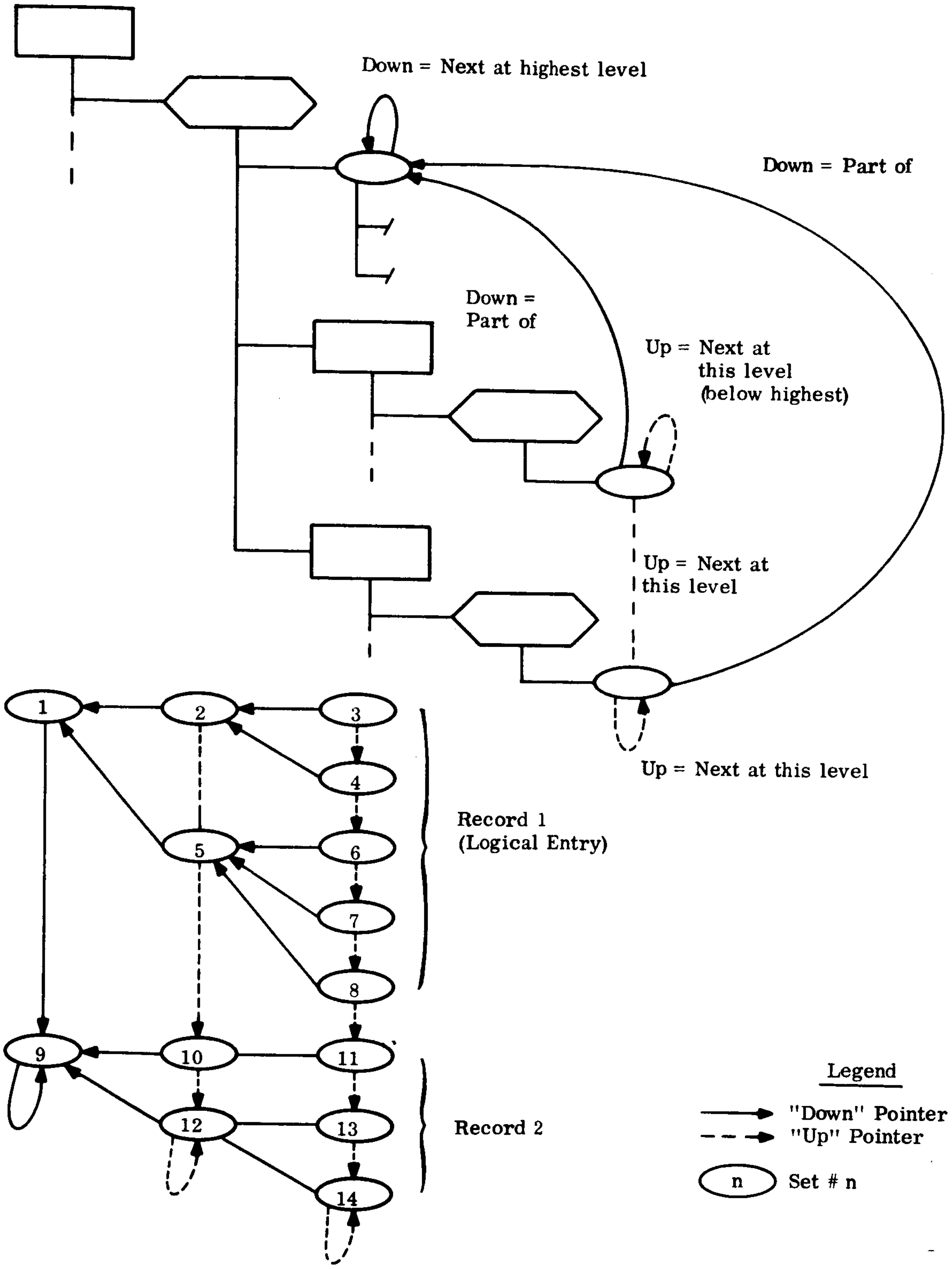


Figure 7. Related Structure Pointers in CFIND Table of TDMS

2.3 Query Language[1]

TDMS has a well-developed query language which, except for a logical error to be discussed in Paragraph 3.3, provides for retrieval of arbitrary data subject to a general boolean logical condition. Using the data definition and data file example given in Paragraph 2.1, the following queries can be posed in TDMS:

(1) Print Job Title where Position = Project Head

(2) Print Job Title where Job Class has Position = Project Head.

The first query is in the form: PRINT field WHERE condition; the second is in the form: PRINT field WHERE scope modifier HAS condition.

The interpretation of these queries is:

(1) Print the value of job title in every job history record in which position is Project Head.

(2) Print the job title in every job history record of the job history file if the job history file contains a position of Project Head in any record.

Thus, the where clause establishes a condition and implies the scope of that condition. If there is no scope modifier in the form "<field name> HAS" after the "WHERE," then the condition must be satisfied individually in each record at the level established by the items named in the condition which follows "WHERE." If "HAS" is used, then the condition is considered satisfied if any record at the level below the field named in the HAS-clause satisfies the logical condition in the WHERE-clause. Other permissible queries are as follows:

(3) Print salary where position has salary eq. 701...799

(4) Print salary where job title has every salary gr 600.

In Item (3) "HAS" carries the meaning "has any." Any job with a salary within the given range will qualify.

3. EVALUATION

3.1 Structure

TDMS appears to be a system that a non-programmer can easily use to define, update, and query a structured data file. A language is provided to evoke system-defined actions which define new data elements, load an initial file, maintain it, respond to queries, and produce reports. It appears, however, that there is no easy way for a programmer to interface with TDMS and provide for non-standard actions in these areas; that is, there is no discussion of the programmer as a user of the system, and no direct data access services offered to the programmer as a user. As a consequence of this lack of interaction of TDMS with a programming system, it appears that batch-oriented or non-standard data processing functions will have to be performed with a specially prepared version of the file, using the compose/produce facility. If new data arises as a result of the "foreign" functions, it would be entered through the Maintain facility.

These constraints lead to a system which is rather closed and rigid in structure and, therefore, slow to adapt to changing or new requirements.

3.2 Search Strategy

The data base of TDMS is highly indexed; normally, there is a complete concordance of all fields. This means that the file is indexed by essentially each field value which occurs in the file, a strategy that minimizes search time in responding to queries. The results of making retrieval considerations dominant in the design are that a large amount of space is devoted to the index tables and a large amount of time is spent updating these tables when records are added or modified. These results are a manifestation of the usual space-time tradeoffs which are normally available to the programmer. If fast response to random queries is the most critical requirement for the query system, then TDMS indeed optimizes the right element. However, in a multi-file data base there is usually a variation of response times which can be tolerated across the files. Some files have faster response requirements than others; and even within a single file, a predictable subset of properties will often enter into the conditional statement of queries while other properties are rarely qualified. It appears that a system which allows a tradeoff of these factors at the time a file is defined or loaded will be more efficient in the overall utilization of space and time. If the selection of which fields and field values are indexed can be made dynamically during the life of the system, then this tradeoff is adaptive to changing usage patterns or to experience with the use of the system.

3.3 Query Language

The TDMS query language has one property which is extremely desirable in data management systems. It allows the interrogator to make effective use of the information he has concerning the structure of the data base which he is querying. The ability to qualify the scope of the condition through use of the "item name HAS" clause would be rendered more effective, however, if the item name were the name of an embedded file or record rather than the name of a field at the level of the embedded file within which the condition is directed.

For example (referring to the structure defined in Figures 3 and 4), the TDMS query: "Print employee, job history where job class has job title equal programmer analyst and position equal section head," is actually ambiguous. In a sense "job class" does not "have" anything since it is a terminal item (field) in the structure. As it is, even the intent is ambiguous because it is not clear whether the condition (job title and position) must co-occur in a single job history record or whether they each may occur individually anywhere in the job history file of a single employee. A more effective syntax for the HAS-clause would be:

$\langle$file name$\rangle$ [RECORD | FILE] HAS

For example, with the new syntax the above query would read

> "Print employee, job history file where job history record has job title equal programmer and position equal section head,"

if the intent were to print the entire job history file when any one record contained the specified job title and position. (The word "record" would be replace by "file" if the other interpretation were desired.)

The inability of TDMS to take this approach seems to stem from a lack of recognition that file and record are two distinct entities and levels in the structure. In TDMS the phrase "repeated group" appears to be used indiscriminately to refer to either. Thus, the characteristic of the query language discussed above appears to stem from the limitations of the data definition language. Another feature which may improve the data definition language is the ability to use the indentations of the outline format to signal levels in the structure instead of the requirement that the parent repeated group be mentioned in each subsumed item.

4. BIBLIOGRAPHY

(1) Bleier, Robert E. Treating Hierarchical Data Structures in the SDC Time-Shared Data Management System (TDMS). ACM 20th Anniversary Conference (August, 1967).

(2) Franks, E. W. "A Data Management System for Time-Shared File Processing Using a Cross-Index File and Self-Defining Entries". AFIPS Conference Proceedings, 28: 79-86 (1966).

(3) Grant, E. E., and DeSimone, P. A. The Language Specifications for the Define Operation of TDMS. SDC Document TM-3370/003/00, (April, 1967).

(4) Raucher, V. L., and Schwimmer, H. S. The Basic Language Specifications for TDMS. SDC Document TM-3370/001/00 (April, 1967).

(5) Vorhaus, A. H., and Wills, R. D. The Time-Shared Data Management System: A New Approach to Data Management. SDC Document SP-2747 (February, 1967).

(6) Williams, W. D., and Bartram, P. R. "COMPOSE/PRODUCE: A User-Oriented Report Generator Capability within the SDC Time-Shared Data Management System. "AFIPS Conference Proceedings.

(7) Ziehe, T. W. "Data Management; A Comparison of System Features." TRACOR, Inc. (August, 1967).

PREFACE TO GIS

The discussion of GIS is based on references (listed in the bibliography) released in 1965 and 1966; although the report itself has been largely taken from Data Management System Survey, AUERBACH Report 1280-TR-1. Although IBM withdrew the specifications of GIS in May, 1967, and announced that major revision would be reissued in April, 1968, the design concepts reflected in the documents upon which this report is based represent important contributions to data management technology. It is from this point of view that the report on GIS is included.

GENERALIZED INFORMATION SYSTEM (GIS)

1. OBJECTIVES

GIS (Generalized Information System) is a comprehensive Data Management System that is being developed by IBM Federal Systems Division. It is designed to operate within Operating System/360 to support a wide variety of data processing applications by providing facilities for defining, maintaining, and processing data files and unformatted text.

GIS provides a user-oriented interface which permits the description of logical data relationships and processing tasks without concern for the processing devices, physical formats, and device operations involved in manipulating the data.

A number of variable parameter statements, which together constitute the GIS language, are available to allow a user to establish and modify the more static control tables, to enter task specifications, and to control task execution. The parameter content of the control tables and task specification are combined at execution time to direct and control the flow of program operations. A wide range of file structures and organizations can be established, maintained, and used.

2. IMPLEMENTATION AND STATUS

GIS is a set of programs for System/360 that are designed to perform data file establishment, maintenance, retrieval, and presentation operations that are common to many data processing applications. GIS is expected to function on a range of System/360 configurations under control of OS/360. It may be operated in either a job sequential mode or a teleprocessing (multiprogramming) mode with remote terminals. The minimum configuration for the job sequential mode is Model 30F (65,536 bytes of core storage) and, for the teleprocessing mode, a Model 40G (131,072 bytes of core storage). In addition, GIS programs require approximately 500,000 bytes of direct access secondary storage and an interval timer.

GIS programs are run as jobs under the OS/360. The implementation language for the GIS translator and system programs is Assembler Language. Schedule of availability will be issued in April 1968.

Although quite different in concept and design from its predecessors, GIS is the latest in an evolutionary line of IBM file control systems, represented by TUFF, TUFF/TUG, FFS, and IPS, developed for military operations control centers by IBM Federal Systems Division. GIS will be supported by an IBM Applications Group (Type II support).

3. SYSTEM DESCRIPTION

3.1 User Languages

3.1.1 General. Although GIS runs as a job within OS/360, it has many of the features of an operating system itself. A GIS job is made up of a series of JOB control cards followed by a task specification. Within each task, GIS will exercise control over a number of GIS and non-GIS (user supplied) programs.

OS/360 provides the overall control at a given installation concerned with receiving, queueing, and executing jobs. Core memory allocation, channel time allocation, program loading, and multiprogramming functions are all handled by OS/360. The GIS run is initiated by OS/360 in response to the GIS JOB cards. GIS itself has a control mechanism which interprets the task specification.

The task specification contains task control information, the names of generalized GIS service functions (e.g., CALL and QUERY), the parameters needed to perform the desired action, and user-supplied (non-GIS) programs.

There are two basic modes of GIS operation: JOB-oriented and TP-oriented (telecommunication processing-orientation). The TP-oriented mode takes advantage of the capability of OS/360 to handle multiple messages from the telecommunication devices attached to the system. In the TP-oriented mode, multiprogramming controls in OS/360 make it unnecessary to initiate a new job for each new task specification. In the JOB-oriented mode of operation, the user may enter the job from a variety of devices, including telecommunication devices, but, as stated above, must initiate a new job for each task specification. The job is then processed in accordance with the priority assigned by the originator.

3.1.2 Job Specification Language. GIS provides a language for specifying the tasks that compose a job, their parameters, and the condition for their execution. Task specifications can be saved by GIS and executed when called by the user.

The format for the request to save a task specification is given below:

SAVE INPUT STATEMENT SET <task name> <task specification>

A task specification in the GIS Language is a series of commands and conditional statements to execute GIS and user-supplied (non-GIS) programs. A task specification may contain open (formal) parameters whose values must be supplied by the task specification which names them, or by the user at request time.

A GIS program named CALL will bind the formal parameters of a program. Its parameters are a task name and its parameter value list:

CALL <task name> <parameter list>

The parameter list must supply values for all the formal parameters of the task named. Formal parameters in the task specification are named

1, 2, ...

indicating the sequence in which their values must be supplied.

Operation of this program or any other GIS program in a job requires a command of the form:

```
OPERATE GIS PG <program name>
PARAMETER EQ <parameter list>
```

For example, the statement:

```
OPERATE GIS PG CALL PAYROLL PARAMETER EQ AUG 7
```

will cause the GIS program "CALL" to bind the user supplied program "PAYROLL" with its parameter "DATE" = AUG 7

For user (non-GIS) programs in which no parameter binding is necessary, the command is of the form

```
OPERATE <program name>
```

3.1.3 Query Language. As part of the job specification and request language discussed in the preceding paragraph, GIS has a condition-defining capability which permits a logical (Boolean) condition that must be satisfied by each record before it is eligible for further processing. Conditional statements can be inserted in any task specification as a condition for further processing, such as retrieval, data reduction, updating, reporting, etc. When combined with a retrieval, reporting, or tabulating function, GIS accomplishes what is called a QUERY. In a query, certain functional and data reduction operations to be carried out, as well as the format in which the data is to be presented, can be specified.

A QUERY task specification begins with the word QUERY, followed by a file name, a logical condition to be satisfied by records of the file, and a statement specifying the data to be extracted and the processing to be carried out on eligible records. Because of the file name requirement, a QUERY is limited to a single, identifiable file.

3.1.3.1 Conditional Statements. The conditional statements involve terms which take the following form:

```
<name of field> <comparison to be made> <basis of comparison>
```

These terms may be combined with AND, OR, and AND NOT operators. Parentheses may be used as required. For example,

IF AGE EQ 21 AND HAIR EQ BLOND OR BROWN

< name of field > may be any valid field name in the file being addressed.

<basis of comparison> may be any field value.

<comparison to be made > may include any of the following operators:

{= , > , < , ≠ , ≥ , ≤ , BETWEEN, SCAN, MASK, CHANGE }

The SCAN operator tests the field for the particular pattern of characters specified. The MASK operator permits "don't-care" positions in the scan. The CHANGE operator does not require a basis of comparison. It will indicate "true" for any value which is different from the value of the same field in the previous record.

The truth value of each term is evaluated with respect to the appropriate data elements. These truth values are then combined according to the condition (IF - statement), following the usual rules of Boolean algebra with respect to the AND, OR, and AND NOT connectives.

Those data items for which the condition is true are subject to the processing given in the remainder of the QUERY task specification, as indicated below.

3.1.3.2 Summary Operations. A facility exists to create summaries subsequent to retrieval but prior to final output (i.e., in a QUERY). The following operators are included:

TOTAL - causes the summation of the non-blank entries of the specified field for all selected records.

COUNT - counts the number of non-blank entries of a specified field.

TALLY - counts the number of times a procedure encounters the TALLY statement.

AVERAGE - causes an average to be calculated for a specified field.

3.1.3.3 Data Presentation. Two commands may be used to obtain data presentations during an update or retrieval task.

> LIST - distributes across the page the data (under associated field headings) from the selected fields. A standard format is used and the output appears on the device associated with the query input device.
>
> FORMAT - permits interrupted field headings and data lines, variable width, height, and spacing, and output device selection.

Either command can be pre-stored in the task specification for the query or added at run time.

3.1.4 Data Description Language. GIS gives the user (e.g., the programmer) the ability to define a file once and then refer to the file and its records and fields, by name (in task specifications), without further reference to its storage format or organization. GIS stores and refers to the data description so that tasks can be executed which reference data elements symbolically. Data organization or format may be changed without necessarily invalidating established task specifications which refer to the modified data descriptions. The data descriptions and file processing controls are stored in a standardized table structure for each file.

3.2 Data Organization and Structures

3.2.1 Data Base Organization. The most generic data element in GIS is the file, and each data description must begin with a file name and is, in effect, a file description. The types of structures which can exist in a file, however, are quite varied and general.

A file is composed of an arbitrary number of items, called records, of the same logical structure. A record may have a multilevel structure which may be described as follows: a GIS record is composed of a single "segment"* followed by some defined number of embedded files, where a segment is a string of a defined number of fields (at the same level in the record).

*The word segment is GIS terminology. It is a statement whose subitems are fields.

This defines (recursively) what is called a file in GIS. The general logical structure of a GIS file is illustrated in Figure 1.

When there is more than one embedded file in a record (at the same level), each embedded segment must be identified by one of several options (to be described

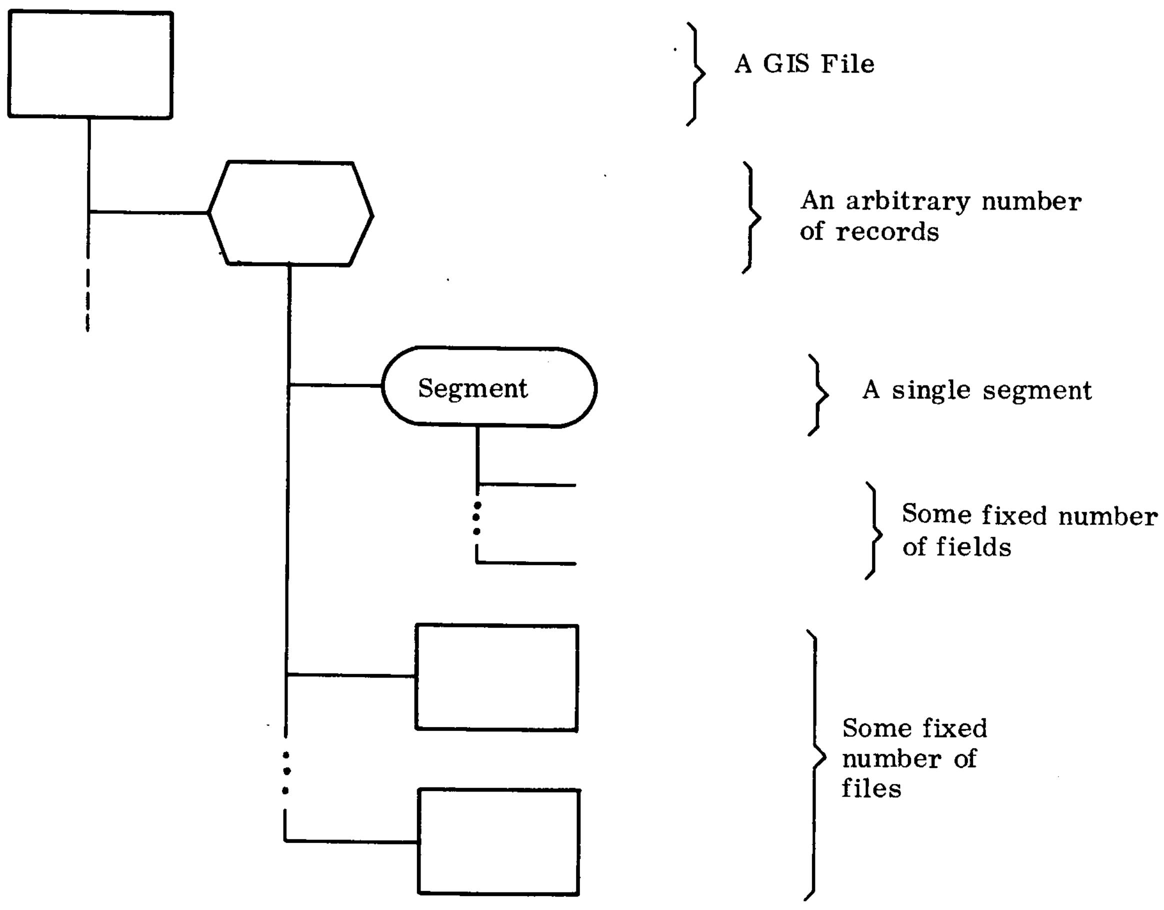

A GIS File
An arbitrary number of records
Segment
A single segment
Some fixed number of fields
Some fixed number of files

Figure 1. A GIS File

below) such as a key field which is common to all segments at that level and which takes a unique value for each embedded file. An example of a GIS file organization is shown in Figure 2, which shows the structure of a hypothetical file called File Zero. The records of File Zero contain a Level 00 segment (made up of fields C, D, and E) and two subfiles. These embedded files, Subfile 1A and Subfile 1B, contain Level 01 segments whose first field is a key field whose value is "A" for Subfile 1A and "B" for Subfile 1B. The Level 01 segment in 1A records contain the additional fields F and G. The Level 01 segment of 1B records contains the field J in addition to the key. The 1A records also contain a Level 02 File. The Level 02 segment need not contain a key field as there is only one Level 02 segment type in the structure. The Level 02 segment contains the information fields H and I.

Since the size of a subordinate file may be different in each parent record, a definition of file size (subrecord occurrence control) must be established for each file. Three options for subrecord occurrence control are provided by GIS, as illustrated in Figure 3. In the first option, Figure 3(a), each segment must contain a record-count field for each subordinate file (File 1A and File 1B).

In the second option, a key field which is present in every segment must be defined. In Figure 3(b), this key is shown as the first field of every segment. Its value is "Z" in every File Zero record, "1A" in every File 1A record, and "1B" in every File 1B record. Thus, the key field serves as an identifier of every segment, thus fulfilling the needs of subrecord occurrence control at the expense of the space for this field in every segment.

In the third option, a unique one-character termination code is used to signal the end of every file. As indicated in Figure 3(c), the File Termination Code occurs as a single character after the last record of every file. While this device is efficient in space, it is usable only where it can be guaranteed that the bit pattern representing the File Termination Code cannot occur as the first character of a segment.*

A field is a terminal GIS item - the smallest unit of data that can be referenced or described. The required descriptive elements for a field are its relative position in

*The GIS Application Manual implies that this character may not occur in any field - a considerably stronger restriction.

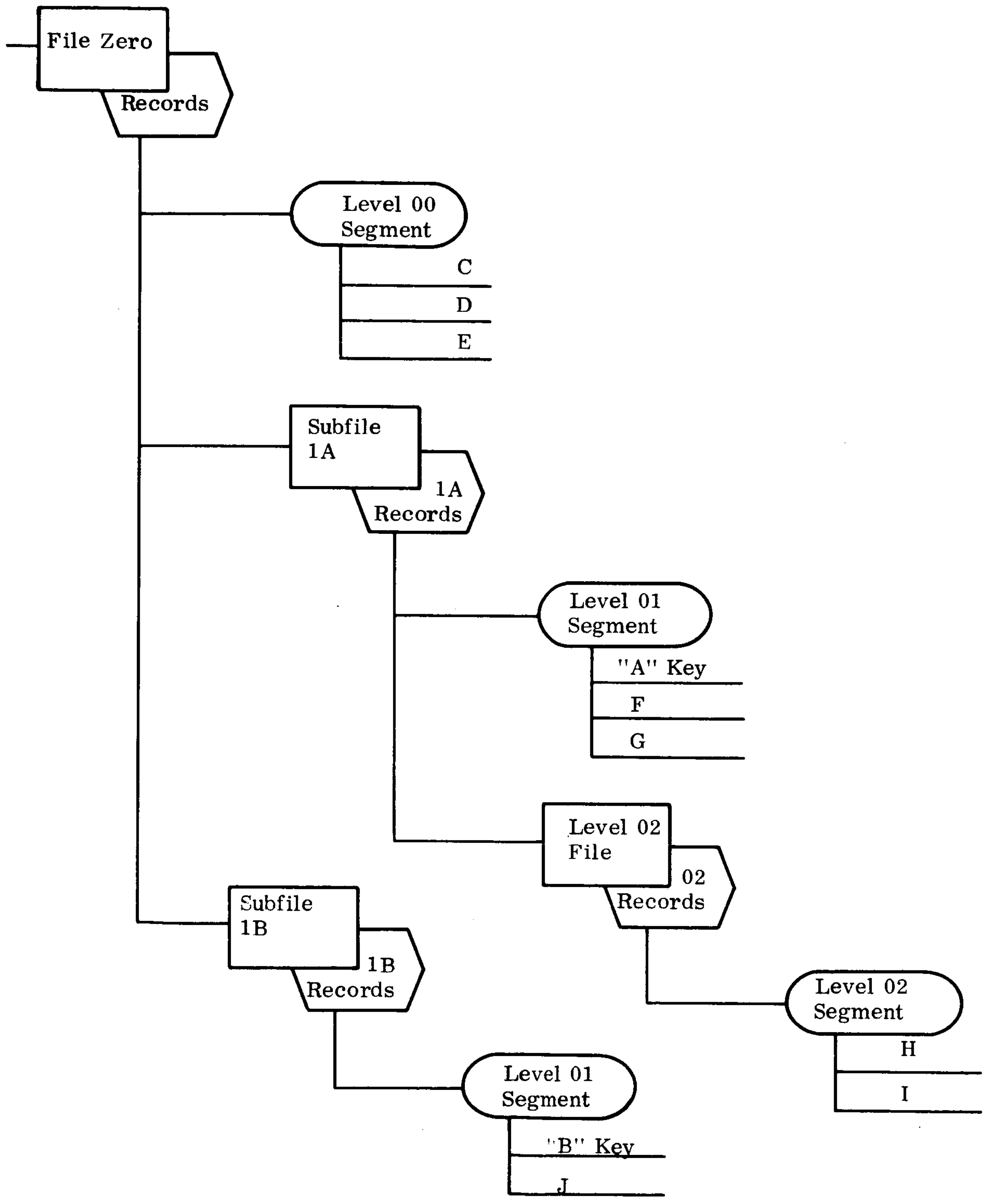


Figure 2. A GIS File Example

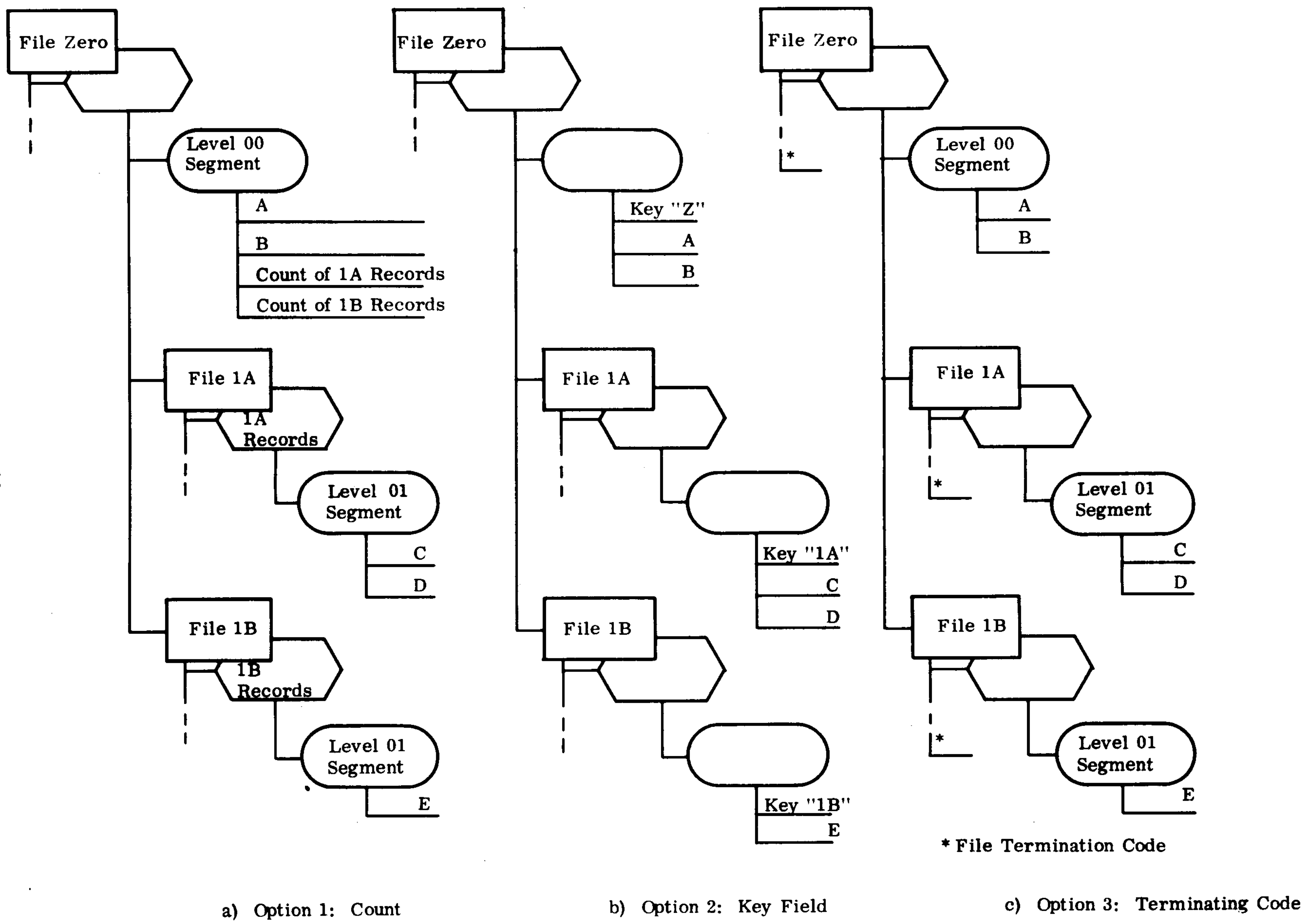


Figure 3. Subrecord Occurrence Control

the segment, its name (and synonym, if any), its type, and its length. Field type may be any of the following:

(1) Binary

(2) Decimal

(3) Floating point

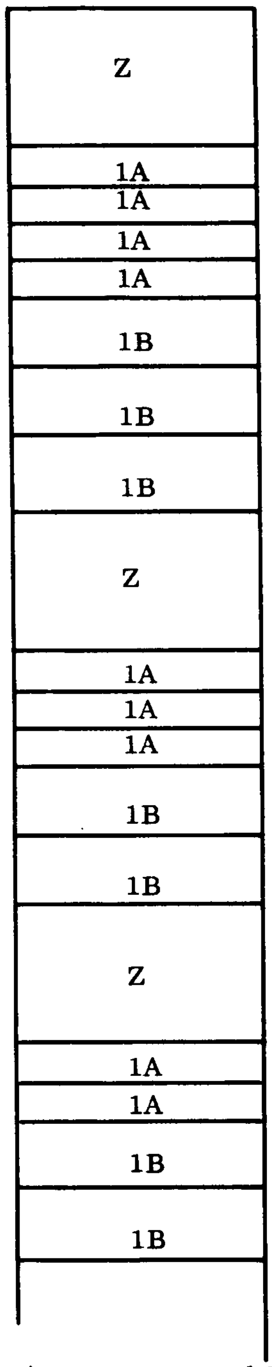
(4) Alphanumeric.

