IBM CONFIDENTIAL

SDD POUGHKEEPSIE
March 1, 1971

Memorandum to: Recipients of Advanced Future System (AFS) Proposal

Subject: Poughkeepsie AFS Proposal

Enclosed is a copy of the present draft of our proposal for AFS architecture. Though a great deal of progress has been made, this is a report on progress and not a final report. In one sense the ideal approach would have been to let the two principal authors of the conceptual work, Steve Zilles and John Sowa, proceed to the production of a complete document stating a consistent set of fundamental concepts with absolute academic purity. Then others could propose and describe practical implementations. Such an approach would have two overwhelming flaws:

1. The resulting proposal would suffer from a lack of contact with realities of what is required to implement a system, no matter how elegant the document.

2. No individual or pair of individuals is fully equipped to deal with all of the diverse disciplines that are part of the design of a full hardware-software system.

Instead, our approach has been to simultaneously develop pragmatic detail and abstract conceptual foundation. This admittedly has resulted in false starts and frequent rethinking of basic issues. The great benefit has been in the testing of concepts and of implementation against the sternest measure of all -- an orthogonal point of view. This approach has resulted in a study which is technically both broad and deep.

The process of synthesis is not yet complete. It has resulted in much change in both points of view. Only when this change has stopped will we consider the proposal complete, consistent, and final.

The two principal parts of our proposal are entitled "Fundamental Concepts and System Language" and "System Architecture". These will be referred to as the SL report and the SA report or as SL and SA. They were prepared simultaneously with related but different points of departure.

The purpose of the System Architecture study was to explore the feasibility of this approach and to discover and solve problems arising during implementation.

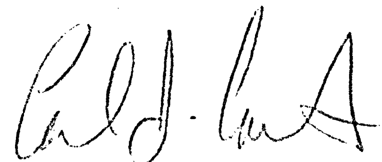
Creating a simple, generalized, logical foundation and finding a way to create a system language that can meet the objectives of AFS and to serve as an adequate system control and task management language and as an adequate target language for translators from high level languages was the purpose of the System Language study. The objective of producing a simple, coherent, design with a minimal number of concepts and conventions is primary.

The goal of discovering how to create a high level language that is a suitable target for translators seems within our grasp. Much supporting evidence is contained in this report in the form of definitions of functions which support fundamental language properties. Of particular importance are the functions and definitions that are aimed at supporting language facilities that have not yet been specified but will be important by the time this product reaches the market. More evidence is contained in ASP 051, "Arithmetic Operations in AFSIL".

Representing our case for feasibility is the System Architecture report. There are two aspects. On one hand, we created a simulator that dealt with all the problems associated with the critical parts of PL/I. To do this we couldn't leave any unsolved problems hidden under vague phraseology since the simulator runs. In the SA report are the solutions to problems that are glossed over or even ignored in the SL report. It may be less evident that the advanced approach in SL can be implemented with an extension of the same approach. However, close study of the two reports and discussion with the contributors will I hope convince you, in fact, what we propose can actually be done.

The second major topic of SA is efficiency. We are gratified to find that our intention of creating an efficient system appears to be quite reasonable in view of the performance being predicted through simulator usage.

Reading and understanding this report will be a sizable undertaking. To assist you, key contributors can be made available to present whatever is required.



C. J. Conti

CJC:cpl

DO NOT REPRODUCE - FOR FURTHER COPIES, PLEASE CONTACT
C. J. CONTI's OFFICE - EXTENSION 3-2531

AFS SYSTEM ARCHITECTURE MANUAL

February 26, 1971

This document contains information of a proprietary nature. All information contained herein shall be kept in confidence. None of this information shall be divulged to persons other than: IBM employees authorized by the nature of their duties to receive such information, or individuals or organizations authorized by the Systems Development Division in accordance with existing policy regarding release of company information.

IBM CONFIDENTIAL

TABLE OF CONTENTS

Table of Contents 2

Foreword 3

CHAPTER 1 INTRODUCTION 4
- 1.1 Purpose of the System Architecture Manual (SAM) 4
- 1.2 Principal System Features 5
- 1.3 System Structure and Principal Components 6
- 1.4 Using the System 10

CHAPTER 2 THE LOGICAL MACHINE 12
- 2.1 Overview 12
- 2.2 The Program Tree 16
- 2.3 The Activation Tree 19
- 2.4 Dynamic Storage 22
- 2.5 Data Objects and Linking 26
- 2.6 The Interpreter 35
- 2.7 Building a Logical Machine 43

CHAPTER 3 THE LOGICAL SYSTEM 45
- 3.1 The Logical Machine Supervisor 45
- 3.2 System Facilities 47
- 3.3 The Logical Input/Output System 53

CHAPTER 4 THE PHYSICAL SYSTEM 56
- 4.1 Control 58
- 4.2 The Storage Management Subsystem 64
- 4.3 The Program Processing Subsystem 79
- 4.4 The Source-Sink Subsystem 93

CHAPTER 5 MODELING 103
- 5.1 Description of Models 103
- 5.2 Model Usage Results 105
- 5.3 An Instruction-Level Machine Compared with a Higher-Level Language Machine 114
- 5.4 Model Plans 116

CHAPTER 6 GLOSSARY 117

FOREWORD

This document has been prepared as a manual and thus order of presentation is given in reference form. For a reader who is being exposed to the system for the first time, it may be useful to read the document in a different order. The suggested procedure is to first read the Introduction (Chapter 1); then Section 2.1, the Overview of the Logical Machine; then Chapter 3, the Logical System; then Section 4.1, the Control Description of the Physical System. With this preparation, it is hoped that the reader will be able to use the document as intended.

The authors have used several conventions to aid the reader. When a significant term or phrase is introduced for the first time it is underlined. Most terms which are capitalized can be found in the Glossary (Chapter 6).

All readers are invited to submit their commentary on the system and this document. Please contact A. Peacock, Department B11, SDD Poughkeepsie.

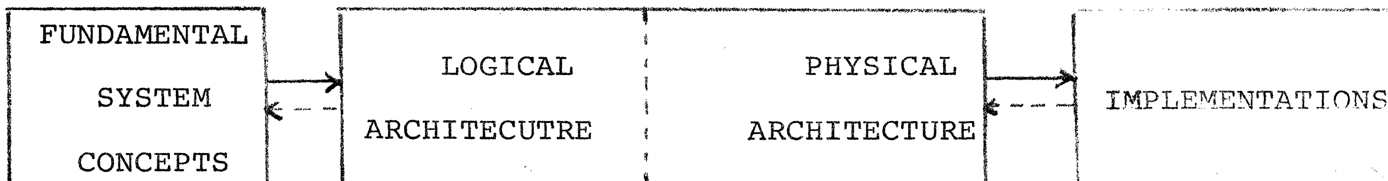
IBM CONFIDENTIAL

CHAPTER 1

INTRODUCTION

1.1 Purpose of the System Architecture Manual (SAM)

SAM is one of a set of documents prepared by the Advanced Systems Group in Poughkeepsie in response to the AFS requirements and objectives which were issued jointly by Carl Conti (Manager of Advanced Systems, Poughkeepsie) and Al Magdall (Manager of Advanced Systems, Endicott) on January 19, 1971. Two other major documents in this set are the System Language Manual and a subset implementation (described in ASP memos 015, 046, and 049)*. The relationship of these documents is simply illustrated:



A major purpose of SAM, then, is to provide a communication interface between the abstract, analytical, description of the system in the System Language Manual and one or more implementations. A second major purpose of SAM is as a convenient introduction to the system for skilled, but unfamiliar, readers.

It is inevitable, and even desirable, that the early versions of these three document sets are not completely compatible. Some differences, primarily in nomenclature, between the Systems Lanaguage Manual and SAM are discussed in a Glossary of terms (Chapter 6 of this Manual). More serious technical differences are, or will be, discussed in ASP memos. Ultimately, SAM will become, as these differences are resolved, an approved Reference Manual for the system.

* Advanced System Proposal Department B11 SDD Poughkeepsie

1.2 Principal System Features

The novel features of this system are designed to bring about a major improvement in programmer productivity; to make the system easier to use and maintain ; and to penetrate new market areas, particularly Data Base Operation.

Central to the heart of the system is the concept of Data Independence. Specifically, the executable code references data objects by a logical name rather than physical location; the logical descriptions of data objects are maintained with the data values and not with the code; and the physical representations of data objects (both descriptions and values) are not known, implicitly or explicitly, to the executable code. Thus, the operators of the executable code are generic and may be applied to a class of data objects without regard to their current physical representation, or location.

The system has been designed to make possible the faithful support of the major high level languages; in particular COBOL, PL/1, APL, RPG, and FORTRAN. Thus a user of such a language can be provided with a logical machine that appears to him to be directly executing statements in that language. The system will detect all the errors defined in the language and report back to him in terms of his source statements. In light of these errors, he may modify his source text and, where this is meaningful, continue execution from the point of error discovery. As a natural consequence of this mode of operation, the execution unit treats a statement (i.e. a list of operators and operands) as the unit of execution rather than an individual instruction.

The system also features a new System Language which combines the semantic power of the best procedure oriented languages, with the operators necessary to support system programmers in their task of building and maintaining application programs, operating systems and language compilers. Since the necessary operators to build software support are directly supplied in the system language, the language controls and defines the whole system in the same manner that System/360 Principles of Operation controls the main-frame hardware of System/360.

A last important feature of the system is the clean separation of the logical (user-oriented) definition of the system from the physical (implementation-oriented) definition,

with rigorously controlled interfaces between them. This makes possible a wide variety of implementations and extensions without jeopardizing the logical foundation.

1.3 System Structure and Principal Components

1.3.1 The Logical Machine

Every user Job is processed in a separate Logical Machine (LM). A Job is a major unit of work in the system. All Jobs are scheduled and run independently; all communication and sychronization between Jobs is the responsibility of the system users, using explicit communication mechanisms provided by the system.

An LM performs processing on behalf of the user by activating one or more Modules which are contained within the Logical Machine in a structure called The Program Tree. Both nested and parallel activation of Modules may occur within the LM, the current activation status being recorded dynamically in The Activation Tree. Connectivity to the current generations of variables used during processing is made by a set of Storage Anchors, one set being provided for each parallel activation.

The active mechanisms within a Logical Machine are called Interpreters. Every parallel activation is supported by a separate Interpreter; implicit communication (through common variables) is possible between Interpreters in the same Logical Machine. The work accomplished by a single Interpreter is called a Logical Task. Thus a Job consists of one or more Logical Tasks, the number of Tasks existing at anytime being identical to the current number of parallel activities.

THIS IS A STRICTLY PROCEDURE-ORIENTED VIEW OF JOB.

Explicit communication with the other logical components of the system (i.e. System Facilities, the Logical Machine Supervisor, and the Logical I/O System, is provided by a series of specialized communication modules accessible from the root (System Node) of the Program Tree.

Just below the System Node in the Program Tree is the External Node. The Module at this node is called the External Module and is activated (i.e. supplied with an Interpreter) when the Logical Machine starts a Job. This activation includes passing a parameter to specify the source of initial commands for the machine.

1.3.2 The System Facilities

Every Logical Machine has access to a number of special system functions. These capabilities are called System Facilities and include the ability to catalogue data, obey system commands, edit catalogued data, introduce procedures, etc. All of the logical objects in the system (cataloged data structures, Modules, Logical Machines, Logical I/O devices, Logical Users, etc.) are themselves owned by another logical object. The set of nested ownership relations is reflected in the Ownership Tree which is maintained by the System Facilities. Access to objects is monitored by System Facilities (this monitoring is automatic since it is only achievable through the communication modules of the LM's) to ensure that the accessor is authorized to access the object and to provide the necessary interlocks required by the type of access (read-only, modify-exclusive, etc.). Many of the leaves and nodes of the ownership tree are themselves complex structures. This fine structure is not directly reflected in the Ownership Tree which resolves only to the detail necessary to provide unique ownership and access rights.

AUTHORISATION WOULD BE A BETTER WORD

Any object in the Ownership Tree can be created, copied or modified by an active Logical Machine that has the appropriate authorization by using its communication modules. Thus one LM can create, copy, modify or pass messages to another LM. An active Logical Machine can in fact be thought of as a logical object in the Ownership Tree that is in the modify-exclusive state.

1.3.3 The Logical Machine Supervisor

One specialized LM, called the Logical Machine Supervisor, is placed at the root of the Ownership Tree and activated at System Generation time with its External Module connected to an operators console. Other specialized LM's are provided by the systems programmers (i.e., users with the appropriate authorization) to support the normal system functions of translating (i.e. creation of a Module from a Data Object), Logical I/O , Data and Load Module editing, etc.

The principal logical task of the supervisor is to activate Logical Machines (including parametising their External Modules) and to monitor their activities in order to handle exceptional conditions such as job completion. Depending on the type of user activity this may involve providing the user with a new Logical Machine, connecting him to an existing LM or making a fresh copy for him of an existing LM.

Awow!

The supervisor also has the important task of communicating with the physical control system to initiate Physical Tasks for either a Program Processing Unit or a Source-Sink Processing Unit. The communication interface between the supervisor and the physical system consists primarily of a set of tables representing the current status of work in the system. The supervisor maintains logical (user-oriented) tables representing the users' Jobs and their subdivision into Logical Tasks (parallel activations within a Logical Machine) which occur upon execution of attach operators in a Logical Machine.

These Logical Tasks are analyzed and monitored by the supervisor which breaks them into one or more Physical Tasks and places them into queue tables for Processing Units to be scheduled by the Physical Control System. A Physical Task includes in its definition the kind of processor required, any physical resource requirements, the priority of execution, and the scheduling discipline (batch, time-shared, etc.).

1.3.4 The Logical I/O System

The Logical I/O System is responsible for all transfer of information to and from a Logical Machine. This includes communication with a user, another LM, the System Facilities, Source/Sink devices and other systems. Logical I/O is provided by a series of specialized Modules, which are placed at the System Node of each LM's Program Tree. Thus any LM can obtain Logical I/O by a simple call or an Attach of one of these Modules.

Logical I/O does not necessarily result in the creation of a Physical Task for a Source/Sink Processing Unit. Some requests can be satisfied by the temporary incorporation of an object in the Ownership Tree into a Logical Machine's environment. Others result in copying or creating such an object which can be done as a Physical Task by a Program Processing Unit. Source/Sink Processing Units are, however, required for Source/Sink I/O, user communication, and communication with other systems.

1.3.5 The Physical System Control

The Physical System, which supports all the structures and activities of the Logical System, consists of three major sub-systems: The Storage Management Subsystem, The Program Processing Subsystem, and The Source/Sink Processing Subsystem. The harmonious cooperation of these sub-systems and their allocation to the Physical Tasks which represent the

activities of the Logical Machines is ensured by a set of hardware and software capabilities known as Control.

Control, at the lowest level, consists of a Storage Bus which enables the Processing Sub-systems to communicate with the storage Sub-system and hence, indirectly, with each other. A minimal direct signalling capability, called the Interrupt Bus, is also provided at this level. At the next level, control consists of a number of task queues in storage (one per processor type) together with the basic operators to build, search, and control access to the queues. Thus processors have the basic ability to transfer tasks to other processors and search for tasks for themselves. At a higher level, control is a high priority Physical Task which is responsible for physical resource allocation and normal and abnormal task termination. Finally, at the highest level, control is a set of physical tasks which constitute the Logical Machine Supervisor.

1.3.6 The Storage Management Sub-System

The Storage Management Sub-system is an automatic, paging hierarchy of storage devices ranging from the highest speed electronic memory to lower speed electro-mechanical devices and out through operator controlled shelf-storage. It differs from conventional 'virtual memories' in three important respects.

Firstly, rather than one, the system provides a practical infinity of independent linear spaces, each of which can grow or shrink independently. Secondly, the system only manages real spaces (i.e. spaces that contain data) and only to the current physical length of the space. Finally, spaces are created and grow in the page buffers of the highest speed electronic memory so that 'Get Main' is a very fast operation.

1.3.7 The Program Processing Sub-System

This Sub-system consists of one or more Program Processing Units (PPU) which are similar in function to a conventional CPU, but which operate at a higher level, close to a Procedure Oriented Language. They process statements rather than instructions; provide a wide range of control operators, and use described data rather than simple bytes to represent data values.

The PPU is essentially designed as a Prefix-Polish Stack Machine, extended by a set of Storage Anchor Registers which perform the function of base registers, and with built in vector and structure handling operations.

1.3.8 The Source-Sink Processing Sub-System

This Sub-system consists of one or more Source/Sink Processing Units (SSPU) together with the conventional source/sink devices and communication lines. The SSPU is a complete processing unit containing most of the capabilities of a PPU rather than a simple data channel. Thus it is capable of handling the complete task of data-transmission to and from storage without interrupting a PPU for intermediate processing.

The principle functions of the Source/Sink Processing Units are line control, network management, source/sink device management, control of the system clock and handling "wake-up" type interrupts to and from the PPU's. Both local and remote source/sink devices are handled by the SSPU, so that the difference is "transparent" to the casual user.

1.4 Using the System

The system is designed for a wide variety of users and uses, ranging from continuously running automatic jobs, through conventional application programs, to the design, debugging and operation of complex operating systems and data bases. This large skill range is provided for in several ways.

Firstly, a very powerful set of control operations is provided (obviously some of the more complex and infrequent of these are implemented in IBM supplied software or micro-code). Secondly, however, to preserve system integrity and provide high performance, many of these functions (such as Storage Management) have been built in, with rigorously controlled interfaces which do not allow for alternative approaches.

Superficially it would appear that naive users, provided with such rich capabilities, could overwhelm the entire system. A very wide degree of control is provided, however, by controlling the Node in the Ownership Tree to which a user is connected, the kind of Logical Machine that is placed at that Node, and the priority he is assigned.

Some idea of the scope of possibilities is provided by some simple examples:

1.4.1 Sensor-Based Applications

NO ONE HERE EVER SAW A SENSOR.

These applications are performed on continuously active Logical Machines which are simply waiting on I/O interrupts. They have modules of code in their External Nodes which interpret commands to change parameters but do not build or modify themselves, or other Logical Machines.

1.4.2 Standard Batch Application Programs

These programs differ only slightly from Sensor-Based ones, in that their Logical Machine is inactive between runs. Thus, to be started, they require a sign-on and the intervention of the Logical Machine Supervisor. The addition of the sign-on capability extends the potential users of any application; however, this can be restricted to the required degree by placing the Logical Machine in the appropriate node in the Ownership Tree.

1.4.3 Problem-Solving and Debugging Simple Applications

The user operating in this mode signs on to a Logical Machine which initially contains merely a System Node and an External Node. The System Node contains access to the complete range of System Facilities but his capability to use them depends on the interpretive range of the External Node. In general, this node can interpret the full command language of the system but the designer of the Logical Machine may choose to restrict this capability by supplying a specialized External Node (e.g., he may only wish to supply the command language of the APL/360 system). The user once signed on to such a Logical Machine will typically be able to improve and specialize its processing capabilities by Connecting Modules below the External Node. These modules may be already in the Ownership Tree or may be introduced by him in any of the languages supported by the system (i.e., any language for which a Logical Machine that translate source text to a module exists).

Subsystem Builders

The designer of a subsystem creates one or more Logical Machines which are to be placed into the Ownership Tree for the use of less sophisticated people. When the designer signs on for this purpose he is provided with a Logical Machine with a complete System Node and External Node. The commands which are particularly important to him are the ability to create a Logical Machine (specifying the Module to be placed in the External Node and the position in the Ownership Tree) and to call or attach one Logical Machine from another Logical Machine. Additionally, he needs to be aware of the physical system supporting a Logical Machine in order to achieve the appropriate cost performance.

CHAPTER 2

THE LOGICAL MACHINE

2.1 Overview

A Logical Machine (LM) is that part of the system provided for processing each independent job. The LM's function is to execute code in the proper environment on behalf of the user. The LM, in addition to executing code, possesses the capability to communicate with the other logical components of the system (e.g. System Facilities, the Logical Machine Supervisor, and the Logical I/O System).

Logical Machine Components

In order to execute code in the proper environment, a Logical Machine must contain the mechanisms to access the proper operands at execution time, provide the necessary operators, maintain program control, and accomplish the above while being "faithful" to the rules of the language used by the user. The mechanisms of the LM used to accomplish this are: a Program Tree to define the static connection of Modules, an Activation Tree which defines the dynamic linking of Modules, a Dynamic Storage Mechanism to provide the generation identification capability for data objects and Interpreters to execute the code.

Interpreters are the mechanisms which perform all operations required, and the Program and Activation Trees, and the Dynamic Storage Mechanism provide them the necessary information to obtain the proper results.

Each LM contains certain built in functions which are provided as special nodes in the Program Tree.

Program Tree

The Program Tree is the mechanism for name resolution. The nodes of the tree are modules which have been generated by Translators and grafted by the Connector. Each node contains the list of its symbols in a Local Symbol Table (LST), an accompanying Local Declare Table (LDT) describing the symbols, and a Local Link Table (LLT) for linking the symbols to their

values. Each node usually contains executable code.

Activation Tree

Each node in the Activation Tree corresponds to an activation of a node in the Program Tree. The Activation Tree is a mechanism for maintaining the required information during Tasking, as well as Calls and Returns.

Dynamic Storage Mechanism

The Dynamic Storage Mechanism provides a set of named Storage Anchors used as starting points for generations of variables. These point to the appropriate Reference Table for each variable. A System Storage Anchor is automatically supplied for each lexical level in the Program Tree. User Storage Anchors are supplied as required by user allocated variables.

Interpreters

Interpreters are the logical executors of code and are the source of all functions/operations in the logical machine.

An interpreter executes on a statement basis, maintains the statement counter, and provides inter-statement control as determined by the code (e.g. IF, DO, GO TO) and intra statement control using the operand descriptor information provided (e.g. vector operations).

Use of the Logical Machine

Logical Machines are activated by the Logical Machine Supervisor. This will occur when a job is created; for instance, when a user signs on the system. The LM then implements the functions required to process the user's program. A general flow of creating and executing a user's program is shown in Figure 2.1.1.

Translate takes the user's source code (APL, PL/I, FORTRAN, COBOL, RPG, etc.) and transforms it into an internal form of the System Language. The translate process accepts source code one line at a time, checks for syntax, and builds a Module. A Module consists of executable code together with information tables.

IBM CONFIDENTIAL

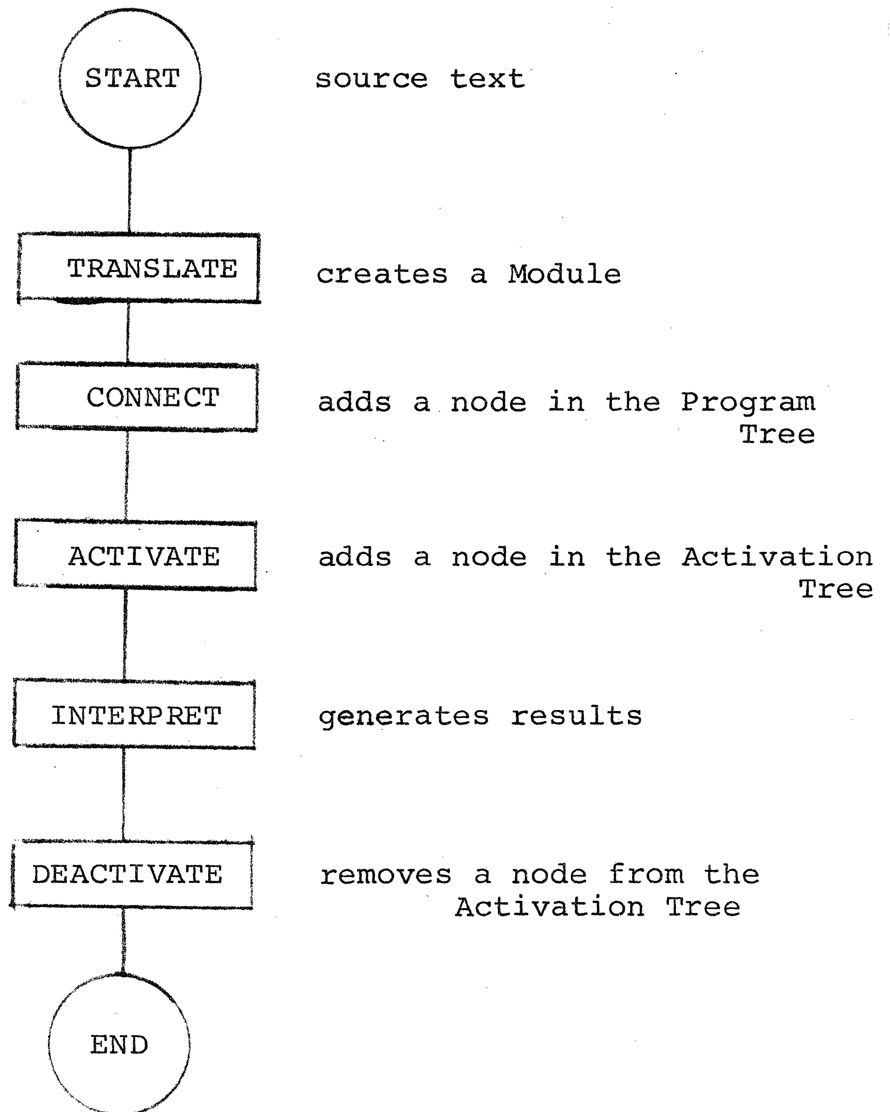


Figure 2.1.1 CREATING AND EXECUTING A PROGRAM

Connect takes a Module and places it into the Program Tree which represents the static connectivity between the Modules of the program.

Activation of a module is the process of creating a node for it in the Activation Tree. The activation tree represents the dynamic structure of the program.

Names are resolved using the Program Tree. However, a name may have several generations of values associated with it. The process of determining the appropriate generation is resolved by the Dynamic Storage Mechanism which consists of a number of Storage Anchors from which the appropriate generations are chained. The result of activation is the initiation of an interpreter.

The Interpreter executes the code which is in the appropriate node of the Program Tree. Each parallel task has a separate interpreter. Interpretation is the process of executing the code in the appropriate module in the Program Tree. The result of the interpretation is the "answer" as defined in the user's code for that module.

Deactivation removes the node from the Activation Tree and removes the generation of variables associated with the activation.

2.2 The Program Tree

Each Logical Machine has a Program Tree. The purpose of the Program Tree is to contain the static connectivity between the Modules of the Job executing in the Logical Machine. (An example of a Program Tree is shown in Figure 2.2.1) Each node of the Program Tree represents a Module and contains the executable code for that Module, together with the Local Symbol Table containing all the Symbolic Names referenced in the Module, and the associated Local Link and Declare Tables. Each node also contains the names of, and the connectivity to, the Modules which have been connected into the Program Tree directly below that node. The Program Tree is used by the Linker in the resolution of the Symbolic Names referenced in the executable code of the Module. Further nodes can be created in the Program Tree for a particular Logical Machine by the execution of a Connect command in that Logical Machine.

There are two nodes in the Program Tree which have special properties and functions. The root node of the Program Tree is termed the System Node and contains the names of, and connectivity to, the system functions which are available to the Logical Machine. The System node also contains connectivity to the External Node. The External Node contains the Local Symbol, Link and Declare tables for all the external names of the program. The executable code of the Module represented by the External Node has the function of interpreting the Command Language. Whenever a new Logical Machine is created, it will be initiated to be executing the Module, associated with the External Node. The source of the command to be interpreted must be specified as a parameter when the Logical Machine is initiated. For example, it might be an interactive terminal, or a catalogued data set.

2.2.1 The Use of the Program Tree for PL/I and APL

For a PL/I program there is a node in the Program Tree for each Procedure or Begin block. The Program Tree reflects directly the static block structure of the program. The External Node contains the names of, and connectivity to, every external procedure and every external name of the program. The name resolution for the symbolic Names referenced in each Procedure or Begin block is done by a search of the Program Tree. The subsequent steps required to obtain a reference to the correct generation of the variables by means of a storage anchor are discussed under Dynamic Storage Control in Section 2.4.

For an APL program there is a node in the Program Tree directly below the External node for every function. All the names referenced in the APL program appear in the External Node as user allocated external variables.

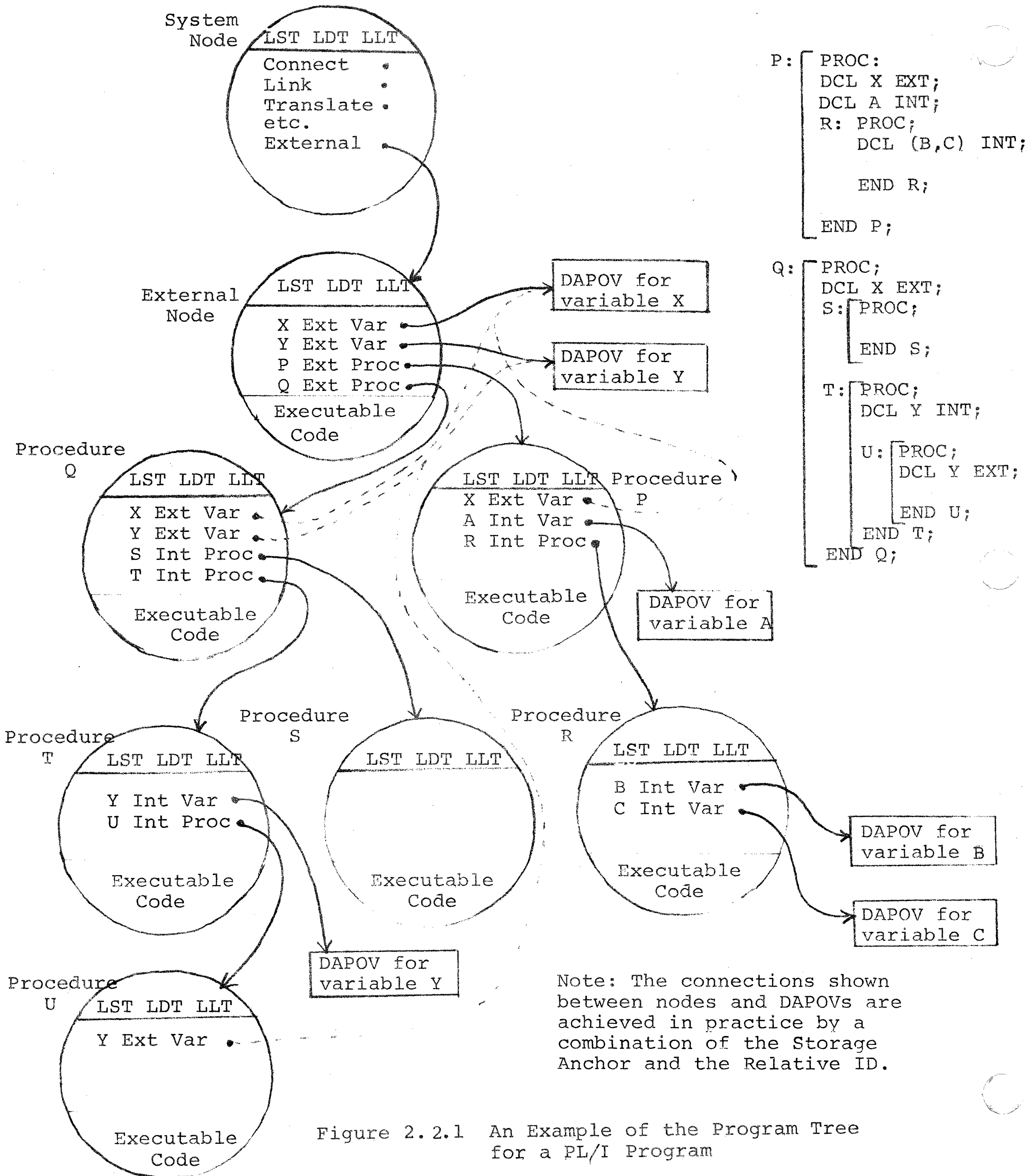
Hence, there will be a Storage Anchor name for each symbolic name in an APL program (see Section 2.4).

2.2.2 An Example of the Program Tree

The diagram in Figure 2.2.1 shows an example of the Program Tree for a Logical Machine which has had the PL/I program shown to the right of it connected into it.

Note: The Program Tree shown has been simplified by the omission of the nodes corresponding to the "imaginary" blocks which are said to contain the external procedures of a PL/I program. The purpose of each imaginary block is to contain the external entry names of the external procedure so that they can be seen by any reference statically contained within the procedure, but not by references statically contained within any other external procedure. The entry names, being external, will of course appear in the External Node, but the PL/1 language rules say that they will only be found there by explicit declarations of the name as "external". These rules can be incorporated into the Program Tree by the inclusion of a node between the External Node and the node for each PL/I external procedure. These nodes correspond directly to the "imaginary" blocks.

IBM CONFIDENTIAL



Note: The connections shown between nodes and DAPOVs are achieved in practice by a combination of the Storage Anchor and the Relative ID.

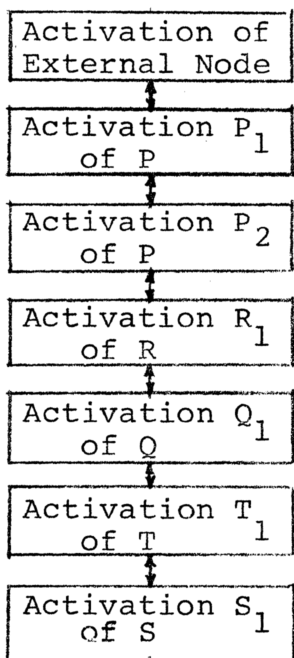
Figure 2.2.1 An Example of the Program Tree for a PL/I Program

2.3 The Activation Tree

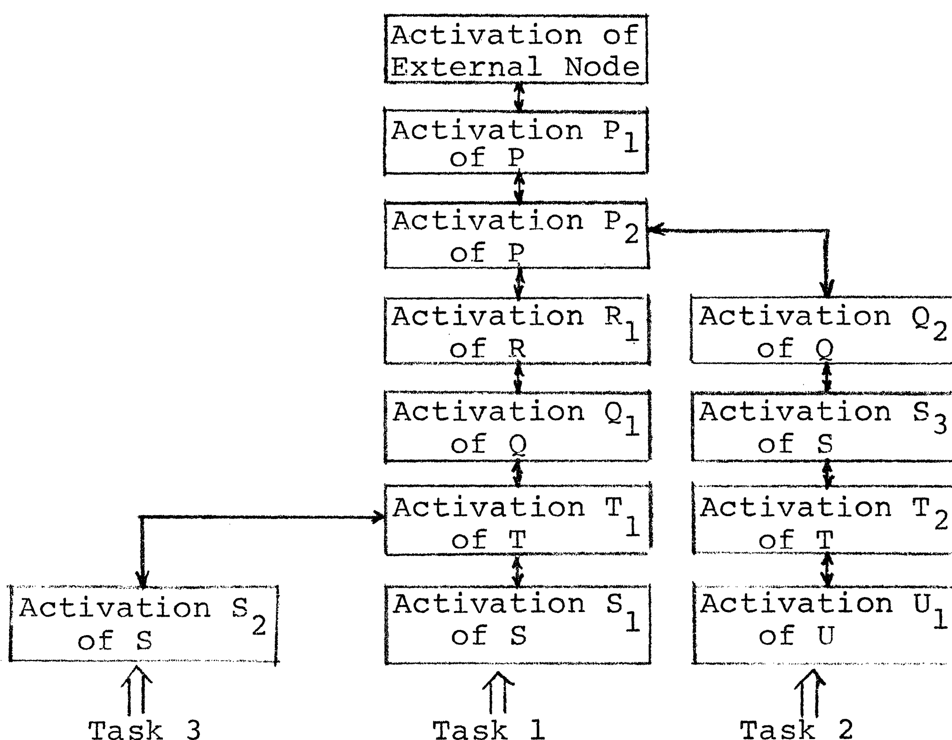
Each Logical Machine has an Activation Tree. The purpose of the Activation Tree is to contain information about which Modules are currently active and the order in which they called or attached each other. Each node of the Activation Tree corresponds to an activation of the Module associated with one of the nodes of the Program Tree. The root node of the Activation Tree corresponds to the activation of the Module associated with the External Node of the Program Tree. This node is activated by the creation of the Logical Machine. When there is no multi-tasking within the Logical Machine, the Activation Tree has only a single limb.

The following examples represent Activation Trees which could exist during the execution of the PL/I program shown in the example of the Program Tree in Section 2.2.1.

Non-Tasking Case



Multi-Tasking Case



2.3.1 Invocation by Call or Function Reference

The invocation of a Module by the execution of an explicit call or of a function reference, causes a new node to be added to the branch of the Activation Tree representing the task in which the invoking operation was executed. Information is stored in the node of the activation tree to provide a link to the node of the Program Tree corresponding to the Module activated, and to enable a return to be made to the point immediately following that from which the Module was invoked.

Two types of Return are possible, these are usually termed "normal" and "abnormal" Return. Normal Return occurs when the current activation is terminated, and execution is resumed at the point remembered in the node of the Activation Tree. Abnormal Return occurs when a GO TO statement nominates a non-local label value as its operand. The current activation, and any intervening activations are terminated, and execution resumes at the label point specified in the GO TO statement.

2.3.2 Invocation by Attach

The invocation of a Module by the execution of an explicit Attach causes a new branch to be added to the Activation Tree. At this point the only element in this branch will represent an activation of the Module specified in the Attach. There is no return information associated with the first activation of a task. Once established, a new branch in the Activation Tree can grow and shrink in the same way as the main branch and can itself become the root of new branches.

Two types of task termination are possible. Normal task termination occurs when a Return statement or the last statement of the Module attached has been executed. The Task will be abnormally terminated if the activation which initiated its first activation itself terminates.

2.3.3 The Handling of Exceptional Conditions

The handling of exceptional conditions falls into three parts: the establishment of the action to be taken if an exceptional condition arises, the search to determine the action to be taken when an exceptional condition arises, and the invocation of the Module which is to perform the specified action.

Each condition name will be associated by the Linker with a stacking mechanism of the type used for user allocated variables (see Section 2.4). This stacking mechanism will be used to contain the entry value (Module, entry point, and environment) of the action which is to be taken if the condition arises. This action may have been defined by the user (e.g., a PL/I on-unit) or by the system (e.g., Standard System Action for a PL/I condition).

The action to be taken when the condition arises will be determined from the latest entry in the branch of the stack corresponding to the current task. The invocation of the Module giving this user or system provided action will follow exactly the protocol for the invocation of any other entry value (see Section 2.4.1).