An alphanumeric field may be either fixed or variable in length, whereas all others must be fixed-length. Length control for a variable-length field may be provided by one of two options: count or terminating code. In the count option, each occurrence of the variable-length field will be preceded by a "length" field indicating the number of characters for this occurrence of the field. In the terminating-code option a user-selected special character is employed to mark the end of each occurrence of the variable-length field.

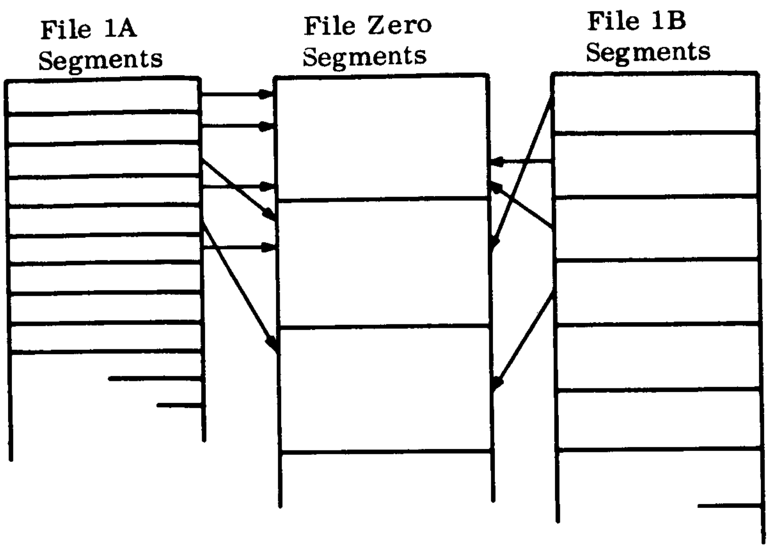
3.2.2 Physical Data Formats. In order to accommodate various processing requirements, GIS provides three physical format options for files shown in Figure 4. In the first option (linear record), shown in Figure 4(a), the data in each record is physically stored in the same sequence as its logical definition. That is, referring to the file shown in Figure 3, the fields of the Level 00 segment are followed by all Level 01 segments of File 1A, followed by all Level 01 records of File 1B. (If the 1A records had contained a Level 02 File, those segments would have preceded the Level 01 segments of File 1B.)

In the second and third options (split records), only the segments in the same level of each file are stored contiguously. In option two (link type split record), shown in Figure 4(b), and Figure 5(a), a link field occurs as the last field of each embedded segment. This field points from each embedded segment to the location of the segment to which it is subordinate (detail-to-master linkage).

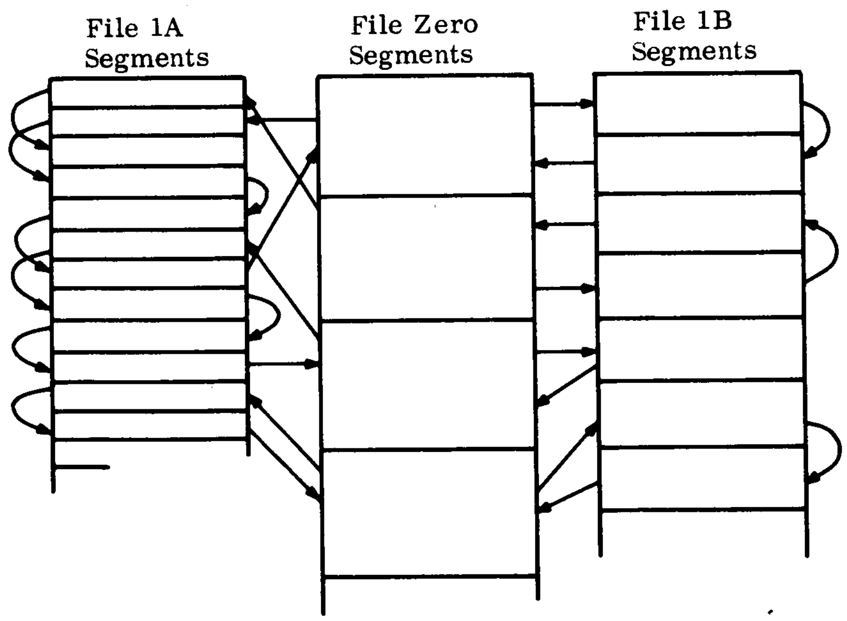
In option three (chain-type split records), shown in Figure 4(c) and Figure 5(b), a chain of links exists which originates in each master segment and threads through each record of an embedded file and back to the master. Thus, a master segment would contain a link field for each of its embedded files and, also, a link field to the next record (or its master, if it is the last record) at its own level.



a) Linear Record Format

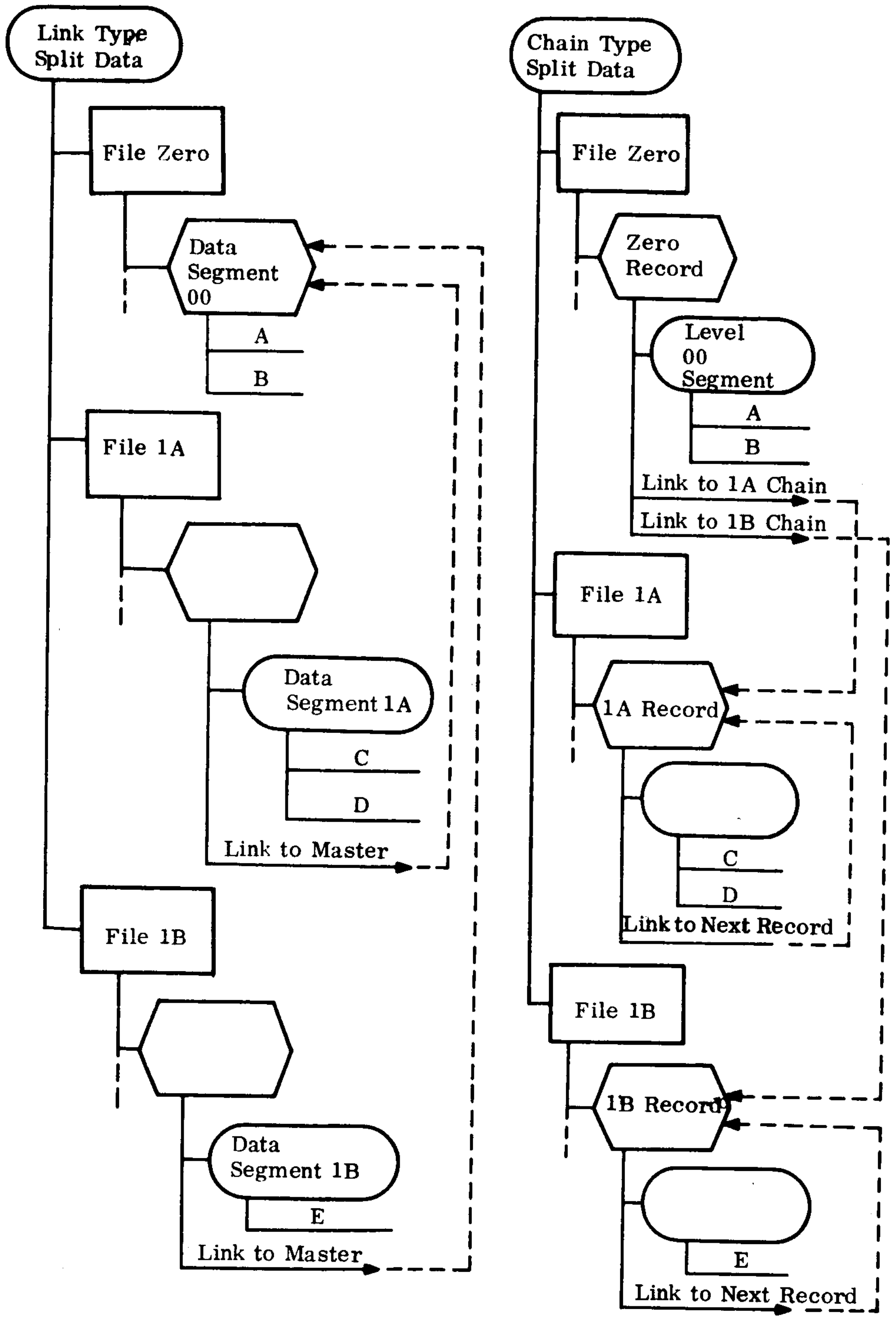


b) Link Type Split Records



c) Chain Type Split Records

Figure 4. Physical Formats



a) Link Type b) Chain Type

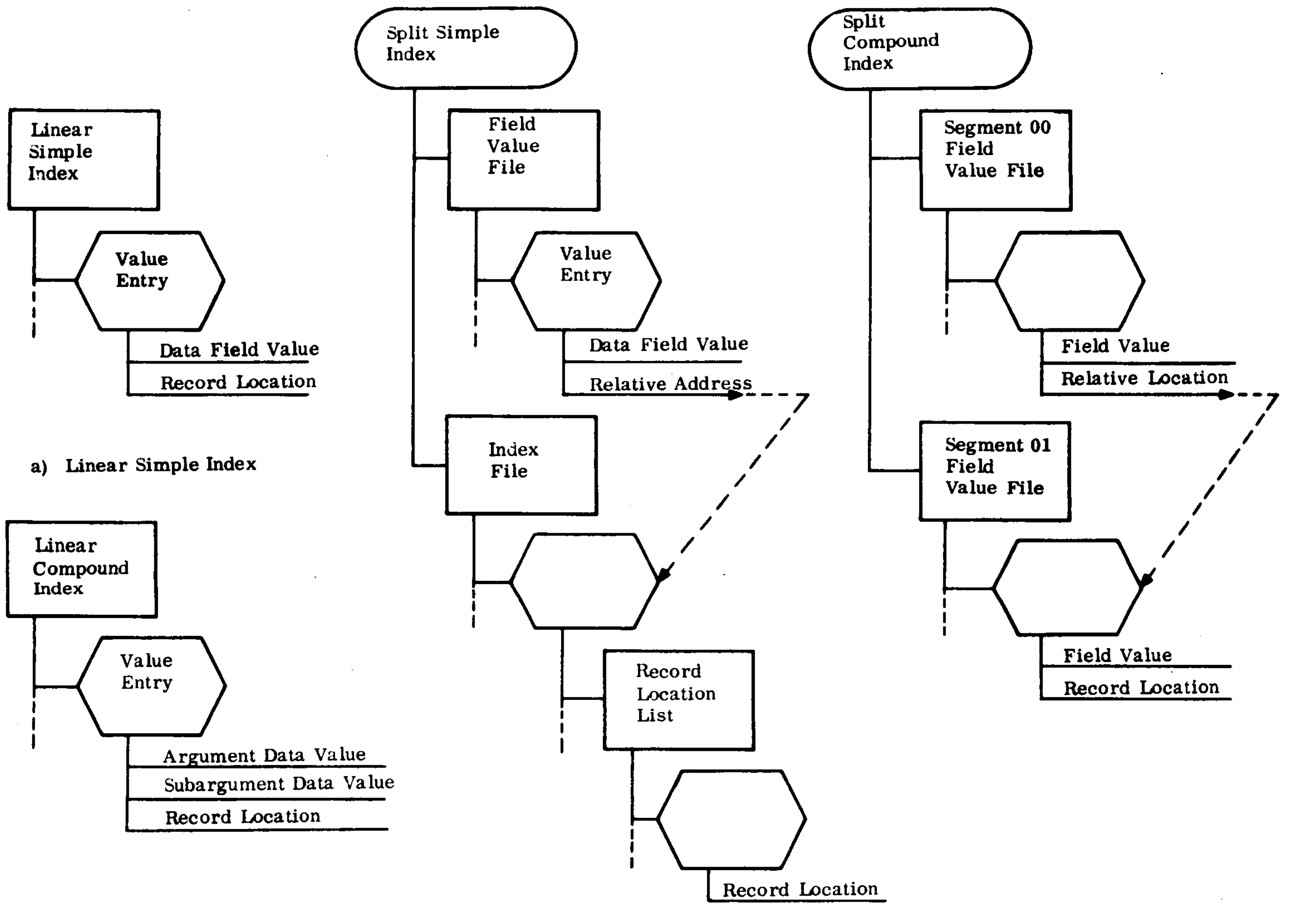
Figure 5. Split Records

3.2.3 Indexing. Indexes in GIS can be organized in any of the four ways shown in Figure 6. The simple indexes shown in Figures 6(a) and (c) will index fields occurring in the Level 00 segment of a file. The compound index shown in Figures 6(b) and (d) can index fields occurring in either a Level 00 segment or a Level 01 segment. Either a linear or split physical organization can be employed in a simple or compound index. The index value entries are ordered by data-field value. In the case of a linear compound index, the entries are ordered on both the argument (a Level 00 field value) and subargument (Level 01 field value).

4. EVALUATION

GIS will offer a programming language, a user-oriented query and command language, and a set of basic data management functions such as data definition, update, and query. GIS programs are compiled by the GIS translator and become jobs which run within Operating System/360. Thus, although constrained to a particular operating system, GIS provides the user with a data management system of broad capability. These is a capability of incorporating user-supplied non-GIS tasks in the GIS job. These may be written in other procedural languages such as FORTRAN or COBOL but since they may make no use of GIS services, and since they are bound to parameter and data specifications at compile time, they do not have the advantage of the data independence of GIS programs.

GIS has a query language with capability of expressing a logical condition of complete generality, and indexes which can be used for efficient retrieval of data which satisfies the conditional expression. In GIS, the indexes use physical, rather than logical, addresses to refer to data, thereby implying more direct access to data but additional maintenance due to physical data movement. In GIS, several index structures are offered, but it is not clear from available documentation whether these are options open to the programmer or whether the choice of index structure is implied by the type of data structure used. It is likewise unclear whether the simple index structure is used efficiently (e.g., logical product searches) to answer complex queries. GIS offers the capability of using masks on the compared values to provide for a "don't-care" capability within a field. Also available within the GIS query is the use of several summary operators such as COUNT and AVERAGE.



a) Linear Simple Index

b) Linear Compound Index

c) Split Simple Index

d) Split Compound Index

Figure 6. Index Types

In summary, GIS is a high-capability data management system with data and index structure and query and job languages. Considerable differences exist, however, in the approach taken in the two systems in the manner of interface of the data management system with the programmer and with the operating system. The GIS programmer must use the GIS language as his procedural language and compile an object program which runs as a job within the operating system. The DM-1 programmer uses standard procedural languages such as COBOL. The data services of DM-1 are available to the programmer as resident executive level services that are integrated as an operating system/data management system environment.

INFORMATION CONTROL SYSTEM (ICS)

1. INTRODUCTION

The Space Division of North American Aviation, Inc. and International Business Machines, Inc. have combined forces to design and develop the Information Control System (ICS) for the IBM System/360. ICS is designed to permit simultaneous execution of teleprocessing programs and conventional batch processing programs on a single computer.[1*] ICS is the result of on-line data base systems development at North American. This development dates back to 1964 when North American and IBM produced a 1460/1801 on-line data base processing system.

ICS is a message-driven (i.e., terminal-source) data management and application scheduling system. As such, its design focus is on the upgrading of the generalized Operating System/360 to a performance level that uses the terminal system as its primary control point.

*The superscript numbers denote references in the bibliography.

Several versions of ICS are planned. Each new version expands the services and data base description capabilities of the preceding versions. The initial version of the system will be operational in the first quarter of 1968, with succeeding versions planned but not scheduled.

It should be noted that ICS is a proprietary system. Questions concerning its availability must be directed to North American Aviation, Inc.

2. SYSTEM ELEMENTS

Figure 1 depicts ICS as a total system, including its four functional areas:

(1) Communication Control. This function provides the interface between the remote terminals using the system and ICS itself. Tasks performed by this function include:

- (a) Enabling the user to describe the systems lines, terminals, and their usage via communication descriptive language
- (b) Scheduling communication tasks
- (c) Opening and closing lines
- (d) Character set translating
- (e) Time and date stamping
- (f) Logging for restart and audit
- (g) Transmission error handling
- (h) Input and output queuing
- (i) Polling terminals
- (j) Receiving messages
- (k) Addressing terminals
- (l) Sending messages
- (m) Buffer allocating.

(2) Application Scheduler. This function initiates execution of messages processing programs based upon messages received. These message types are predefined to the Scheduler which then performs the function of allocation

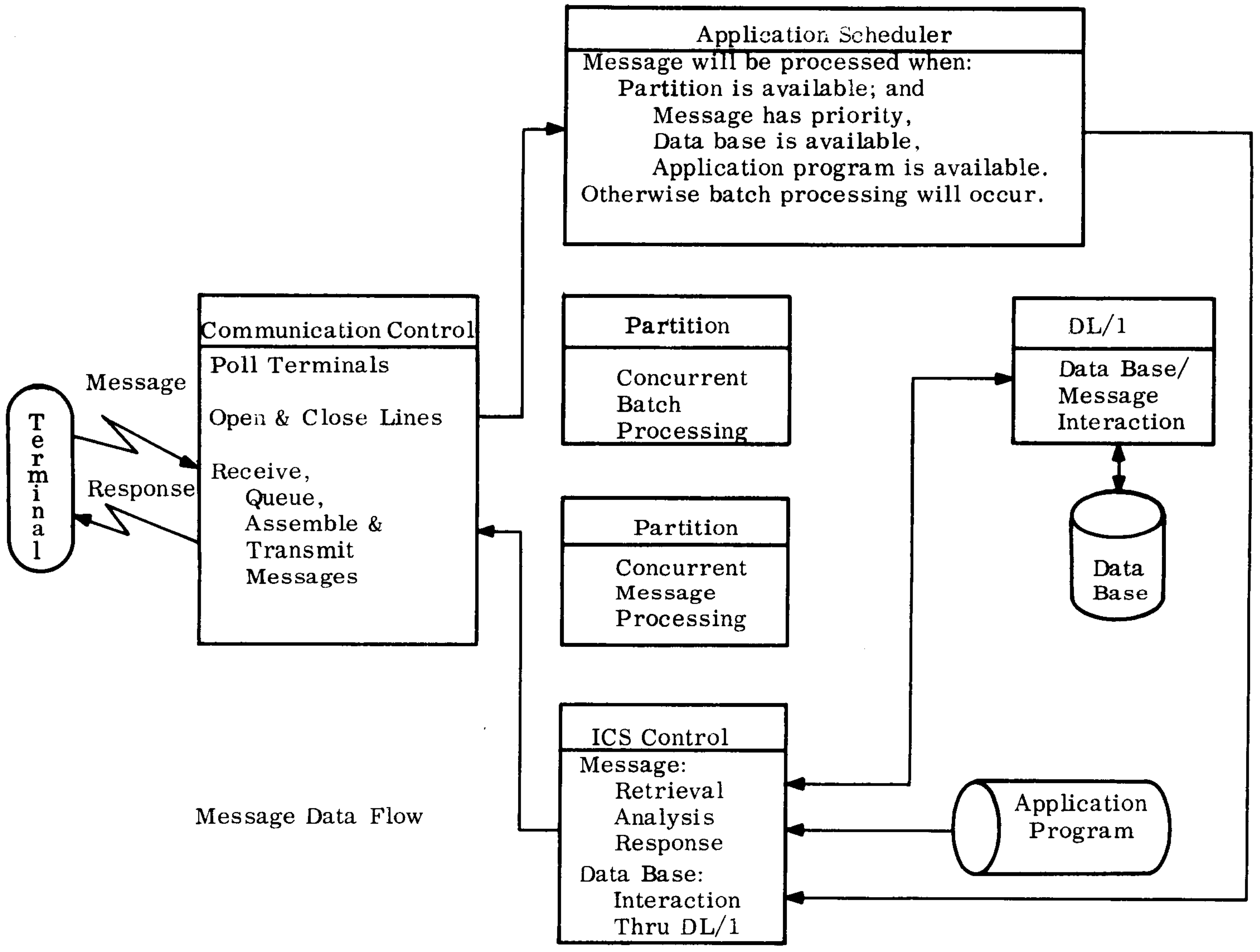


Figure 1. ICS Message Processing

of available and required resources to process the message (i.e., memory, appropriate priority of the message, data base required to process the message, and application program to process the message).

(3) ICS Control. This function acts as the interface between the application program, the application scheduler, the communication control modules, and the data base. Message response and analysis, along with data base retrieval, are the two functional task areas processed through ICS Control.

(4) Data Language/I. This function, performed under the control of ICS Control, comprises the bulk of the remainder of this report. Through Data Language/I (DL/I), the programmer is able to describe and process his data bases without concern for the housekeeping functions normally attributed to I/O operations. Calls for "segments"* of data from the data base are made using DL/I calls. These calls are interpreted by the DL/I Processor, operating in conjunction with the remaining services that comprise ICS.

Figure 1 also indicates the two modes of processing supported by ICS: concurrent BATCH processing and concurrent MESSAGE processing. A feature of ICS is the co-residence of both types of processing operating under the same system control facilities.

2.1 Software Elements

Operating System/360 (OS/360) forms the software foundation of ICS. The initial implementation of ICS operates with the OS/360 Interim Sequential Partitioned System. To effect this linkage, ICS requires a modification to OS/360 to allow for inter-partition communication. This is needed to service the ICS functions described above, each of which occupies a separate OS/360 partition, or part of one.

The ICS-OS/360 Software package is constructed from, and operates with the following elements:

(1) Data Language/I for data base definition, creation, maintenance, and reorganization

(2) Data bases defined, organized and created within Data Language/I conventions

*Segments are defined in Paragraph 3.5.

(3) Application programs written in conventional high-level procedural languages (i.e., COBOL and PL/1)

(4) ICS routines to perform the functions described above.

2.2 Hardware Elements

The first ICS versions will support a communication network comprised of IBM 1030, 1050, and 2740 terminals. The minimum hardware configuration to support this network under OS/360 must include 256,000 bytes of core memory. This memory level will support approximately 60 remote terminals, 5 message processing programs, 50 different types of messages, and 5 different data bases.

3. SYSTEM DESCRIPTION

ICS can be described as an extension of its host operating system, OS/360. The intent of the system appears to be the augmentation of services provided by OS/360 toward an easy-to-use set of input/output operations available to the programmers operating under ICS. The requirement for a standardized form I/O manipulation facility is obvious to a large programming and systems staff. ICS, through Data Language/I, provides that interface.

As a consequence of this principal system intent, some of the facilities associated with generalized data management systems are not part of the services provided by ICS-DL/I to terminal users. That is not to say that these features could not be added at a later time because the basis of its design makes ICS an "expandable" system.

3.1 Operations

ICS system users are defined as those persons accessing the data base through one of the remote terminals and programmers who provide the application program code that is processed through the terminals.

Figure 2, Terminal Users, illustrates the operational environment for ICS. Note that the "master terminal" and "slave terminal" system design philosophy is used in ICS.

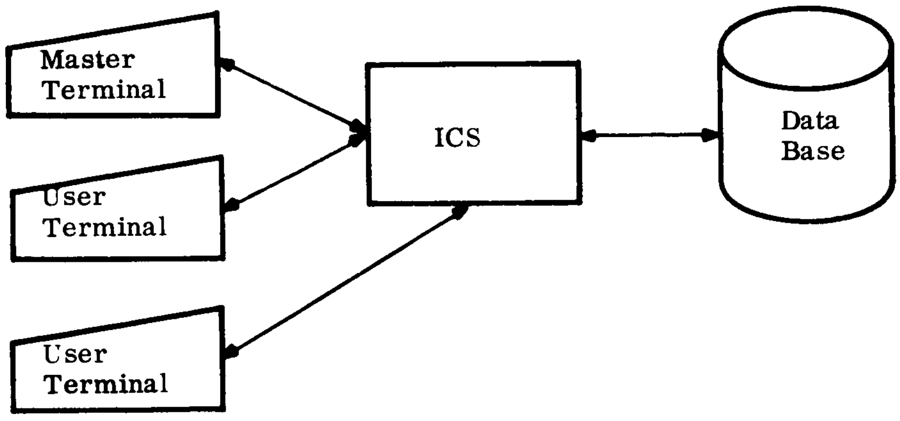


Figure 2. ICS Terminal Users

3.1.1 Master Terminal. The master terminal serves to control the operation of the system. This terminal has complete control of the ICS system with respect to communications, message scheduling, and data base operations. The Master Terminal must activate the system at start time, monitor the system during operation (through displays of status, etc.), and alter the operation of the system. A master terminal language comprises the following commands to facilitate system control:

(1) Shutdown the System. Causes a halt to processing and queues the time of initiation

(2) Discontinue Processing Only. Provides a halt to processing only, but allows the queues of messages to continue to build up

(3) Refuse Further Input. Allows everything to finish that has been queued by the system, but refuses further input to the system

(4) Initiate Operations. Provides for the starting of the system

(5) Print Control Information. Provides the ability to cause the contents of the system tables to be displayed on the master terminal

(6) Alter System Tables. Provides a command that is used to relate the following to each other:

   (a) Transaction code of message to program

   (b) Password terminal name to transaction code

   (c) Terminal name to line number, terminal address, or terminal name.

3.1.2 Slave Terminals. The slave terminals are the means by which operators communicate data-handling information to the system. A terminal command language is provided to augment the application messages required by the application program:

(1) Terminal-to-Terminal Message Switching. Allows messages to be sent from one terminal to another

(2) Character Correction. Permits input message character corrections by a backspace of the carriage

(3) Message Retransmission. Provides the ability to request output (from the system) message retransmission

(4) Terminal Testing. Allows terminals to operate in a test mode. No messages are processed, and any messages received will be returned until the terminal is subsequently attached to the system.

(5) Exclusive Line Control. Notifies Communication Control that this terminal is to be given exclusive control of the line; other terminals on the line are not to be polled by the system until this terminal has relinquished exclusive control.

(6) Protection Provisions. Provides the facility to lock and unlock terminals, data bases, transaction codes, and programs from a terminal via their name and password.

3.1.3 Programmers. Figure 3 illustrates the interface mechanism between the program written for ICS and the system. Because ICS is intended as an interface only between application programs and terminal users, this interface does not include

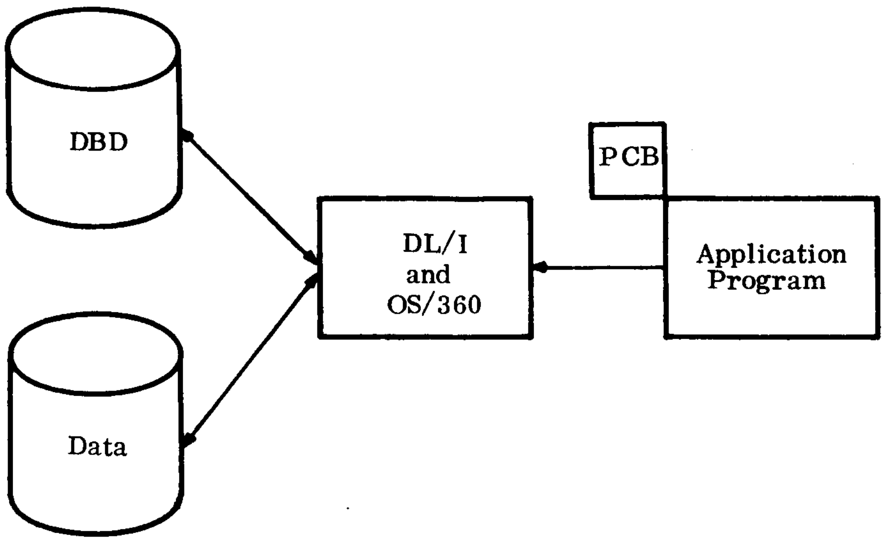


Figure 3. ICS Program Interface

generalized retrieval services, report generation facilities, etc. attributed to generalized data management services.

The interface mechanisms illustrated in Figure 3 highlight the following features of the Data Language/I system to be detailed later in this report.

(1) Data Base Directory (DBD). The central directory used by DL/I to describe and locate data in the data base. A separate Data Base Directory exists for each data base. The central directory is comprised of a data set of all individual Data Base Directories. A utility program, running as a separate job step, creates the DBD from definitions of segments, segment type codes, and tables defining the hierarchical relationship of the segments of the data base.

(2) Program Control Block (PCB). The interface between the application program and DL/I. When the PCB is completed during job initiation, it contains the name of the data set and the DBD associated with it. It further defines buffers and a work area loaded by DL/I and used by application programmers. Communication between the application program and DL/I is maintained through a field in the PCB.

(3) Interface With OS/360

ICS interfaces with OS/360 through the previously mentioned ICS Control function. Data calls executed by application programs (using the PCB) are interpreted by the system, which calls upon OS/360 Data Management services for segment retrieval, writing, and indexing.

(4) Job Management Interface

The terminal system described provides the linkage between the terminal users and the application programs running under ICS. The messages processable by these programs, along with the system terminal command language just described, provide the means for job management within the system.

3.2 System Services

The generalized system services discussed here are available through the User Interface (i.e., terminals and their command languages) or the Programmer Interface (i.e., the application programs and DL/I services). Functions mentioned below are included in the system according to the documentation reviewed.

(1) Directory Updating

Directory updating is provided by the DL/I directory generation utility program. This job lets the application programmer describe the structure of his particular data base.

(2) Data Search

Data search functions are provided through the conditional expression of segment retrieval under DL/I. One of the calls that may be made by the application program requests a search of the data base attached to the program. This search (possibly executed with a condition stated) is carried out by DL/I until either the appropriate segment is found or an indication that the segment is not in the data base is delivered to the application program through the program's PCB.

(3) Data Search Look-Ahead

Data Search Look-Ahead is provided by the "get next occurrence of a segment" call executed by the application program.

(4) User Accountability

User Accountability is incorporated into the system through the Master Terminal's assignment of a terminal to the system. This assignment includes the designation of a password to be used by the terminal operator to request data from the system.

(5) Job Restart Points

Job Restart Points are maintained by the system through the Communication Control function. Messages, i.e., implicit job requests, are queued by the system during their receipt. The system may be restarted at any point in the message queue by the master terminal operator. Messages that have remained in the system queue after system shutdown are brought into the system for processing subsequent to restart.

3.3 Data Structure

While ICS is a system that operates as a unit under OS/360, its utility is realized through the system's data base organization philosophy: Data Language/I. DL/I is the title that the authors of ICS apply to the set of mechanisms and languages provided for the description, maintenance, and processing of the data used by the system's application programs. In DL/I, data base definition is external to the programs that update the data base. Before a programmer can reference or create his data base in a batch or message processing program, he must define the program's data requirements (i.e., format, etc.) and communicate these requirements to DL/I. DL/I utility programs are used to communicate these requirements to the DL/I directories.

3.4 Data Definition

The data structures processable under DL/I follow the form of header-trailer segment structure. The principle of data definition and processing implemented under DL/I depends upon the ability to describe a file or set of files as a data

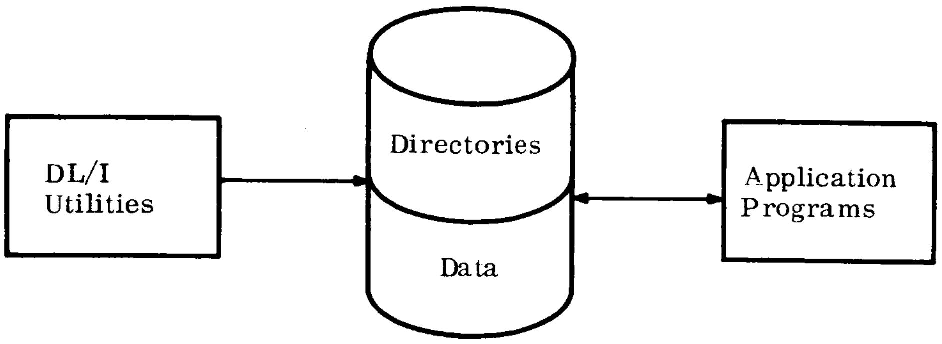


Figure 4. Data Language/I

base. The "header-trailer" segment structure readily transforms definable data to hierarchical form referred to as a "tree structure."

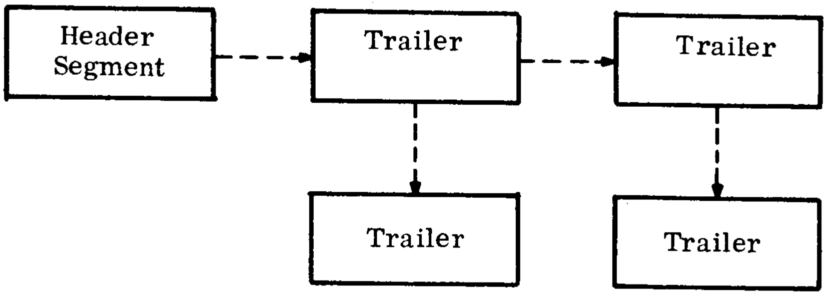


Figure 5. The Header-Trailer Segment Structure

The "header segment" in DL/I is defined as the Root Segment.

The logical description of the data base records to the system is not enough to process the individual occurrences of the defined records. The link between the logical description of the data base and the individual records that form the data base is through the specification of a KEY within the records. This key (e.g., account number, name, etc.) is indicated to the system through the Data Base Directory.

3.4.1 Simple Files. The records in a simple file contain fields of information arranged so that each record in that file contains exactly the same fields in exactly the same arrangement. These fields may be fixed or variable length; i.e., the same field may or may not vary in length from record to record within the file.

As an example, consider the following hypothetical personnel file:

	Man #	Name	Address	SS #	Pay
Rec. 1	445	J. Doe	1234 Westland Dr. L. A.	333-11-222	1500.00
Rec. 2	446	D. Smith	1356 Eastland Dr. L. A.	111-22-333	1700.00
Rec. 3	447	W. Walton	1589 Eastland Dr. L. A.	326-10-766	450.00
Rec. 4	448	R. Jones	1342 Ardis Ave. Dny.	275-05-001	600.00
Rec. 5	449	B. White	1275 Clark St. Dny	300-99-221	650.00

This file contains only the man-number, name, address, social security number (SS #), and pay scale for five persons. Each row represents one record of the file.

3.4.2 Segmented Files. The segmented file is one whose record contents have been regrouped into smaller pieces called "segments," a process similar to chopping a record into several pieces, with each piece containing only a portion of the original record. Each of the records in the file was chopped into the same segments; i.e., all the segments contain the same field ordered in the same way although the value of the contents of the fields may vary. For example, if the simple file described earlier were split into two segments, it would look like this:

Segment #1				Segment #2	
	Man #	Name	Address	SS #	Pay
Rec. 1	445	J. Doe	1234 Westland Dr. L. A.	333-11-222	1500.00
Rec. 2	446	D. Smith	1356 Eastland Dr. L. A.	111-22-333	1700.00
Rec. 3	447	W. Walton	1589 Eastland Dr. L. A.	326-10-766	450.00
Rec. 4	448	R. Jones	1342 Ardis Ave. Dny.	275-05-001	600.00
Rec. 5	449	B. White	1275 Clark St. Dny.	300-99-221	650.00

3.4.3 Hierarchical Files. Hierarchical files are used by the system to eliminate repeated data between and within files. The conceptual hierarchy is illustrated in the sample layout below.

Line #	Record Information			
1	445	J. Doe	1234 Westland Dr.	L. A.
2		17589	500.00	
3			Hospital	300.00
4			Doctor	200.00
5		18811	200.00	
6			Doctor	200.00
7		333-11-222		1500.00

Note: 1. Each line represents one segment.
2. The amount of indentation signifies degree of dependence or level.
3. The line numbers will be used in the text that follows when referencing this example.

Starting with the root segment (line #1) and progressing through the file, successively lower level segments are encountered through line #3 in the preceding example. This constitutes a hierarchical "path." A requirement of the hierarchical file concept is that the segments which constitute a hierarchical path must be in sequence according to descending hierarchical level.

A segment is classified by the system according to its "type"; two or more segments are of the same type if they contain the same fields, and those fields are ordered alike. In the above example:

Type 1:	Line 1
Type 2:	Line 2, Line 5
Type 3:	Line 7
Type 4:	Line 3, Line 4, Line 6

3.5 Data Base Directory Creation

Each data base is described to the system through a utility program provided for that purpose. The input to that program is a set of DL/1 control cards that describe the programmer's data base.

3.5.1 MODE Control Card. This must be the first DL/I control card in the setup for the job step. It determines how the step will operate (i.e., TEST, GO, PRINT) and what will be produced from it. There may be only one MODE control card in this job step.

3.5.2 DBD Control Card. This card names the data base to be described and provides DL/I with preliminary information concerning its organization and disposition. There can be only one DBD control card in this job step.

3.5.3 DMAN Control Card. Each DMAN control card describes one data set that is to be set up by Data Management to accommodate the data base being described. There may be up to nine data sets specified for a single data base.

3.5.4 SEGM Control Card. The SEGM control card defines a segment to be contained in the data set defined by a preceding DMAN control card. There may be a maximum of 255 segments defined. SEGM control cards must be entered in hierarchical order. The segments will be physically stored in the data base record in the same order in which these cards are entered.

3.5.5 FLD Control Card. The FLD control card defines each of the fields in the segment defined by the preceding SEGM control card that may appear as part of a qualification statement.

3.6 Item Structures

As is frequently the case with a large system, certain features of the system's architecture are not included in the initial implementation. Those restrictions, imposed by the initial version of DL/I, that apply to its data base defining capability are as follows:

"1. There shall be only one root segment per data base. This implies that there shall be only one sort key and hence, only one sort order per data base.

2. The total length of the root key in bytes shall be equal to or less than 255 bytes.

3. There may be "N" dependent segments per root segment as long as the restriction listed below is not violated.

4. The sum of the number of the different named segments under a root segment shall be less than or equal to 254.

5. Each segment, be it root or dependent, shall be a single fixed length. The length may vary from segment type to segment type, but a single named segment shall have a fixed length.

6. It will be possible to have up to 15 levels of dependency plus the root segment in any single logical record.

7. Provisions will be made for private data bases only. The special provisions required for automatically operating a shared data base across several projects will not be included in Version 1. As noted above, the sensitivity codes will be included for dependent segments."

3.7 Program Interface

The principal language elements of ICS are its application program messages, its terminal command language (discussed earlier), and its programming languages.

3.7.1 Programming Languages. Application programs written for execution under DL/I may be programmed in either COBOL or PL/I. In order for DL/I to process requests from an application program, certain types of entries must be provided in that program so that the program's data base calls will be communicated to DL/I routines that perform the functions of retrieval.

The Program Communication Block (PCB) is a set of entries providing the following information:

(1) The name of the data set to be processed

(2) The specification of the DL/I functions which will be used

(3) Indication of the types of segments that will be processed

(4) Areas for receiving responses from DL/I.

The individual PCB entries are explained in the following paragraphs. Figure 6 is referred to by using the "ref" column numbers as pointers, shown in parentheses after each entry name.

Name of this PCB (ref 1). This is a label name which the programmer selects.

Ref	Control*	Description
1	ACL	Name of this PCB
2	ACL	Name of Data Base Directory (DBD)
3	DCL	Segment Hierarchy Level Indicator
4	DCL	DL/I Results Status Codes
5	ACL	DL/I Functions to be Used
6	DCL	Reserved for DL/I JCB Address
7	DCL	Segment Name Feedback Area
8	ACL	Length of Feedback Key
9	ACL	Number of Sensitive Segments
10	DCL	Key Feedback Area
11	ACL	Name of Segment Type Highest in Hierarchy (Root Segment)
12	ACL	Name of Segment Type at Next Lower Level
		Complete hierarchy of all segments to be used in the application program must be accurately shown.
13	ACL	Name of Segment Type Lowest in Hierarchy

*ACL = Data Provided by Application Programer.
DCL = Data Provided by DL/I.

Figure 6. The Program Communication Block

Name of Data Base Directory (DBD) (ref 2). This name is obtained from The Directory of Data Base Descriptions. It is the name assigned to the directory of the data base which will be processed by the application program.

Segment Hierarchy Level Indicator (ref 3). DL/I loads this area with the level number of the lowest segment encountered in its attempt to satisfy a request from the programmer to retrieve a segment.

DL/I Results Status Codes (ref 4). Flags, in character form, are placed here by DL/I and remain here unchanged until this PCB is referenced by another DL/I CALL. These flags tell the application programmer when certain conditions arise, e.g., end of the data base encountered when sequentially processing the data base.

DL/I Functions to be Used (ref 5). The application programmer places character codes here prior to opening the data base to tell DL/I which functions will be used during data base processing.

Reserved for DL/I JCB Address (ref 6). DL/I uses this area for its own internal linkage related to this particular application program.

Segment Name Feedback Area (ref 7). DL/I puts into this area the name of the lowest segment encountered in its attempt to satisfy a segment retrieval request.

Length of Feedback Key Area (ref 8). This entry specifies the length of the area required to contain the completely qualified key of any segment to be processed.

Number of Sensitive Segment Types (ref 9). The application programmer must tell DL/I which segment types he will process in his program.

Key Feedback Area (ref 10). DL/I places into this area the completely qualified key of the lowest segment encountered in its attempt to satisfy a segment retrieval request.

Name of Segment Type Highest in Hierarchy (ref 11). As noted previously, the application programmer must tell DL/I the names of all the different segment types he will process. He must further indicate the hierarchical relationship of these segments. The segment names and their places in the data base hierarchy can be obtained from the Directory of Data Base Descriptions. Beginning with this entry, the name of the root segment is entered. Then the names of the segment types to be processed are listed in exactly the same order as they appear in the Directory of Data Base Descriptions.

Name of Segment Type Lowest in Hierarchy (ref 13). This entry is made in accordance with the preceding instructions.

3.7.2 Program Calls. Communication between the application program and the data base is explicitly requested through the DL/I program calls. These calls are used whenever the application programmer requests DL/I services from the system. Operating through the Program Communication Block defined by the programmer, the calls perform services of OPEN and CLOSE of a file; file reading across a data base or along the leg of the segment tree; and segment INSERT, DELETION, and REPLACEMENT.

3.7.3 Qualification Statements. Records may be requested from the DL/I data base described by the programmer using a form of conditional retrieval specification. When the program call is supported by a qualification statement of the form described next, the segment will be presented to the requesting program only if the indicated qualification has been met in the segment. If the segment is requested and it does not meet the required qualification criteria, it will not be delivered to the application program; DL/I will continue the scan of the file to provide a segment that does meet the qualification requirements.

When the application programmer requests DL/I to perform functions, it is frequently necessary for him to specifically identify a particular segment by its own key field and the key fields of all parent segments along the hierarchical path leading to the segment. These key field references do not appear directly in the call statement; instead, a label given in the statement points to an area in the user's program which contains the actual segment-search-argument (SSA).

The SSA consists of two kinds of entries, the segment name and (as required) a segment qualification statement. The segment name points the system to that entry in the Data Base Directory which contains the characteristics of the segment and its key field.

The segment qualification statement contains entries which DL/I uses to test the value of the segment key field for determining whether the segment meets the user's specifications. The segment search argument portion that contains the qualification statement is illustrated in Figure 7. Its components are discussed in the following paragraphs.

```
(Segment - Field - NameⓇCompara-
tive - Value)



Ⓡ  = Relational Operator
```

Figure 7. The Segment Qualification Statement

(1) <u>Segment-field-name.</u> This is the name of a segment key field which appears in the description of that segment type in the Directory of Data Base Descriptions. The name is eight characters long, with trailing blanks as required.

(2) <u>Relational-operator.</u> This is a set of two characters which expresses the manner in which the contents of the field, referred to by the segment-field-name, are to be tested against the comparative-value.

Operator	Meaning
b=	must be equal to
b>	must be greater than
b<	must be less than

NOTE: As used above, the lower case b represents a blank character.

(3)	Comparative-value.	This is the value against which the contents of the field, referred to by the segment-field-name, are to be tested. The length of this entry must be equal to the key field in the segment; i.e., it includes leading or trailing blanks (for alphanumeric) or zeros (usually needed for numeric fields) as required.

4. EVALUATION

An evaluation of ICS as a total system must take into consideration the reasons for its architecture and use. The documentation reviewed conveys the impression that ICS is a system developed to support telecommunications processing of specific applications. ICS is not a revolutionary attempt to change the character of the programming and systems design functions; programmers and systems designers still exist under and work with ICS and DL/I just as before. There is one major exception to that statement; they will be using the features of OS/360 in a standard manner dictated by the conventions of data base description and processing imposed upon them by DL/I.

4.1 Ease of Use

DL/I appears to be a system that will be reasonably easy to use since the standard programming languages of COBOL and PL/1 are continued under DL/I. The changes incorporated in the programmer's normal attack on a programming job appear only in his specification of the Program Control Block, his definition of the data base structure through the data base definition utility provided with the system, and his altered execution of I/O statements in his program. Certain restrictions implied in the use of the initial version of DL/I would seem to limit the programmer's use of the system; however, even these restrictions appear minimal and should be corrected in future versions of the system. The terminal command language appears to be minimally structured to fit the specific requirements of the set of applications.

4.2 Versatility

Considering the features normally associated with a generalized data management system, ICS would have to be considered a special-purpose system with the capability of growing into a large, general-purpose system.

While many features of the general-purpose system have not been incorporated in the documentation reviewed on ICS, it is felt that these features could be incorporated as the system grows with use. The general-purpose Query Language, for example, could be added when the data base directory system is enlarged to accommodate a sufficiently large corpus of data.

4.3 Economy

It is here that ICS appears to offer the most advantage. The relative simplicity of the system enables its implementation at a predictable pace. Expansion on the system can then come as the need requires.

A second form of economy is seen in the relatively simple way DL/I interprets the data upon request from the application program. As mentioned in the documentation reviewed, "there is some overhead involved, but it should be minimal." The simple forms of data base structures that are describable and processable by the system impose a form of operational economy not realized by more complex systems.

4.4 Responsiveness

The ICS-DL/I system appears to respond to the requirements of the environment for which it was structured. The straight-forward handling of data calls, described previously, does not change the job of the programmer to any major extent. The terminal system, with its associated command language, is probably sufficient for the environment for which it was designed.

Finally, it should be remembered that ICS - DL/I is a system built by and for North American Aviation, Inc. The dictates of the corporate requirement are seen in the design of the system.

The system is evolving to enlarge the spectrum of services available to the user of the system. According to the documentation reviewed, ICS-DL/I is to be expanded as follows:

(1) Dependent segments will be stored physically on separate devices.

(2) The tree-structured hierarchy of files will be augmented to include file linkage. With this change, it will be possible to hold the data itself in a single sequence, strip off the several keys to that data, and hold these several keys as related (but compressed) indexes to the main file.

(3) The growth of the system will impose requirements for including system features to manage and control large data bases. These services will be designed to support a "data base manager" with the following types of functions:

   (a) Operational records for control of persons authorized to access the data base.

   (b) Operational facilities for the control of system operation by maintaining records of runs that were made, length of the runs, types of operations performed, etc.

   (c) Security protection facilities to allow for individual control of the data placed in the data base.

(4) The incorporation of a more sophisticated job control facility than currently exists, i.e., allowing the console operator to obtain a running time estimate before a job is set up for production.

(5) DL/I-based utility program which, when given the name of the data base and the layout of desired reports, will obtain the appropriate directory, generate the proper program, and execute the program just generated to produce one or more formatted reports from a stored data base.

5. BIBLIOGRAPHY

(1) Data Language No. 1 (DL/1) Encyclopedia, Pub. 2541-F New 8-66, North American Aviation, Inc. and IBM

(2) Unpublished paper: DL/1

(3) ICS: Information Control System, North American Aviation, Inc. and IBM

DATA MANAGER — I (DM—1)

1. INTRODUCTION

This appendix describes the generalized data management system designed and developed by the AUERBACH Corporation and identified as Data Manager - I (DM-1). The system has been developed for Rome Air Development Center's Reliability Central program and commercial applications. The development of the system extends from 1963 when initial conceptual work was done through an internal research activity funded by the Corporation.

1.1 System Objectives

Data Manager - I (DM-1) is a generalized data management system focusing on virtually all aspects of the management of the data processed by a computer installation...including those "data-associated" functions of program specification and control, job management, data base querying, and data base maintenance. The system, operating in conjunction with a host operating system (in the current implementations), provides a spectrum of data management services in the areas indicated above.

The documentation reviewed* describes the system as providing:

(1) A rationale for the structuring of a large, dynamic data base. Data can be integrated into the DM-1 data pool under structural specifications which retain the individuality of the diverse data items while manifesting the relationships among the items.

(2) A repertoire of generalized system operations that provide for the management of the DM-1 data pool. Users can modify the data and its descriptive parameters to accommodate new data elements, or new relationships, or to adjust to changing requirements and operational experience.

(3) A mechanism for retrieving selected data to meet specified information needs. Users and programmers can access the precise data items needed to meet on-demand requirements for information. DM-1 can display the results of an inquiry issued at a console, develop a data item with specified characteristics for further processing, or deliver the specified data to a program operating with the system.

(4) A procedure which assists an inquirer in defining his information needs. A user can perform a dialogue with the system. From displays presented by the system, he chooses the pertinent attributes to describe the item about which he needs information. Each display identifies the classes of information available in the DM-1 data pool. The displays proceed from generic to specific identifiers in response to the inquirer's selections. After the attributes are defined, the user provides limiting conditions, which define the properties of the individual units of information he needs, by selecting characteristics to define the pertinent data from another series of displays.

(5) A library of application-oriented programs and jobs. The user can add new programs to the DM-1 repertoire to perform special-purpose processes on the data. New units of work (jobs) may be defined as combinations of existing system and application programs to meet recurring needs for data processing.

(6) A mechanism for selecting data processing routines from the library and applying them to specified data elements in the data pool. The user can issue commands from a console for the execution of any job in the DM-1 library. He can specify the data items in the data pool to be operated on by the job.

*See Section 6, Bibliography.

(7) A method of maintaining data and programs as independent, mutually complementary resources. The data in the data base need not be oriented to any specific set of programs. The application programs need not be limited to operation on specific items of data. DM-1 maintains structural information about the data in its directories, and it maintains information about the data requirements of the programs in its library. If the requirements of the program fail to match the characteristics of the data, the system can transform the data to the format required by the program.

(8) A set of system routines to control the execution of DM-1 jobs and to service the data needs of programs during their operation.

(9) A mechanism to provide for data integrity and security. Users are protected from unauthorized access to their data.

1.2 Implementation Status

DM-1 is being implemented (at this writing) on the Univac M1218 computer and on the IBM System/360, Model 50. (Another version, described by the implementers as "machine independent" at the moment, is being produced for the Bureau of Employment Security).

At this writing, both implementations are proceeding through the initial stages of development. The systems are in operational "test" status, providing basic directory manipulation services to programs coded to test the operation of the DM-1 Directory System. The plans are to proceed to complete implementation during 1968, with delivery of the completed systems to be made during late 1968 or early 1969, depending upon the implementation involved.

2. SYSTEM DESCRIPTION

DM-1 is a console-directed system, which operates in response to user commands issued in the system's Job Request Language. The user treats the systems's Job Library as a set of operators at his command. He treats the data pool as the operands to be bound to the operators. The jobs that he invokes may be lengthy processes that consist of many tasks to be executed over large files of data or simple functions that consist of a single operation on a small unit of data. The user may execute a job by specifying its name and binding its input and output parameters. He may supply literal values for the parameters at the console, he may give the names of the data pool items containing the values, which may be qualified by conditions the data must meet, or he may specify secondary jobs which will supply the arguments for the primary job.

Figure 1 provides a graphic description of the mechanisms (i.e., routines and jobs) that perform the services requested by the user. Note that the system includes, in addition to the Directory mechanism, a library of connectible generalized "jobs" which are used by user and system jobs within the system.

2.1 Query Operations

The user has at his command a set of system jobs which enable him to query the data pool. The query job may be used to answer questions about the state of any items in the data pool. A condition specification of arbitrary complexity may be used to select the items of interest.

The user may engage in a dialogue with the system to determine the names and structural relationships of items in the data pool, to select items of interest, and to specify conditions for selecting the values of those items for display or processing. When he can formulate a formal query, the user may call on the query job to create a new item for processing or to produce a report or display of selected items, based on the values of those items which meet a condition of arbitrary complexity.

2.2 Programs

A program is described to DM-1 by executing the program description job. The names and the precise structure of the formal input and output items are specified at the console. This information is translated by the program description job into an entry in the library. Once a program has been described to the system, it automatically becomes a job in the system's repertoire. It may be called and executed on specified data from a console, used as a component in another job, or called and executed as a subroutine of another program.

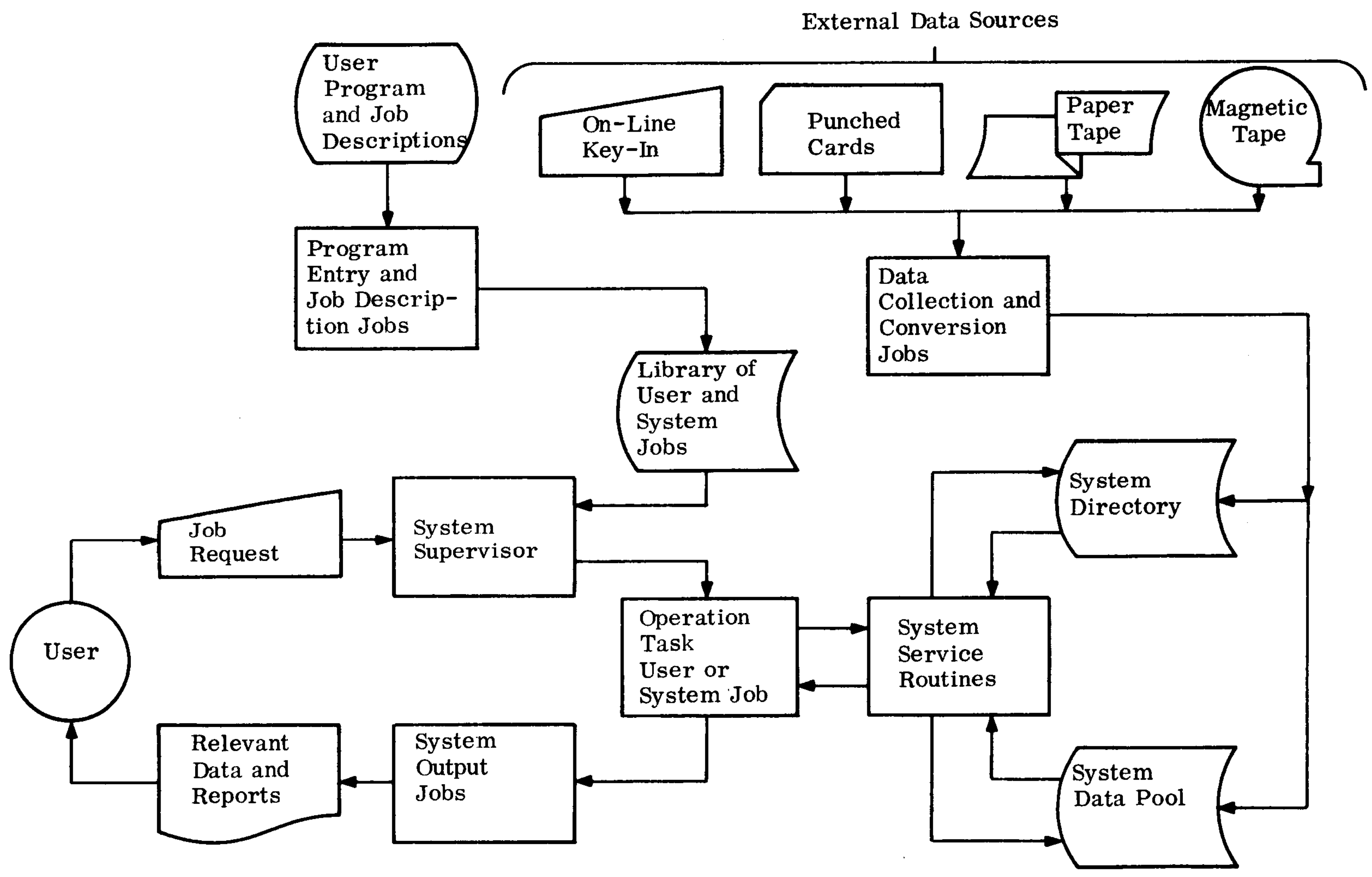

External Data Sources
User Program and Job Descriptions
On-Line Key-In
Punched Cards
Paper Tape
Magnetic Tape
Program Entry and Job Description Jobs
Data Collection and Conversion Jobs
Library of User and System Jobs
System Directory
Job Request
System Supervisor
Operation Task User or System Job
System Service Routines
User
Relevant Data and Reports
System Output Jobs
System Data Pool

Figure 1. DM-1 System Overview

Programs may be written and compiled or assembled in the programmer's normal programming language. Calls to DM-1 data management services are made as executive service requests with arguments transmitted by the language processor as literals. These are executed interpretively by the DM-1 resident service package.