2.4 Dynamic Storage

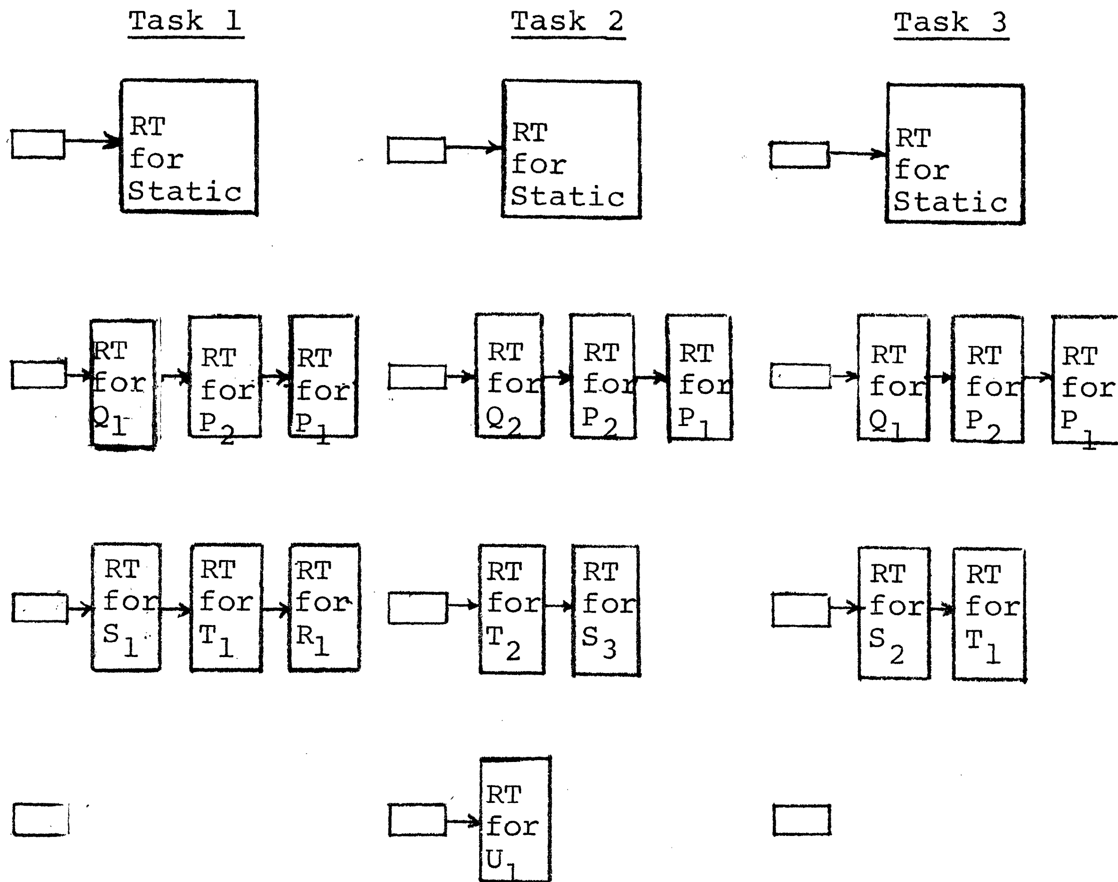
The purpose of the Dynamic Storage Mechanism is to provide storage for the Data Objects and to provide execution time addressability to the appropriate generations of the Data Objects referenced by Symbolic Names in the Modules of the program. Each generation of a Data Object consists of a Value Descriptor and value (DAPOV) as defined in Section 2.5. These DAPOV's are allocated in a Reference Table and each one can be addressed within that Reference Table by its Relative ID. Each Reference Table can contain DAPOV's for one or more variables. Addressability to each Reference Table is provided by a Storage Anchor. The total set of Storage Anchors can be divided into two parts, the set of System Storage Anchors and the Set of User Storage Anchors.

The System Storage Anchors are used to address the Reference Tables directly associated with the activations of the Modules represented by the nodes of the Program Tree. Each of these Reference Tables contains the DAPOV's for the variables which are allocated at the activation of the Module. Since only the names contained in one node at each level in the Program Tree can be referenced directly at any time, it is sufficient to have one System Storage Anchor for each level in the Program Tree. The names of the Storage Anchors to be associated with the nodes of the Program Tree will be filled in by the Connector. All the Reference Tables which use the same Storage Anchor must be chained together so that the value of the Storage Anchor can be updated as the Reference Tables are freed.

The User Storage Anchors are used to address the DAPOV's for those variables whose allocation and freeing is under the control of the user. The names of the Storage Anchors for these variables will be filled in by the Linker.

There must be a set of Storage Anchors for each Task executing in the Logical Machine, since the Reference Tables containing the current generation of the variables may be different in each task. When a Task is attached, unless there is a change of environment as described in Section 2.4.1, the only Storage Anchor to be changed is the one corresponding to the Module being activated. The rest of the Storage Anchors must be copied to provide addressability from both the Tasks to the Reference Tables associated with any Modules further up the Program Tree, and to any variables which have their own User Storage Anchors.

The System Storage Anchors for the multi-tasking example shown in Section 2.3 can be illustrated as follows:



Note: Although there is a Storage Anchor in each Task identifying the "RT for Static", there is only a single Reference Table for static variables; it can be addressed from any Task. The same comment applies to "RT for P_1" and the other Reference Tables which appear in the chains of a Storage Anchor in more than one Task. "RT for P_1" represents the Reference Table for activation P_1 of P, whilst "RT for P_2" represents the Reference Table for activation P_2 of P.

2.4.1 The Use of Storage Anchors for the Support of Entry Values

The environment which must be assigned with an entry value, together with the value of the entry point, can be seen

as the values of the Storage Anchors associated with each of the statically containing nodes. Again using the example of Section 2.2 and 2.3, the assignment of entry point R to a static external entry variable EV in activation P_1 involves making a copy of the System Storage Anchors containing the values "External" and "P_1", and a note of to which Storage Anchors (or lexical levels) each of these values belongs.

At the call of the entry variable the values of the Storage Anchors which were copied at assignment time must be used to set the appropriate environment into the Storage Anchors, for the invocation of the entry variable. The current values of the Storage Anchors changed must of course be restored on return. In our example, assuming that the call of EV is in activation S_1 of S, the value of the Storage Anchors for Task 1 before and after the activation of R_2 would be as follows:

Before the call of R	After the call of R
External	External
Q_1	P_1
S_1	R_2

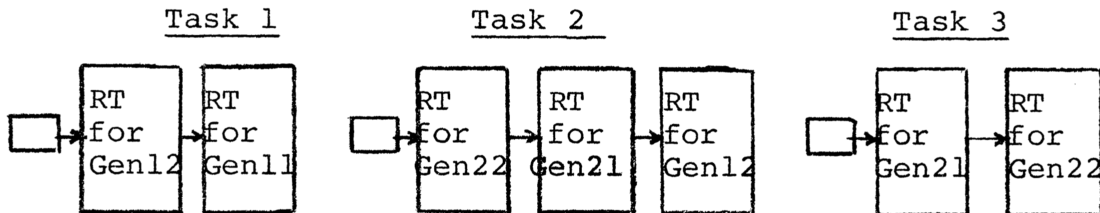
2.4.2 The Use of Storage Anchors for Support of Label Values

The implementation of label values with an environment component falls out as a simpler case of entry values. The difference, or simplification, is that for label values it is only necessary to switch the environment once and for all back to that which existed at the time of the assignment of the label value. All the activations which have come into existence between the time of this assignment and the branch to the label value will be wound up as part of the process of switching the environment. This approach can be used for the implementation of PL/I label variables.

2.4.3 The Use of User Storage Anchors for PL/I Controlled Variables

The Linker must associate a User Storage Anchor with each PL/I variable having the controlled storage class. This Storage Anchor will be used as an anchor for the Reference Tables containing the DAPOV's for the generations of the variable. The value of the Storage Anchor in each Task at any time will give direct addressability to the current generation of the variable in that Task.

As an example, suppose that there was a controlled variable in procedure Q of the example in Section 2.2 and 2.3. At the point shown in the Activation Tree there might be two generations allocated in Task 1, two in Task 2, but none in Task 3. The relationship between the User Storage Anchor and the Reference Tables for the generations of the variable might be as shown below. (Note that the exact picture depends on the order of allocations and Task attaches.)



Note: "RT for Gen_{ij}" represents the Reference Table for the ith generation allocated in Task j.

2.4.4 The Use of User Storage Anchors for PL/I Based Variables

Since any remaining generations of PL/I based variables allocated in a Task must be freed at the termination of that Task, it will be useful to have a User Storage Anchor for all such generations. Thus the DAPOV's for all the generations of based variables allocated in a Task will be contained in a Reference Table addressable from this User Storage Anchor.

2.5 Data Objects and Linking

A Data Object has three components: a name, a descriptor, and a value. Included in the discussion in this section are the two forms of a name, i.e. the Symbolic Name and the Logical Name, as well as the two forms of a descriptor, i.e. the Value Descriptor and the Generic Descriptor. The Value Descriptor is always bound to the value, thus leading to the term DAPOV, meaning descriptor and, pointer or value, where the word pointer denotes a system supplied means of reaching a value.

2.5.1 Descriptors

A descriptor is stored logically with each Data Object in the system. The word "descriptor" is used in several different contexts.

- The Value Descriptor is stored with the value of a Data Object. It includes:
  - A description of the instantaneous physical representation of the Data Object, and
  - the authorization requirements for access.
- The Generic Descriptor is associated with the name and exists in the Local Declare Table (described fully in Section 2.5.2.3). In general, the Generic Descriptor includes:
  - The set of logical declarations that are permitted for this name at varying points in time.
  - The initialization expression(s) that will be used. These may be user declared. If not, defaults are supplied by the language translator and utilized if the name is not resolved at Link time to an already existing name.

This information is then sufficient to return to the Interpreter at execution time a described value when the object is read, and to permit error checking on receipt of a described value at write. The reading of an object by the Interpreter utilizes the Value Descriptor, and the writing into an object will first confirm that the described data to be stored is compatible with the Generic Descriptor.

Value Descriptors may be factored, such that like data elements may have a common descriptor. Access to a Data Object will be via a level in the factoring that will permit a reconstruction of the full descriptor to be returned to the Interpreter. A specific example of factoring, for example, may occur in the executable code, in which only a short, partial descriptor distinguishes Logical Names from literals or operators.

The appearance of a Data Object to the user of a Logical Machine is as if the externally presented representation (e.g. input or output to a terminal) is that contained within the machine. However, specific internal representations may be important to the skilled user, e.g.

- Structure or array stored as a vector of vectors.

- Descriptors factored

Descriptors are not directly manipulatable by the user of a Logical Machine. They are indirectly manipulated by implied descriptor changing (or creating) statements: DECLARE, ASSIGN, ...

2.5.1.1 Descriptors as Related to Access Machines

Descriptors are a form of access machine (as defined in the Fundamental Concepts and System Language Manual). Examples of descriptors which act as built-in access machines are a floating point number, arrays, and structures. Logically, an array is treated as a vector of vectors, although physically, a more efficient implementation may be required. Access will be made via a descriptor and will yield a described value from the representation. A descriptor also provides the capability of the "follow" mechanism of an access machine in that it may contain the value, "System Pointer", which implies that the pointer should be followed to the next DAPOV.

2.5.2 Names

All names have two forms, a Symbolic Name and a Logical Name. The Symbolic Name is a character string, as written by the user, used for all communication with the user. A Logical Name is the internal encoding which replaces each Symbolic Name. In the executable code of each Module, each unique Symbolic Name is replaced by a unique non-negative integer, called the Logical Name. The system may be required to supply additional Logical Names in certain special instances, for example, for unnamed BEGIN blocks or iteration

variables. A Logical Name may be used as an index to a unique position in the Local Symbol Table, the Local Link Table, or the Local Declare Table (each defined below). Once program text is encoded, it is required that the Local Symbol Table be available to reconstruct the original names.

It will not be required that an application programmer be aware of the fact that his Symbolic Names have been encoded as Logical Names, but a system programmer may need to be aware of the existence of Local Symbol Tables and their role in name resolution in the Program Tree.

The connectivity between the Logical Name in the executable code and the Symbolic Name, the Generic Descriptor, and the DAPOV are contained in a set of three tables in the Module. These are the Local Symbol Table (LST), the Local Declare Table (LDT), and the Local Link Table (LLT).

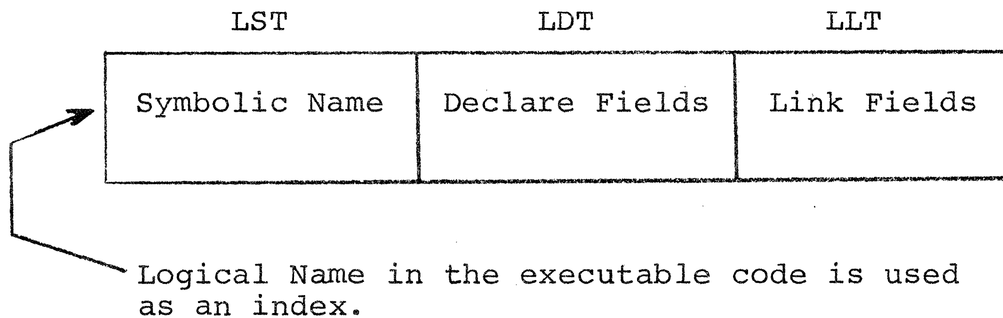


Figure 2.5.1

2.5.2.1 <u>Local Symbol Table (LST)</u>

The <u>Local Symbol Table</u> contains one entry for each unique (variable length) Symbolic Name, as written by the user, that is either declared or referenced within a single unit of translation. The Logical Name of any symbol can be determined by the Interpreter by searching the LST in sequence. Conversely, the Symbolic Name can be recovered by using the Logical Name as an index to the LST. (See Figure 2.5.1).

2.5.2.2 <u>Local Link Table (LLT)</u>

The Local Link Table contains one entry for each

Logical Name, in the same order as the LST. (See Figure 2.5.1). Each entry contains two fields which are filled in by the Linker. The first field will contain the Storage Anchor Name to connect each Logical Name with the Reference Table containing its execution time DAPOV. A zero in this field implies that this entry has not been linked yet. The second field will usually contain the Relative ID of the DAPOV within the Reference Table. (The Relative ID is the logical location of the DAPOV within the Reference Table.) However, a User Storage Anchor Name is contained in this field, if the first field has maximum field value.

2.5.2.3 Local Declare Table (LDT)

The Local Declare Table also contains one entry for each Logical Name, in the same order as the LST. (See Figure 2.5.1). Each entry contains seven fields which are filled in by the Translator. The name of each field, their possible values, and meanings follow:

Field	Value(s)	Meaning
Defined Flag	Defined	A declaration for this Symbolic Name has been made in this module.
	Undefined	No declaration made in this module.
Search Node	Internal	No search of Program Tree required for Linking
	External	Search External Node when Linking
	System	Search up to System Node (by-passing External Node)
	Above	Search one node above (see Section 2.5.2.4).
	External +1	Search up to one node below External Node.
Storage Anchor	Here	Use this Module's System Storage Anchor for allocation.
	Ext SA	Use System Storage Anchor of External Node for allocation
	User	Use User Storage Anchor for allocation.
	External +2	Use System Storage Anchor of node two below External Node.
Mandated Storage Anchor	Yes	Declaration requires Storage Anchor field value to be used.
	No	Storage Anchor field contains default to be used only if search is unsuccessful.

Field	Value(s)	Meaning
Task FREE'd	Yes	Will be FREE'd by task termination if not earlier FREE'd by user.
	No	Task FREEing not required.
Template Indicator	Real	Symbol represents a value
	Based	Symbol represents a descriptor only.
Generic Descriptor		(See following paragraph)

The last field in the LDT, termed the Generic Descriptor, itself contains ten subfields, some of which are variable in length. These subfields contain either the user declarations or the language defined defaults if at Link time a search is required and the name is not found in one of the containing blocks in the Program Tree. The names of these subfields, their possible values, and meanings follow:

Subfield	Values	Meaning (if not self-explanatory)
Data Type	Any, Character, {Fixed / Float} {Decimal / Binary}, Pointer, Offset, Label, Entry, Event, Task, or Statement	
Shape	Scalar, vector, Array, or structure	
Con-Var	Constant, or Variable	
Precision	Numeric value	
Scale	Numeric value	
Array Bounds	Upper expression and lower expression	Two expressions for each dimension of an array to define the limits of the array.
Initial	Expression	Initial value expression
Locator Qualifier	Expression	Used for pointer expression declared with based variable

Subfield	Values	Meaning
Parameter	Yes	Symbol unusable unless associated with an argument at entry.
	No	Symbol usable
Generic Desc. Pointer	Pointer	For undefined symbol, pointer to Generic Descriptor of defining occurrence of symbol. Filled in by Linker.

2.5.2.4 Entry Names

Procedure (or function) names and entry names are properly not considered to be declared within the Module that make up the statements of that procedure. Rather, they are considered as being declared as a name in the containing node of the Program Tree. Thus, to prevent the possibility of duplicate names in the LST, a second set of tables (LST, LLT, LDT) is created by the Translator which contains information relative to procedure or entry names. The search node field of the LDT contains the value "above" for these names. The Connector is responsible for merging the "above" tables of the procedure being connected with the tables of the procedure into which the connection is being made.

2.5.2.5 Object Code in the LST, LLT, LDT

The object code for each source language statement will be treated as a variable. The Symbolic Name is the statement number. The line directory will contain the Logical Name of each statement number. The LLT fields will point to the start of the initialized object text for each. Fields in the LDT will be set to defined, internal, ext SA (if PL/I) or here (if APL), and will contain an initializing expression which is the encoded object text created by the Translator.

2.5.2.6 Treatment of APL Symbolic Names

All APL Symbolic Names will be declared by the Translator as defined, external, user. Further, for each local APL symbol in a function, the Translator will insert a user allocation statement in the entry code to initialize the DAPOV to unassigned, and a user free statement in the exit code. As with a PL/I procedure, it is only necessary to Link a function

once. The inserted statements for local symbols will create the proper visibility of symbols as required by APL.

One area requiring special mention is that a function name, when Connected, will be established as the oldest generation for that given Symbolic Name, even though there are other generations (of local variables) already in existence.

2.5.3 Linking

The function of the Linker is to resolve the static nesting of symbols so that at execution time a Logical Name will lead via the fields in the LLT to a DAPOV in a Reference Table. In this manner, the name of the Data Object is bound to the DAPOV of the Data Object. Linking is accomplished at first activation. In order to fill in the Relative ID, it is necessary to initialize and allocate variables while Linking.

For a name containing a value in the search node field of the Generic Descriptor of internal, one of three actions will be taken depending on the value of the storage anchor field of the Generic Descriptor:

1. Value of storage anchor field is Ext SA. The initial value is generated using the expression in the initial field of the Generic Descriptor, and the Reference Table associated with the External Node is extended to include the DAPOV. The Storage Anchor Name and Relative ID are filled into the LLT, thereby marking this name as linked. (This case is used for PL/I static internal.)

2. Value of storage anchor field is here. Generate the initial value and extend the RT with the DAPOV. The RT is that identified by the System Storage Anchor for this Module. The Interpreter will have already stored a newly created Space Name in this Storage Anchor. The name of this Module's System Storage Anchor and the Relative ID to the new DAPOV are filled into the LLT. (This case is used for PL/I automatic.)

3. Value of storage anchor field is User. Obtain the next User Storage Anchor name, and insert as its value the next available Space Name. Then extend that Space with a pointer indicating the previous generation is non-existent and a DAPOV for an un-assigned value. Finally, the fields of the LLT

are filled in. It may be noted that these variables are initialized to catch a reference prior to user allocation as an error. (This case is used for PL/I controlled internal.)

For a name containing a value in the search node field of either external, system, or external +1, the Linker will extract the Symbolic Name from the LST using the Logical Name as an argument. The Symbolic Name is used as a search argument in the specified search node(s). (In the case of system or external +1, the search will start at the node above this module's node.) If found, the fields of the LLT entry where found are copied into this Module's LLT. For a defined symbol, the corresponding LDT entries are checked for consistency. For an undefined symbol, a pointer to the LDT entry of the symbol found is placed in the Generic Descriptor pointer field. If not found, one of three actions will be taken depending on the value of the storage anchor field:

1. Value of storage anchor field is Ext SA. Proceed as in Case 1 above. Then extend and fill in the fields of the LST, LDT, and LLT of the External Node with a copy of the entry from this Module's tables. (This case is used for the first occurrence of a PL/I static external variable.)

2. Value of storage anchor field is User. Proceed as in Case 3 above. Then extend the fields of the LST, LDT, and LLT of the External Node with a copy of the entry from this Module's tables. (This case is used for the first occurrence of a PL/I external controlled variable or APL symbol.)

3. Value of storage anchor field is External +2. Two subcases apply.

   a) If this Module is at a node below external +2, initialize and allocate the DAPOV by extending the RT of the System Storage Anchor of the node two below the External Node. Fill in the LLT fields for the entry in this Module, and then copy the LST, LDT, and LLT into an extension of the tables in the external +2 node. Mark the storage anchor field in that LDT with the value,'Here!

   b) If this Module is at the external +2 node, proceed as in Case 3a, but do not extend the LST, LDT, LLT tables, since the entry already exists there. (These cases are used for an undefined variable.)

The Link process for names with a template value of based requires that a link be established to a Storage Anchor containing all such names, so that the names may be freed by the system if the user fails to do so prior to task termination.

After Linking all the entries in the LLT, the Linker will turn on the link bit for this Module. Subsequent activations will only require allocation and initialization of those Logical Names with a storage anchor field value of 'Here'. Procedure editing will turn the link bit for this Module off, as well as the link bit for any Logical Name affected by the editing. Additional detailing is required to state which Logical Names are affected, and the effect on Logical Names in other Modules.

2.6 The Interpreter

The execution of code in each active Logical Task of a Logical Machine is performed by an Interpreter. Each such Interpreter is capable of executing a set of Built-in Operators, and is provided with a separate set of mechanisms consisting of: An Operator Stack, an Operand Stack and a Task Status Table. Each interpreter also makes use of the facilities of the Logical Machine in which it is active, i.e., The Program Tree, the Activation Tree, and the Dynamic Storage Mechanism.

2.6.1 The Built-in Operators

The Built-in Operators which can be executed by each Interpreter can be classified as those which handle computation, those which handle the flow of control between statements of the active Module and between different Module activations, and those which provide special system functions.

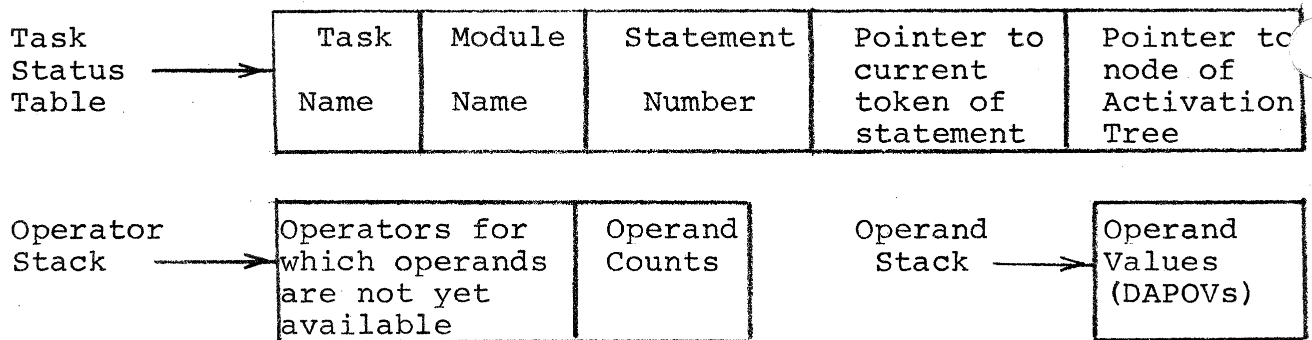
The computational operators are generic with respect to the data type and aggregation of their operands and provide a combination of the logical and arithmetic operations available in APL, PL/I, FORTRAN, COBOL, and RPG. The logical operators include: ASSIGN, SUBSCRIPT, CONCATENATE, COMPARE EQUAL/NOT EQUAL/LESS THAN/ and GREATER THAN, AND, OR, NOT, SUBSTRING and single character SEARCH, TRANSLATE, SIZE and SHAPE (like APL ρ). The arithmetic operators include: ADD, SUBSTRACT, MULTIPLY, DIVIDE, EXPONENTIATE, LOGARITHM, ABSOLUTE VALUE, COMPARE EQUAL/NOT EQUAL/LESS THAN/ and GREATER THAN, FLOOR, CEILING, MAXIMUM, MINIMUM, ROUND and ASSIGN. Discussion of the language requirements for these operators is contained in memos ASP-045 and ASP-051. There is also a requirement for a number of operators to handle editing and conversion between different types of data.

The operators which handle the flow of control between statements of an active Module and between Module activations include: GO TO, IF-THEN-ELSE, DO, CALL, RETURN, ATTACH, and EXIT.

The operators which provide the special systems functions include: LINK, CONNECT, CREATE LM, CALL LM, ATTACH LM, ENTER/ LEAVE EDIT MODE and other Operators to support System Facilities and the Logical I/O System.

2.6.2 The Interpreter Mechanisms

In executing the code (which is in a prefix polish form) the Interpreter makes use of three mechanisms which are illustrated schematically below:



The Task Status Table contains five fields:

1. The first field contains the name of the Logical Task being executed by the Interpreter. This provides access to the Logical Task Control Block (LTCB) and hence to the set of Storage Anchors for this task.

2. The second field contains the name of the Module currently being executed and provides access to the various components of this Module which are contained in the node of the Program Tree (i.e., LST, LLT, LDT, executable code and line directory).

3. The third field contains the number of the statement which is currently being executed, and will be updated as each statement is completed.

4. The fourth field identifies within the statement code the token (i.e., operator, operand, or literal which was fetched most recently).

5. The fifth field identifies the latest node in the branch of the Activation Tree for the task being executed, and will be used to store the information contained in the Task Status Table into the Activation Tree when the call of a new Module is executed.

The Operator Stack contains a push down list of those operators which have been fetched, but for which all the operands are not yet available, together with counts of how many operands are still needed for each operator in the stack.

The Operand Stack contains the values, in DAPOV form, of all those operands and literals which have been fetched, or computed, but which cannot be used by their operator until its other operands have been fetched or computed.

2.6.3 Statement Evaluation

The basic protocol for statement evaluation is presented in the Figure 2.6.1. The equivalent of an instruction fetch is the token fetch loop which starts at the block labeled A and continues through block B for an operator token, or blocks C and D for a literal, or blocks E and D for a logical name. The DAPOV fetch in block E is discussed in further detail in the next section. In block D the "appropriate" DAPOV in the case of an assignment argument is a pointer to the generic descriptor in the LDT rather than the fetched value descriptor and value.

An execution cycle represented by blocks F, G and H is taken whenever the operand count of the operator on top of the operator stack is zero. Note that the evaluation of control and system operators will often result in leaving the basic evaluation loop. Simple cases of these operations are discussed in subsequent sections.

After execution a check is made to see if the Operator Stack is empty. In prefix polish notation this should be the end of the statement. To protect against a badly translated Module, an End of Statement token is required in the code and the Operand Stack must also be empty. If these conditions are met the first token pointer of the next statement is fetched by blocks I and J. This requires using the statement counter as an index to the Line Directory in the Module. If no statement is found, the Interpreter takes an automatic RETURN.

2.6.4 Protocol for Value Fetch from Variables

Whenever a reference type of operand is encountered during statement execution its value (DAPOV) must be fetched into the operand stack from the appropriate Reference Table. The process involved is described below and illustrated in Figure 2.6. 2.

The Logical Name appearing in the code is used as an index to the LLT (contained in the node of the Program Tree) to yield a Storage Anchor name and Relative ID for the required DAPOV. The Storage Anchor name and Task name (contained in the current status table) are used to obtain the value of the

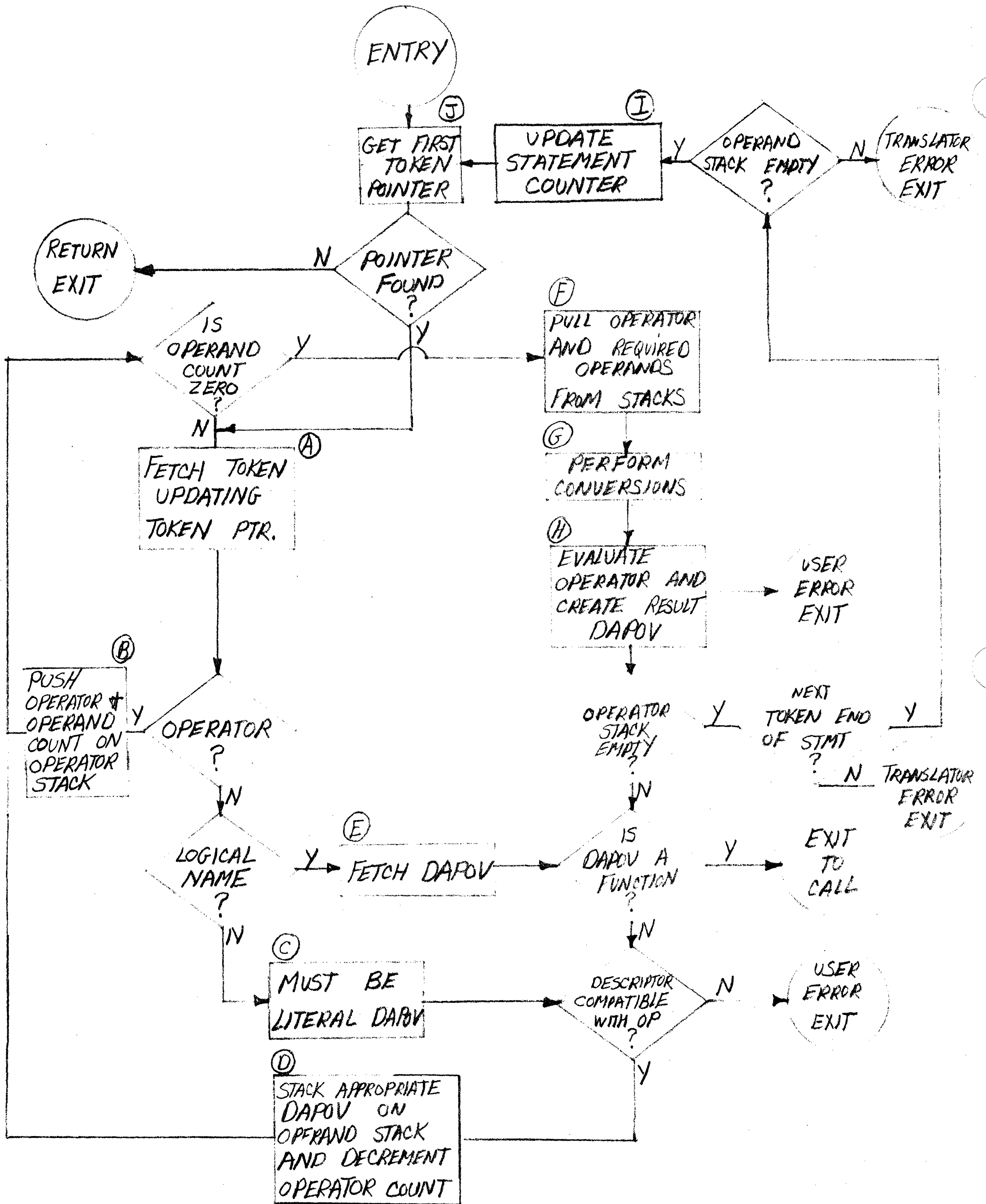


Figure 2.6.1 STATEMENT EVALUATION

Storage Anchor relevant to the current task. This Storage Anchor value identifies the required Reference Table. The Relative ID, already obtained from the LLT, is now used to access the required DAPOV from this Reference Table. If the DAPOV obtained turns out to contain an indirect reference to some other DAPOV the chain of indirection will be followed automatically to its end.

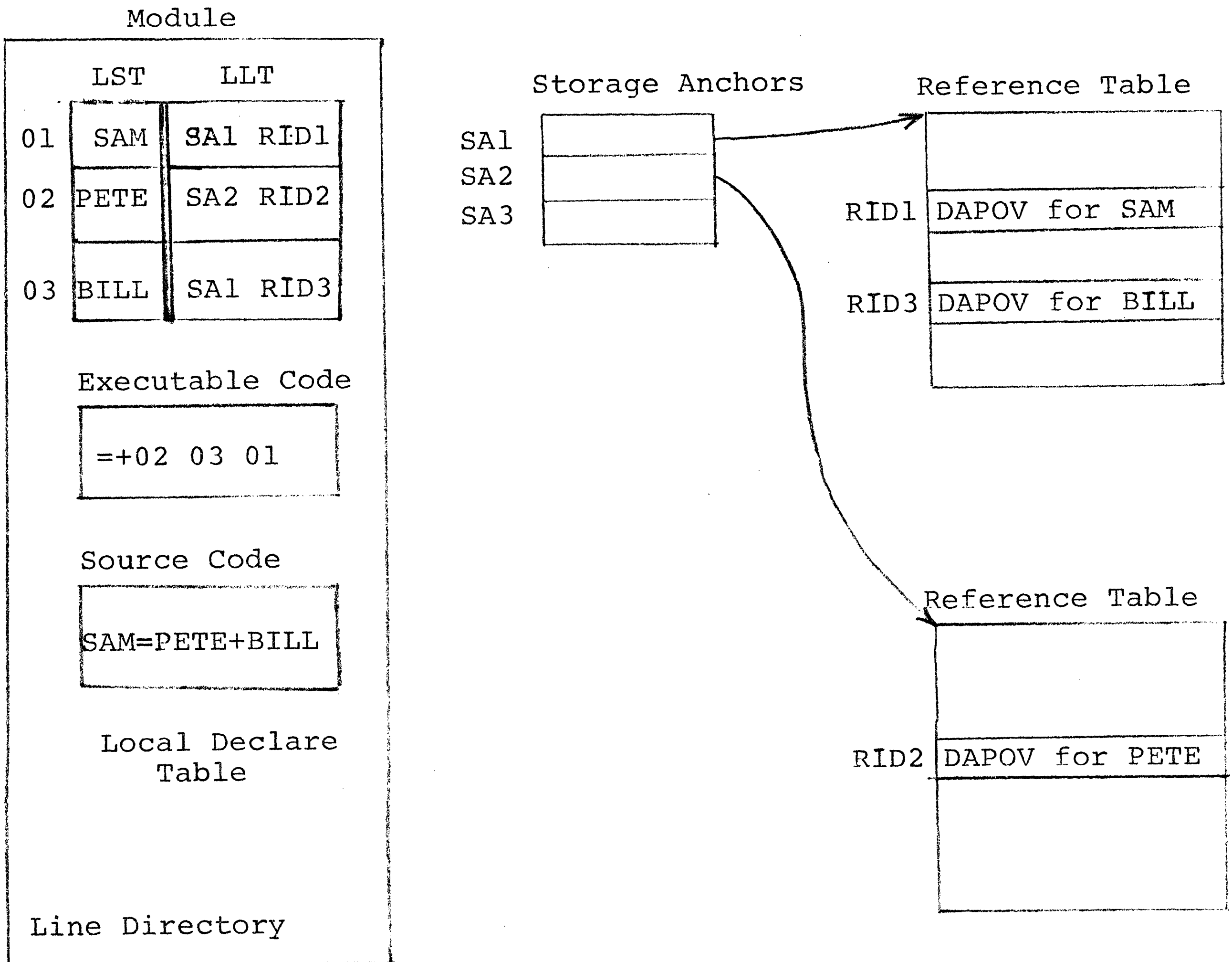


Figure 2.6.2 Access of DAPOVs

2.6.5 Protocol for Call

The following set of actions must be performed by the Interpreter whenever a Call operator or function reference is encountered:

1. Evaluate any arguments and place the values or references on the operand stack.
2. Update the statement number field, and save the information contained in the Task Status Table in the node of the Activation Tree identified by the fifth field of this table.
3. If invocation involves a switch of environment, also save the values of Storage Anchors to be modified, and set these Storage Anchors from the environment information of the entry value.
4. If the Module to be invoked has not previously been linked, invoke the Linker to link the Symbolic Names referenced or declared in the Module.
5. Create a new Reference Table for this activation, set its chain field to the value of the appropriate Storage Anchor, and set this Storage Anchor to identify the Reference Table just created.
6. Carry out initialization of variables allocated in this Reference Table.
7. Set up addressability to any parameters using the argument values contained in the operand stack.
8. Set the fields of the Task Status Table and start execution of the code of the invoked Module.

2.6.6 Protocol for Return

The following actions must be performed by the Interpreter when a Return operator is encountered:

1. Evaluate any return value expression and place the result in the operand stack.
2. Restore the appropriate Storage Anchor from the chain field of the Reference Table for the activation being terminated, and then free this Reference Table.

3. If the invocation had involved a switch of environment, restore the Storage Anchors switched from the values saved.

4. Restore the Task Status Table from the information which was saved in the Activation Tree, and remove the node of the Activation Tree associated with the activation being terminated.

5. Continue execution according to the information now contained in the Task Status Table.

2.6.7 Protocol for GO TO

There are two cases:

a) A GO TO within the scope of the current activation:

   1. Set the value of the statement number field of the Task Status Table to the value given by the operand, and the value of the current element pointer in the current status table to the start of this statement.

   2. Carry on with execution of this Module.

b) A GO TO with a destination in some suspended activation.

   1. Search the Activation Tree to ensure that the destination activation still exists.

   2. Terminate as many activations as necessary until the destination activation becomes the current one, freeing the Reference Tables for these activations and taking the other actions described in the protocol for Return.

   3. Proceed with protocol for case a.

2.6.8 Protocol for Attach

The following actions must be performed when the Interpreter encounters an Attach operand:

1. Evaluate any arguments and collect the values or references in a special table.

2. Call upon the Logical Machine Supervisor to set

up and initiate a new Logical Task passing to the LMS the name of the attaching task (i.e., LTCB), the name of the Module to be invoked as the first activation of the new task, the values of the Storage Anchors, and the table containing the arguments to be passed to this activation. (The Logical Machine Supervisor sets up a new Logical Task which will itself be executed by an Interpreter.)

3. Activate the Module in the new task performing actions 4 through 8 of the protocol for Call.

IBM CONFIDENTIAL

2.7 Building a Logical Machine

Skilled users of the system may wish to modify the Logical Machine to which they are connected or to create a new Logical Machine for subsequent use. Modifying a Logical Machine requires invoking the system functions of Connect and Disconnect which add or remove Modules from the Program Tree. A separate system function is used to create a new Logical Machine.

The ability of a user to perform these functions depends upon the capability of the Logical Machine to which he is connected. The functions are always available at the System Node in the Program Tree; invoking these functions can only occur from some Module lower in the Program Tree; and recognizing commands from the user to cause this invocation is dependent on the interpretive power of the External Node in the Program Tree.

2.7.1 The Connect Function

The Module to be connected may either be in the Ownership Tree or supplied by the user as program statements entered through the Logical I/O System (e.g. from a keyboard).

If the user choses to enter statements from the keyboard, he must enter Edit mode. Statements entered in Edit mode are translated and, when the user is finished entering the statements, a Module is built. The statements can be written as nested procedures. If this is done the procedures will be nested into the Program Tree in the same way they were written.

Before connection into a Program Tree, a procedure exists as a Module. When the connection is initiated, information from the Module is used to construct the LLT, LST, and LDT. None of the variables of the Module are linked at this time, entry names, however, are resolved to the node of the Program Tree under which the Module is being connected.

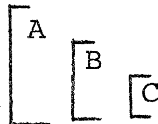
If the Module (call it JOE) is to be nested below another Module (call it PETE), then the entry names for JOE may be found in the LST for PETE.

If they are found, PETE's corresponding LLT entries are set to identify an element of the Reference Table corresponding to JOE's entry names do not show up in PETE's LST, then they are added to PETE's LST and LLT.

The static nesting can be specified by the connect command. For example, the command sequence

Connect A
Connect B in A
Connect C in A. B

results in A being the outermost procedure with B and C nested below:



2.7.2 <u>The 'Create Logical Machine' Function</u>

The 'Create Logical Machine' function requires three parameters, a name for the Logical Machine, the position in the Ownership Tree at which the LM is to be created, and the name of a Module in the Ownership Tree which is to be placed at the External Node of the new LM.

The name of the LM will be checked in the Ownership Tree to insure uniqueness. Also, the receiving node in the Ownership Tree must authorize this kind of access. It is, of course, always possible to create a Logical Machine directly beneath the creating Logical Machine.

The choice of Module for the External Node will also be checked for authority and indeed this is one of the prime mechanisms for defining the skill level of users. Clearly at least one Module will be defined to interpret the full SL Command Language. Who is allowed access to this Module, and where they are allowed to place a Logical Machine which they create, constitute basic control mechanisms on the users of the system.

CHAPTER 3

THE LOGICAL SYSTEM

3.1 The Logical Machine Supervisor

The Logical Machine Supervisor is an active Logical Machine at the root of the Ownership Tree with two primary functions. Firstly, it is the interface between the system users and their work, as it is processed by one or more Logical Machines. Secondly, it is the interface with the Physical Control System and loads this system with requests for physical processing as generated by the Logical Machines.

3.1.1 The User Session, Jobs, and Logic Tasks

The major unit of work in the system is a User Session. In a simple interactive session the User Session is the processing performed between sign-on and sign-off. It is possible however, for a skilled user to initiate more than one independent activity and have these Jobs controlled by the supervisor as batch activities. Clearly at any time in a session, only one Job can be running in interactive mode for each active terminal. Each Job may result in one or more dependent parallel activities called Logical Tasks. The first task in a Job is the Master Task; all subsequent tasks are subordinate to this Task (or to other Sub-Tasks). In any job the Master-Task is connected to the User's Console (interactive mode) or to a Catalogued Data Set (batch mode). Every Job runs in a separate Logical Machine controlled by the Logical Machine Supervisor.