2.3 Data Storage and Retrieval Services

A program reads data from the data pool and writes data into it by calling on routines of the Service Package. The Service Package is analogous to an input/output control system. It contains a resident interpreter and a set of service routines. The routines are reentrant, so that they serve more than one user at a time.

The program may retrieve an item by giving the formal name assigned by the programmer or the actual data pool name and supplying a buffer to receive the data. If a formal name is used, the system translates it into the actual item assigned to it by the job description or the job-run request. An item may be written into the data pool in a similar way. The program places the data in a buffer and calls the appropriate service routine, giving the formal name or data pool name of the item to be written.

When a program reads or writes parts of the same item repeatedly, it may initialize the system by opening the item for reading or writing. The translations to internal item identifiers are accomplished when the item is opened. Later operations on the item are more direct. This service is especially valuable when reading or writing the records of a file.

The significant characteristics of the storage and retrieval services are the system's ability to transform item structures and the use of an invariant name in the program. The name remains the same in the program, no matter which items of the data pool are bound to it for a given run. It is completely independent of the location of the item or of the characteristics of the storage devices.

2.4 Operational Use

DM-1 operates in association with an operating system which controls a multi-programmed environment. The operating system recognizes DM-1 jobs as a class. When a DM-1 job is requested, the operating system ensures that the Service

Package is in memory as part of the operating system's input/output control package. One set of Service Package routines can serve any number of jobs in a time-shared mode.

Figure 2 depicts the relationships among the system components in operational use. A DM-1 job is requested by keying the job-request message (command) at the console and signaling the operating system that a DM-1 job is to be executed. The operating system assigns memory to the job according to its own scheduling algorithm.

The first step in a DM-1 job is accomplished by the Request Processor. This is a part of the DM-1 Supervisor which interprets the job request, prepares the job parameters, and initiates the job. The Request Processor uses the job name to retrieve the job description from the library. It develops a task list containing the identifiers for the sequence of component tasks (jobs or programs). Using the job images from the job description and the input and output specifications from the job request, it builds the input/output binding lists to relate the parameters of each task to data pool items. Any item transformation or conditional selections required in relating actual items to formal parameters are scheduled by the Request Processor.

The Job Manager is the part of the DM-1 Supervisor that controls execution of the job tasks. It reads the next task from the task list, loads it, and gives it control. The task maintains control until completion, unless the operating system's scheduling algorithm interrupts it. When the task is completed, control returns to the Job Manager to load the next task. The last task in the task list is the Request Termination task, which performs final housekeeping and terminates the job.

While a task is operating, it uses the Service Package to retrieve and store its data. The Service Package translates the formal parameter names used by the task to data pool items by using the binding lists. Data retrieval and storage are accomplished by the Service Package, which uses the system directories to locate and interpret the data pool items.

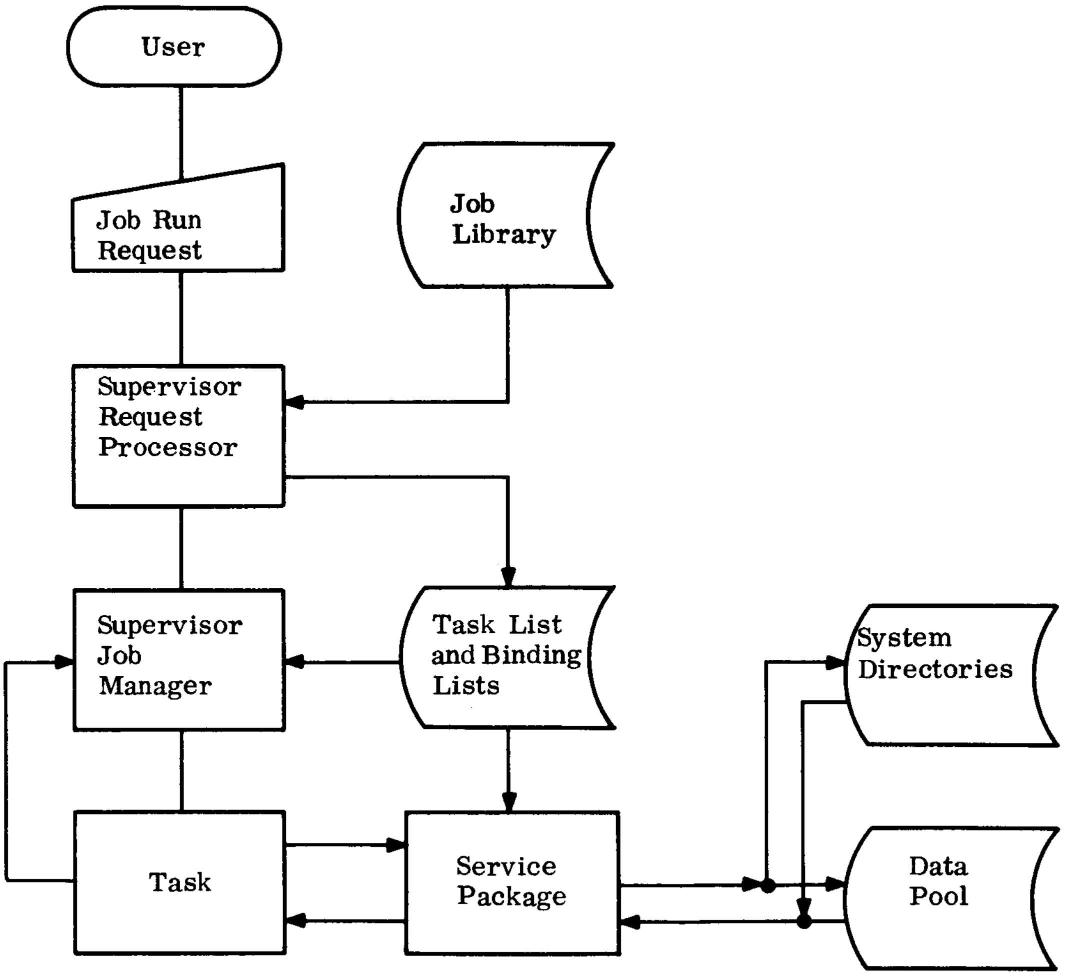


Figure 2. System Operation, Block Diagram

2.5 System Program Modules

The DM-1 design and implementations are based on modular construction of the system. The interface between the modules is maintained by a standard interface mechanism (i.e., table) that links an operating program to the system's data services.

3. STRUCTURES

DM-1 was conceived as a data management system to describe and process complex data structures. The environment of data acquisition and processing operations involving a large-scale data base processing calls for a system that is able to describe and process data in virtually any logically structured form. Because of this requirement, DM-1's design focus is on a set of mechanisms (i.e., directories and directory manipulation jobs) to describe and process multi-level hierarchical structures of data. The tree-structure representation of data noted in Figure 3(a) is one subset of the hierarchical structures processable under DM-1.

3.1 Data Items

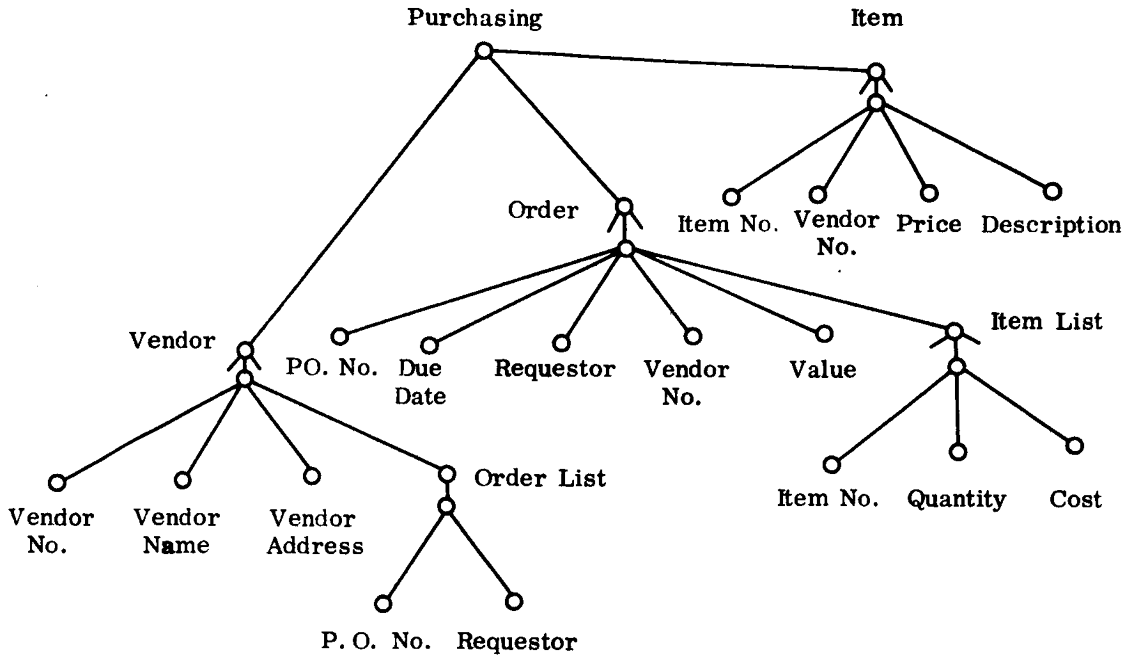
The generic term "item" has been applied by DM-1's designers to represent several types of structural entities that are combined to form a working data base. The items represented by the system fall into several classes:

(1) Field. A field is a terminal item; that is, it contains no substructure. It is defined to the system by its name, type, size, and units that its contents represent (e.g., volts, amperes, etc.)

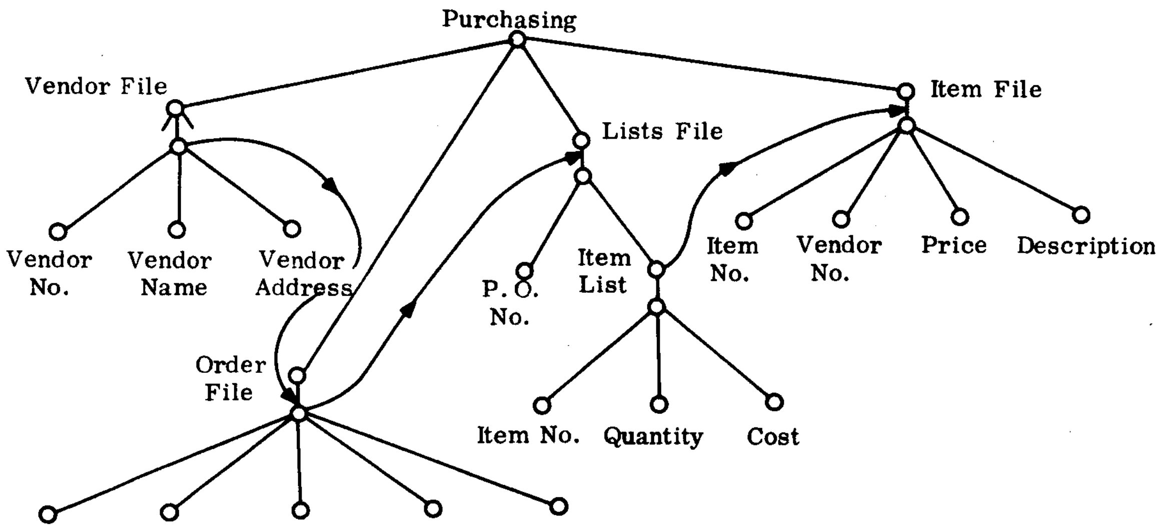
(2) Statement. A statement is an item which subsumes other items. Its subitems may be fields, files, or other statements. The statement is a mechanism for associating several items to show their relationship and to permit the system and the user to treat them as a named unit.

(3) File. A file is an item which subsumes an arbitrary number of subitems, each of which has an identical structure. Its subitems are called records.

(4) Record. A record is the subitem of a file and is defined to the system when a file is defined. It is a "logical" item unrelated to physical blocking.



(a) Pure Tree Structure



(b) Linked Network

Figure 3. Tree and Network Diagrams for the Purchasing Item

(5) Null Node. A null node is a terminal item representing a position in the logical structure. It is a means of reserving a location in the logical structure of the data base.

(6) Link. A link is an item which logically subsumes a set of items on another stem in the tree structure. It permits a logical connection between two branches of the tree and gives the structure a "lattice-like" network character. See Figure 3(b) for an example of a linked structure.

3.2 Structural Generality

The design of the system prescribes virtually no limit to the size or the structural complexity of the data base that it may process. The system will process data recorded on both magnetic tape and random access devices. Because the directory system for DM-1 is global (i.e., all data bases that it processes may be described to and retained by the system directories), it is possible to retain several levels of physical data storage that are called by the system when a program or query has requested a particular file. That file need not be on-line to be processed.

The degree of file structure nesting within DM-1 is not limited by its design. An implementation consideration has limited the number of items that may be directly subsumed by one item to 255. Each of the 255 structures may represent a separate file, statement, or field that is required by the parent item during processing. This restriction does not limit the size of a file, i.e., the number of records it may have.

3.3 Directory Structures

The structural description of the data is maintained by DM-1 in the system directories. The primary directories are the Item List, the Term List, and the Term Encoding Table. These three directory structures are used to focus on the data in the data base and to describe its structure. Processing of the data proceeds with the use of these tables.

(1) Item List. The Item List is a file with a record for each item (node) in the data pool structure. The records are ordered by the internally-used Item Class Code of the item. Each entry in the Item List contains the item type and the size of the item. The size of records and statements is the number of subitems they directly subsume. The size of a field is the number of units (e.g., bits, bytes, etc.) that comprise its physical length.

(2) Term List. The item names and units are maintained in a Term List file, which is parallel to the Item List file (i.e., record-for-record).

(3) Term Encoding Table. This table is retained for each item in the data pool so that the system can translate the item's name (externally used) to its internally coded form.

An optional feature of DM-1 is the ability to index selected fields. When a field is indexed, the system maintains a subsidiary directory table relating the values assumed by the field to the numbers of the records in which those values occur.

Indexing provides a tradeoff between the speed of retrieval and the size of storage required. When a field used in a conditional statement is indexed, the system can focus rapidly on the pertinent items without searching the data stream.

The DM-1 facility for data protection employs two separate, but interacting mechanisms: the security level and data access/modification rights. Each data item class is assigned a security level for access and another level for modification. Likewise, each user receives a clearance level which gives him access and modification rights to all items at and below his level.

4. LANGUAGE

DM-1's language considerations are broad in scope. The system is designed to interface with programs written in procedural languages (i.e., assembly language, COBOL, FORTRAN, PL/1) and, in addition, provides a set of languages for its operation and maintenance.

4.1 User Languages

The principal language provided with the system is the Command (Job) Language.* All other operational languages are subsumed under, and controlled by, the DM-1 Command Language:

(1) Program/Job Specification. The language provided for this facility is the communication medium between the systems Program Directories and Job Directories and the

* See Appendix B for a discussion of DM-1 Command (and Query) Language.

user. A program is described to the system by an indication of its name, input parameters (literal string or data base item), and output parameters (data base item). This description is retained by the system, and the code for the program is maintained by the host operating system. DM-1 and the operating system, working together through the DM-1 Command Language, perform the functions of binding the programs together to form "jobs." Job specification proceeds with a language like the Program Specification Language, but includes the programs that are to be bound to form the running job.

(2) Program Execution. This language invokes job specifications and item definitions; it also establishes a control link between the job and the host operating system. Because most of the elements of a job specification are retained in the DM-1 directories, the format and use of the language are extremely simple.

(3) Data Definition. The directories that focus on the data in the data pool are established and maintained through a generalized DM-1 job that processes the data definition language. The specification of data base structures may be effected through a console or from a card reader using a data definition form.

A DM-1 data definition can be written in an indented outline format which follows directly from a tree-like representation as shown in Figure 4. This figure shows the structure illustrated in Figure 3(b) in standard DM-1 form. Rectangular boxes represent files, hexagonal boxes represent records, and oblate ovals represent statements. A link item is a statement with a source or target pointer at one end.

(4) Query. The DM-1 Query Language operates as a subset of the Command Language. Provision is made in the language for complex specification of retrieval conditions, data reduction (e.g., sort) operations, and output formatting.

4.2 Programming Languages

DM-1, as mentioned previously, interfaces with the programming languages commonly used. In addition, complex data management functions are provided with the system as generalized DM-1 jobs that may be bound with the application code to form a working program. These generalized jobs take the form of, for example, Update, Retrieve, and Seek. In addition, the DM-1 program/job directory system, operating in conjunction with the Command Language, provides the facility for the

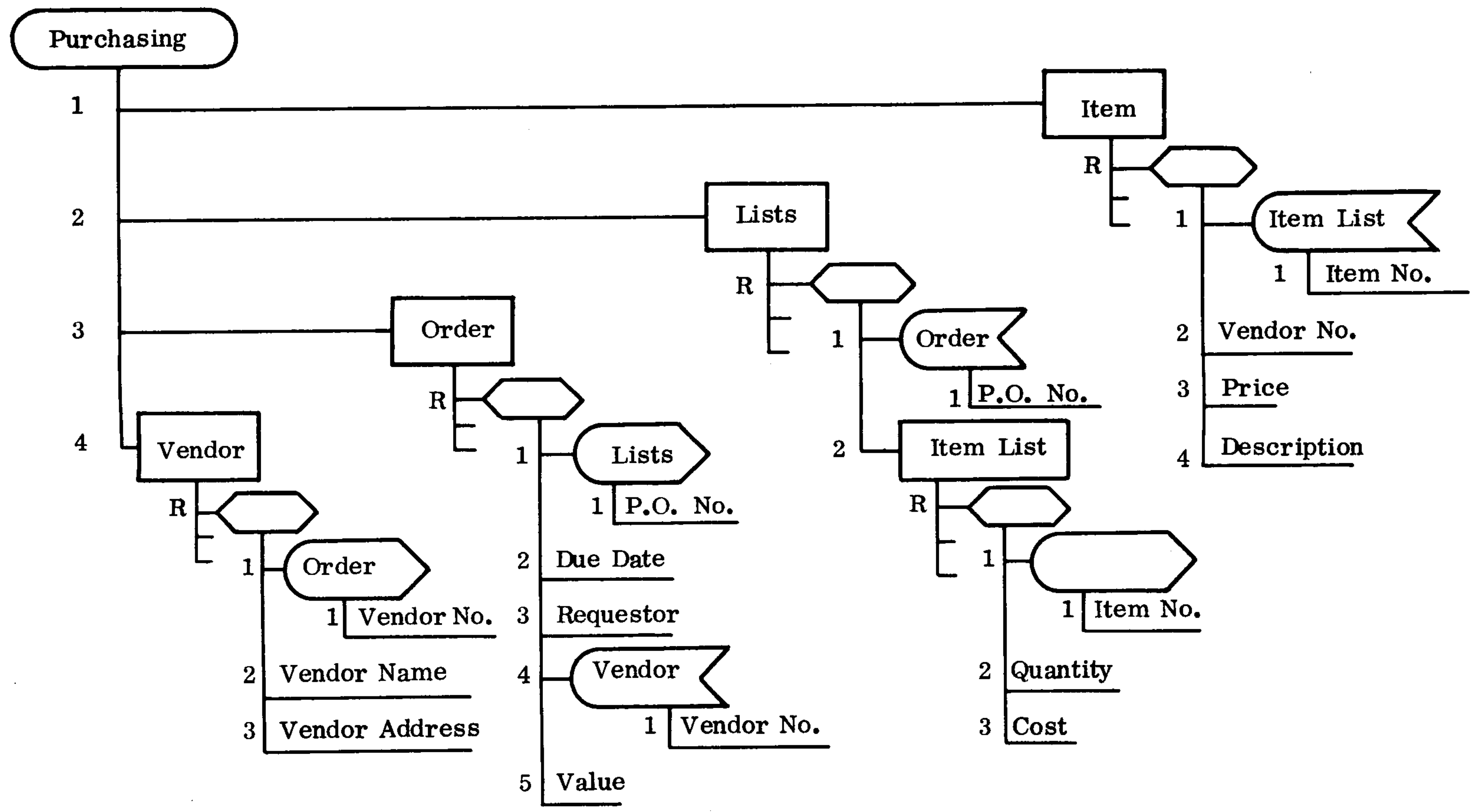


Figure 4. Purchasing Item with Links

user's continuing development of the generalized job (or program) to fit the needs of the installation. This facility would not be available were it not for the system's focus on generalized services.

5. EVALUATION

5.1 DM-1 Design

The third generation of computer systems and the forthcoming time-sharing oriented system are not proceeding in an ideally evolutionary way. The problem of providing a workable computer-user environment for data base systems is fast becoming a major sub-area of computer design and development. By intent, many of the mechanisms of this generation are becoming too complex for the average programmer to conceive of, let along manipulate in a meaningful manner. The first and second generation utilization of the computer has been primarily to solve the clerical aspects of work...and provide straightforward and simple calculational facilities. The third generation, however, brings with it the problems associated with a changeover from the simple to the complex. Data recorded and manipulated in a hierarchical form is not as conceptually simple as it used to be. It is from this point that the DM-1 designers evolved the system. It is from this point that the system's utility can be seen: the separate management of data and programs with a mechanism for linking data to programs and jobs will, by intent, extend the life of the programs. The management of data, however, implies the ability for flexible description and processing of that data. The evolution of data from one format to another, the addition of fields to records, the transformation of a recording format from integer to floating point, for example, are all characteristics of the environment for which DM-1 was developed.

Once the above environment is assumed, it is not difficult to conceive of a system and a design that will fulfill the needs of that environment. DM-1, with a few exceptions that are being corrected through implementation, has apparently fulfilled the needs of that environment.

The directory system, with its associated optional extensions to incorporate data field indexing and data item linkage, provides an associative mechanism that does not begin with the problems of the data's recording format. It does not begin with these problems because the scope of the system's design assumes the interpretive task of translation between the logical format and the physical format.

The concept of generalized programs is another design characteristic that should prove extremely useful in the development of software and application systems during the third and succeeding computer generations. One of the principal concerns of every programming or systems manager is the cost involved in the clerical aspects of programming. The same logical functions are repeated by application programmers utilizing varying algorithms. Generalization in DM-1 can provide both a means for reduction of the application system development effort and a mechanism for programmer standardization.

5.2 Implementation Progress

As with other systems of comparable complexity, DM-1 implementation is not proceeding as fast as the designers may have hoped. With an associative directory system as complex as that involved in DM-1, it is sometimes difficult to accept the extended times required to retrieve items from the data base, but the implementers have made specific attempts to upgrade the operation of the system. Speed, as well as flexibility, has now been written into the basic system design as the result of timings taken during test. The system's basic structure, however, still stands.

5.3 Ease of Use

The system, as conceived, is easy to use. The designers' documentation has focused on the system's mechanism and has not stated the specifics of those system features that make it easy to use.

The programmer interface through a non-special language is an obvious feature that does not demand that the programming staff be retrained to use the system.

The DM-1 Command Language goes far in replacing the complex parameterization job languages that have evolved with the third generation of systems.

5.4 Versatility

The two system features of separate data management (through the directory system) and generalized jobs (manipulated through the program/job languages) make the system versatile by intent. The separate management of data and programs implies the facility for changing (or evolving) either entity without the corresponding and expensive problems of program modification.

Another area of versatility involves modifications to the data described in the directories. The optionally-used facility to link data base structures provides the mechanism for cross-file retrieval and processing.

5.5 Economy

The versatility just described carries with it a price tag. DM-1 is an interpretive system and, as such, operates as a translator of program-data requirements, among other functions. One of the principal concerns of the implementation team has been the reduction of the time required to process records through DM-1. It has succeeded in modifying the internal mechanism structure (i.e., routines and linkages) to optimize the record or item retrieval time. The team is giving further consideration to providing the often-used data services routines through read-only memory; another attack has been through the combination of fields in the program-data linkage mechanism to reduce the number of instructions executed with a program's data request.

Another economic consideration was mentioned; that of reducing application system development time through the use of generalized services and user-coded function programs. It is here that DM-1 finds its principal economy. As the problems of interpretive processing time are solved, the utility of the system will increase. The generalized program manipulation facility will be used to produce application systems in shorter time frames, thereby bringing them into the computer at a quicker pace...to be modified and developed through the changing application at a pace that meets the needs of the application.

5.6 Responsiveness

The system's principal design, as stated previously, incorporates the mechanisms for responding to the requirements of several types of system "users." DM-1 concerns itself with operations personnel and analysts (through the Command Language), programmers (through its program and job manipulation system), system designers (through the ability to combine the system's generalized programs), and managers (through the system's Query Language).

If the system is finally delivered in an operational state that reflects its designers' intent, DM-1 will be one of the most powerful generalized data (or environmental) management systems conceived. Whether, in fact, it will perform as specified remains to be seen. The system's basic architecture has been demonstrated as implementable and usable; the system's full scope has yet to be demonstrated.

6. BIBLIOGRAPHY

(1) Sable, J. et al. Design of Reliability Central Data Management Subsystem. Final report. AUERBACH Corp., Philadelphia, Pa., July 1965, vol. 2. (RADC TR-65-189-Vol 2) (AD-469 269)

(2) Sable, J., W. Crowley, M. Rosenthal, S. Forst, & P. Harper. Reliability Central Automatic Data Processing Subsystem. Vol. 1: Design Specification Report. Final report. AUERBACH Corp., Philadelphia, Pa., Aug. 1966, 131 p. (Report no. 1280-TR-Vol-1) RADC TR-66-474-Vol-1 (AD-489 666)

(3) Sable, J., W. Crowley, M. Rosenthal, S. Forst, & P. Harper. Reliability Central Automatic Data Processing Subsystem. Vol. 2: Design Specification Report (Cont'd). Final report. AUERBACH Corp., Philadelphia, Pa., Aug. 1966, 577 p. (Report no. 1280-TR-Vol-2) RADC TR-66-474-Vol-2 (AD-489 667)

(4) Sable, J., and J. Minker. Reliability Central Automatic Data Processing Subsystem. Vol 3: Data Management System Survey. Final report. AUERBACH Corp., Philadelphia, Pa., Aug. 1966, 50 p. (Report no. 1280-TR-Vol-3) RADC TR-66-474-Vol-3 (AD-489 668)

TIME—SHARING SYSTEM/360 (TSS/360)

1. GENERAL

TSS/360 stands for "Time-Sharing System" for the IBM System/360, Model 67. This system, currently being developed by IBM, is intended to support simultaneous interactive access to a system by several users at terminals as well as a batched job stream.

The important distinctions between the Model 67 and other S/360 models are two capabilities which are needed to make time-sharing a practical operating concept:

(1) Virtual Memory Addressing

The programmer may address 2^{24} bytes of virtual storage. This virtual store is organized as 16 segments, each segment consisting of 256 pages of 4096 bytes each. Each user is permitted to use as much of this space as necessary. Virtual storage addressing is made practical by the base register method of address computation (available in all S/360 models), and by eight high-speed associative registers which hold the most current logical-to-physical page address translations.

(2) Multiprocessing

One or two central processors share jointly addressable core memory modules. Up to two channel controllers may be employed. Each communicates with both processors and all memory modules; they may share peripheral devices. (The formerly announced goal of four processors and four channel controllers has been retracted).

TSS/360 provides the user a set of on-line programming services through which he may prepare programs for use in the system, execute his own programs or system service programs, and maintain his own data structures within the system.

A conversational FORTRAN IV compiler and a conversational symbolic assembler will be available for interactive program preparation. The terminal command language is the link between the user and the system services available to him, and is the vehicle by which he creates and manages data within the system.

In such a system, the user at his terminal has certain advantages over the programmer using a batch-only programming system. The many steps involved in preparing a program or maintaining data bases are carried out in a conversational mode and at the user's own pace. Since the user receives feedback as he carries out his steps, he can verify his actions, minimizing the number of iterations normally involved in these tasks, and in general make the most effective use of the programmer's time.

A disadvantage to the remote terminal user in TSS/360 is that he is limited to conversational FORTRAN IV and assembly language for coding his programs interactively. (Conversational PL/I has been retracted and COBOL, previously limited to batch only, has now been withdrawn altogether.)

(Other limitations in TSS/360 are the withdrawal of a remote job entry facility and, compared to other S/360 operating systems, the withdrawal of SORT/MERGE, graphics support, checkpoint restart, and support of the 2321 Data Cell Drive and the 2260 Display Station.)

2. THE EXTERNAL FILE SYSTEM

2.1 Introduction

Data Management in TSS/360 is the facility for the physical management of data, and thus meets our definition of an External File System. The Data Management facilities include three "access methods" for use by problem programs, and system services for the on-line user through a command language.3*

2.2 Identification of Terms

The following terms should be clarified before proceeding:

(1) Volume

A physical unit of secondary storage media, e.g., a disc pack or a reel of magnetic tape, is called a volume.

(2) Sequential

A volume is sequential if, when it is in use, the read or write mechanism is in contact with only one storage position, and may reach other points by only one path through storage. Examples of sequential volumes are magnetic tape and paper tape.

(3) Direct-Access

A volume is direct-access if its access capabilities are more precise than those of a sequential storage device; for example, data cell, disc, or drum are direct-access volumes. The improvement in access time may be due to:

(a) mechanisms which are in contact with many storage positions simultaneously

(b) addressing, which enables the access mechanism to be placed in the ballpark (cylinder in the case of a disk) of the data

(c) a combination of the two. Direct-access generally denotes direct addressing of a block rather than a byte (as in core memory).

(4) Data Set

The symbolically referenced data entity in TSS/360 is called a data set. A data set is a collection of one or more records; it may subsume other data sets or be part of another data set. A data set is "in" the system if it is stored on a volume.