3.1.2 Session, Job, and Task Protocols

A Session commences when a User gains access to a physical terminal and port and is recognized (e.g., key-board unlock) by the Physical Source-Sink Sub-System. The code activated by this process is in fact a Physical Task running on a Source-Sink Processing Unit, but is logically a new Logical Task in the Logical Machine Supervisor. This Logical Task creates a Session Control Block (SCB) and accepts the sign-on from the user. The sign-on code is searched in the Ownership Tree provided by System Facilities to determine that the user is authorized and selects an appropriate Logical Machine for his Session. Accounting information, priority, etc. is also determined at this time, and

the whole entered in the Session Control Block.

Once the Session Control Block is prepared, an initial Job Control Block (JCB) is set up and pointed to by the SCB. The JCB specifies which Logical Machine is activated by the session and whether it is interactive or batch. Now the supervisor activates the Logical Machine by parametising the External Node for batch or interactive communication and chaining a Logical Task Control Block (LTCB) from the JCB. This LTCB specifies the active node to be the External Node. Finally, the supervisor creates a Physical Task Control Block (PTCB) for a physical processor, chains this PTCB from the LTCB and enters the PTCB in the processing queue of the Physical Control System.

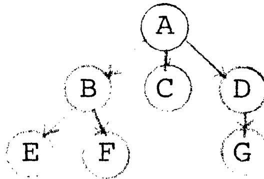
During the running of a Physical Task, the Processing unit may encounter operators such as Attach which result in the creation of further LTCB's and even commands from the user which create further Jobs. These activities and the normal, or abnormal, termination of Jobs and Tasks, result in the automatic intervention of the Logical Machine Supervisor via built in Modules in every Logical Machine.

3.2 System Facilities

All of the Logical Objects in the system (Logical Machines, Data Sets, Modules, Logical Source-Sink Devices) are connected together in a single structure called the Ownership Tree. This tree, together with the ability to monitor communication between the Logical Objects, and together with operators to Create, Destroy and Modify Logical Objects, constitute the System Facilities.

3.2.1 The Ownership Tree

The objects in the Ownership Tree are connected in such a way as to define ownership and access capabilties. These concepts are best explained by examining this simple tree:

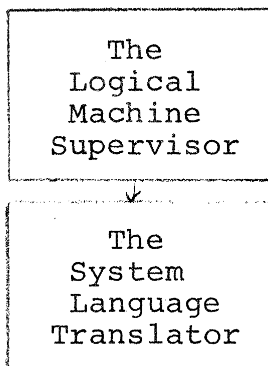


Here A owns A, B, C and D and through them E, F and G
B owns B, E, and F
D owns D and G
and C, E, F and G own only themselves.

Any Object can access an Object it owns; additionally it can access any Object which is a predecessor in the Ownership Tree.

Thus (for example) G can access itself, D and A but not B, C. E and F. The fact that an Object can access another does not necessarily authorize it to extract information from it or to modify it. These authorization rights are a property of the accessed Object and may be permanent bars against certain classes of access (e.g., Read Only), or complex functions controlling privacy and security, or temporary bars against access of one class while another is proceeding (e.g., Modify - Exclusive State). Only an owner of an Object can change the authorization rights.

When a system is first generated it contains a simple Ownership Tree composed of two Logical Machines.



With this start and the operators of System Facilities, the rest of the complete operating system can be constructed.

3.2.2 Monitoring Object to Object Communication

Communication between a Logical Machine and other Objects is handled by special Modules in the Logical Machine called the Logical I/O System (Section 3.3). These Modules achieve communication by built-in operators which automatically check for access and authorization rights and constitute part of the System Facilities. Communication between a Logical Machine and the Logical Machine Supervisor is direct since the access and authorization rights are built in. From the Ownership definition it is apparent that any Logical Machine can do anything it wishes to itself and, therefore, the System Facilities do not monitor internal Logical Machine actions.

3.2.3 Creating and Modifying Another Logical Object

Any Logical Machine can create another Logical Object and becomes the Owner of it. Because of the complex structure of many objects and to avoid revealing their internal bit reresentations, System Facilities supplies a number of operators and functions to perform these tasks.

3.2.3.1 Operations on Logical Machines

These operations can be considered as system commands for languages which do not have these functions within their syntax. There are three series of functions - Logical Machine parameter setting, Information Commands, and Dynamic Commands.

IBM CONFIDENTIAL

The Logical Machine parameter commands are used to give information required by the Logical Machine. Illustrative examples of this type of command are:

- Name or rename current Logical Machine

- Set limits on LM size

- Set limits on compute time

- Set limits on number of bytes of data catalogued

- Set defaults
  - language
  - computational precision
  - origin

- Set width of typed output

- Set trace

Information Commands are used to read information the user desires for knowing his status, debugging, etc. The list below is illustrative of this type of command.

- Variables visible to suspended Module.

- Modules in the LM

- Name of catalogued objects

- Names of active LM's

- Read LM parameters

Dynamic Commands are used to control the contents of a Logical Machine. Illustrative examples of dynamic commands are given below.

- Deactivate an LM and Save

- Save a copy of suspended LM

- Catalog a Module

- Clear an LM
- Sign off
- Erase a Module
- Connect a Module into LM
- Drop a catalogued object
- Send a message
  - to operator
  - to another LM

3.2.3.2 Entry of Source Code and Text

To enter a source code or text, the user enters EDIT mode. When the user enters EDIT mode, he indicates whether it is text or source code he is entering, and the name of the object to be entered (or edited). The Logical Machine has a default language. If the user specifies source code entry and no language, the LM default language is assumed.

The name of the object is checked to determine -

a) if the name is new, or
b) if it has been previously defined.

Once the editing has been initiated then the editor requests a line of input. The user then types in his next line. If this is source code entry, the Translator for the language then translates the line. If any syntax errors are detected, the Translator returns a code for the user to re-enter the source.

There are facilities for changing lines or inserting new lines. When the user is finished, he types a command signaling he is finished. The editor then causes a Module to be built (if this is procedure entry). The user can request that the Module be catalogued if he so wishes.

The default is to connect the Module to the External Node of the Logical Machine in which it was entered. The user

IBM CONFIDENTIAL

may issue commands to connect it nested under some Module in the Program Tree if he wishes.

3.2.3.3 Creating a Module

A Module is built by the action of the Translator on a number of statements written in one of the accepted source languages (e.g., APL or PL/I).

The unit of translation is a sequence of statements in which all uses of the same name refer to the same object. Translation of this unit becomes a Module. This unit corresponds to a Procedure Block or BEGIN Block in PL/I or a Function in APL. Nested procedures are handled as separately translated units. A statement entered for direct execution will also be treated as a unit of translation.

The Module resulting from a translation consists of a number of distinct components together with a table giving addressability to each of them. The components of the Module are the source code, the line directory, the executable code, and two sets of Local Symbol, Link and Declare tables.

The source code contains the unmodified form of the source statements as they were received by the Translator. The line directory has an entry for each line of the source code and gives the relative starting location of the executable code produced for this line. The executable code consists of the sequence of operators, operands in the form of Logical Names, and numeric or character literals, which, when executed, will give the semantic actions defined for the source language statements. The form of the executable code is prefix Polish. The first set of LST, LLT, and LDT contains an entry for each Symbolic Name declared or referenced within the statements of the Module. The second set of LST, LLT and LDT contains an entry for each Symbolic Name which defines an entry point of the Module. The Connector is responsible for merging the items in this second set of tables into the corresponding tables of the Module represented by the node of the Program Tree into which this Module is connected. The LST, LLT and LDT are described in detail in Section 2.5.

3.2.3.4 Modifying a Module

The process of changing Modules will be like APL but with fewer restrictions on allowed changes. When a change is to be made the execution must be stopped. The Logical Machine goes into a direct mode.

A variable can be read by typing the name of the variable. This will return either the value of the variable or a message telling why the variable was not printed. One can also attempt to assign a new value to a variable. If the attempted assignment is invalid a message will be returned to the user.

The source statements of a Module can be changed by entering EDIT mode. Lines can be added, deleted or changed. When finished the modified load module is in the Logical Machine only.

After changing the Module, it must be fitted back into the Logical Machine environment with appropriate changes to the environment to reflect changes made to the Module. There will be restrictions on changes to Modules which exist other than as a single leaf of the Activation Tree.

3.3 The Logical Input/Output System

3.3.1 Introduction

Every Logical Machine is provided with a Logical Input/Output System comprised of a number of specialized I/O nodes. These nodes are placed under the System Node of the Program Tree for the Logical Machine. Information is transferred to and from the Logical Machine using the facilities provided by these nodes. These facilities provide the Logical Machine with the capability to communicate with:

another Logical Machine,
Catalogued Data,
Catalogued Modules,
the Logical Machine Supervisor,
Logical Source-Sink units,
and other systems on the network.

Only Logical Source-Sink nodes and "other system" nodes have the property enabling communication via the Source-Sink Subsystem. Input and Output buffering is used to provide the link between Logical I/O and physical Source-Sink (which is discussed in Section 4.4).

3.3.2 Protocol

The Logical Machine initiating communication via its appropriate I/O node is in control of that communication. The Logical Machine Supervisor, because of an explicit or implicit command, will activate the system utility procedure to cause a transfer of Data Objects or Modules.

The operators which create the logical I/O task are contained in the executable code associated with the appropriate I/O node. A Logical Task will be created and the Logical Task Control Block (LTCB) will be placed in the job queue of the Logical Machine Supervisor. The Logical Machine Supervisor will observe that a Logical Task has been created and will initiate one or more tasks on the physical system. The Logical Machine Supervisor obtains addressability to a particular object via the Ownership Tree.

Logical Machine to Logical Machine A LM may, when authorized, access the Data Objects or Modules contained in another LM either by obtaining a pointer or by making a copy. However, the controlling LM cannot alter the LM being accessed.

Logical Machine to Catalogued Data A LM may access and/or modify catalogued data. There are various levels of authority (e.g., Read Only, Read and Copy Only, Read and Write). When a LM is in the process of modifying the Catalogued data, other potential users of the data are locked out. Authority may be allowed to a collection of Data Objects or only to individual Data Objects.

Logical Machine to Catalogued Module Accessibility to Catalogued Modules is the same as to Catalogued Data. Locking also takes place when the Catalogued Module is being modified.

Logical Machine to Logical Machine Supervisor An active LM communicates with the LMS to initiate some system functions such as initiation of a Logical Task.

Logical Machine to Logical Source-Sink Device Output to a Logical Source-Sink Device may be spooled in one of three ways which are selected either by a default condition or specification in the application program.

One way is to spool the data in its internal format for later manipulation by the physical Source-Sink Subsystem when output actually takes place. This maintains the independence of the application program from the physical Source-Sink Device.

A second method of spooling (specifically chosen by the user) is for the application program to specify the output format and hysical device at which the data is aimed. This will cause a Physical Task to be set up by the LMS to precompute the appropriate format and data blocking which is then spooled for eventual physical output.

The third method is to provide for the data to be edited and formatted at the sink. The data is spooled in the internal format. A tag signifying this condition is passed to the Source-Sink Processing Unit (SSPU) with the data at the time physical output is to take place.

When the spooled data is actually passed to the SSPU, the LMS I/O System also points to the appropriate code to be used by the SSPU to perform physical I/O.

Logical Machine to Other Systems on the Network Like systems may be treated in the same way as Logical Source-Sink

Devices. Unlike systems will be handled by emulation. The details of handling emulation have yet to be worked out.

Broadcasting is used on output, when a Symbolic Name being sought is outside the system and thus is not known to the Ownership Tree. The LMS will set up a Physical Task for output via the Source-Sink Subsystem which will broadcast a message to the other systems in the network requesting them to reply with the location of the Symbolic Name if they are aware of it. The response to the requesting LM is either a "Symbolic Name Not Known" or an explicit address or list of explicit addresses where the name may be found. In the third case, the user must personally investigate which of the responding locations holds the Symbolic Name for which he is looking.

The SSPU may act as a network node and perform a store and forward function. Store and forward is the action of passing data through to allow communication between two other systems. When there is not a direct line between two systems that desire to communicate, the sending system's SSPU obtains from the Ownership Tree the address of the next possible stop along the way. The Source-Sink Subsystem will make connection with one of them and ship the message with the desired termination address. The Source-Sink Subsystem at the store and forward node receiving this message will temporarily store it and establish connection in the same way, with the next node. This will continue until the desired termination system is reached. Answer back, if required, will work the same way.

CHAPTER 4

THE PHYSICAL SYSTEM

4. The Physical System

The Physical System consists of three major subsystems:

The Storage Management Subsystem
The Program Processing Subsystem
The Source-Sink Subsystem

The Storage Management Subsystem (SMS) provides storage facilities for all the processing and source-sink units. The Program Processing Subsystem consists of one or more Program Processing Units (PPU). A PPU executes programs and provides overall control of the system. The Source-Sink Subsystem consists of one or more specialized Source-Sink Processing Units (SSPU) to provide input and output facilities, communication with user terminals, or communication with other systems of the same or different type.

These subsystem units are interconnected by physical lines and control hardware, and their interaction requires common control software and tables. The control software is executed on any available PPU.

The breakdown of the Physical system is described here in functional terms and does not necessarily correspond to the packaging of an actual implementation. The implementations of such a system may range all the way from a single physical unit, with SMS, PPU, and SSPU functions all rolled into one and sharing common hardware, to a very large system where each subsystem unit is broken down into several more specialized subsystems. It may also be found desirable to package some SMS functions in a physical unit primarily devoted to processing functions. Specific implementations may make different trade-offs between hardware, firmware, and software execution of a particular function.

Thus the physical description is an architectural one, to which all implementations adhere. An overall diagram of the physical system is shown in Figure 4.1.

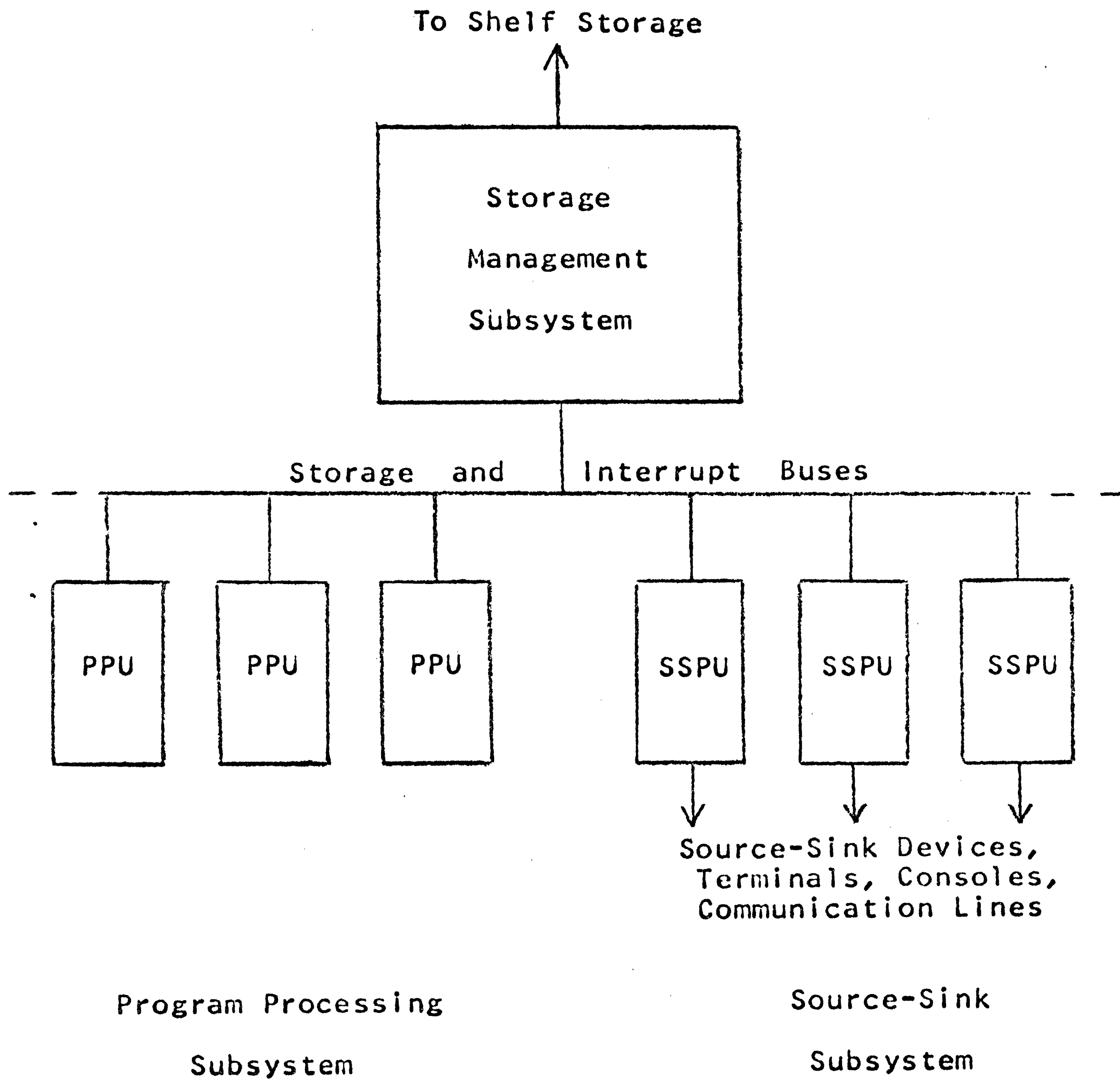


Fig. 4.1 PHYSICAL SYSTEM

4.1 Control

4.1.1 Hierarchical Design

The inherent complexity of a large system can be greatly reduced by adopting a hierarchical design where each level of the design supplies one or more basic functions for use at all higher levels. A function supplied at one level is not duplicated at higher levels. Each level is kept simple enough, with separate modules and carefully controlled interfaces, that exhaustive tests can be devised, and the level can be completely debugged. As each level , starting at the base (level 0), is completely tested, its functions can be used with complete confidence at the next higher level.

As the design levels for AFS have not yet been defined, the levels used by E. W. Dijkstra, who first described the merits of such a hierarchical design (CACM, vol. 11, no. 5, May 1968, pp. 341-346), may serve as an example for a relatively simple system.

Level 0: Real-time clock interrupts and processor allocation. Above this level the processor identity disappears and all processes become sequential.

Level 1: Storage paging mechanism. Above this level only the virtual memory is addressed.

Level 2: Console operation to allow sharing of the console among different tasks at higher levels.

Level 3: Input-Output device allocation and buffering. At higher levels the user addresses virtual devices.

Level 4: Independent user programs.

4.1.2 Interrupts

There are two kinds of interrupts: external interrupts and priority task switching interrupts. Both cause an interrupt signal to be sent by the hardware to an appropriate processing unit (PPU or SSPU) for action.

<u>External interrupts</u> are caused by asynchronous signals from external sources. Examples occur when a pre-set time interval has elapsed, a pre-set time of day has been passed, an attention button has been pressed, a telephone port has been dialed, or an input-output unit has been made ready. External interrupts are sent to an SSPU of the proper class to handle that interrupt.

The selected SSPU switches from its current task to the interrupt processing task. The SSPU may complete the interrupt procedure or request PPU action by placing the interrupting task in the PPU physical task queue and raising the priority task switching line. The SSPU then returns to its former task.

<u>Priority Task Switching</u> (PTS) interrupts are used to activate an idle PU and to ensure that higher-priority tasks, when ready to go, are attended to before tasks with a lower priority. Normally a PU switches tasks only when its current task terminates or reaches a waiting point. The unit then searches a common physical task queue for the highest priority task in its job class that is ready to go next. Thus processors are normally <u>queue driven</u> and kept busy as long as there is work to do. This normal flow of work may be altered by a PTS interrupt using the common PTS lines.