*Superscript numbers denote references in the bibliography.

2.3 TSS Data Structure

The most important goal of an external file system is to provide symbolic referencing of data entities in order to shield the user of these services from hardware considerations and burdensome "housekeeping" chores. Thus, a discussion of how data is structured and what elements in the structure are symbolically referenced is in order so that the user's interface can be properly presented.

2.3.1 Data Set Names. The user assigns a symbol (name) to each data set. The data sets are strictly tree-structured,* and the name of any data set carries the names of all its supersets, thus setting up a symbolic identification of its superstructure. The composite name takes the form of simple names separated by periods, e.g., VERTEBR. MAMMAL. CANINE. DOGS. This composite name has 26 characters; the limit is 35 characters and 17 levels.

In this example, VERTEBR. MAMMAL. CANINE, VERTEBR. MAMMAL, and VERTEBR are also data set names. The system appends a user ID to each data set name; these ID's are kept in the system catalog.

Provision is made for alternative names, called aliases, to be assigned to data sets. This feature is convenient when a data set is shared among several users, each of whom may wish to choose names which are most meaningful in the context of his problem.

2.3.2 System Catalog and Volume Table of Contents. The System Catalog is a special data set maintained for on-line direct-access devices. It represents the directories of the data structure, and is the bridge between a data set name and the volume on which it resides (and the relative location on the volume if magnetic tape). The restrictions on user access to data sets also reside in the system catalog, which comprises a master index and the user's catalog. The former lists the users currently active in the system. The users' catalog contains the indexes of the defined data sets to whatever level their names imply.

*In the material reviewed there is no mention of linking between data sets stemming from different branches or roots, i.e., lattice points.

In the example, the full data set name might be JOEBLOW. VERTEBR. MAMMAL. CANINE. DOGS. The master index lists JOEBLOW as a user and points to the user catalog address which lists his data sets. In the first level index there will be an entry for VERTEBR, and this entry will point to a second level index, containing entries for MAMMAL, MARSUPIAL, etc. The entry for MAMMAL will locate the list containing CANINE and FELINE, etc.

At the lowest level, the entry for DOGS will identify the volume containing the data set. Each volume has a volume table-of-contents (VTOC). In the example, one entry in the VTOC will exist for the data set JOEBLOW. VERTEBR. MAMMAL. CANINE. DOGS, and will point to the physical starting address of the data set for that volume.

Provision is made for convenient naming of a data set type which may have successive versions generated and present in the system, e.g., weekly payroll data sets. This type is called a generation data group; the data sets retain the same name but are assigned generation numbers, with zero representing the most recent generation.

2.3.3 Data Set Organization. The data sets are of two types:

(1) Physical Sequential. The unit of data exchange is the block, which comprises one or more logical records intended for processing by OS/360. Also, sequential volumes (magnetic tape) are restricted to this organization.

(2) Virtual Storage. The unit of data exchange between the direct-access device and virtual memory is the page (4096 eight-bit bytes). There are three organizations of Virtual Storage data sets:

    (a) Virtual Sequential. Logical records are arranged in the order created, and assigned to blocks (pages). Reference made to a block by a user program results in return of the retrieval address. Thus, within a block, the user programmer may supply his own direct-access criteria or his own indexing when Virtual Index Sequential (discussed next) does not meet his needs.

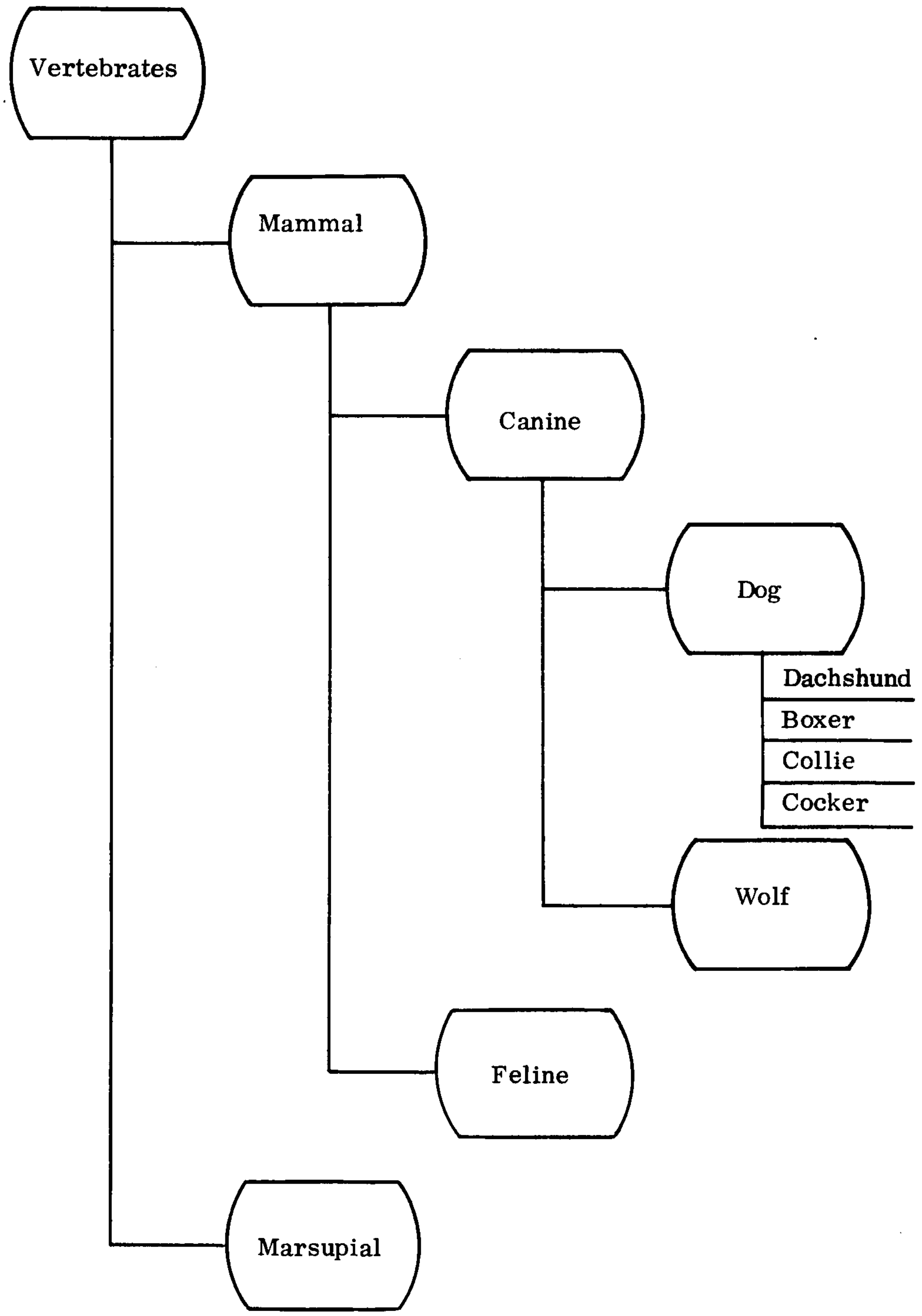


Figure 1. Sample Data Set Structure Representation

(b) Virtual Index Sequential. Key data with each record locates the record or locates overflow record key data. A page directory contains the value of the key in the first record of the page. The user may allocate data set records to pages in a way that leaves room for convenient expansion, thus more efficient operation. (Otherwise, overflow records are created, and more steps are required during retrieval.)

(c) Virtual Partitioned. Data groups organized separately under the previous methods may be combined into one data set. The separate groups are called "members" and may be mixed within the virtual partitioned data set. The name of a member then consists of the name of the virtual partitioned set followed by the name of the member enclosed in parentheses. The first entry in the data set is the partitioned organization directory, which locates and describes the members.

2.4 The User's Interface

The user's interface to the data management facilities is at two levels:

(1) the macros employed by his programs during execution for access to the data sets and related needs

(2) the command language through which the on-line user may reference his data sets or call upon programs that do so.

2.4.1 Access Methods from Programs. Several access methods, corresponding to the types of data set organization, are provided for program input/output.

For each data set used, a data control block (DCB) must be established within the problem program. The DCB contains various parameters describing the data set and the way it is to be processed. The actual processing of the data set begins with the issuance of an OPEN command and continues until the issuance of a CLOSE command.

The access methods and brief descriptions follow:

(1) Physical Sequential Data Sets

BSAM (Basic Sequential Access Method). BSAM is used for physical sequential data sets which are primarily on magnetic tape. The basic operations are READ and WRITE, which only initiate the reading or writing of a block. The user is responsible for his own buffering, but may use the CHECK macro to check for completion of the read and write operations. Also, GETPOOL, FREEPOOL, GETBUF, and FREEBUF macros are provided for allocating pools of buffers and individual buffers within pools.

(2) Virtual Storage Data Sets

(a) VSAM (Virtual Sequential Access Method). VSAM is used for virtual sequential data sets. The principal operations are as follows:

(i) PUT, used to store records. Records are stored in the order in which they are delivered. When a record is stored, the system gives the storage address back to the user, who may use it for later retrieval.

(ii) GET, used for sequential retrieval of records; i.e., a series of GET's will retrieve the records in the same order as they were stored by the original series of PUT's. Both PUT and GET can be used in either move or locate mode. In move mode the user points to the record; in locate mode the system points to it.

(iii) SETL sets a pointer to the retrieval address of a specific record in the data set. Retrieval is then sequential from that point.

(iv) PUTX, used to replace a record retrieved by a locate-mode GET macro.

(b) VISAM (Virtual Index Sequential Access Method). VISAM is used for virtual index sequential data sets. The principal operations are as follows:

(i) PUT and GET, for sequential storage and retrieval by key; i.e., the records are delivered to or from storage in key sequence.

(ii) WRITE and READ, for non-sequential storage and retrieval. WRITE causes the system to insert the record at the proper point in the key sequence, but requires more processing than PUT. READ retrieves a specific record by key or retrieval address.

(iii) SETL sets a pointer to a specific record in the data set by key or retrieval address. Processing is then sequential.

(iv) DELREC deletes a specific record from the data set by key or retrieval address.

(c) VPAM (Virtual Partitioned Access Method. VPAM is used for VP data sets. The principal operations are FIND and STOW.

(i) FIND is used to prepare a particular member of the data set for processing. The processing of the member is then done via VSAM or VISAM depending upon whether the member is a VS or VIS data set.

(ii) STOW puts the resulting member into the partitioned data set and updates the Partitioned Organization Directory. The actual operation accomplished by STOW may be the addition, deletion, or replacement of the member; or the changing of the member's name.

2.4.2 Data Management Through the Command Language. The command language is the medium through which the on-line user communicates with the system. It is used for scheduling his tasks, preparing programs for execution, requesting information, and manipulating data.

Individual users may be granted access to a TSS/360 system by means of JOIN commands issued by privileged system personnel. Each user is provided with an identification code, a password, charge numbers, a priority level (for use by the scheduling algorithm), a privilege class (manager, operator, administrator, user), and a user authority level (user, privileged system programmer, non-privileged system programmer). A user's access rights to the system may be terminated by means of a QUIT command.

A conversational task begins with a LOGON command issued from the user's terminal and continues until a LOGOFF command is issued. A non-conversational task may be initiated from a conversational task by issuance of an EXECUTE command, which calls for execution of a command procedure that is prestored in the system as a data set. The command procedure must being with a LOGON and end with a LOGOFF command.

The portion of the command language which is of primary interest is that used for managing data. This capability is subdivided and presented in the following sections.

2.4.2.1 Data Set Creation and Related Functions. Nine commands are provided for manipulating the data set structure.

(1) CATALOG - Enter or change a data set name.

(2) DELETE - Delete a data set name (not necessarily the data set).

(3) DATA - Create a data set (data supplied from terminal or through card reader).

(4) MODIFY - Delete and/or insert lines in a VISAM data set.

(5) ERASE - Remove data set and its references (implies DELETE function).

(6) DEFINE DATA - Describe a data set for linkage to a program.

(7) RELEASE - Nullify previous DEFINE DATA effects; also release I/O device.

(8) CALL DATA DEFINITION - Refer to a data set containing DEFINE DATA commands and act on those.

(9) COPY DATA SET - Duplicate a data set.

The commands may result in a dialogue between user and system involving prompting, informative responses, or diagnostics until the function is complete.

2.4.2.2 Data Set Protection and Sharing. These commands are used to control user access to data sets:

(1) PERMIT - Grant other users access to one's data set, or withdraw permission granted previously.

(2) SHARE - Establish access to a data set belonging to another user who has granted it via the PERMIT command.

Without the PERMIT command, a data set is accessible only through the user ID associated with its creation.

Data set sharing may be specified at three levels; "read only," "read and write," and "unlimited." (Any user may ERASE as well as read and write.)

Protection mechanisms are provided to control simultaneous reference to data by different users. These interlocks, when activated by one user, delay any other reference to a data set which he is writing, and delay any other writing of a data set which he is reading. The setting and resetting of these interlocks according to level and the unit of data protected are shown below:

Level (Data Unit)	Access Method	How Set	How Reset
Data Set	All	OPEN	CLOSE
Member	VPAM	FIND	STOW/CLOSE
Page	All	GET/READ	CLOSE/Reference another page

2.4.2.3 Hard Copy Capability. Three commands are provided for dumping data sets for punching or listing:

(1) PUNCH causes specified data set contents to be produced on punched cards.

(2) PRINT causes printout of the specified data set.

(3) WRITE TAPE causes data set contents to be copied on tape in a format suitable for printing.

3. EVALUATION

3.1 Secondary Storage Data Addressing Capability

In the tree-structure of data sets within virtual storage, the ability to reference any branch symbolically is good, and the limit set at 17 levels should be sufficient for anyone. The (partial) ordering within the structure being implied by the symbol (name) of the branch is useful, both from the point of view of the user working without the help of an internal file system and from the standpoint of efficient operation, especially in changing the directories for additions and deletions.

Within a data set which is an indexed sequential file, logical records may be retrieved in the order stored, or in the order dictated by key data, as specified by the user.

3.2 Safety

(1) None of the material referenced mentions any backup and restore capability; surely some capability is mandatory. One can only conjecture that at least a manually initiated utility is provided for this purpose.

(2) The controls for restricting access to data sets seem adequate. A program which finds itself locked-out, however, has no way of knowing whether it is practical to proceed. Carrying version numbers (edition numbers) would improve this situation.

3.3 Data Structure Accessing and Manipulating

The limitation to a tree-structure is not as severe as it may seem on the surface, for other structures may be set up within that framework (although efficiency of operation is bound to suffer somewhat).

The directory structure allows for minimal overhead in deleting or adding branches. The data set name-to-physical address translation takes place when the data set is "OPENed", and the volume identification and physical address are used for subsequent references. As indicated previously, within the tree-structuring of the data, the operation seems well-conceived.

3.4 Program Interface

Detailed information is not available, but an IBM report[2] indicates a full set of standard services, including label checking. Worthy of note is the fact that one may build external file system services (with special data structures) of his own, and co-exist with TSS/360 access methods. The supervisor will accept channel programs for I/O through the Input/Output Request (IOREQ) facility.

3.5 User/Data Interface in a Time-Sharing System

For a system with no internal file system, TSS/360 provides about as much data management capability to the user as can be expected. The command language[3] seems easy to use. Notable is the facility to produce a hard copy of selected data. With no internal file system, any on-line query capability must be built by the user.

3.6 Degraded Operation/Operational Adaptability

In the material reviewed, no mention is made of fail-soft features or ability to cope with a dynamic reconfiguration.

3.7 Multi-Level Secondary Storage Management

This capability is claimed by Comfort,[1] with "activity" being the criterion for moving data up or down the storage device ladder. It is not clear how refined this capability was to be or what was meant by "activity." There is no mention of level changing in more recent material. Further, support for the 2321 Data Cell Drive is withdrawn. Further, although "direct-access device" is the term used in discussing data set management, the impression is given that this means disc storage, and that the drum is strictly for core-image programs and page-swapping.

3.8 Adaptability of Design

The design seems to place no limit on the growth of the data base, nor does it seem to suffer inefficiencies because of growth. The design seems highly dependent upon 360 equipment, however. Although much future IBM software will be coded in PL/I, it is assumed that 360 assembly language is used. The only user terminal supported is the 2741 Communications Terminal. Independency of equipment is difficult (perhaps undesirable) in external file system design, and is a more appropriate criterion for evaluating the internal file system design.

3.9 Support of Internal File System

For the portion of the data structure which should be physically tree-structured, the access methods could be used to advantage by higher level file system programs. For other structures one would have to build his own external file system, which in turn uses the IOREQ facilities of TSS/360.

In developing a powerful general-purpose file system for a time-sharing environment, the Model 67 and basic TSS/360 services and language translators would furnish fairly good support. But the file system per se is only a small subset of the structure that would be needed for the management of an on-going data base.

4. BIBLIOGRAPHY

(1) Comfort, W. T. "A Computing System Design for User Service." Proc. FJCC, (1965), pp. 619-626.

(2) IBM System/360 Time Sharing System, Concepts and Facilities. IBM Corp. (1966), Form C28-2003-0.

(3) IBM System/360 Time Sharing System, Command Language User's Guide. IBM Corp. (1966), Form C28-2001-0.

(4) IBM System/360 Time Sharing System, Command Language for Administrators and Operators.

BERKELEY TSS (BTSS)

1. INTRODUCTION

The time-sharing system developed at the University of California at Berkeley is part of a project (supported by the Advanced Research Projects Agency (DOD)) devoted to exploring and developing techniques in man/machine interaction. The time-sharing facility at Berkeley is now operational; it utilizes an SDS 940 computer, which is basically a copy of the SDS 930 with hardware modifications designed by the Berkeley group to make it more suitable for time-shared operation.

The designers[6*] state that their objectives were to construct a time-sharing facility for the following purposes:

(1) to develop and test some ideas in time-sharing a digital computer

(2) to develop a useful facility for a variety of experiments in man/machine interactive areas.

It should be emphasized that the time-sharing system, although general in nature, is an experimental system intended to give great flexibility and fast response to a limited number of users. (In particular, it is not designed to serve a large

*The superscript numbers indicate references in the bibliography.

number of users over a broad spectrum of problems, such as a utility approach to time-sharing.)

Each user of the Berkeley Time-Sharing System (BTSS) is given a "copy" of a slightly modified SDS 930 with 16K of fast memory. The "copy" differs from the normal computer in the following respects:

(1) an obvious impairment of some real-time capabilities

(2) the substitution of system I/O commands for machine I/O instructions

(3) the addition of software interpreted instructions, system services, and large-scale interactive subsystems, such as an editor (QED), computation system (CAL), assembler (ARPAS), debugger (DDT), and others.

2. SYSTEM DESCRIPTION

2.1 Hardware

The hardware configuration for BTSS is shown in Figure 1. The SDS 930 is a 24-bit, fixed point machine with one index register, multi-level indirect addressing, a 14-bit address field, and 32K words of 1.75 microsecond memory in two independent modules. Briefly, the modifications to the 930 are as follows:

(1) Addition of Monitor and User Modes. Since the role of the system monitor is unique among the programs which use the system, the SDS 930 has been modified to permit the use of a certain class of privileged instructions and unrestricted memory only for programs that are part of the monitor. Attempts to execute privileged instructions in user mode are trapped by the hardware. Privileged instructions are those which (a) affect peripheral equipment, (b) halt computation, or (c) interfere with rapid response to interrupt requests.

(2) Addition of a Hardware Mapping Mechanism. In monitor mode all memory references are absolute. In user mode, however, all core references are made through a hardware mapping mechanism so that an attempt to access an unassigned location results in a trap.

(3) Addition of System Programmed Operators (SYSPOP). This is an extension of the normal 930 feature called the programmed operator (POP). SYSPOP's and POP's are invoked by setting a bit in the instruction word. During

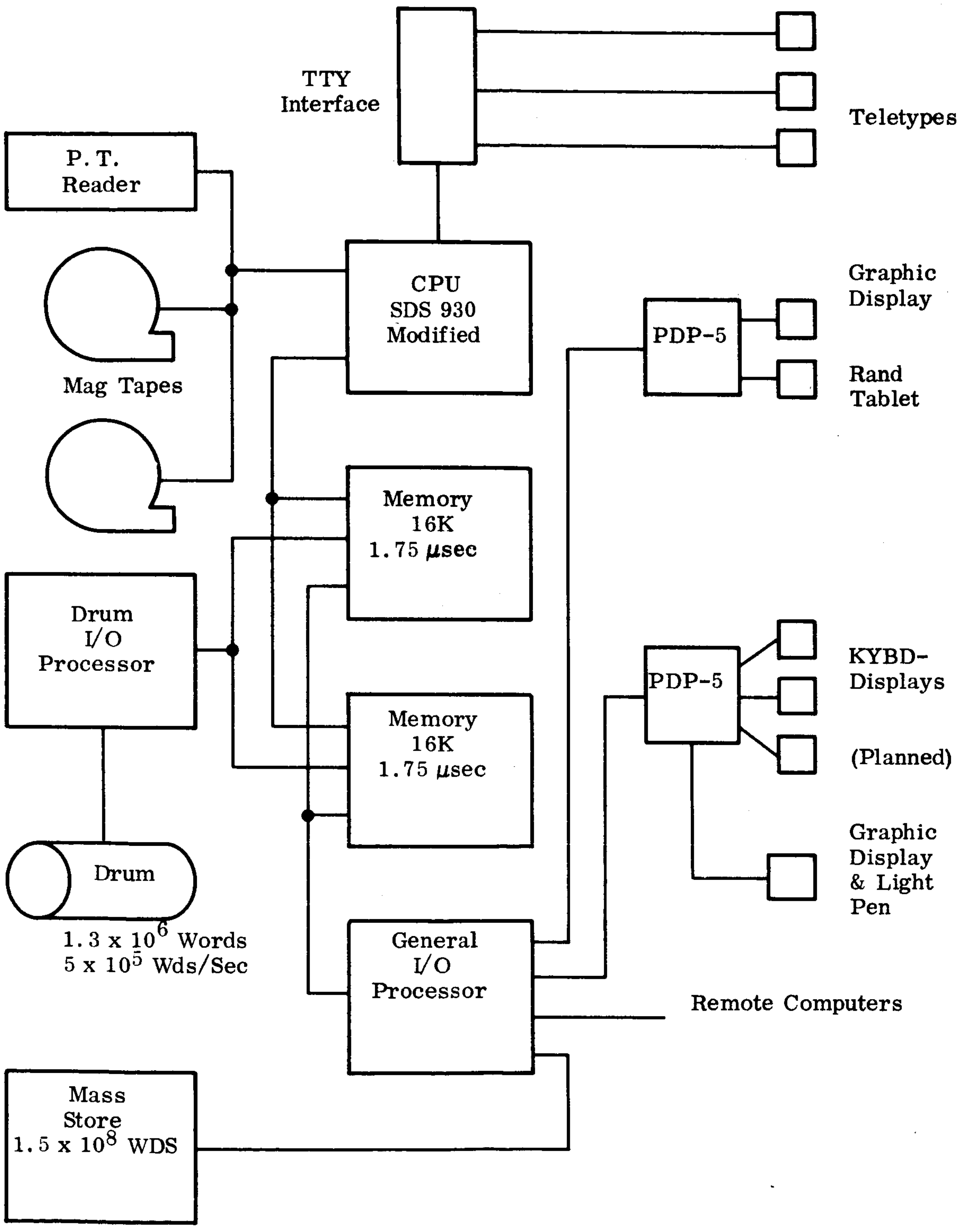


Figure 1. Configuration of Equipment

execution, the operation code of these instructions is taken to be the relative address in a transfer vector stored in a fixed location. Thus, these operators function as a special kind of subroutine call. The SYSPOP's, however, give the user 64 new "machine instructions," which are automatically executed in monitor mode and do not require him to allocate memory or provide linkages.

2.2 Monitor and Executive

BTSS provides two distinct levels of system supervisory software with which programs can interface. The lowest level, called the monitor (and operating in monitor mode), provides the basic time-sharing mechanism using a time quantum clock and a hardware page mapping technique so that each user has a "machine" which can be controlled by the execution of instruction sequences at the machine language level. The "machine" configuration is very similar to that of a standard medium-sized computer.

The next level of software, called the executive, operates in user mode and provides for multi-user access to the machine with an expanded set of software interpreted instructions which control memory sharing, execution of multi-process jobs, and an external file system. The executive controls access priority and scheduling, using a software storage map for each process which is independent of the hardware (core) map mentioned previously. It might be said that the Monitor (and hardware) provides the mechanism for time-sharing but the Executive provides the policy and strategy. This stratification of control into two levels (which does not include the programming system level with which the normal "non-toolmaking" user would interface) means that BTSS can be viewed as a general-purpose computer which executes machine level programs a little slower than the non-time-shared computer. (This distinction would be important only for real-time applications with critical timing constraints.) This view is especially important for programmers who are developing software tools for higher level users or perhaps modifying the executive system itself.

2.3 File System

2.3.1 Input/Output. The primary emphasis of the (software implemented) user input/output instructions in BTSS has been to make all input/output devices interface identically with a program. This results both in programs which are independent of the environment, and in a simpler implementation of the system.

Each input/output instruction names a file or is associated with a file previously named. Files are associated with physical devices by the system, and a file in BTSS is either sequential or random.

2.3.1.1 Sequential Files. A sequential file is a sequence of records of arbitrary length, each of which is made up of a string of cells of fixed length. The three instructions, CIO (character input/output), WIO (word input/output) and BIO (block input/output) specify the type of item considered as a cell in each case. Thus, input/output for a sequential file handles a string of "cells" of fixed size. The files may physically originate on teletype, drum, disc, or tape. A full-duplex physical interface is provided for each device (generally invisible to the using program). A character is 8 bits, a word 24 bits, and a block 256 words. The file is identified as input or output at the time it is opened.

The sequential file system for auxiliary storage (drum, disc, or tape) provides for files whose records are composed of one or more blocks. The location of these records is stored in an index file which contains a pointer to each block in the file in order, an end-of-record indicator and an end-of-file indicator for each block. Blocks of the index file are chained to provide the required capacity. Available space on the drum is managed with the aid of a drum map, which is a two-dimensional array of one bit cells. The 72 columns of the array represent the tracks of the drum, and the 64 rows represent the 64 256-word sectors (blocks) around the track. Thus, each bit in the table represents the availability of one of the 4068 blocks on the drum. If a new block is required, the system reads the rotational position of the drum, and searches that row in the table for the appearance of a 0. The column in which a 0 is found indicates the track on which a block is available. Because of the way the row is chosen, this block is immediately accessible.

2.3.1.2 Random (Addressable) Files. Indirect addressing of secondary storage is provided by a set of special instructions, in which the addressable cell is either a word or a block. The record is synonymous with a block in this case so that the end-of-record indicator is not meaningful.

2.3.2 File Commands. It has been mentioned that all input/output in BTSS is full-duplex*, a situation which permits, in particular, a unique reaction to users at a keyboard. The full-duplex character-by-character (CIO) capability means that the user can interact with his program at the character level. For example, this capability permits the following activities:

(1) The program can substitute characters (or strings of characters, including the null string) as echoes for characters (or strings) received.

(2) The keyboard can be used while information is being output on the display.

(3) Transmission errors can be readily detected.

(4) The system can recognize command and file names on the basis of the minimum number of input characters required to discriminate among alternatives.

The system can be set to three modes of name recognization, as follows:

(1) Expert. Minimum string recognition applies.

(2) Beginner. Full name is required if it is three characters or less.

(3) Novice. Full name must always be used.

Name recognition for file names can always be disabled by using quotes around the name. Files are "owned" by a particular user but may be declared to be either public or restricted. If a reference to a public file is made by other than the owner, it must be made in the following way:

(JONES) ABLE

where JONES is the owner of the file and ABLE is the file name. Reference to a private file must be made as follows:

(JONES, TERCES) ABLE

*Full duplex means that the information transmitted by the source is interpreted by the destination and retransmitted to the source over a separate channel.

where "TERCES" is Jones' password (called a group name). The string ",TERCES" will not be echoed by the system so that the typed display looks exactly like the first example.

If name recognition is enabled (expert mode), the system response to the preceding string would be as shown in Table 1. Names of new files must be designated by quotes.

TABLE 1. NAME RECOGNITION

User Types	System Action	Display
(	Name search initialized	(
J	Directory is searched for name beginning with "J". More than one is found.	(J
O	Only one JO... is found. System types "NES"	(JONES
,TERCES)	Password is checked. Only ")" is echoed.	(JONES)
A	Jones' directory is searched for file name beginning with "A". Only one is found. System types BLE	(JONES) ABLE

Some file manipulation commands in BTSS are as follows:

```
DEFINE <name> TO BE '<name>'. (defines a synonym)

DELETE <name>

LOAD    <name>

COPY    <name> TO <name>

DUMP ON <name>

FILE DIRECTORY (produces a listing of the user's directory.
```

2.3.3 Directory Structure. Access to user and file directories in BTSS is based on name randomization, a process which transforms the name into an address within a directory area known as the hash domain. Each entry in the hash domain consists of a string pointer (the addresses of the first character minus one, and the last character of the string) and a pointer to a description block (the "value" entry). The string is the name which has been randomized. The general format of the directories is shown in Figure 2.

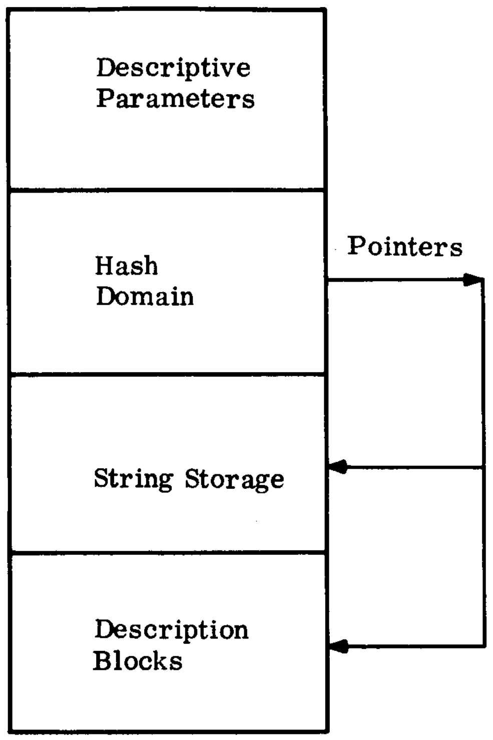


Figure 2. Directory Format

The descriptive parameters are the following items:

directory length

special group names (passwords)

tape system numbers

drum address for this directory

user number

tape parameters

In the user directory, the description blocks contain the following items for each entry:

pointer to hash table entry

drum address of file directory

maximum drum allowance

user status

hash code for his password

accounting information:

total computation time

log-in time

In the file directory, the description blocks contain the following items for each entry:

length

drum address of file

date last written

type: sequential or random

access restrictions: private or public

group accessibility (password)

status

3. EVALUATION

The designers of BTSS started out with what they said was an experiment to test some ideas in time-sharing and man-machine interaction. What they accomplished must certainly be regarded as a highly successful operational system. The following can be listed as its accomplishments:

(1) It has introduced some important innovations in second-generation hardware for time sharing; these compare favorably with what is being introduced in third-generation systems.

(2) It has an easily used interface with the user and has effectively supported the development of user-oriented subsystems such as an interactive editor, compiler, and computation system.

(3) It has been widely adopted by commercial time-sharing users.

The three-level structure of BTSS--monitor, executive, and subsystems--has contributed to the ease of developing user-mode software (including the executive and subsystems). The concept of providing an interface at the monitor/hardware level provides a general-purpose computer to the machine-language programmer and undoubtedly enhances the effectiveness of the system programmers who developed the higher levels of the system. This is an attractive principle which has a bootstrapping effect and perhaps should be more generally adopted in developing new time-shared systems.

The user interface (and system flexibility) provided by the combination of full-duplex operation and name recognition has much to recommend it. However, the structure of the system directories, particularly the use of randomized addressing in these tables, does not seem ideally suited to name recognition, which depends on the examination of partial argument strings and determination of whether they are unique. A table whose arguments were indexed (or chained) alphabetically would appear to be a better match to this function.

The structure of the sequential and random input/output operation is well conceived but there does not appear to be a need for, or an effective way to utilize, the very rudimentary variable record capability which is described. (None of the input/output instructions or file commands appear directed towards the access or processing of a variable length record.)

4. BIBLIOGRAPHY

(1) Lampson, Butler W. Interactive Machine-Language Programming. ARPA Document No. 30.50.11 (October, 1966).

(2) Lampson, Butler W. Reference Manual/Time Sharing System. ARPA Document No. R-21 (March, 1967).

(3) Lampson, Butler W. Scheduling and Protection in Interactive Multi-Processor Systems. ARPA Document No. 40.10.150 (January, 1967).

(4) Lampson, B.W., Lichtenberger, W.W., and Pirtle, M. W. "A User Machine in a Time-Sharing System," Proc. IEEE 54, 12: 1766-1774 (December, 1966).

(5) LeClerc, J.Y. Memory Structures for Interactive Computers. ARPA Document No. 40.10.110 (May, 1966).

(6) Pirtle, M.W., and Lichtenberger, W. W. A Facility for Experimentation in Man-Machine Interaction. Proc. FJCC (1965) and ARPA Document No. 40.20.20 (January 1966).

PREFACE TO MULTICS

The following description of the MULTICS file system is based on unofficial preliminary specifications which are understood to be subject to change and, in many cases, are incomplete. It is presented not as a description of a system to be implemented but as a set of concepts which are important to consider in formulating an overall framework for any data management system.

MULTICS

1. INTRODUCTION

MULTICS is a general-purpose, time-sharing system based upon the GE 645 computer, an extension of the GE 635 adapted to a time-sharing environment. The MULTICS software system, a joint project of GE, MIT, and BTL, is to provide a comprehensive set of programming services with a high degree of reliability, simultaneously to a variety of users, both absentee and on-line. This goal, along with an evolutionary capability, is the major influence in the design of the MULTICS system. In this line, most coding is being done in the PL/I language in order to maintain modularity and machine independence.

The MULTICS system incorporates segmentation and paging features which, because they are invisible to the user, aid in program writing, sharing of programs, and sharing of data under the control of protective mechanisms. Through the framework provided by the MULTICS systems, users can build a repertoire of commands and programs to facilitate their use of the system. MULTICS, as designed, provides the user with the capacity to construct a high-level programming system adapted to his needs.

The need for a versatile, on-line external file system in such an environment is one of the main concerns of the MULTICS system. In the environment of a large time-sharing system, a large-capacity, quick-access external file system is essential. Storage strategies must efficiently incorporate the concepts of information usage and multi-level storage hierarchy. This, in addition to recovery capability, controlled access, and machine-independent and device-independent manipulation, is the goal of the MULTICS secondary storage (external file) system.

The file system has two primary responsibilities in the MULTICS system, segment management and multilevel storage management.

Segment management involves the maintenance of directories; the creation, truncation, and deletion of segments; access control; and the retrieval of needed information. Multilevel storage management involves recovery back-up procedures, movement of files through the secondary storage hierarchy, and salvaging of unused storage space.

Before proceeding further into a discussion of the MULTICS file system, an understanding of certain terms is necessary.

● STORAGE HIERARCHY	- the ranking of addressable storage devices according to their access times and rates of data transfer
● FILE	- a linear array of entries residing in secondary storage
● SEGMENT	- a linear array of words or pages known by a symbolic name. In secondary storage a file and segment are equivalent. Segments may be paged or unpaged; unless paged, a whole segment must be brought into primary storage for processing.
● PAGE	- a subdivision of a segment (64 or 1024 words) which is known to the system but invisible to the user
● DIRECTORY	- a system file each of whose entries is the name of another file

2. STRUCTURE OF THE FILE SYSTEM

The file structure is basically a tree hierarchy of data and directory files across which links may be added to facilitate access to other files. To the file system, each file is a linear array of formatless entries of known size. The format is supplied by higher-level modules or by the user. Each user knows a file by a symbolic name and may reference any element of that file by the file name and the linear index of the element in the file. At the root of the tree structure is the root directory which is pointed to by the file system and is unknown to all other directories (see example in Figure 1).

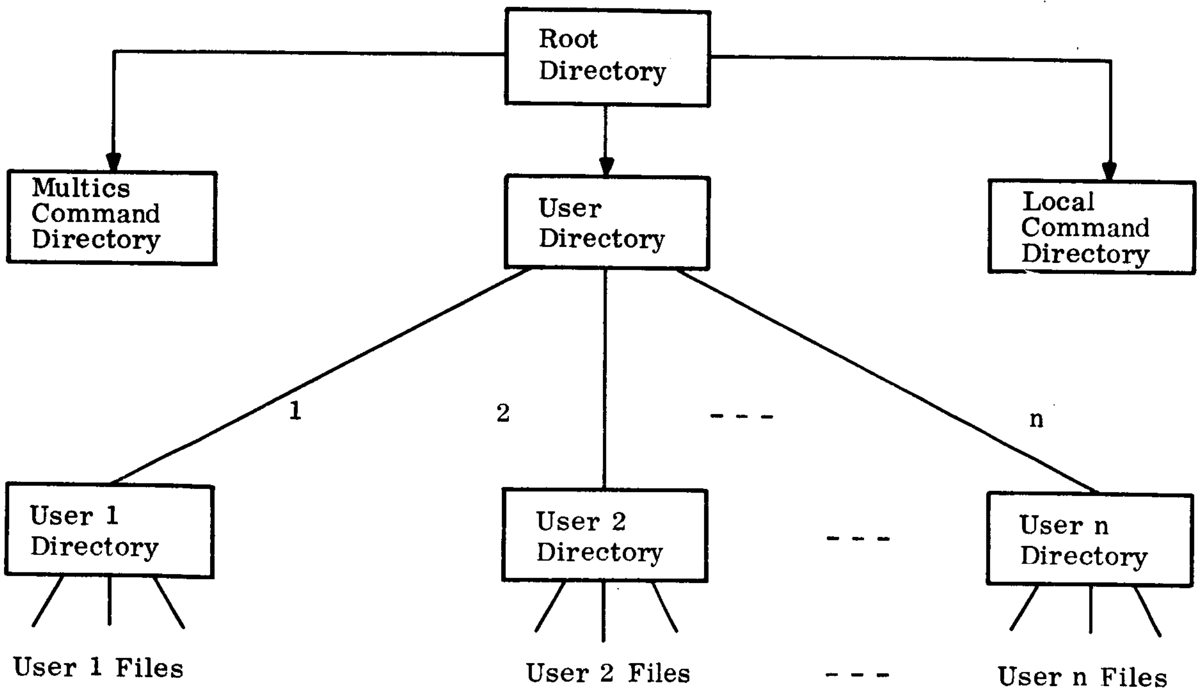


Figure 1. Example of Directory Structure

Each entry in a directory is the symbolic name of a file and need be unique only in the directory in which it occurs. The entry is a pointer either to another entry (in the same or another directory) or to a file. Pointers to entries are called links, and pointers to files are called branches. Links have no purpose other than pointing to an entry. A branch contains descriptive information such as the physical address of the file, the time the file was last modified, the time the file was last used, the access control information, and the current status of the file.

With such a complex structure in a heavily used system, it is desirable to keep users from encountering large amounts of directory searching. As a result, the file system is so arranged that each user works in one directory at one time; one such scheme appears in Figure 1. More than one user can have the same working directory, and the inferior files of one directory can be inferior files of other directories, i.e., common files. Any node (file) in the tree structure can be named by the sequence of entries needed to reach the file from the root. In Figure 2, the names for files 1, 2, 3, and 4 are D:F, D:F:H, D:E, and A:C, respectively; the colon is used to separate entry names. If one adds the asterisk to mean the level of immediate superiority and adopts the convention that names relative to a working directory have an initial colon, then naming can be accomplished in terms of an arbitrary directory.

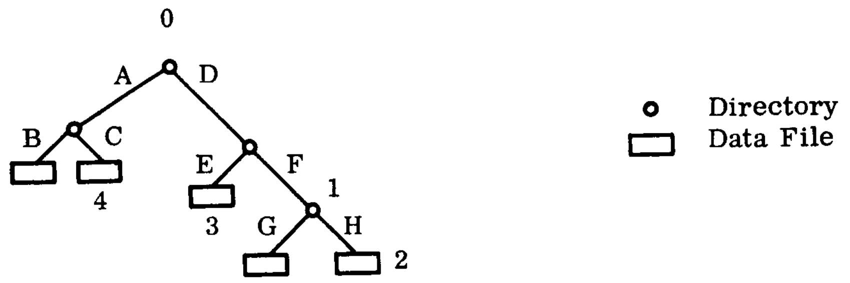


Figure 2. Tree Structure Example

In the preceding example 3 relative to 2 would be :*:*:E, and 4 relative to 2 would be :*:*:*:A:C.

If one allows links to be established between entries, then files may have different names for different users. Since the file structure restricts each node to only one immediate superior, links provide the facility for common files without duplication. Consider the example in Figure 3.

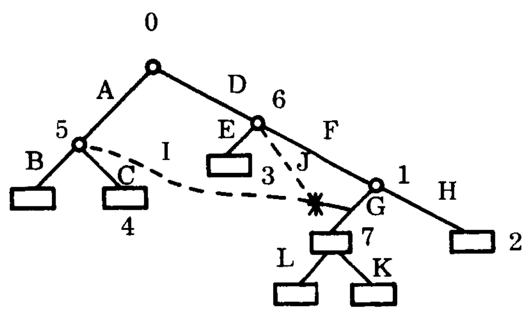


Figure 3. Tree Structure with Links

The entry J in directory 6 is a link to the branch G in directory 1. The entry I in directory 5 is a link to the entry J in directory 6, thus acting as a link to G in directory 1. A pathname (analogous to the tree name of Figure 2) relative to the root for directory 7 can be D:F:G, D:J, or A:I. The pathname similarly may be defined relative to the working directory if the link is included in the concept of superiority. As an example, suppose the working directory has the pathname A (directory 5); a command such as CHANGEDIRECTORY :I results in a new working directory 7 with pathname A:I. The command CHANGEDIRECTORY:* would result in returning to pathname A (directory 5).

3. ACCESS CONTROL IN THE FILE STRUCTURE

Access to files in the MULTICS file system is controlled by the access control module (ACM). Whether a certain user has the right to access a given branch is determined by the mode of that branch with respect to the user. To determine this, an ACL (access control list) is maintained with each branch containing the list of users with their mode. (The ACL can be modified, and is maintained in a non-redundant form.) The mode consists of five attributes, each of which can be on or off: trap, read, write, execute, and append.

The trap attribute is the most significant and is checked first. If the trap is on, the ACM calls the procedure which is the first entry in the trap list (TL) associated with the user in the ACL that caused the trap. The procedure, using pertinent information from the branch and the user reference, specifies the values of the other four attributes. The initial state of these four attributes (the usage attributes) is called the apparent mode; the final state is called the effective mode and is dominant. If the trap is off, the apparent mode is the effective mode. By means of such commands as LOCK FILENAME, KEY, and UNLOCK FILENAME, the user can create a trap which excludes or partially restricts all users who do not know the KEY.

The usage attributes, if ON, allow the user to perform the indicated operations on the file pointed to by the branch. If the file is a directory, the meaning of each attribute is accordingly different. The meaning of read and write is clear. Execute means to execute the contents of a file as a procedure and, in the case of a directory, to search it. Append, in both cases, indicates the addition of new information without altering any of the original contents.

Thus, access is dependent upon the source of the access attempt and the mode of the user with respect to the particular branch involved. Hence, if a file may be reached in more than one way, the access along each path can be different.

4. THE FILE SYSTEM PROGRAM STRUCTURE

Whenever reference is made to a segment by means of symbolic name or segment address, the file system must make the segment available to the user's process. Figure 4 is a block diagram of the modular structure of the basic file system which describes the general flow of action.

With the exception of system interrupts and errors, the user calls the file system in two ways:

(1) The File Coordination Module. To manipulate entries in the user's working directory

(2) The Segment Management Module. To make segments available to a user's process.

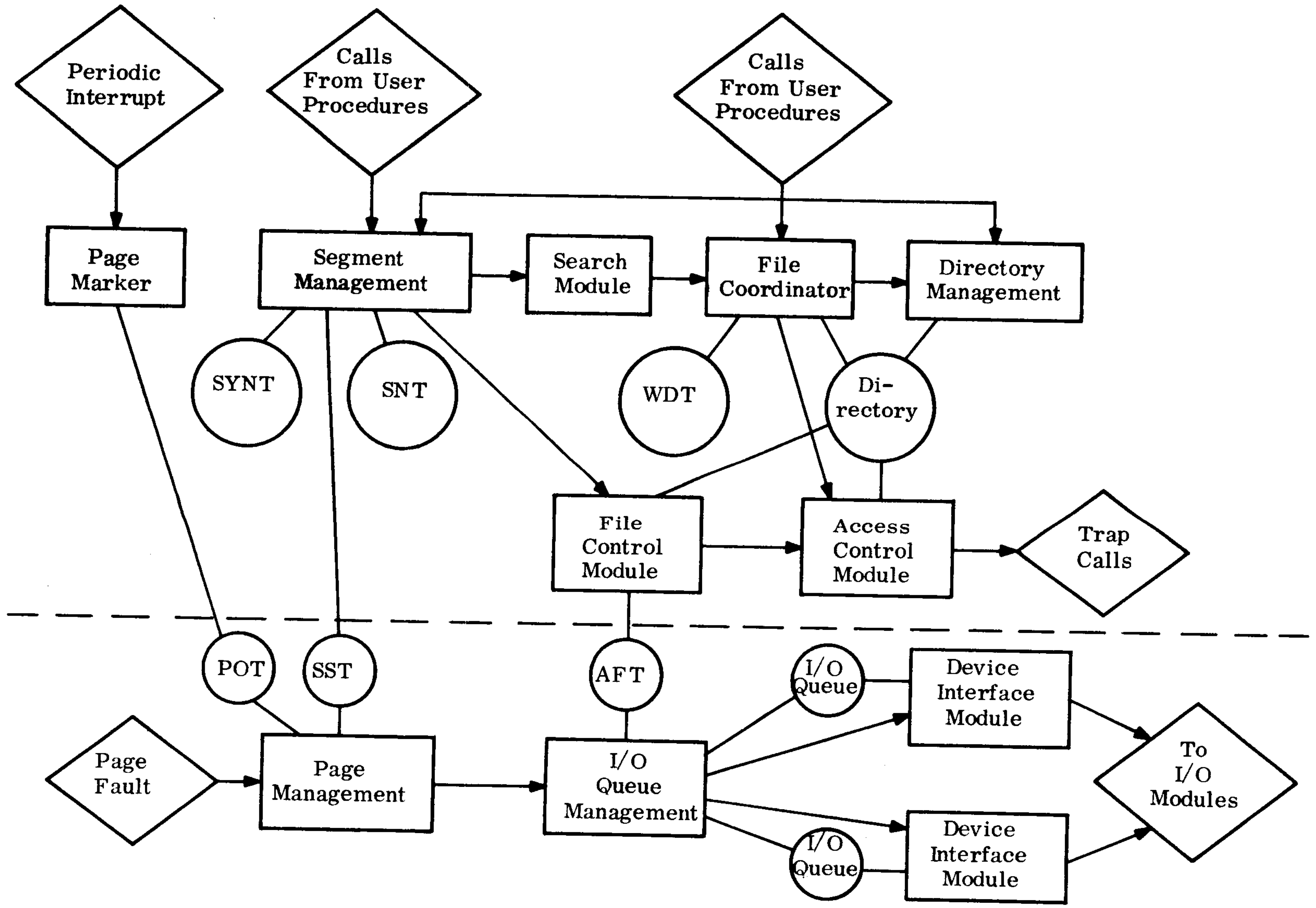


Figure 4. The Basic File System

4.1 The Segment Management Module (SMM)

The SMM maintains a record of all segments known to the current process. (A segment number has been assigned for this process.) For each known segment, an entry is made in the Segment Name Table (SNT) containing the call name (a symbolic entry name), tree name, the segment number, and other pertinent information. If the segment is active (page table in core), an entry exists in the Segment Status Table (SST) for that segment; it contains such information as the number of processes to which it is known and the files to receive I/O resulting from paging this segment in and out of core.

When a user calls for a segment for the first time, a fault occurs. The linker module picks up the call name and transfers control to the SMM which searches the SNT. If the call name is found, the segment number is returned to the process via the linker. If the call name is not in the SNT, the SMM, via the search module, locates the segment in the user's directory, assigns a number, updates the SNT, opens files to receive I/O as a result of paging in this segment, updates the SST, establishes a page table and segment descriptor if the segment was not active for another process, and finally returns the number to the calling procedure. The SMM maintains a Synonym Name Table (SYNT) for synonyms for a segment name. If calls use a synonym, the appropriate file name is substituted before any processing is carried out.

The SMM also provides facilities to create and work with copies of an original file called execution files as well as a facility to ask questions and make declarations about segments known to the user's process.

4.2 The File Coordinator Module (FCM)

The FCM provides primitive facilities for the user and system to manipulate entries in directories in the following ways:

(1) Create an entry

(2) Delete an entry

(3) Rename an entry

(4) Return status on an entry

(5) Change the ACL for an entry

(6) Change working directory.

The validity of the user's attempt to use the FCM is determined by reference to the ACM. A Working Directory Table (WDT) is maintained for the tree names of the user's working directories.

4.3 Other File System Modules

As can be seen in Figure 4, there are many other elements in the program structure of the MULTICS basic file system. The following list summarizes these modules and the services they provide.

- SEARCH MODULE. A module which is called by the SMM and uses the FCM to search a user's directory hierarchy for a particular file
- DIRECTORY MANAGEMENT MODULE. A module called by the FCM to find a particular entry in a directory
- FILE CONTROL MODULE. A module called by the SMM to open and close files for I/O operations which maintain status in the Active File Table (AFT)
- ACCESS CONTROL MODULE. A module called by the FCM or the SMM indirectly to perform access evaluation (Section 3)
- PAGE MARKET MODULE. A module which periodically resets page usage bits to maintain a Page Out Table (POT) of likely candidates for removal when space is needed
- PAGE MANAGEMENT MODULE. A module which services requests for pages missing from an active segment. A free page is assigned by using the POT or, if it is empty, by random removal of a page of appropriate size.
- I/O QUEUE MANAGEMENT MODULE. A module which services calls for I/O operations for entries in the AFT. Requests are placed in the appropriate queue for a device interface module which will handle the request.

- DEVICE INTERFACE MODULE. A module for each type of secondary storage device which executes the proper strategy for I/O operations on its device. These modules maintain STORAGE ASSIGNMENT TABLES, which reside on the storage device and determine where to store or retrieve information.

  In addition, there are other file system modules which, although not basic parts of the file system, are an integral part of the file system capability.

- MULTILEVEL STORAGE MANAGEMENT MODULE. An independent supervisory level process, this module collects and evaluates usage information on stored files, and uses this information along with accounting information to shuffle files through the secondary storage complex to appropriate level devices.

- UTILITY MODULES. A library of modules which provide the necessary functions for manipulation of links and branches and for direct I/O operations from secondary storage.

5. THE SECONDARY STORAGE BACKUP SYSTEM

Devices which can be removed from the system, e.g., tapes, data cells, and disc packs which are used by the system as an extension to the on-line file system, are known as the file backup storage system. In MULTICS this backup system uses magnetic tapes to retain copies of all files known to the system. New files (created or modified) are copied onto a pair of tapes; by restricting these tapes to an N-hour period (expected to be three hours), catastrophic failure effects can be confined to an N-hour dumping period at most. Every week there is a weekly dump of all files used in the preceding W weeks (expected to be four weeks). This dump is done in two parts; the first for all files essential to the operation of the basic system, the second for all files used in the last W weeks. These tapes need be retained for only a few months, while the incremental tapes must be kept indefinitely (periodically consolidated to remove obsolete files).

If a catastrophe should destroy the on-line storage systems, unloading can be accomplished using the most recent weekly dump tape and the incremental dump tapes since the last weekly dump. If the first part of the weekly dump tape (and the incremental tapes if a change has been made in the basic system since that time) is reloaded, the system can proceed to load the incremental dump tapes and then the second part of the weekly dump tapes. Usage and modification information assures that only the most recent of a set of redundant files will be reloaded.

An on-line storage salvage procedure is provided to eliminate conflicts in directories which result from system failure prior to completion of file system update procedures. In addition, the system provides procedures which allow the user to wait for retrieval, process other material while waiting, abort the process requesting a file, or delete the directory entry in the event that a file requested is not in on-line storage. If the file is desired, the system informs the operator of the date and time of the appropriate incremental dump tape and awaits loading. If a file is deleted, its directory entry is destroyed, but a copy remains on the incremental tape and can be accessed if the name, date, and time are known.

6. MULTILEVEL STORAGE POLICIES

The file system assigns all secondary storage dynamically; no areas are permanently assigned to a user. In no way does the system restrict the amount of information a user may keep within the file system, only that which may be in on-line storage devices. An accounting module has the responsibility of regulating the amount of on-line storage given to any one user at one time. The location of information is invisible to the user. Storage devices are classified by the file system according to levels which correspond to the data transfer speeds (greater speed, higher level number). These level numbers are used by the Multilevel Storage Management Module to make on-line storage capacity more efficient.

7. PHYSICAL FILE STRUCTURE

The MULTICS system has the capability for storing data in four structures:

(1) Serial Linear Files

(2) Random Linear Files

(3) Serial Logical Record Files

(4) Random Logical Record Files.

Both types of linear files have the same format; they are linear arrays of entries (called elements) with a predefined size (in bits). Elements may be referred to by an index relative to the beginning of the file. The only difference between serial and random is that in any I/O operation, a serial file is referred to by an index relative to the current element, while a random file is referred to by its element number. For a file to be random linear it must be stored on a random access medium.

Both types of logical record files have the same format; they are a collection of records each having a record number. Each record is made up of a variable string of elements of predefined size. The length of each record may be different and can be changed. A block header contains the number of elements in each record. (The concept of serial and random for logical record files is the same as for linear files.)

A distinction between handling serial and random files is that serial files are considered to end with the last non-null entry. A random file, on the other hand, is considered to be of fixed size, with null records between the last non-null record and the end-of-file.

8. EVALUATION

This MULTICS file system can be discussed only as a set of design concepts as the system has not yet been implemented nor is there any assurance that the version implemented will bear a strong resemblance to the preceding description. The MULTICS file system design, however, is inclusive, and has overlooked few features of significance in the external data management area. Since it permits a very general hierarchy of file naming and structural membership, there is little need, if any, of file duplication for control purposes. Files and programs are sharable among specified users, with proper safeguards, and adequate provision is made for backup and recovery.

The only adverse criticism of the file system is that it is complicated by logical (internal) file management services which may have been easier to handle within the framework of an internal file management system.

INTEGRATED INFORMATION PROCESSING SYSTEM (INTIPS)

1. INTRODUCTION

INTIPS (INTegrated Information Processing System) is a configuration in which various dissimilar computers and peripheral devices may be interconnected via an electronic switch. The external file system,* which will reside in one computer, is to provide a centralized facility for the management of data in the system's secondary storage. While a facility must be provided for operational use, it is recognized that the file system is also experimental in nature, and will evolve with experience.

Design goals include considerations such as:

(1) Accommodation of new computers, devices, and memory in an essentially open-ended configuration

(2) Multi-level secondary storage management. The ability to review periodically the usage of data and its distribution over storage devices of differing access times (and costs), and change the assignments to allocate resources most intelligently

* E. W. VerHoef: Design of a Multi-Level File System. Proc. Natl. Conf. ACM, 1966, p. 66.

(3) Ability to handle growth of data base to a projected maximum of one billion blocks* with no design changes and minimal loss in performance

(4) Ability to cope with dynamically changing device availability, also core memory in which the file system resides.

2. SYSTEM DESCRIPTION

The file computer is currently the PDP-8, with a basic core memory of 4096 words, expandable to 32,768 words. The file system design is not confined to the file computer, as companion programs must exist in the executive control program of the computers serving the problem processes. Besides conveying control information and other data between user and file computer, the executive provides certain services pertaining to the status of the control information. Otherwise, file system functions are the domain of the file computer.

The report used for this study** deals with the functions of the file computer. The file computer accepts operations to be performed from other computers, and returns the results. This dedication of a processor to the external file system function has advantages, and some disadvantages, but in any case is an important and distinguishing feature of INTIPS.

Modular design of programs is emphasized. Nearly all routines must be serially re-usable and able to be relocated during execution.

2.1 Data Structure

The unit of symbolically addressable storage in this file system is called the block (6144 bits). Every block is a member of one and only one collection of blocks, called a file. Each file is a member of at least one collection of elements called an aggregate. The elements of an aggregate are files and/or aggregates. Any aggregate may be a member of zero or more aggregates. A unique name is associated with each aggregate in the system. A file is known by one or more names, each name unique within an aggregate. A block may also be named. Besides reference by name, a block, file, or aggregate may be assigned an ordinal number by which it is

* A block is defined here as 6144 bits, or $3x2^{11}$.

** E. W. VerHoef: Design of a Multi-Level File System. Proc. Natl. Conf. ACM, 1966, p. 66.

distinguished in a collection (for reference purposes, no storing order implied). One could think of collections with "named" elements as unordered sets, and numbered collections as ordered sets.

The index structure consists of four lists: the Aggregate Bucket Index, Aggregate Bucket List, Aggregate Record List, and File List. In order to understand the organization, the notion of the "bucket" will first be developed.

Ignoring the added complexity due to the hierarchical organization of data, the indexing problem is much like the one encountered in maintaining an address book. Starting with a blank book, it would not be wise to enter names indiscriminately in any order. Instead look-up time can be reduced by dividing names into categories. One method is to collect all names beginning with the same letter into a bucket, and index the 26 buckets in the address book. Now within a given bucket, entries can be made in no particular order, and average look-up time is reduced significantly.* However, to be most effective there should be about as many names starting with Z as there are with C. The method for categorizing into buckets should be chosen to produce an approximately uniform distribution. This technique is called "hashing" or hash encoding.

There is a rule, then, used by the file system, to reduce a name to a bucket assignment of 0 through 63. The Aggregate Bucket Index consists simply of 64 entries, each of which locates a bucket in the Aggregate Bucket List. In a given bucket in the Aggregate Bucket List is a list of names. When the match is found by searching the list, a locator stored with each name leads to an entry in the Aggregate Record List. This entry lists the elements of the aggregate. Some elements are aggregates, represented by a name and bucket number, and some are files. The entry for a file will contain a locator of file information contained in the File List. Since the aggregate record may have many entries which have to be scanned (by name or ordinal number), a hashing technique is also used here to reduce the lookup.

The entry in the File List contains a list of block names and locators, and control information. The control information includes such items as a user ID and access restrictions.

* In a book with 676 names, the "indiscriminate" method would require an average of 338 searches per lookup, whereas assuming 26 entries per bucket, the bucket method requires 13 + 13 = 26 searches.

In these descriptions the term "locators leading" to entries in another index has been used rather than addresses. These locators are called in INTIPS System Assigned Symbols (SAS), which are converted to physical addresses through a SAS Index before the next level of reference can actually be made. This extra overhead introduced in referencing a block is directed toward reducing the overhead in carrying out maintenance functions. A change in structure or storage allocation results in a minimum of updating to these indexes; i.e., many index entries lead to the same physical address.