The PTS lines contain the task identifier, priority, and (optional) job class code of the interrupting task. The job class code identifies a specialized processor, such as an SSPU that is attached to the physical equipment needed or a PPU with high-speed arithmetic for a lengthy computing task. A PPU that is otherwise not specialized may be reserved for a certain class of job by assigning a job class code to it. The hardware compares job class and priority of the interrupting task with the job class and priority of its current task for each PU. The PU with the lowest priority within the job class is selected. It switches tasks and selects the the identified task from the physical task queue. If all processing units of the right class are busy executing tasks of equal or higher priority than the interrupting task, the interrupt signal is ignored. The new task will be picked up during a subsequent normal task switch.

The PU selected for interruption will attempt to reach the end of the current statement before switching tasks. If the statement end is not reached before a fixed short response time has elapsed, the PU is interrupted in mid-stream by storing away all current hardware register contents. The response time desired for a task appears in the task control block; it is either set by the user, or a system default value is inserted.

When the interrupted task is later resumed by a PU of the same class, all registers are restored automatically. Such a mid-stream interruption is entirely safe, but it usually takes more time, both to dump and reload the registers and because of additional storage activity.

Exceptions occurring within a task, such as fixed-point overflow, are sometimes called synchronous interrupts, but they are not classified here as interrupts. As a rule an exception condition causes an appropriate exception function to be called, either at the beginning of the current statement or at the end, and the task continues uninterrupted. Some exceptions may cause abnormal termination of the task.

4.1.3 Time-Outs

Timer interrupts via an SSPU may be used to implement time slicing, which allows tasks of equal priority to gain equal access to the available processors. Timer interrupts reflect elapsed real time.

To keep track of active processing time, each logical task has a run time value in its task control block. An internal timer in each processor is used to update that run time whenever the task is active. When the active processing time exceeds the pre-set task time limit, the task is switched out and de-activated; outside intervention is then required to re-activate the task and let it continue. The purpose is to detect and break into possibly endless loops. The task time limit defaults to a system value unless set otherwise by the user. The task run time may be used to update accounting records upon termination of a task.

4.1.4 Task Synchronization

The control program requires facilities from the physical system to permit interlocking of control functions and to gain exclusive control over system resources when necessary. (The Logical Machine functions for the equivalent of the wait-post and enqueue-dequeue facilities of OS/360 are as yet undefined, but they may use the same facilities described here.)

Tasks are interlocked by two operations, ENTER and LEAVE (which correspond to E. W. Dijkstra's P and V operators). ENTER and LEAVE operate on a special semaphore type of integer variable and, apart from initialization, they are the only operators that can change a variable of type semaphore.

ENTER S, given by one task, locks the semaphore S to gain exclusive control over that variable (see the Storage Management Subsystem section on the details of locking) and subtracts 1 from S. If S becomes or remains nonnegative, it is unlocked and the task proceeds. If S becomes or remains negative, the current task is placed on a queue for S and S is unlocked.

LEAVE S, given by another task, locks the semaphore S and adds 1 to S. If the new value of S is still negative or zero, a task waiting on the queue for S is released. In any case, S is unlocked and the current task continues.

Suppose S is an interlock for a number of equivalent resources, specifically a set of N printers. Initially S is set to N. Each task requesting a printer gives ENTER S, thus reducing S by 1. When the N-th printer is allocated, S = 0. The next task requesting a printer sets S to -1 and must wait. A subsequent requestor leaves S = -2. When a task finishes printing, it gives LEAVE S, adding 1 to S and releasing a waiting task as long as S remains negative or zero. Thus at any time a positive S

represents the number of still unallocated resources (no task waiting), a negative S gives the number of tasks waiting (no available resources), and S = 0 indicates everything busy and nothing waiting.

As another example, let S be an interlock for an event. Suppose task A attaches a parallel task B and sets S = 0. Later A needs to synchronize with B. At the point of synchronization in the two programs, A has ENTER S (equivalent to Wait) and B has LEAVE S (equivalent to Post). While running, if A gets there first and gives ENTER S, it sets S = -1 and waits until B gives LEAVE S releasing A. If B gets there first and gives LEAVE S, then S = 1 at the time A issues ENTER S, leaving S = 0, whereupon A proceeds immediately.

4.1.5 System Efficiency and Tuning

Methods of observing system queues will be provided to measure the utilization of critical resources and identify bottlenecks. Neither over-long queues (bottlenecks) nor frequently empty queues (excess capacity) represent an optimum cost-performance relation. Thus queues are important indicators where system tuning may be needed.

Another example is observing high but unproductive activity. Too frequent page transfers between levels of the storage hierarchy may indicate a temporary overload, and performance may increase overall if the input load is temporarily reduced. It may also indicate the need for increasing the capacity of a level. Conversely, too rare a use of a given storage level may indicate over-capacity at the next lower level.

Thus a completely automatic system does not mean an unsupervised system. A system should be tuned to suit a changing environment.

4.1.6 Hands-Off System Operation

Normal system operation, including start-up and shut-down, should require a minimum of routine human intervention. To avoid a serious bottleneck and a common source of error, there is no central system console. Instead any terminal can be parameterized to perform a particular supervisory or operating function. Routine operator

functions will be restricted to simple physical tasks which do not justify full automation. Examples are loading paper for printers and handling storage media in shelf storage. Such actions are performed only upon instruction by the system and are monitored by the system.

Complex decisions are reserved for exceptions requiring supervisory intervention via terminals with proper authorization. An operator would normally not have such authorization.

4.1.7 Error Correction and Recovery

Facilities for error detection and, as far as practicable, automatic error correction will be designed into all parts of the system to assure a high level of reliability and availability. On-the-fly error correction is, of course, costly and cannot guarantee continued operation under all circumstances. For that reason, and because user errors require them anyway, journaling and restart from a checkpoint will be provided so that rapid recovery will be possible.

4.2 The Storage Management Subsystem

The Storage Management Subsystem (SMS), sometimes called the "storage hierarchy", furnishes all addressable storage for the system. This section discusses the automatic storage management function provided by this subsystem to all processing units, the operations and operand addresses by which processors communicate with the SMS, the responses given, and the internal structure and operation of the SMS.

The SMS also provides interlocking and locking functions to allow many tasks to run concurrently in the same or different processors, yet provide for their synchronization when necessary. Other functions designed to improve storage performance are briefly described.

4.2.1 Storage Management Function

The SMS function is purely one of storage management: to allocate storage spaces when requested by a processing unit (PPU or SSPU), to determine the physical location of the stored information, and to provide access to that information. Storage management is completely automatic and requires neither PU nor manual intervention. Several PU's may be connected to the common storage hierarchy, which manages their separate storage requirements or permits them to share access to common information.

The SMS appears to a PU as a single very large and fast storage device. It differs from earlier "virtual memories" in several important respects. The SMS manages all storage, including bulk storage previously classified as input-output, in a uniform fashion. It manages only space allocated to real data, not the unallocated storage capacity (There is no list of free space). New space can be created directly in the fast storage level at the processor without reference to slower levels, as long as pages are available. In fact, the allocation mechanism of a conventional memory is almost non-existent.

Storage is logically divided into spaces of varying sizes. Each space contains a varying number of bytes. The size, or extent, of a space and the number of spaces available are almost unlimited. Each space can grow or shrink independently of any other. The only limitation is the sum of the extents of all active spaces which cannot exceed the total capacity of the entire SMS. Even that limit is not rigid since, with operator assistance, the available capacity can include off-line storage.

Physically the on-line SMS consists of a number of levels, from the very fast, but relatively small, storage unit directly associated with a PU, through levels of increasing size and decreasing speed, to rotating disks and demountable tape strips. Storage areas are subdivided into blocks, or pages, which are passed from level to level to a PU as needed and drift back out when not used. The physical location remains hidden from the PU, which always addresses the SMU as if it were a single-level store.

The SMS is concerned only with storage management, not with the form or contents of the information stored there. Data and task management are PU functions. The procedures needed for this are, of course, stored in the SMS as is everything else.

The SMS performs a storage function that is purely internal to the system. Communication with the outside is a separate function performed by the source-sink subsystem.

4.2.2 Addressing

How?

Each logical name appearing in the translated code is transformed by a PU into a pair of nonnegative binary integers, the space number and the offset, with which it addresses the SMS. The SMS uses the space-offset pair to search for the current physical address of the desired data object. Space numbers are inaccessible to the programmer. (Offsets may have a 1-to-1 correspondence with index values used by the programmer.) Physical addresses are even inaccessible to the PU, let alone the programmer. The inaccessibility of addresses provides a high degree of protection against erroneous or unauthorized accesses.

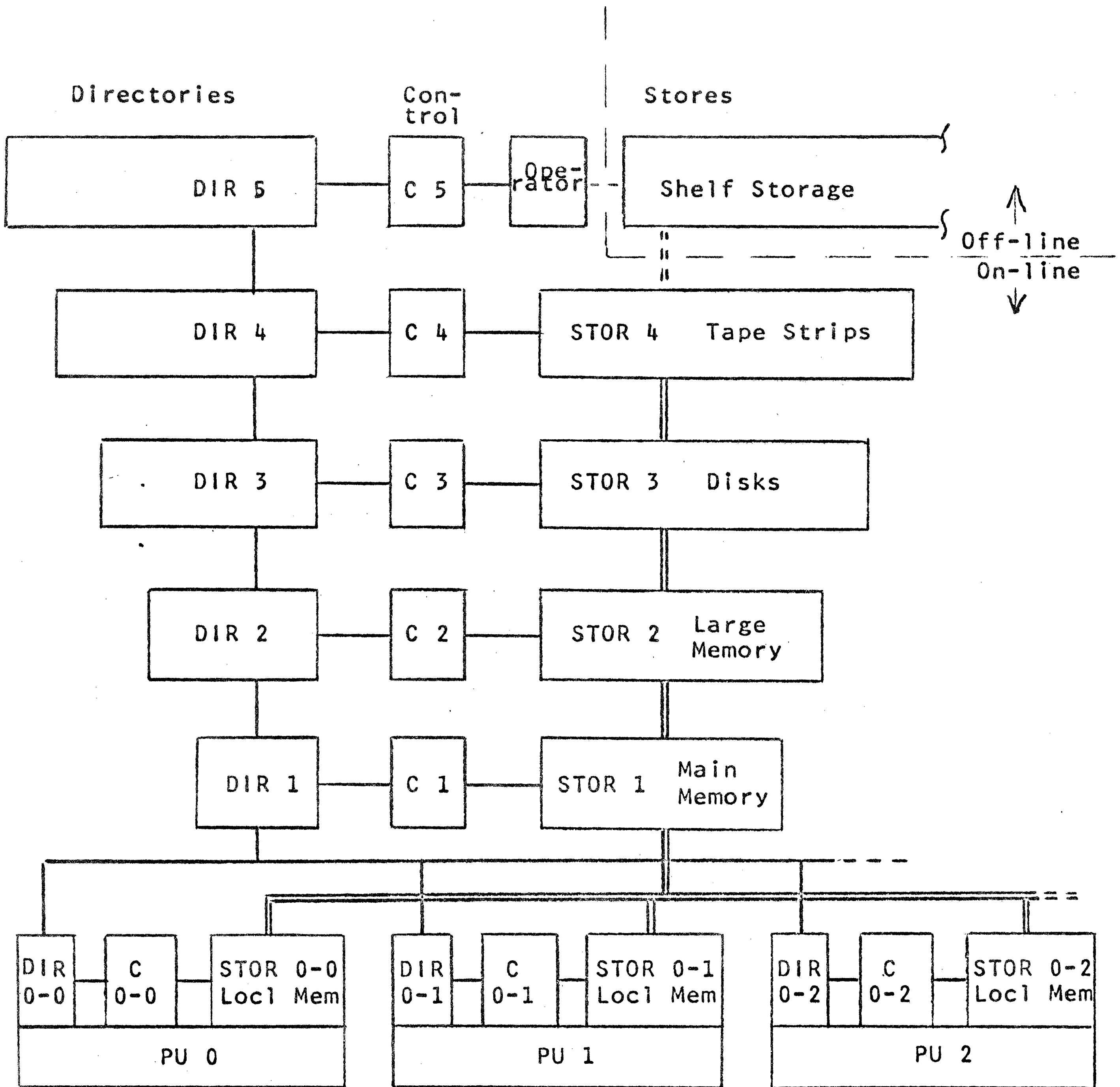


Fig. 4.2.1 STORAGE LEVELS

Space numbers are 52 bits long. When a new space is created by a PU, a unique number is assigned to that space and not re-used after this space is destroyed. Even an inactive space that has migrated to off-line shelf storage retains its unique space number so that it can be restored to the system at any later time without fear of conflict with other spaces.

Offsets are 24 bits long and specify an 8-bit byte within the space. Successive bytes are addressed by consecutive offset integers, starting with 0 for the first byte and ending with X-1 for a space containing X bytes. Offsets of X or greater are not allowed.

The PU can index through a space by doing address arithmetic on the offset as long as the result remains within the current extent. Arithmetic on space numbers has no meaning and is not permitted by the PU.

Protection against invalid space or offset references is a function of the PU hardware. The SMS assumes that all addresses given by a PU are valid and does not, for example, keep track of a space extent. Invalid references that happen to fall in a valid page (i.e., references just beyond the last byte) are not detected by the SMS. References to a non-existent page would trigger off a long search through all levels of the SMS, so that the error response time of the SMS is too long to be a useful response to a programming error. Such an erroneous page reference would only be the result of a hardware error.

4.2.3 Processor-Storage Operations

The PU can request the SMS to perform these operations:

CREATE		
ADD	S,F,C	(F=X, Y=X+C)
WRITE	S,F,C	(F+C < X=Y)
READ	S,F,C	(F+C < X=Y)
DELETE	S,F,C	(F+C = X-1, Y=X-C)
DESTROY	S	
BRING	S,F,C	

where S is the space number,

F the offset for the first byte,
C the count of bytes to be transferred,
X the old extent, and Y the new extent.

CREATE sets up a new space. The SMS sets up an empty page and assigns a new space name, which it returns to the PU. (The PU subsequently initializes the space with a space descriptor and extent using an ADD operation.) Space names are assigned by the SMS from a single 52 bit space name counter that is incremented by 1 whenever any PU gives a CREATE.

ADD stores C bytes at the end of space S, extending the space. F must equal X, the old extent, and the new extent is X+C.

WRITE modifies (updates) C bytes in space S starting at offset F.

READ fetches C bytes from space S starting at offset F.

DELETE fetches C bytes from the end of space S, shrinking the space. F+C+1 must equal X, the old extent, and the new extent Y is X-C.

DESTROY removes space S from storage. All pages belonging to S at any level are freed for other use. No further references to space number S is possible, and S will not be re-used.

BRING causes the pages implied by S,F,C to be "staged" from a slow electromechanical storage level into a fast electronic level to facilitate subsequent access. BRING always continues as far as necessary through the SMS levels. If the SMS finds that an earlier BRING for the same page is already in progress, it ignores the second operation.

CREATE/DESTROY, ADD/DELETE, and WRITE/READ are complementary operations. WRITE and READ both reference an existing portion of a space and do not change the extent. ADD and DELETE expand and shrink an existing space at the upper end of the space. CREATE and DESTROY deal with entire spaces and do not reference bytes stored within the space.

WRITE and ADD differ in two important respects. WRITE assumes modification of an existing page and will cause a possibly time-consuming search for it until found; ADD creates a new page when necessary and bypasses the search. ADD implies a change in the extent whereas WRITE does not. All checking and changing of the extent field is done by the PU using appropriate READ and WRITE commands to access the field.

BRING can specify any part of a space or all of it (F=0, C=X). All other operations using the count field C are limited to a maximum count equal to the page size in level 0 of the smallest implementation (perhaps 64 bytes.) This means that a single operation can cross no more than one page boundary. At the start of an operation a check is made that one or two consecutive pages, as needed, are available before the operation proceeds. Once a reading or writing operation has started, it is allowed to complete without interruption.

4.2.4 Storage Responses

After requesting an operation the PU waits until data transfer starts or one of these responses is given:

Completed — Follows any data transfer and indicates successful completion.

Delayed — Indicates that the search must proceed to a slow, electromechanical and possibly manual, level of the SMS. At the same time, the SMS issues an automatic BRING command for the desired page. (The PU may anticipate the need by issuing an explicit BRING beforehand.)

Space Not Found — When a BRING operation does not find the page previously requested, a dummy page (status Void) is inserted in the outer electronic level. When the PU later tries to refer to that page, this space-not-found response is given.

Error — An uncorrectable error has occurred while accessing the requested page.

When the delayed response is received, the PU may abandon the search or hold its task in a queue. The task may be reactivated after a set time interval elapses or when its turn comes again in the task queue.

4.2.5 The Hierarchical Structure of Storage

The SMS levels are numbered 0,1,2,... Level 0 is the fastest storage device directly attached to the PU. For reasons of speed the level 0 unit may be designed as an integral part of its associated PU, with multiple PU's interconnected via common buses between levels 0 and 1. To be specific, the description will assume multiple level 0 units and single units at higher levels, but other configurations may be implemented that are logically equivalent.

Architecturally there is no limit on the number of levels. That number, and the choice of capacity and speed at each level, are based on cost and performance considerations for a particular implementation. For a large time-sharing system with data bank storage these factors clearly enter into the choice:

1. The system must be open-ended. On-line data will slowly grow, and the system would suddenly come to a halt unless inactive data can smoothly, but not irreversibly, migrate off line. Thus storage management extends to shelf storage.

2. The off-line storage medium must be low in cost to permit indefinite shelf storage. This virtually dictates a flexible, tape-strip medium.

3. Inherently long access times at this level require that active spaces be written at one time to keep them physically together.

4. A large capacity disk store is required, both as random-access buffer for the tape-strip level and as a medium for page overflow from faster levels. Scattering of pages for the same large space is unavoidable at this level. Removing disk packs, as in present systems, becomes impractical and undesirable.

5. The disk level provides a necessary non-volatile back-up for volatile electronic levels.

Each level communicates only with the next level above and below. Data are transferred as pages of fixed size at each level, but the page size need not be the same at every level. The ratio of page sizes between adjacent levels should be a non-negative integral power of 2, the actual size being an implementation choice. A page in the level below becomes a line of a page in the level above. All lines of a page belong to the same space. Normally each space occupies at least one page at any level, and the last page may not be full.

This apparent waste of capacity greatly simplifies storage management. It makes practical the almost unlimited expansion of a single space by adding active pages as needed without disturbing other spaces. At the slowest levels, however, it may be possible to pack and unpack data to gain storage efficiency without substantial loss of performance. Such modifications do not affect the overall design and will not be discussed further.

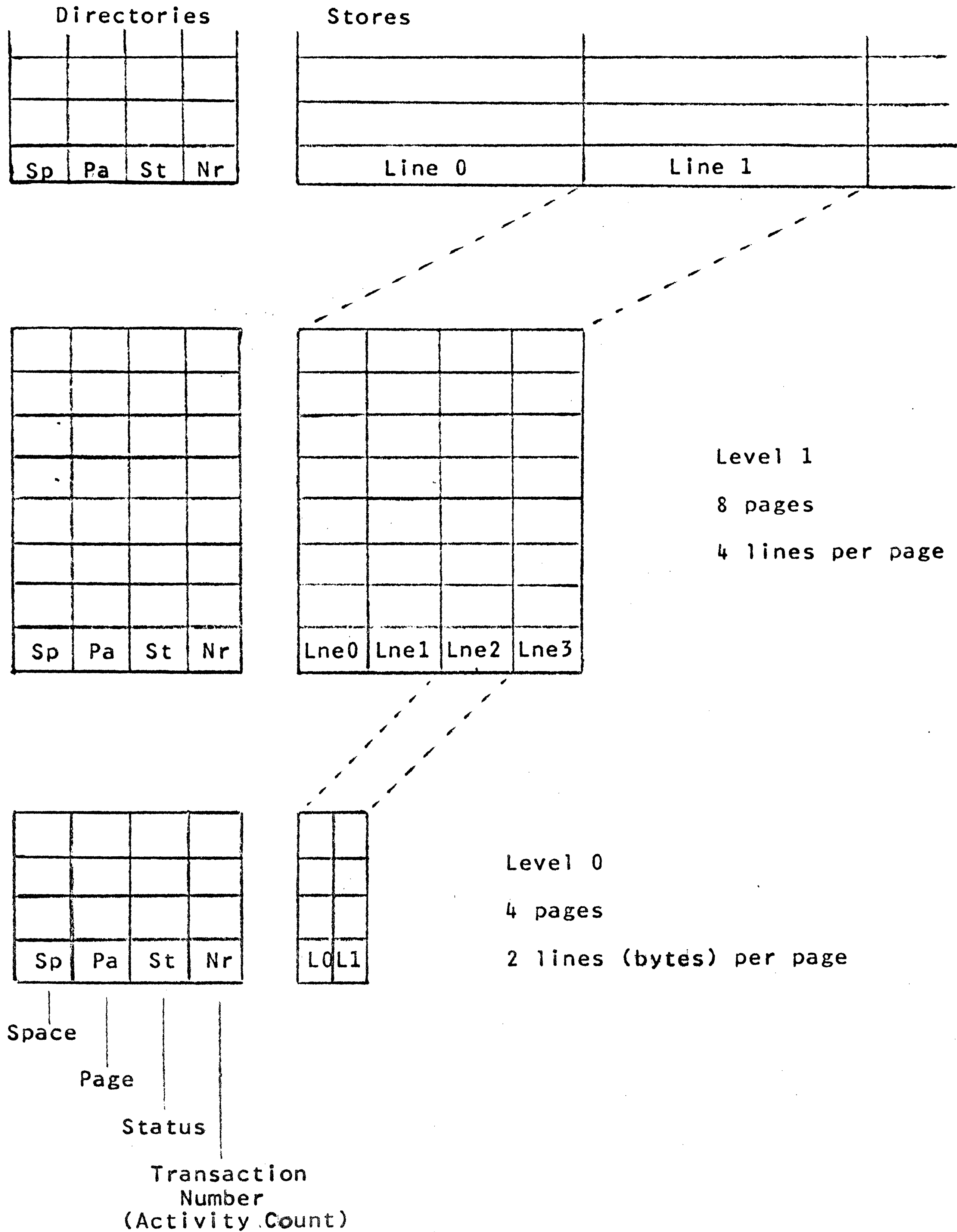


Fig. 4.2.2 STORAGE LEVEL LAYOUT

Each level has, in addition to one or more storage devices, a separate directory storage. The directory contains one entry for each page location at that level. An entry consists of space number, page number, status, and activity count.

The space and page numbers correspond to space number and offset at the PU level. For a page size of 2^n bytes the number of the page containing a given byte is derived from its offset by dropping the low-order n bits.

The page status, at the completion of an operation, may be:

Unmodified -- this page has only been read, and a valid copy also exists at a higher level.

Modified -- this page has been stored into; a copy exists at a higher level but is not up-to-date.

New -- this page has not been moved to the next level since it was created; thus no copy exists at any higher level.

Shared -- See section 4.2.7

Void -- A dummy page indicating page not found (see section 4.2.4); this page is not displaced to other levels.

The activity count is incremented whenever a reference is made to that page. This count is used to determine the least-recently-used page when one must be displaced to the next higher level.

The directory for a level is searched by space and page number to find whether and where a given page is located. A page can be at any location in a level. For greater speed the directory at the fastest levels may be implemented in an associative array. In slower levels a sequential search of a conventional storage array may be sufficient.

Each level is equipped with the controls necessary to implement the various inter-level operations, and with any buffers and interlocks required to synchronize page transfers and to prevent interference from simultaneous requests by the other neighbor. The controls also take care of requirements peculiar to each level, such as error detection, correction, and recovery.

IBM CONFIDENTIAL

4.2.6 Internal SMS Operations

The above listed commands from the PU and responses from the SMS apply at the interface between a PU and its level 0. Operations between storage levels are basically the same, except that page and line numbers replace offsets, and only one page is moved at a time (count=1).

When a directory search of one level reveals that the page desired is not in that level (level i), a search is initiated in the next higher level (level i+1). First a location is freed in level i to receive the page. The directory is searched for the least-recently-used (LRU) page according to the activity count.

A DELETE or DESTROY operation sets this count to zero, so that deleted pages are used first. If no deleted page exists but the LRU page has a status of Unmodified or Void, that page location is used immediately since either a valid copy exists in a higher-numbered level or none is needed. Otherwise the LRU page must first be displaced by initiating a WRITE (if status is Modified) or ADD (if New) command to level i+1. Because the displacement operation may take time, the original request, coming from level i-1, is queued to permit other operations for level i to proceed. When the displacement to level i+1 is completed, the original request is resumed by issuing a READ request to level i+1. This READ may trigger a similar chain of events in level i+1, so it may be queued again to await completion. Interlocks meanwhile block any subsequent requests from level i-1 for this page, so that there will be no premature use of the incomplete page and no additional search request for the same page.

If the page size in level i+1 is greater than in level i by a factor of 2^k, the page number passed up from level i is converted to a page number for searching the level i+1 directory (along with the space number) by dropping the low order k bits. When the page in level i+1 is found, these k bits form the line number which determines the position of the level i page as a line within the level i+1 page.

The WRITE or ADD request resulting from a displacement in level i triggers off a search in the level i+1 directory. If the operation is ADD and this is the first line of a page (the low-order k bits are all zero), a new page is started in level i+1 at a location to be freed as before. If the ADD operation refers to other than the first line, the line must be stored into an existing page; likewise a WRITE implies changing an existing page. The existing page must then be found. If it is not in level i+1, the same cycle of events

described above is repeated to get the page from level i+2.

Thus the most complex chain of events for level i, in response to READ, WRITE, or ADD request from level i-1, consists of:

1. Search directory i for page desired by i-1. If not found, queue up request from i-1.

2. Search directory i for LRU page. Block the page.

3. If not free, queue up READ request for i+1. Initiate displacement of LRU page by WRITE or ADD request to i+1.

4. When displacement completes, send queued READ request to i+1.

5. When READ completes, finish i-1 request by transferring the desired line. Unblock the page.

One or more of the steps in this sequence may be skipped depending on conditions.

A DELETE operation is similar to READ in that the desired page must first be located and the specified line transferred to the level below. If this is the first line of a page, however, the directory entry is marked as deleted and a DELETE request is passed to the next higher level. The chain of DELETE requests ends when the line number is not zero (the lines below being still valid) or when the page found has the status New (there being no corresponding line at a higher level).

DESTROY simply causes all page entries for the specified space to be deleted in every directory. (Other modes of DESTROY may be found desirable, such as a DESTRUCT command which also causes the storage pages to be overwritten for extra security.)

A CREATE command by the PU initializes a page marked New in level 0. CREATE does not appear at higher levels. When a page so created must be displaced to level 1, an ADD command is generated as with any other page marked New.

It may be noted that ADD or WRITE commands to one level may trigger off a READ command for the same page to the next higher level but not another ADD or WRITE for that page. ADD or WRITE commands are always the result of a displacement action for a different page. ADD or WRITE may be generated automatically by a level when it is not otherwise busy and (1) the supply of empty or inactive unmodified pages is low, so as to anticipate

future needs for free pages, or (2) to create backup copies of pages at higher levels for security. Such automatic displacement action leaves the displaced pages in Unmodified status at their old location.

As mentioned before, a PU ordinarily waits for its level 0 to complete any request. Thus no other interlocks are required between them. At higher levels, however, multiple PU's may independently initiate requests for the same page. Also a PU request for a page may come just after a level has initiated an automatic displacement action for that page. Hence the need for interlocking multiple page requests at higher levels.

4.2.7 Interconnections for Multiple Processors

Each PU is connected directly to its own level 0 storage device and directory for transfer of operations, data, and responses. There is no direct connection between PU's for storage operations. At the interface to the rest of the SMS all level 0 units and the common level 1 unit are interconnected via common directory and data buses and control lines.

The PU's operate simultaneously and independently of each other unless they make reference to the same page. At that point the hardware will provide temporary interlocks so that the PU's in fact proceed sequentially. Logically the result is the same as if all PU's were connected to a single storage unit. The interlocks are transparent to the user. (See also ASP Memo 067 for details.)

The previously stated rules for searching the levels are modified somewhat for parallel PU operation. If a PU requests a READ for a page not in its level 0, a search for a copy of that page is initiated via the common bus in the level 0 units of all the other PU's. If one is found, a copy is sent to this level 0 and both directory entries are marked in Shared status. Thus multiple copies can exist in different level 0 units for simultaneous reading without further interference.

When a PU requests a WRITE or ADD operation for a Shared page in its level 0, a cancel request is broadcast to all other level 0 units to delete all Shared copies of the page before the page modification proceeds. A WRITE or ADD for a page not in this level 0 causes a search in all other level 0 units for a copy to be transferred, but all other copies are cancelled. If there is no copy in level 0, the request is directed to level 1. Thus for page modification a given PU takes sole control over that page. Subsequent

READ requests by another PU causes the modified page to be written into level 1 so as to make shared copies available to the level 0 requestors.

Level 1 and higher levels are not affected by the sharing and do not need to "know" which PU, if any, has a copy of a page. Shared status appears only in level 0. (The possibility of multiple units at higher levels for protection against failure is not being addressed here.)

The net effect of these interlocks is that there is no restriction on simultaneous reading of the same page by several PU's or on simultaneous operations on different pages of the same or different spaces. Simultaneous writing of the same page, when needed, suffers only the loss in time for transferring the page from one unit to another. At any instant a READ operation by any PU is always guaranteed to give the most up-to-date version of the information.

4.2.8 Page Locking

To synchronize different tasks in the same PU (multiprogramming) or in different PU's (multiprocessing), two further operations are provided in level 0 only:

```
LREAD  S,F,C
UNLK
```

LREAD (Lock and Read) is similar to READ with the following additional properties.

1. It seizes exclusive control of its page, as described above for WRITE, by cancelling all other copies in level 0 units.
2. It causes that page to be locked in its level 0 unit.
3. Interrupts for this PU are disabled while any page is locked.

Locking a page means that no other PU can gain access to that page and that the page cannot be displaced. The PU that locked it continues to have free access to that page. If another PU requests the locked page, it must wait; it cannot get the page from level 1 or proceed to another operation. The locked page is that containing the offset specified by LREAD. If the operation crosses page boundaries, the second page is not locked. Additional pages could be locked by separate LREAD operations, however.

UNLK (Unlock) releases all locks for its level 0, enables interrupts for this PU, and allows other waiting PU's to proceed. UNLK has no operands. Only one UNLK is required to release all locks left by previous LREAD operations. To prevent permanent system locking due to a failure, locks are released also after an automatic time-out.

LREAD followed by WRITE allows the inspection and setting of a task interlock bit or count (semaphore). If the task being synchronized has to wait, it can be placed on a task queue before giving UNLK to allow other PU's to test the same task interlock. The control program providing this synchronizing function must be carefully written to avoid deadlocks and to reduce the lock time to a minimum. The hardware merely enforces sequential operation and cannot itself create deadlock situations. (See also ASP Memo 067.)

4.2.9 Lookaside

For extra speed a PPU may contain a logical-name array that permits look-aside to minimize storage accesses on repeated reference to the same information. This array contains, with each short-hand logical name, the physical address in level 0 where the corresponding byte is currently located. This physical address is then passed to level 0 instead of a space-offset pair for a directory search.

Because it is closely tied to other PPU functions, this lookaside feature is described in more detail in the PPU section. Its function, however, is also tied closely to the level 0 portion of the hierarchy. The physical address is provided by level 0 after the first search for the desired item, and level 0 must cancel an entry whenever the corresponding page is displaced. Thus there is no basic contradiction with the principle that physical addresses in a hierarchy level are "known" only to that level.

4.2.10 Other SMS Functions

The SMS will be provided with thorough checking facilities and, where practical, with automatic on-the-fly error correction. Automatic restart after an uncorrectable hardware failure probably requires some form of automatic journaling device that records all changes since the last checkpoint.

To speed up purely sequential operation it may be desirable to have a "sequential" modifier on READ or WRITE operations that would prompt the SMS to pre-fetch additional consecutive pages of the same space.

At the outer levels additional operations may be needed to read and write complete spaces in a highly compressed format under supervisory program control.

4.3 The Program Processing Subsystem

The Program Processing Subsystem consists of one or more Program Processing Units. These units communicate only by use of the interrupt bus and shared storage.

The Program Processing Unit (PPU) of the AFS system is the physical counterpart of the logical interpreter of section 2.6. In this section, we discuss the structure of the PPU, its relation to other subsystems, and the framework of a large machine implementation.

4.3.1 An Overview of the PPU

The job of the PPU is the evaluation of a statement, or an ordered list of symbols. These symbols may be built-in operations, literals, or logical names. A built-in operator produces as its result, an architecturally-defined function of its arguments. A literal is a data unit which represents itself (e.g., 5.3) and cannot be assigned a value. A logical name may represent a value, or name a function. In order to distinguish these two possibilities, the current attributes and values of the referenced data element must be retrieved from storage.

The PPU is also required to communicate with other subsystems by initiating and responding to "wake-up" signals or interrupts.

4.3.2 Data Representation

The fundamental concept of data representation of the system is that all data is self-defining. This implies that attributes are associated with the data in storage. Any reference to data implies an examination of the attribute set, in order to determine if the requested operation is applicable, whether implicit type conversion is required, whether the operand is scalar or an array or structure requiring special sequencing control.

The PPU supports a number of data types which are detailed in Table 4.3.1. The principal separation is between problem data and program control data. The attributes of a data item define its value at any instant. On the other hand, any assignment of a new value to the item requires the use of the generic descriptor in the LDT (section 2.5). The generic descriptor specifies what the value descriptor can be. The value descriptor indicates what the value is.

Applicable Attributes	Type	Scalar Other	Length of Element in Bytes	Scale Factor	Value
Problem Class	binary real	X	n	q	v
	decimal real	X	n	q	v
	binary complex	X	n	q	v
	decimal complex	X	n	q	v
	bit string	X	n		v
	character string	X	n		v
	unassigned	X	n		v
Program Control Class	semaphore		n		v
	pointer	X	n		v
	offset	X	n		v
	label	X	n		v
	entry	X	n		v
	event	X	n		v
	task	X	n		v
	module		n		v
	area		n		v
	symbol table		n		v
	reference table		n		v
	system pointer		n		v
	system offset		n		v
	undescribed		n		v

Table 4.3.1 Primary Value Descriptor

An entry in a column indicates that the choice is applicable

If the data item is a scalar, its complete descriptor appears in a two-byte field. For more structured data, additional descriptive information is required including array/structure, level within a structure, dimensionality and array bounds or index set specifications. The exact formats for the array and structure descriptors have not yet been determined.

Any or all of the attributes of non-scalar data may be factored, or represented once for the entire data collection, instead of with each individual element. For example, a numeric array might be represented with all attributes but scale factor and value factored into a common descriptor for the array.

Arithmetic data may be represented in any combination of binary/decimal, and real/complex choices. Each arithmetic item is associated with a precision, p, the number of digits which appear in the quantity, and a scale factor, q, which specifies the position of the radix point relative to the low-order digit. A negative scale factor signifies that the radix point is to the left of the low-order (right most) digit, a positive scale factor to the right of an implied zero digit to the right of the low-order digit.

String data is another variety of problem data. The two forms of string data are bit string and character string. Instead of precision, string data possesses a length attribute, the number of characters or bits in the item.

Data of program control type is not directly used by the programmers, except for certain assignment statements. Table 4.3.1 presents a list of the program control data types currently envisioned. We note that certain of these data types may appear in structure or arrays.

The final aspect of the data descriptor is the value of the object. The value is represented in storage by a string of bytes of an appropriate length, but the interpretation of these bytes is possible only by means of the descriptor. In most instances, the complete Descriptor (including value) for a data item will be found in a Reference Table, as discussed in section 2.5. If the item is too large to fit reasonably into the Reference Table or the number of bytes of storage required is expected to change, the item will be assigned by the system to a newly created space. The Reference Table entry will then be a System Pointer to the space containing the actual item.

4.3.3 Space and the Addressing Mechanism

The virtual storage of the system is composed of spaces. Each space contains a known number of bytes, m, which are numbered from 0 through m-1. Each space, as a whole, carries its own length and descriptor information in its first four bytes. The first three bytes contain a binary integer specifying the number of bytes currently allocated to the space. The fourth byte is used to tell what the type of the space is. Currently, we recognize spaces for Link Tables, Declare Tables, Symbol Tables, Reference Tables, and Undescribed. A space is called undescribed whenever a system user logically knows a priori the type of space which he is using. In other words, an undescribed space means that " if you have to ask what type of space this is, you shouldn't be here."

Space descriptors, like data descriptors, are transparent to the user. They are tested and manipulated only by well-controlled system functions.

Each object in storage has a unique IID or internal identifier, which consists of a 52 bit space number and a 24 bit byte offset within the particular space.

Seen from the PU, storage is an access machine which responds to certain calls, including READ, WRITE, CREATE, DESTROY, ADD, and DELETE. The arguments for these calls consist of the IID mentioned above, and a value for WRITE. The hierarchy responds with a completion message, and perhaps a value, as in READ.

The logical mechanism required to convert between the programmer's symbols and the IID representation has been discussed in Section 2.3 and 2.6. That scheme requires a table reference to convert the LLT offset to a space-offset pair in a Reference Table, followed by fetching the specified object.

If the actual data is located in the table, the search is over, but otherwise, a pointer chain must be followed until the data (or its space-offset identifier) is obtained.

Any request to storage is expected to employ a valid space-offset pair, which means that the address mechanism must read the extent field of any space, compare the extent to the specified offset, and make the request only if the extent of the space is not violated.

4.3.4 System Addressability Considerations

In this section we discuss the physical embodiment of the Activation Tree and Dynamic Storage control which were described in Sections 2.3 and 2.4. The goal of the discussion is to clarify the process of obtaining addressability to the various spaces which are referenced by a module.

We have previously seen that the LTCB provides identification for four spaces associated with the task:

a) That branch of the Activation Tree associated with this task.

b) The Interpreter currently assigned to this task.

c) The set of System Storage Anchors

d) The set of User Storage Anchors.

In turn, the Interpreter identifies the task's Execution Stacks, the current Module (in the Program Tree), statement number, and token within the statement.

4.3.4.1 The Call Stack

The Call Stack corresponds to that portion of the Activation Tree associated with the given Logical Task. Thus, the flow of control in the task is represented by this stack, and the next unit of work for this task is indicated by the last entry in this stack.

The essential information in the Call Stack consists of a pointer to the current node of the Program Tree and the offset in the calling block to which control is to be passed on exit. The connectivity to the Program Tree provides indirect access to the LLT, LST, LDT and Reference Table for the activation.

The Call Stack operates logically as a pushdown stack or extensible space in storage. However, it is desirable to scan successive entries without deleting them, so that attempts to branch to no-longer-existing blocks can be effectively diagnosed.

4.3.4.2 The Execution Stacks

The Execution Stacks for a task consist of an Operator Stack and an Operand Stack, required for executing prefix Polish code. These stacks are spaces in storage, but are referenced so frequently that special hardware is warranted for their top levels. A further discussion of a proposed implementation will be found in Section 4.3.7.

4.3.4.3 The Storage Anchor Registers

The discussion in Section 2.4 treats the logical requirements on the Storage Anchors in two phases, System Storage Anchors and User Storage Anchors. We recall that the System Storage Anchor contains the space number of a Reference Table, and a User Storage Anchor, the space number of a space containing one DAPOV.

The physical implementation of System Storage Anchors consists of a set of 254 Storage Anchor Registers. We regard this number of levels as infinite. Each such register contains the space number and for checking purposes, current extent of the corresponding Reference Table.

The User Storage Anchors are implemented by placing all such, for the given task, in a space, whose identification is placed in Storage Anchor Register 255. The Relative ID field of the LLT selects a particular Storage Anchor (System Space Pointer) from the designated space.

The Space referenced by a User Storage Anchor is constrained to contain at most one user DAPOV. However, successive generations of a particular variable must be chained together, by a means transparent to the user. No explicit offset to the user DAPOV is provided.

Clearly, we must have enough Storage Anchor Registers in high-speed hardware to permit high-speed operation. Several possible implementations include:

a) Restricting the number of System Storage Anchors to 14 or so, instead of 254, and providing 14 actual registers.

b) Using a block storage method, where certain sections of the complete table are automatically swapped into a small buffer.