2.2 Operations

In Table 1, a summary of operation descriptions, let B stand for "block name or ordinal number," or "block" according to context. Similarly let F = file, A = aggregate, and C = user-supplied data.

TABLE 1. SUMMARY OF OPERATION DESCRIPTIONS

User Program Specification	Optional Specification	Function(s)
C, F, A	N_B	Store C as B in F of A. Number it N_B.
B, F, A		Retrieve B of F of A.
A	F	List elements of A, or list all block names in F of A.
F, A		Lock or release F of A (end of run also releases).
F, A, C,	N_F	Create F record of size C for A. Number it N_F.
F_1, A_1, F_2, A_2	N_F	Include F_1 of A_1 as F_2 of A_2. Number it N_F.
F, A		Delete F's membership in A.
A		Define new aggregate name A.

TABLE 1. SUMMARY OF OPERATION DESCRIPTIONS (Cont.)

User Program Specification	Optional Specification	Function(s)
A_1, A_2	N_A	Let A_2 be an element of A_1. Number it N_A.
A_1, A_2		Delete A_2's membership in A_1.
A		Delete A and all nodes which it uniquely includes.

These services are available to the user program in two modes of operation; in Random Mode there is presumably no useful relationship between the parameters in one command and those of the next. Each command therefore contains explicit values (either name or ordinal number) for the parameters A, B, and F. Control Block Mode is used when an orderly sequence of references is intended, such as sequentially stepping through the blocks in one file, then again through those in the next file of the same aggregate. In this case the user takes advantage of services provided for ordinal number increment and decrement. After a "Ready Aggregate" or a positioning command, subsequent references may specify B of F by relative position to the previous reference.

2.3 Other Features

2.3.1 Access Rights. The user may allow others to access his data at several levels. Currently planned are read-only, write-only, and read and write. Also the executive programs may protect their control information from all users. Explicit access specifications must be given to restrict the use of data since, without them, the data is public.

2.3.2 Multi-Level Secondary Storage Management. Blocks are allocated to files in groups of 8. With each group is associated a count which is increased upon retrieval of any block. At regular intervals these usage counts are plugged into an algorithm which yields a usage measure based upon the previous value of usage measure and the current count. (Current counts are forced to values between 0 and 1 by dividing by the largest count in the time period before using the algorithm. The reason for this

AUERBACH

is not clear if the intervals are truly "regular.") The resulting usage measures for the groups indicate the way they should be distributed over storage devices of varying access times and costs; "semi-permanent" reassignments are made accordingly. The user has some privileges in that he may place upper and/or lower bounds on the storage device level to which his data is assigned and have his data "visit" a higher level for the duration of a run. This advantage protects the infrequent user who nevertheless has good reasons for fast access during execution.

3. EVALUATION

The ability to manage data structured as a lattice is a significant degree of generality over systems which confine data structures to a tree. However, the more flexibility in the structure, the more critical becomes the design of the directories or indexes and the procedures for lookup. One of the challenges remaining in designing file systems with non-simple data structures is the discovery of ways to cut down the time needed for successive references to the indexes in locating an item. It is apparent that the INTIPS file system designer is at least conscious of this problem, as exemplified by the use of hash encoding. A technique which would be especially effective in this situation, where a satellite computer is dedicated to file management, is to utilize whatever core memory is available to hold copies of indexes or selected portions thereof. This procedure would enable a significant reduction in the time to access a block. In the material reviewed, however, there is no indication of the operational techniques employed.

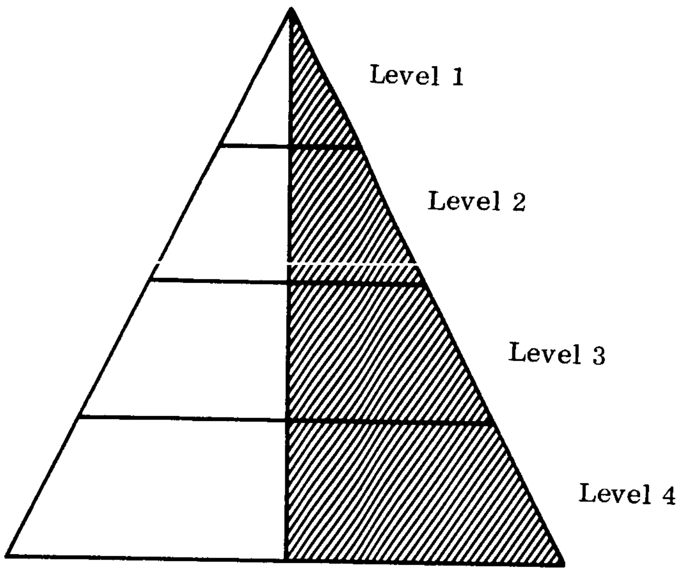
Although the SAS mapping tables, needed for going from one index to the next, are a considerable overhead factor during retrievals, this price is paid to reduce overhead during operations which move data or restructure the indexes. In a system with a static or slowly changing data base, this might not be justified. Such systems are exceptions rather than the rule, however, and INTIPS is not seen to be one of the exceptions.

In the material reviewed, there is no mention of a machine-independent language used for developing the file systems programs. It is assumed that machine language is used, and that a new host machine would mean a new coding job. This assumption is not meant to be a negative point, as the price of machine-independence is high overhead, which is a severe enough problem in file system design without compounding it. The design concepts seem amenable to implementation on any machine having appropriate I/O capabilities.

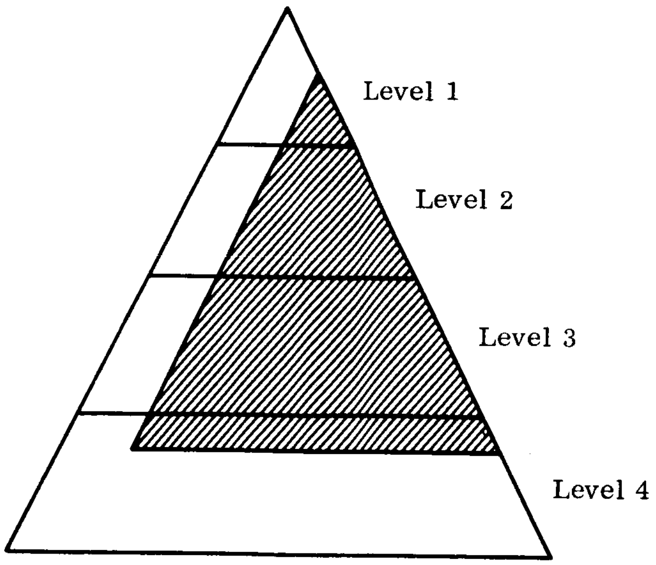
A surprising omission is the apparent lack of a backup system. While this may be elaborate (automatically triggered backup or restoration of selected files) or simple (specially scheduled manually initiated utility program), some capability to protect against loss of a data base should be considered a requirement. No doubt at least a minimal capability is intended, as the design goals include immunity from catastrophies due to losing devices. This could be a significant cost area in a system which aspires to a data base of 10^9 blocks, as the number of tape reels needed to hold the backup data is in the neighborhood of half a million.

Although the delegation of file system functions to a dedicated processor has many advantages, the arguments which would normally justify an elaborate level-changing capability hold to a lesser extent in this arrangement. That is, the argument is basically that one is willing to pay a bit of overhead and additional program development cost for the ability to match high usage data to fast-access devices in the interest of better performance. The total access time (beginning with user making I/O request and ending with user in control and operation complete) is approximately expressed as the sum of two terms, one a "constant" overhead term C, and the other a term D proportional to the latency of the device holding the data (and indexes). If the data is on a device at level 1, the access time $T_1 = C + D_1$ is less than if it is at level 2 ($T_2 = C + D_2$). The relative gain of T_1 compared to T_2 due to difference in storage device is significant only if C is a small contribution to the expression. Examples of overhead contributing to the value of C are the shipping back and forth of the Control Block between computers, and the processing of the Control Block which precedes the accessing of the host device.

Borrowing the representation of storage levels used in the VerHoef reference, the INTIPS approach to data distribution over storage device levels is depicted in Figure 1(a) (assuming 4 levels). In this illustration, the shaded area represents how storage requirements at 50 percent of capacity would be distributed. This solution seems wasteful of the unused storage at levels higher than 4. Yet one cannot go to the extreme of packing everything at the top, for a new demand for level 1 would precipitate a series of shifts in order to make room. The obvious compromise is depicted in Figure 1(b). This arrangement leaves buffer space at each level but does not assume, as in (a), that the best distribution is one of uniform percentage occupancy.



(a) INTIPS Rule



(b) Another Approach

Figure 1. Allocation of Multi-Level Secondary Storage

APPENDIX B. INSCAN: A SYNTAX—DIRECTED LANGUAGE PROCESSOR *

by

Mark Resnick and Jerome Sable

* Submitted for presentation at ACM National Conference, Las Vegas, Nevada, August 27-29, 1968.

ABSTRACT

This paper describes the operation of Inscan, a generalized language processor whose operations are controlled by action graphs. The action graphs are capable of specifying both the syntax of the language and the actions necessary to translate or otherwise process it.

The concept of syntax-directed processing is introduced, and the requirements of a user-oriented metalanguage are discussed. The operation of the Inscan processor is explained. Action graphs are described in two forms: a pictorial form convenient for design, and a string-language form (STAG) convenient for computer processing. Several examples of action graphs are given, including one which specifies the syntax of the STAG language.

APPENDIX B. INSCAN: A SYNTAX—DIRECTED LANGUAGE PROCESSOR

B.1 INTRODUCTION

B.1.1 Concept of Syntax Directed Processing

The problem of building a language translator can be partitioned into two areas. The first is concerned with the specification of the syntax of the source language and the actions to be taken upon recognition of each syntactic type. The second is concerned with an algorithm for simultaneously scanning the source language string and the syntax specification and producing an object language string. That the problem could be broken up in this way has been known for some time and has been extensively reported in the literature. See, for example, the published surveys (Refs. 2, 7, 8). One of the theoretical results of partitioning the problem in this way is that the same scanning algorithm can be used with each of several syntax specifications for parsing and translating several languages. Although this has been theoretically understood and often mentioned in the literature of syntax-directed compiling, very little use has been made of multi-language translators in a system context. The ADAM system (Ref. 4) and the AUERBACH generalized data management system called DM-1 (Ref. 6) are two systems which have attempted to do this.

The purpose of this paper is to discuss some of the possible modes of use of a syntax-directed processor, and the design and use of a particular syntax-directed language processor, called Inscan, which is part of DM-1.

It is believed that Inscan is unique in that it offers a convenient user-oriented language for language and language processor specification. Our premise is that the route to optimal language interface with the users of a system (users at all levels, programmers, analysts, experimenters, and managers) is to provide a universal processor rather than a universal language (or even several universal languages). In Inscan, the syntax of the language to be processed, and the actions to be taken, are specified by the designer in a chart called an action graph. Inscan allows an experimental or adaptive approach to language development and, furthermore, permits both interpretive and translational modes of operation.

B. 1. 2 Processing Framework

A syntax-directed language processor can be used in several modes within a given system context. Consider, for example, the language processing needs of a hypothetical user-interactive system which manages an on-going data base. Within this context there may be several types of user languages:

(1) Compiler language

(2) Macro-assembler language

(3) Command language

(4) Data description language

(5) Data language

(6) Job description language

(7) Data service language

(8) On-line computational language.

It should be possible to perform a large part of the input stream scanning, analysis, and interpretation of these languages with basically the same, or similar, syntax-directed processors. Yet an examination of the language processing contexts for most of the preceding languages shows that the roles of their processors are quite different.

In some, such as the compiler, macro-assembler, and data language processors, the language processor behaves as a translator. This mode is illustrated in Figure B-1. In others, such as the command and data service language processors, the processor must function in the interpretive mode, executing the command or service request as it is recognized. This mode is illustrated in Figure B-2. Inscan has been designed to function in either mode, depending on the nature of the action graph which is controlling it. More generally, Inscan can be used in a mode which combines both translational and interpretive aspects, as illustrated in Figure B-3.

An important by-product of using a syntax-directed processor for the language scanning functions in a user-oriented system is that the system languages themselves can become the object of experimentation. In order for this to occur, it must be easy to introduce a new action graph (or a new version of an old one) into the system without impact on existing system or user programs. In DM-1 this is done by using Inscan in a translation mode to translate an action graph specified in a symbolic language called STAG (STring Action Graph) into an absolute form called the Action Graph Table (AGT), which is the form actually used by Inscan. Thus, the use of Inscan can be viewed as a two-stage process. In the first stage, corresponding to assembly time, an AGT is generated from a STAG source language specification (or "program"). In the second stage, corresponding to execution time, a user language text is scanned by Inscan under control of the AGT. These phases of user language processing are illustrated in Figure B-4.

B.2 LANGUAGES FOR LANGUAGE SPECIFICATION

B.2.1 Metalanguage Requirements

Since our basic assumption was that a language should respond to, and adapt to, the needs of the user, and since the syntax and semantics of languages are specified in a metalanguage, the characteristics of an appropriate metalanguage will be examined. Our interest is in the metalanguage as a design language for the user, rather than as a form in which the language rules may be expressed internally in the processor, such as transition list structure, transition matrix, substitution table, etc.

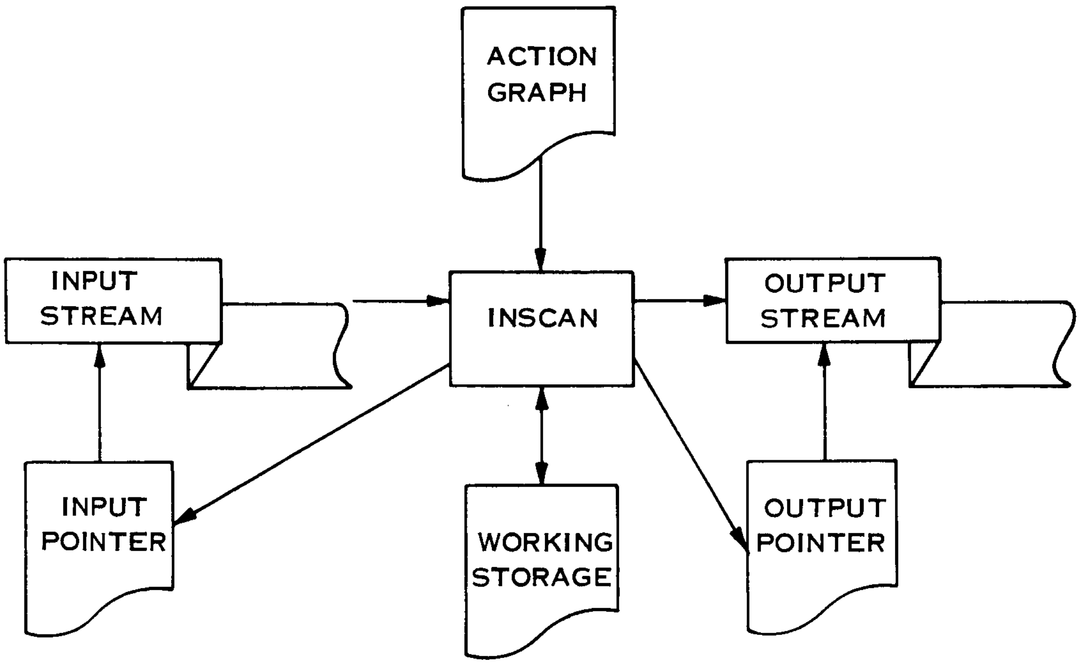


Figure B-1. Inscan: Translation Mode

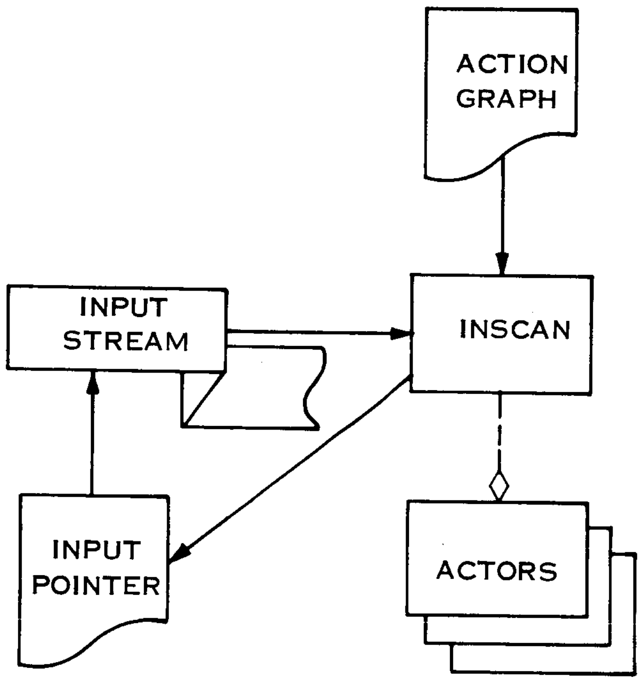


Figure B-2. Inscan: Interpretive Mode

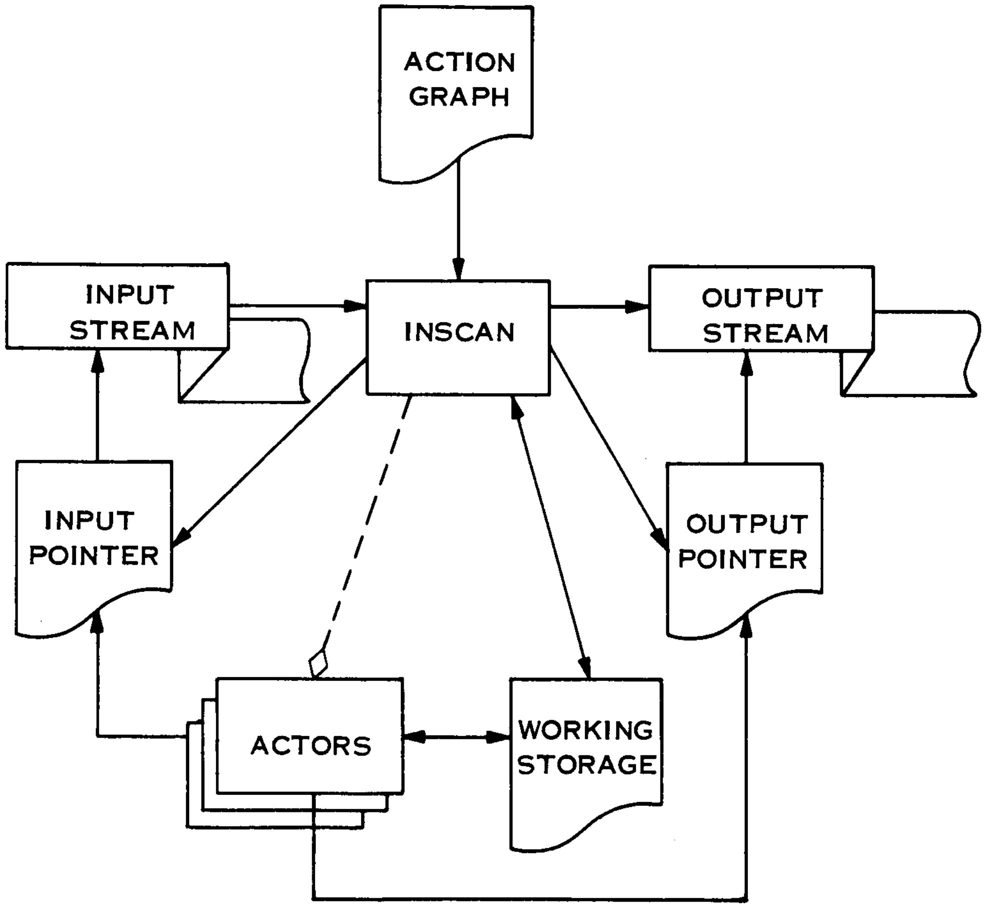
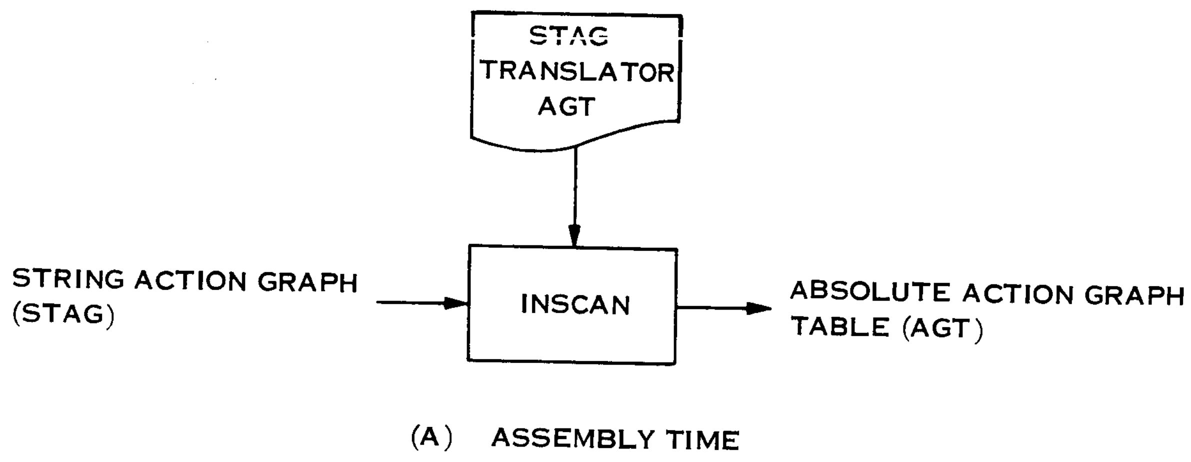


Figure B-3. General Inscan Configuration



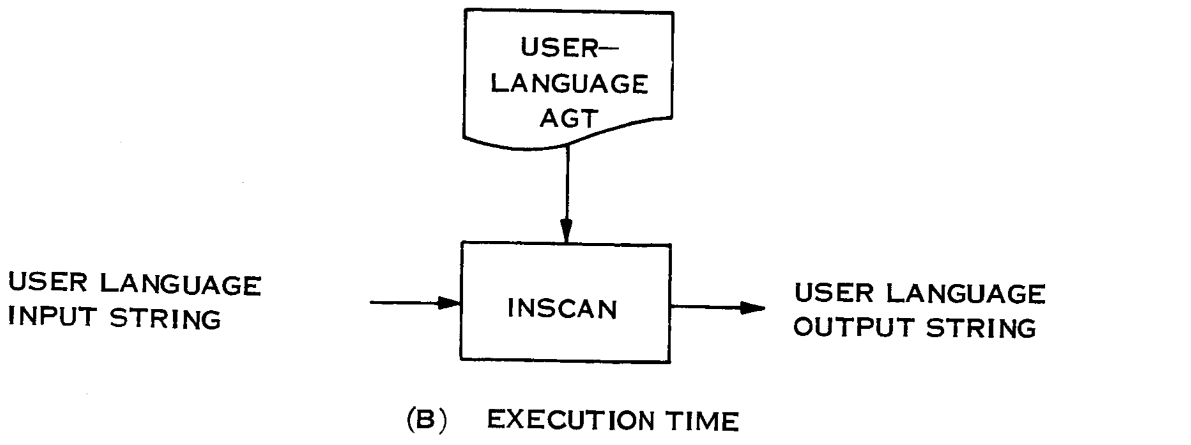


Figure B-4. Phases of User Language Processing

There are several properties that the metalanguage should have:

(1) It should be capable of expressing the notation and syntax of a context-free language (Ref. 3).

(2) It should be capable of expressing the operations of a push-down automaton which will generate the output stream.

(3) It should be capable of expressing a subroutine call to a processor which can take some interpretive action, such as execute a command, respond to a modal operator (or pseudo-op), or expand a macro-call in the output stream.

To this list of necessary characteristics should be added the following desirable features:

(1) It should be capable of being represented in an easily read graphical format that exhibits the structure of the language much in the same way in which a flowchart exhibits the structure of a program.

(2) It should be capable of several levels of detail so that the overall structure of the language can be expressed before linguistic subtypes need be defined.

(3) It should possess a readable linear string version amenable to computer input, and it should be easy to translate from the graphical to the linear version, and vice versa.

B. 2. 2 Metalanguage Examples

Some examples of specification languages are given in Figure B-5. The figure shows how two simple languages, called jm and paren, are defined in four metalanguages. (The jm example is taken from Steil (Ref. 12.) Some examples of sentences in the language jm are:

JOHN MARSHA, JOHN JOHN MARSHA MARSHA, etc.

Examples of paren are:

( ), (( ) (( ))), (( ) ( )), etc.

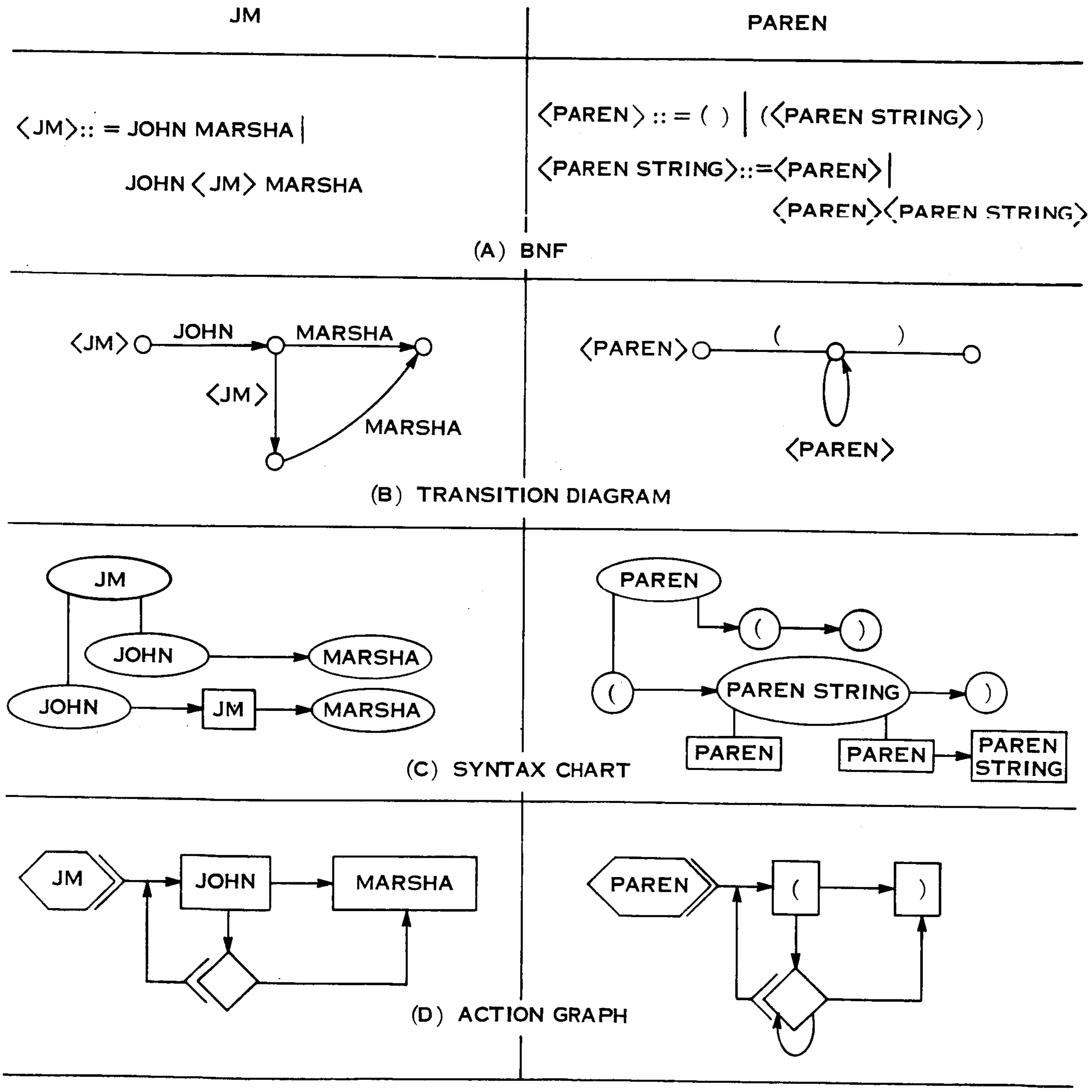


Figure B-5. Examples of Syntax Specifications

The metalanguages illustrated are:

(1) BNF (Ref. 1)

(2) Transition diagrams (Ref. 5)

(3) Syntax charts (Ref. 13)

(4) Action graphs.

Several general references books on metalanguages have appeared (Refs. 7, 9, 11).

The remainder of this paper will describe the operation of Inscan and its associated action graphs in some detail to show that action graphs effectively fulfill the metalanguage requirements outlined previously.

B. 3 INSCAN AND ACTION GRAPHS

Inscan may be thought of as a processor that processes an input string in accordance with a set of instructions contained in an action graph. Thus, Inscan corresponds to a computer, the action graph to a program running in the computer, and the input string to data being processed by the program. The nature of the processing may vary widely, but two distinctive features of Inscan, to be described in more detail next, make it particularly suited to a certain class of applications. These features are the automatic scanning control with respect to the input string and the recursive executive control with respect to the action graphs.

B. 3. 1 The Input String

The input string is regarded as a string of symbols which is scanned from left to right. The input pointer always points to the next symbol to be scanned. When scanning begins, the input pointer is automatically set to the beginning of the input string. Thereafter, it is moved in accordance with the dictates of the various action-graph instructions.

B. 3. 2 The Action Graph

The action graph controls the processing performed on the input string. As seen by the Inscan processor the action graph is a table of instructions (the Action Graph Table, or AGT). Each instruction specifies a test to be made, an action to be performed, the establishment of alternate paths, or a transfer of control. When Inscan is called by a user, the user supplies the input string to be processed and the name of the action graph to be used. Inscan retrieves the specified action graph from a library and begins execution with the first instruction in the table. Thereafter, the instructions are executed sequentially until a choice point is established or a transfer of control occurs.

The most basic instruction dealing with the input string is Scan (Read Input and Match). This tests the input string, beginning at the current position of the input pointer, for equality with a constant. The number of symbols tested is equal to the number of symbols in the constant. If the input string matches the constant, the input pointer is automatically advanced past the matched portion of the input string, so that the next portion may now be tested. If it does not match, the pointer is not moved, so that the same portion of the input string may be tested again.

A choice point indicates that several alternate paths may be taken at that point, and specifies an ordered list of the addresses of the paths. After a choice point has been established, control is transferred to the first path on the list. As long as the choice point is in effect, the failure of any test will cause control to be transferred to the next alternate path.

An action graph may at any time call for the execution of itself or of another action graph. When this occurs, a return pointer to the calling action graph is stored in a pushdown list, the new action graph is called in, and execution of it is begun at its beginning. Certain points in an action graph are designated as end points. When an end point is reached, the execution of the action graph has been successfully concluded. Inscan then "pops up" to the action graph on the next higher level and returns control to the point indicated by the return pointer. If, however, during the execution of an action graph some test fails and no alternate paths have been provided (or all alternatives have been exhausted), the execution of the action graph terminates unsuccessfully. In this case Inscan also pops up to the next higher level, but now returns control to the next alternate choice path on the higher level.

When Inscan pops up from the highest level, control is returned to the calling program, and appropriate status information is provided to indicate the result. A failure at the highest level indicates a syntax error in the input string.

B. 3. 3 Specification of Action Graphs

There are several alternative ways in which action graphs may be described, and these various representations may be translated into the Action Graph Table (AGT) that is actually used by Inscan. One of these representations, the Pictorial Action Graph, will be introduced now. Another representation, the String Action Graph (STAG), discussed in Paragraph B. 4.

A Pictorial Action Graph expresses the notation, syntax, and interpretation of a language in a manner similar to that in which a flowchart describes a program. Pictorial Action Graphs are composed of a number of symbols representing nodes in a directed graph, the nodes being connected by directed branches. There is logically one entrance to and one normal exit from each graph, but a node in the action graph may represent a subgraph which is to be executed.

A list of the symbols used in Pictorial Action Graphs is shown in Figure B-6. There is some leeway permitted in drawing action graphs, and a given language may be expressed in more than one way.

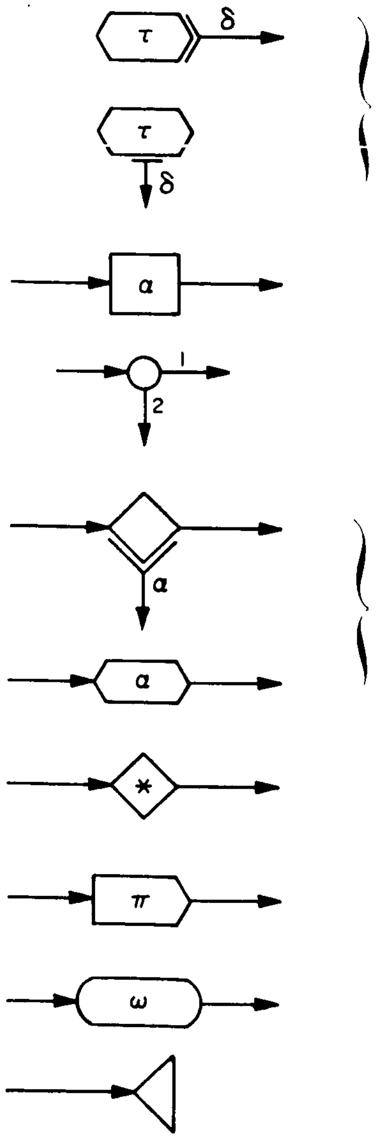
The Action Graph Name symbol, used at the beginning of an action graph, contains the name of the action graph being defined.

A subgraph execution may be indicated by several symbols, as shown in Figure B-6. Use of the diamond symbol with an arrow to the subgraph exhibits the overall structure most explicitly. The asterisk notation eliminates the need for the arrow for recursive definitions. The hexagonal symbol names the subgraph to be executed and is most directly translated to the STAG form.

The Scan symbol contains a literal quantity to be tested for at that point in the action graph.

The Choice Point symbol indicates the establishment of alternate paths. It always has one line entering it and as many lines leaving it as there are alternate paths at that point. Choice points may also be indicated implicitly simply by having several paths leaving a node of another type.

The End symbol is used to mark the normal termination points of an action graph. It has one line entering and none leaving it.



GRAPH NAME: τ IS DEFINED BY PATH δ

SCAN: READ INPUT SYMBOL AND MATCH α

CHOICE: TRY ALTERNATIVES 1 AND 2

SUBGRAPH: EXECUTE GRAPH α AND RETURN

RECURSE: EXECUTE THIS GRAPH RECURSIVELY AND RETURN

EXTERNAL ACTION: DO SUBROUTINE π AND RETURN

INTERNAL ACTION: DO OPERATIONS ω

END: RETURN TO PARENT GRAPH

Figure B-6. Action Graph Symbols

B. 3. 4 An Action Graph Example

In order to illuminate the discussion of Inscan and action graphs, attention is directed to the following example.

The example to be presented is a recognizer for a simple infix language involving the letters A and B, the operators + and -, and the left and right parentheses. The recognizer requires two action graphs, called NEST and TERM, both of which are shown in Figure B-7.

The highest-level action graph is NEST. This is the graph that is identified to Inscan by the user, and the one that Inscan first calls in and begins to execute. NEST is the action graph as far as the user is concerned. The fact that it calls on other action graphs is incidental.

To observe the action of Inscan, let us follow the scanning of the input string (A+B)-(A-B). To help follow the action, the nodes of the action graphs (see Figure B-7) and the characters of the input string (see Figure B-8) have been numbered. When Inscan is called, the user supplies the input string to be processed, and the name (NEST) of the action graph to be used. Inscan sets the input pointer to character 1 of the input string, "(", and retrieves action graph, NEST, from the action-graph library. Execution begins at node 1 of the action graph. This node calls for the execution of another action graph called TERM. Consequently, a return pointer to the NEST action graph is established and placed in Inscan's pushdown list. This pointer will point to the next node of NEST to be executed, namely the choice point at node 2. TERM is now called in and execution begins at node 1. This node establishes a three-way choice of nodes 2, 3, and 5. Node 2 is tried first. This node specifies that the current position of the input string should be tested for equality with the character "A". The input pointer is still pointing to character 1, "(", since it has not yet been moved and is not affected by the transfer of control from one action graph to another. Therefore, since "(" is not equal to "A", the test fails. The input pointer is not moved, and the next alternate path is tried, namely node 3. The test for B likewise fails. Next, node 5 is tried. This test succeeds, so the input pointer is advanced to character 2, the A, and control proceeds to node 6 of TERM. This calls for the execution of another action graph, NEST, so a return pointer to node 7 of TERM is placed in the pushdown list and NEST is called in. Node 1 of NEST calls for executing

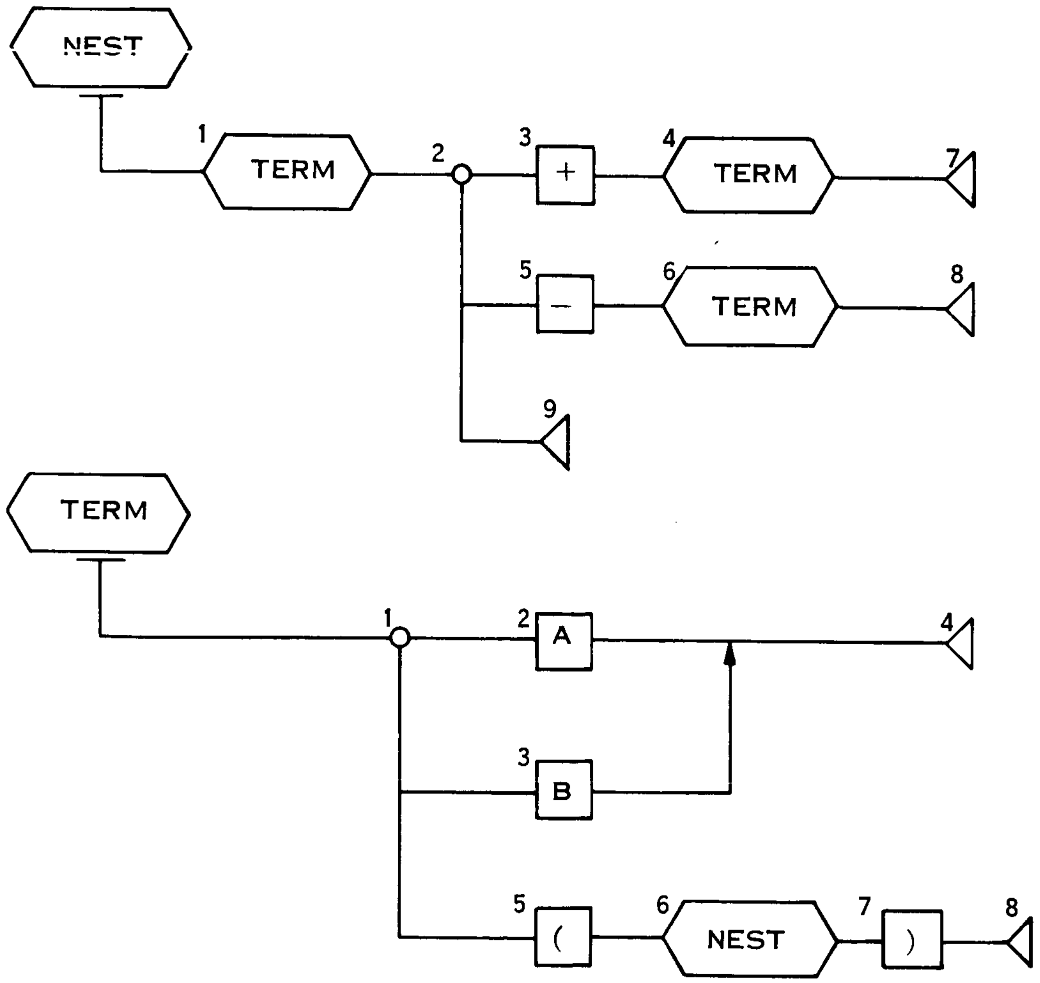


Figure B-7. Infix Recognizer

(	A	+	B	)	−	(	A	−	B	)
1	2	3	4	5	6	7	8	9	10	11

Figure B-8. Sample Input String for the Infix Recognizer

TERM, so a return pointer to node 2 of NEST is created and TERM is recalled. Again, node 1 of TERM establishes a choice of 2, 3, and 5. At node 2, the test for A now suceeds since the input pointer is at character 2. The input pointer is advanced to character 3 and control goes to node 4. This is the termination symbol, indicating that execution of the action graph, TERM, has been successfully concluded. Consequently, Inscan pops up to the next higher level and follows the return pointer. This points to node 2 of NEST. Node 2 establishes a choice of three alternatives: 3, 5, and 9. Node 3 is tried first. This specifies a test for the character "+". The input pointer is still at character 3, which is "+", so the test succeeds, the input pointer is advanced to the next character, and control passes to node 4 of NEST.

The reader is left to complete the details of the example. Figure B-9 gives a list of the various portions of the input string, in the order in which they are recognized during the scan, together with the name of the action graph that recognizes them.

B.3.5 Action Graph Constraints

From the above discussion it can be seen that Inscan is a "top-down" type of analyzer in that it starts at the highest-level graph and does not scan the input until directed by a graph to do so.

Although the algorithm for a top-down analyzer is quite simple and straightforward, it is known that they can be relatively inefficient (Reference 10). In order to provide for efficient analysis, the following two constraints (Reference 5) were adopted and must be observed in designing action graphs for Inscan:

(1) No-Loop Condition

There should be no cycle of subgraph references which returns to the initial graph without scanning an input symbol, for example,

(a) No action graph should execute itself recursively without first reading an input symbol.

(b) No action graph should execute a subgraph which executes the initial graph without first reading an input symbol.

PORTION OF STRING RECOGNIZED	RECOGNIZED BY
A	TERM
B	TERM
A + B	NEST
(A + B)	TERM
A	TERM
B	TERM
A − B	NEST
(A − B)	TERM
(A + B) − (A − B)	NEST

Figure B-9. Operation of the Infix Recognizer

(2) No-Backup Condition

The input symbols that can be scanned and matched at each branch of a choice point should be disjoint sets; that is, it should not be possible to satisfy an input symbol through two different paths at a choice point.

B. 3. 6 Action Points

In order to enable Inscan to take actions as well as to perform syntax checking, the concept of action points has been developed. An action point is a node of an action graph at which an action is taken. Since action points are embedded within the action graph, they are automatically related to Inscan's scanning and control functions. That is, when an action point is entered, the input pointer is at a certain place in the input string, and control is at a certain point in an action graph on a certain level. Action graphs take their name from their ability to contain action points.

Action points are classifiable into two types, called external and internal. When an external action point is encountered, Inscan relinquishes control to an external subroutine provided by the user. The subroutine may then take whatever action is appropriate, and when finished, it is responsible for returning control to Inscan. Appropriate conventions are established for communication between Inscan and the external subroutines.

At an internal action point, Inscan retains control and executes a specified action command which is part of Inscan's own repertoire. The action commands are aimed at the generation of an output string. The output string and the method of generating it are discussed more fully in the following paragraphs.

The symbols used to represent action points are shown in Figure B-6.

B. 3. 7 The Output String

The output string, like the input string, is a string of symbols addressed by a pointer. The output pointer always points to the next symbol position to be filled. When a series of symbols is placed into the output string, the output pointer is automatically advanced to the next position following the last one filled.

The two most basic instructions for placing information in the output string are Write and Copy. Write is used to write constant values into the output string. Copy is used to copy portions of the input string to the output string. The portion of the string to be copied is defined by setting the input pointer to the beginning of the portion to be copied, then saving this setting (by means of a Save-Input-Pointer command), setting the pointer to the next symbol following the end of the portion to be copied, and performing the Copy command.

The use of action points and the generation of an output string, can be illustrated by returning to the example discussed previously. With the addition of only four action points, the graphs of Figure B-7 can be converted from a syntax checker only to a combined syntax checker and infix-to-suffix translator. The new graphs are shown in Figure B-10. The actions can be described as follows: Whenever TERM is entered, the position of the input pointer is saved. If the term recognized is a literal (A or B), then after the recognition the pointer is advanced to the next symbol; so that when the Copy operation occurs, the current and saved positions of the pointer are such that the literal is copied to the output string. Consequently, the suffix translation of a literal is the literal itself. If the term recognized is a parenthesized nest, no action is taken. Therefore, the suffix translation of such a term is just the suffix translation of the nest.

When NEST is entered and an infix nest is recognized (i.e., a pair of terms separated by a "+" or "-" sign), the sign is placed in the output string after the second term has been recognized. Consequently the suffix translation of an infix nest consists of the suffix translations of the two terms followed by the separating sign. If the nest recognized is a single term, no action is taken. Therefore, the suffix translation of such a nest is just the suffix translation of the term. Figure B-11 shows the combined syntax checking and output-string generation performed by the action graphs for the example. The final output string, representing the suffix translation of (A+B)-(A-B), is AB+AB--.

B.4 THE STRING LANGUAGE AND IMPLEMENTATION OF INSCAN

Work with Inscan has shown the need for a language which can readily express action graphs but which can also be easily read and processed by a computer. To fill this need, the String Action Graph (STAG) language has been devised. In STAG an action graph is represented as a linear string of characters comprising a sentence.

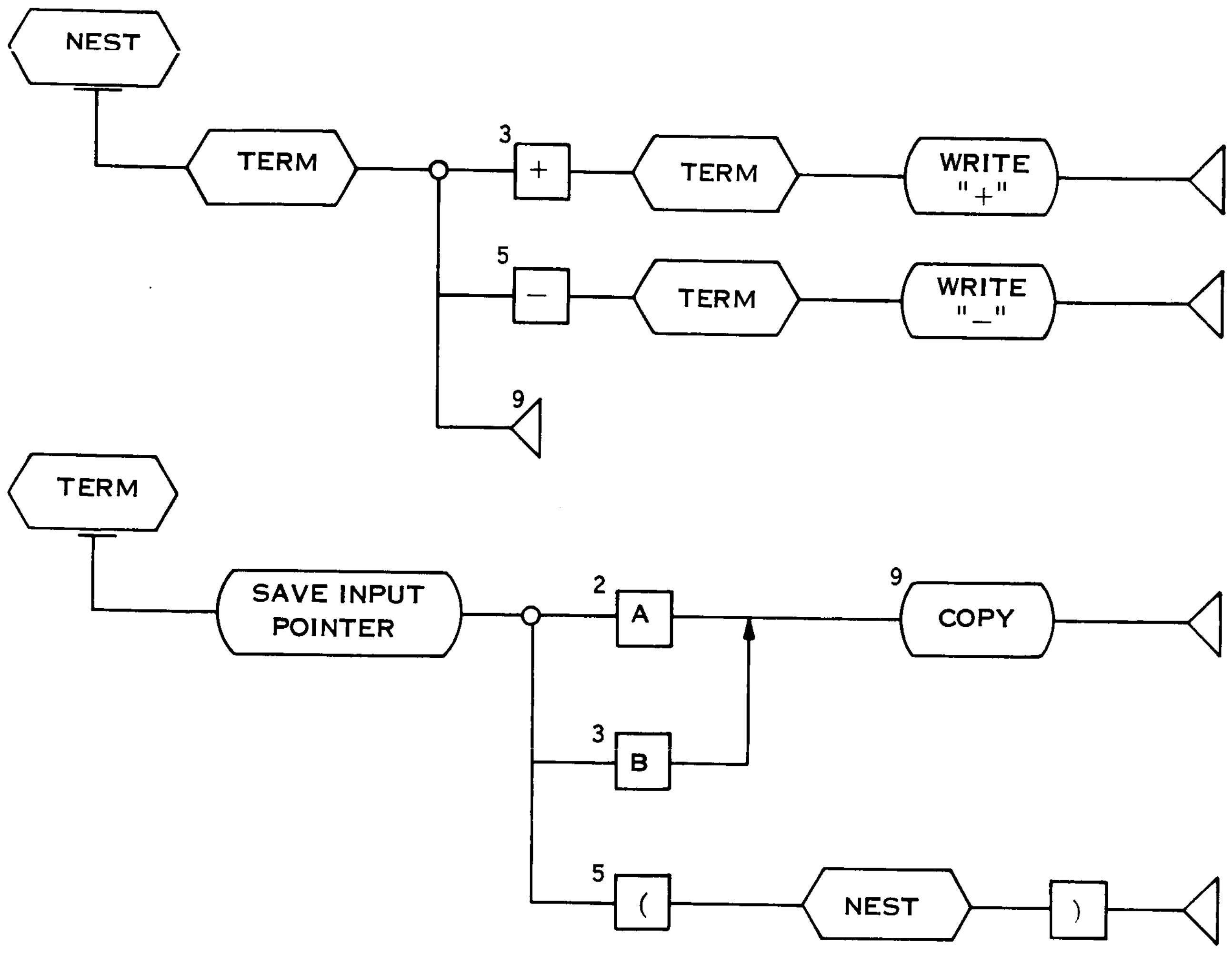


Figure B-10. Infix-to-Suffix Translator

PORTION OF STRING RECOGNIZED	RECOGNIZED BY	SYMBOL ADDED TO OUTPUT STRING
A	TERM	A
B	TERM	B
A + B	NEST	+
(A + B)	TERM	
A	TERM	A
B	TERM	B
A − B	NEST	−
(A − B)	TERM	
(A + B) − (A − B)	NEST	−

Figure B-11. Operation of the Infix-to-Suffix Translator

The sentence begins with the name of the action graph. The name is a string of alphanumeric characters and is set off from the body of the sentence by a colon. The body of the sentence consists of a series of clauses, which contain the actual instructions to be performed, and correspond to the nodes of a pictorial action graph. The clauses are separated by semicolons, and the sentence is terminated by a period.

Since action graphs offer a convenient means of showing the syntax of linear string languages, and since STAG is such a language, the syntax of STAG is shown by the action graphs of Figure B-12. As shown in that figure, a clause consists of an optional tag followed by a choice of various kinds of instructions. The tag is a number used to identify the clause, and is used whenever the clause is to be referred to by other clauses. If used, the tag is set off from the remainder of the clause by a colon. Most of the instructions consist of an operator ("EXECUTE", "CHOICE", "GOOD", etc.) followed by an operand (NAME, LITERAL, CHOICE SPEC, etc.). In some cases, one or the other is omitted.

The correspondence between the STAG instructions and the various Inscan operations described earlier is as follows: EXECUTE calls for the execution of another action graph, the name of which immediately follows. RECURSE specifies that the current action graph should be executed recursively. The Scan operation is specified simply by writing the constant to be tested for. This constant is written as a literal, which is a string of alphanumeric characters enclosed in quotation marks. CHOICE indicates the establishment of a choice point. The various alternative paths are listed in the choice specification, which consists of a series of tags separated by commas and enclosed in parentheses. GOOD represents the successful termination of an action graph. GOTO specifies a transfer of control to another clause of the action graph, the tag of which is written immediately after the GOTO. CALL calls for the execution of an external action point, the name of which immediately follows. The remaining instructions, WRITE, COPY, and SAVE INPUT POINTER, correspond to the various internal action points, described previously, for generating an output string. In the case of WRITE, the constant to be stored is described as a literal and immediately follows the WRITE. COPY and SAVE INPUT POINTER do not require any operand.

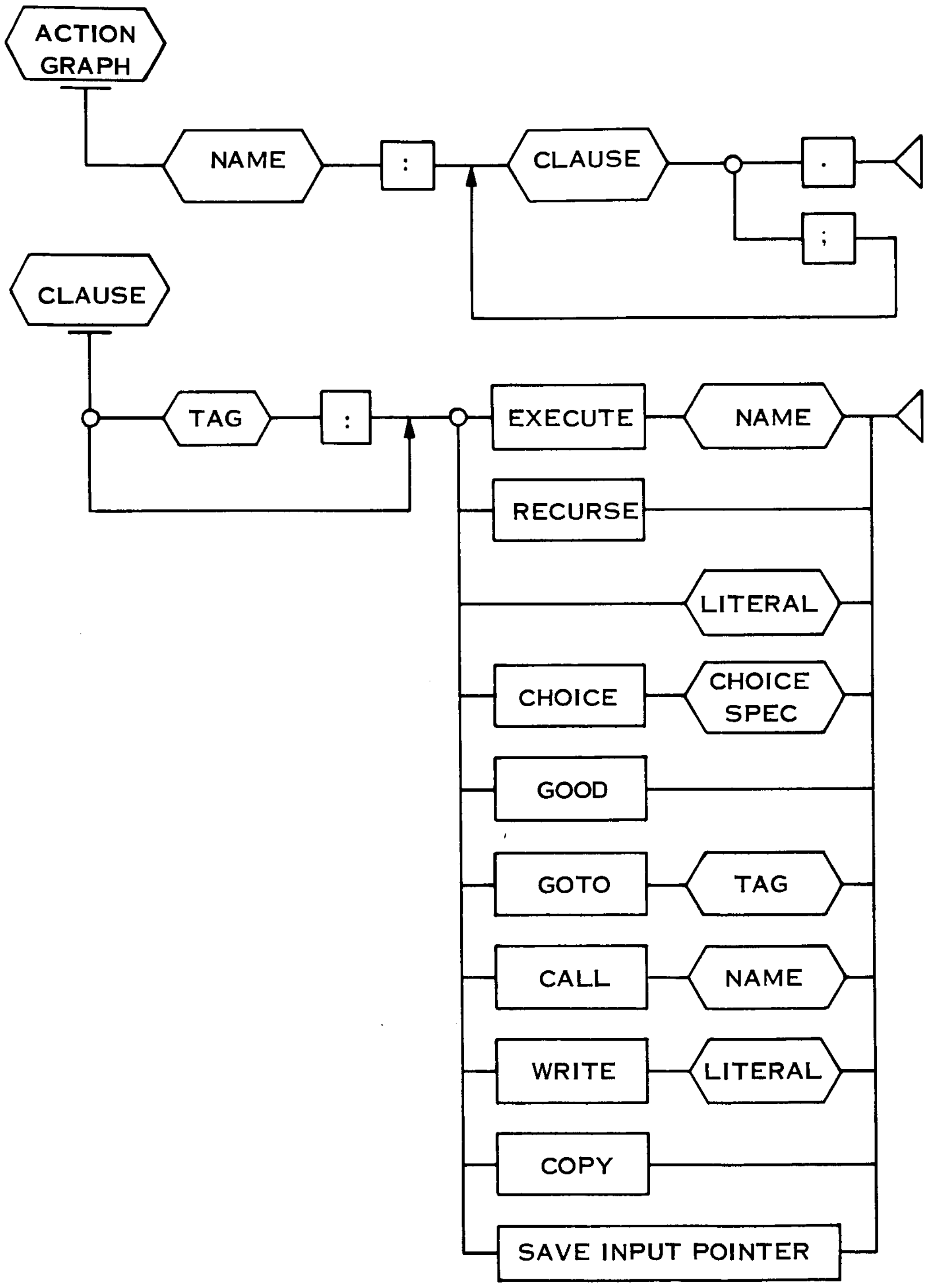


Figure B-12. STAG Syntax

```
NEST:  EXECUTE TERM; CHOICE (3, 5, 9) ;

         3:"+"; EXECUTE TERM; WRITE "+"; GOOD;

         5:"–"; EXECUTE TERM; WRITE "–";

         9:  GOOD.

TERM:  SAVE INPUT POINTER; CHOICE (2, 3, 5) ;

         2:  "A"; 9:  COPY; GOOD;

         3:  "B"; GOTO 9;

         5:  "("; EXECUTE NEST;")" ; GOOD.
```

Figure B-13. STAG — Language Action Graphs for the Infix-to-Suffix Translator

To illustrate the use of the string language, the infix-to-suffix translator of Figure B-10 has been translated into the string language. The result is shown in Figure B-13.

Just as it was possible to describe the syntax of the String Action Graph language by means of an action graph, it is also possible to use an action graph to describe the actions necessary to translate the string language into the Action Graph Table actually processed by Inscan. The relationship of STAG to the AGT is approximately that of assembly language to machine language. In the AGT, instructions are represented by numeric codes, and tags by addresses relative to the beginning of the table. Literals are placed at the end of the table and are referred to by address.

The process of translating STAG into an AGT, like the process of assembly, basically consists of generating the AGT instructions, constructing a "symbol table" relating tags to relative addresses, and then filling the tag references into the AGT. Most of the steps of this process can be handled by action graphs. The approach being taken is to write the necessary graphs in STAG and then hand-translate them into AGT's. These AGT's, processed by Inscan, can then serve as a STAG-to-AGT translator.

B.5 SUMMARY AND CONCLUSIONS

In summary, Inscan, together with the action graphs it processes, is a convenient tool for expressing the syntax of linear languages and for specifying the actions necessary to translate or otherwise process the languages. Inscan has been implemented for two different projects at AUERBACH. One of these implementations provided only external action points and the other only internal action points. The combined configuration, however, is clearly the most general and potentially the most powerful.

Generalized language processors have not been used as widely as possible in user-oriented systems. One of the main reasons for this has been the slow development of languages which combine the ability to specify the syntax and semantics of languages in convenient forms. It is felt that action graphs provide this capability. The Inscan approach to language processor design separates the language scanning and translation function from the details of post-translation processing and facilitates experimentation with the design of languages.

REFERENCES

1. Backus, J. W.: The Syntax and Semantics of the Proposed International Algebraic Language. Information Processing — UNESCO (1960), pp. 125-32.

2. Burkhardt, W. H.: Universal Programming Languages and Processors: A Brief Survey and New Concepts. FJCC, 1965, pp. 1-21.

3. Chomsky, N.: Formal Properties of Grammars. Handbook of Mathematical Psychology, Vol. II (ed. Luce, et. al), pp. 323-418.

4. Connors, T. L.: ADAM — A Generalized Data Management System. SJCC, 1966, pp. 193-203.

5. Conway, M. E.: Design of a Separable Transition Diagram Compiler. Comm. ACM 6.7 (July 1963), pp. 396-408.

6. Dixon, P. J., and Sable, J.: DM-1 — A Generalized Data Management System. SJCC, 1967, pp. 185-198.

7. Feldman, J., and Gries, D.: Translator Writing Systems. CACM. 11.2 (Feb. 1968), pp. 77-113.

8. Floyd, R. W.: The Syntax of Programming Languages — A Survey. Trans IEEE PGEC, Aug. 1964, pp. 346-53.

9. Gorn, S.: Specification Languages for Mechanical Languages and Their Processors — a Baker's Dozen. CACM. 4.12 (Dec. 1961), pp. 532-42.

10. Griffiths, T. V., and Petrick, S. R.: On the Relative Efficiencies of Context-Free Grammar Recognizers. CACM. 8.5 (May 1965), pp. 289-300.

11. Steel, T. B. (ed).: Formal Language Description Languages for Computer Programming. North-Holland Publishing Co., 1966.

12. Steil, G. P.: Pure. ADAM Project Report #87 (27 Sept. 1963), Department D-19, MITRE Corporation, Bedford, Mass.

13. Taylor, W., et al.: A Syntactic Chart for ALGOL-60. CACM. 4.9 (Sept. 1961), p. 392