Option b is attractive, since references will probably cluster about the current level, appear at the external level, or invoke some User Storage Anchor with few references between.

High-speed swapping between sets of Storage Anchors and the SAR's will likely be required at task switch time.

Since the Call Stack points to a node of the Program Tree, which is associated with a System Storage Anchor, any Call Stack manipulation also involves Storage Anchor Register modification.

4.3.5 Computational Facilities

In this section, we discuss the computational requirements of the system, corresponding to E-box functions in present-day PU's. These requirements are associated with the problem data types of Table 4.2.1.

The first distinction is between arithmetic and string data types. Arithmetic data types admit standard numerical algorithms (e.g., add, multiply, exponentiation, arctan) as operators. String data permits such operators as concatenate, compare for equality, substring searches, and for bit string, bit-by-bit logical connectives.

As indicated earlier in Section 4.3.2, storage is byte-oriented, and arithmetic data is not necessarily aligned on "word boundaries", and is of variable length. It is a system requirement that the PPU provide both decimal and binary (or hexadecimal) arithmetic, at equivalent speeds. We anticipate that decimal will be the preferred radix, but emulation of S/360-370 will require floating hexadecimal as well.

In the native mode of the system, a fixed point operand, because of its scale factor, is truly that. In fact, the only distinction between fixed point and floating point quantities occurs in assignment, when the generic descriptor indicates whether the scale factor may be changed. In this interpretation, all numeric computation may be considered internally to be floating point.

String operations appear to cause no new problems for a byte-wide VFL unit. If found desirable, many such operators could admit multiple-byte-wide processors.

The system logical language is prefix Polish. In order to execute this code, the hardware requires an operator stack and an operand stack. As the code is scanned sequentially, each operator token is placed on the operator stack, together with a count field which specifies the number of arguments which the operator requires. When an operand (literal or variable) is encountered during the scan, the value of the operand is placed in the operand stack, and the argument count field of the top operator in the operator stack is decreased by 1.

When this count is zero, the operator may be scheduled for execution. The scheduling means that the operator and its operands may be sent to an appropriate execution unit for the required processing, provided that the unit is available. At the completion of the operation, all operands participating are pushed off the top of the stack and lost. The result and its descriptor are placed in the operand stack and the operator stack

is popped up, exposing a new top operator. We then decrement and test the argument count field for this operator. If at some stage, the decreased operator count is not zero, the code scan resumes. Otherwise, we return to the scheduling step.

This description of the prefix Polish stack algorithm assumes that operators and operands are distinguishable. In order to achieve this property, the execution-time attributes of symbols must be examined. The logical architecture provides a code initialization function, capable of translating the logical language into appropriate execution time code.

The physical hardware language, "under the covers", is not yet defined. The definition will be transparent to the user of the system.

The rich operator set of the logical architecture makes the use of microprogramming attractive. In particular, the use of pageable microprograms, as presented by McGovern, Moore, Willoughby and Kurtz of Department B05, IBM SDD Poughkeepsie, appears to provide a mechanism by which a small, fast control store may perform like a large, fast one.

4.3.6 Lookaside

In attempting to implement this architecture, the system designer is forced to consider the cost/performance implications of its features.

The most important consideration is that of shortening the time between an operand request via the LLT and the return of the actual DAPOV. The proposed method is to provide an associative lookaside memory in parallel with the directory for the level 0 cache. Figure 4.3.1 sketches a cache, with the lookaside memory in the dashed block in the upper right corner.

In this discussion we denote a collection of pointers to the current Module, its LLT, LDT and LST as System Bases. Physically, the System Bases are found in the System Base Registers, a portion of the Storage Anchor Register Array.

Without lookaside, the typical Logical Name search begins with the LLT space number (from the System Bases portion of the array containing the SAR's), and the LLT offset (i.e., Logical Name) from the program used as an index to the cache directory memory. If the search is successful, the physical address of the LLT entry in the cache is read out, and used, in the cache address register, to read out the LLT entry. This entry consists of a storage anchor name and relative offset.

This information, on the cache data out bus, is used to select a Storage Anchor Register and its appropriate offset. Another directory search is made, and, if successful, the physical address of a DAPOV is placed in the cache address register. The resulting cache read operation produces the long-awaited DAPOV. If this DAPOV is itself a pointer, and a value is required, we may repeat the loop, except that the Storage Anchor lookup is bypassed.

The use of the lookaside memory permits us to eliminate at least one pass through the cache. In this mode of operation, the lookaside memory and cache directory memory are searched concurrently. If the lookaside memory finds the physical address of the DAPOV, that address is used immediately as the cache address. This produces the desired output in one cache cycle, not two. More than one cycle will be saved whenever a multi-level pointer chain is referenced.

When the lookaside search fails, the direct LLT readout is performed as described above. When the physical address of the desired DAPOV is generated, it is inserted into the lookaside memory and associated with the Logical Name from which it was derived. Subsequent references to this name will be found in the lookaside, until the entry is invalidated.

In mapping a large index set into a small memory, a replacement algorithm is required for determining the entry to be removed when we desire to place a new entry into a previously full memory. This problem occurs both in the cache and in the lookaside memory. The actual organizations and replacement algorithms in these two cases are implementer's choice.

The two cache data busses present or accept left-aligned data. The Write and Read Data Aligners are used to convert between the wide, fixed width cache word and the variable width, variable initial byte usage of logical storage.

Implementers may choose to combine the two cache data registers, data aligners, or data busses.

4.3.7 A Possible PPU Implementation

In this section we discuss a possible high-performance implementation of the PPU architecture. This is diagrammed in Figure 4.3.2.

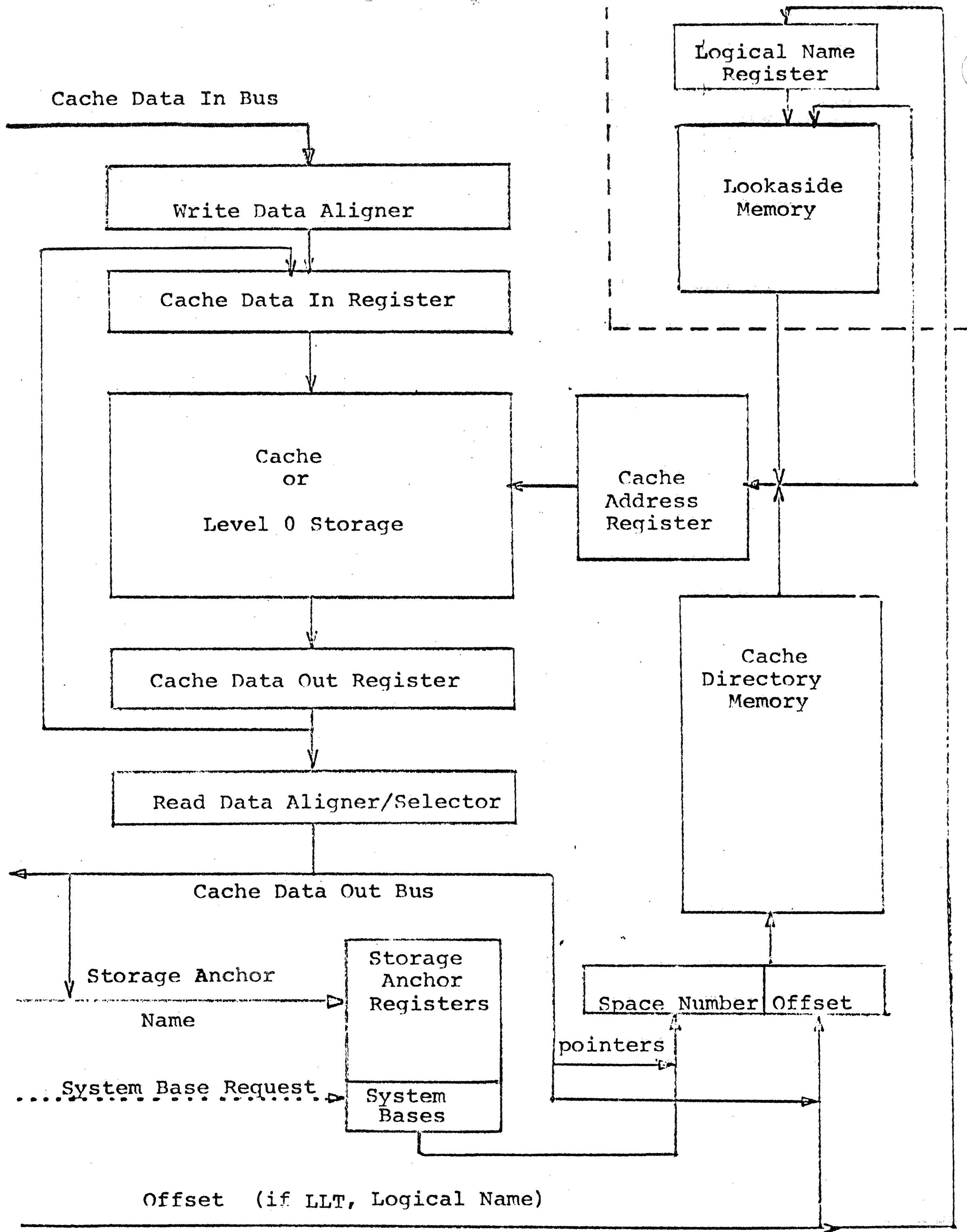


Figure 4.3.1 Cache With Lookaside

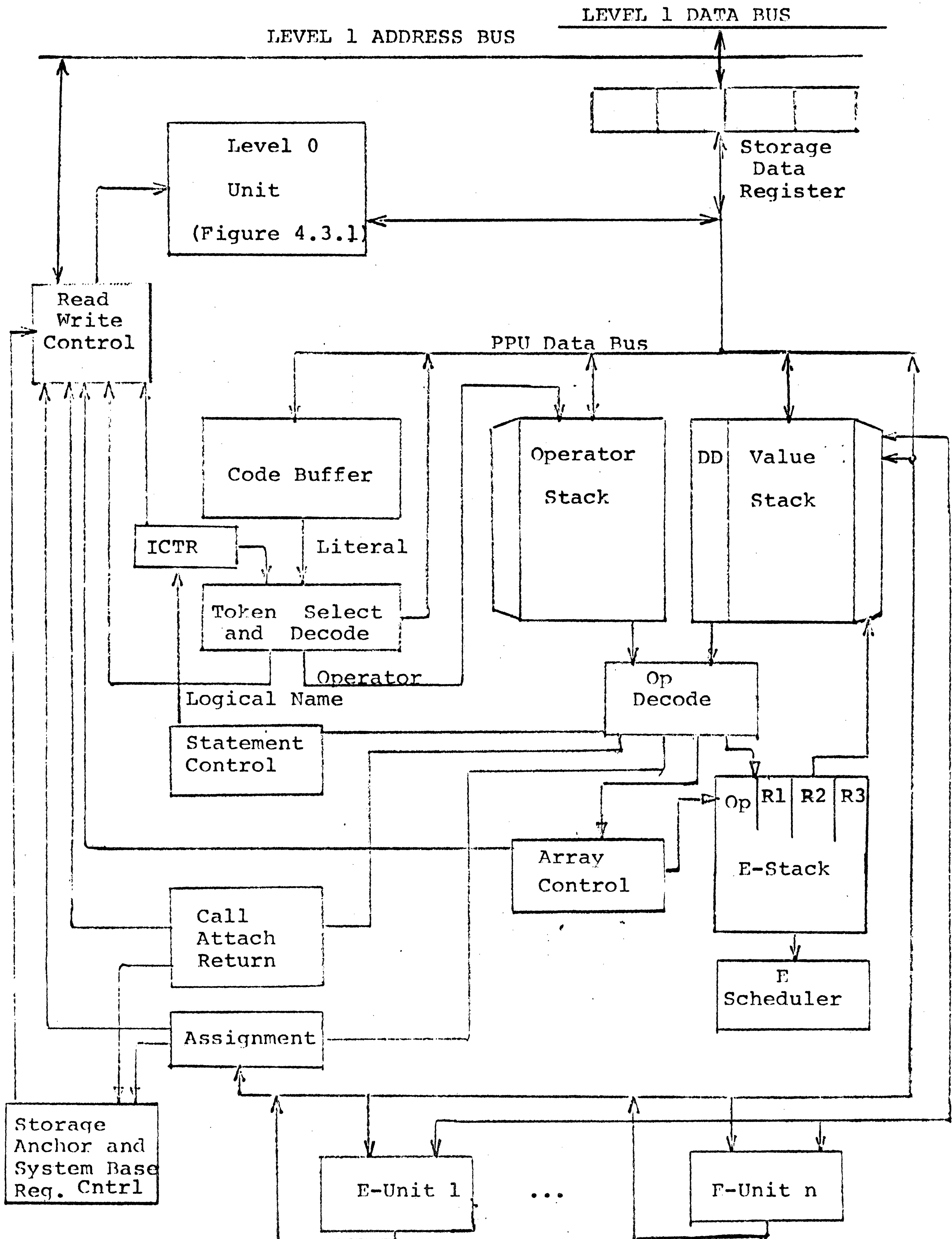


Figure 4.3.2 PPU Implementation

4.3.7.1 Level 0 and the Data Bus

In the Level 0 Unit box, we assume the structure shown in Figure 4.3.1, except that a single bi-directional data bus is shown here. Communication with storage level 1 is achieved by the level 1 address/command bus and data bus.

The Read-Write Control box coordinates all activities at level 0. These include priority determination, block replacement management and level 1 communication. This box also controls the byte aligners, described in Section 4.3.6. It is further responsible for providing multi-level pointer-following lookups, such as occur when lookaside fails, or when data values are not found in a DAPOV of an RT.

The bi-directional data bus is eight bytes wide. Normal usage calls for level 0 to code buffer, value stack or storage data register transfers, and for storage data register, value stack, and new top bus to level 0. There should be provision for bi-directional block transfer between the operator, value stacks and level 0. A path is also shown for placing a literal from the code buffer onto the bus, for transmission to the value stack.

4.3.7.2 I-Box Function

The customary I-box functions of fetching and examining, instructions is carried out using the code buffer, ICTR and token select and decode boxes. The ICTR provides the offset for code in the code buffer, and for fetching new segments. The token select and decode unit examines the byte pointed to in the buffer by the ICTR, and selects an appropriate action. If a value corresponding to a Logical Name is required, the Logical Name is sent to read control with the value stack as its destination address. An operator byte is placed in the operator stack. A literal field causes the entire literal to be placed on the data bus for immediate transmission to the value stack. A Logical Name whose value is not required, as in the target of an assignment, will be placed in the value stack, like a literal.

4.3.7.3 Operator Execution

The theory of Polish stack manipulation was detailed in Section 4.3.5. When the topmost element in the operator stack has found all its operands in the value stack, the operator is sent to the op decoder for classification, and assignment to the appropriate executer.

IBM CONFIDENTIAL

Although the values placed in the value stack are inherently of variable length in bytes, these sizes will be rounded upward to the next multiple of eight bytes. This creates a minor problem in matching data descriptors with their variable length values for easy access in the value stack.

This philosophy also presents a problem in eliminating "noise bytes" transmitted from level 0, before they actually participate in a computation.

We may point out here that the value stack represents a portion of the Operand Stack of each task.

A statement control op will change the ICTR, and probably cause a code buffer refill.

A CALL/ATTACH/RETURN operation will not only affect the ICTR, but will cause the appropriate protocols from Section 2.6 to be obeyed. These will in general involve the storage anchors and system base registers. These registers are shown in an array in Figure 4.3.1, and controlled by the box labeled storage anchor and system base register control, in Figure 4.3.2.

Assignments will be performed using the Logical Name to reference the LDT entry for the destination, to validate the descriptor information or invoke a conversion. For certain types of assignments, some or all of the storage anchor registers will be copied into storage.

For a scalar computational operation, the op and three indices to the value stack will be placed in the E-stack. The E-scheduler controls the issuing of E-stack groups to the appropriate E-unit.

Array computational operations, distinguished by the data descriptors in the value stack, are routed to an array control unit. This unit converts array ops into an appropriate loop of scalar ops, executed from the E-stack. Since arrays will not appear in the value stack, array control must provide the ability to reference storage.

4.3.7.4 Work to Be Done

Needless to say, much work must be done to provide a good system design. We briefly mention some points of interest here.

The data bus philosophy may be overloaded, and prove to be a performance bottleneck. If the literal-to-value-stack path is implemented by a private path, will performance be improved?

There appears to be a possible usage conflict on the busses between the value stack and the several E-units. Although the implicit logical addressing of elements of the value stack is clean, it requires more analysis to detect opportunities for parallel execution.

It is likely that the cache design can be made sensitive to page boundary crossovers. By this, we mean that consecutive references to the same page of level 0 will not employ the directory lookup mechanisms.

Any instance of a local copy of an item in storage means that a mechanism must be provided to invalidate the copy when the original is changed. There remains much room for invention in this field.

The number and functions of the E-units have not been established. A possible choice might be an adder, a multiplier-divider, and a string processing unit.

4.3.8 Interrupts

In any system, there occur interrupts, which might be defined as occurrences which either are asynchronous (anticipated but whose time of occurrence cannot be correlated with the processor activity), or synchronous (an unanticipated but prepared-for event which can be associated with processor activity).

In the normal case, the PPU will handle asynchronous interrupts in a manner transparent to the user. If a process must be halted, the appropriate status information will be dumped into storage, and, at the proper time, reloaded and execution resumed.

An asynchronous interrupt request is passed to the PPU's of a system on a bus which chains through all PPU's. The information on this bus consists of a priority number and a queue indicator. If the nth PPU in the chain is currently working on a task of higher priority than that of the interrupt, he passes the request to PPU n + 1. If the request is of higher priority than the current task, this PPU accepts the request and does not pass it on.

Each such request will be associated with a time interval, by the end of which service must be begun. The PPU will attempt to process to a statement boundary before the end of the specified interval. If no boundary was reached, the PPU drops the current task and takes the top unit of work on the indicated queue as its next task.

4.4.1 Introduction

In this section we discuss the structure of the Source-Sink Subsystem (SSS), its relation to other subsystems, and its functions. The SSS provides the link for communication between source-sink devices and storage. One or more Source-Sink Processing Units (SSPU) are the heart of the SSS. These processing units are similar to the PPU's in the Program Processing Subsystem, except that the SSPU's do not have floating point capability and they are specialized to handle physical Input/Output.

A set of control lines exists between the SSS and the PPS to allow for "wake up" type of signals to be passed either way when an interrupt is called for. Data Objects and Modules do not pass between the SSS and PPS.

Local and remote source-sink devices are both handled by the SSPU in such a way that the local or remote character of the device is transparent to the rest of the system. The source-sink device classes include, card readers and punches, printers, terminals, displays, other systems, special I/O equipment, and tape units and disk drives that are not included in the Storage Management Subsystem.

4.4.2 Source-Sink Subsystem Structure

The Source-Sink Subsystem structure allows communication between a source-sink device and the logical source-sink Task. The functions that must be accomplished in order to allow this communication to take place may be separated as shown in Figures 4.4.1 and 4.4.2. Figure 4.4.1 depicts the functional layers required for communication between a logical source-sink Task and a source-sink device without its own processing unit. Figure 4.4.2 depicts the functional layers required for communication between a logical source-sink Task and an intelligent source-sink device.

The logical source-sink Task is assigned as a physical task by the Logical Machine Supervisor. The Symbolic Name of the source-sink device is known in the Ownership Tree and a table of connectivity in the Task Control Block yields the named destination to the SSPU.

The Function Manager allows the application program to be independent of the source-sink device with which it is communicating. It will map a particular function (such as print) to a particular device (such as a display unit).

The Source-Sink Device Manager defines information that is unique to a particular device. The device commands and control characters are inserted in the data stream by the Device Manager. The Device Manager separates local messages from remote messages.

The Network Manager selects the communication path and controls the network interaction for the system.

The Line Control establishes the connection to a given path, notifies the Network Manager that the connection has been made, brackets the formatted data with control characters and transmits them over the line.

At the other end of the line, the Network Controller, when it recognizes its own address, strips off the line control characters and passes the message on to the source-sink device under control of the device's control unit.

If the source-sink device is another system (as in Figure 4.4.2), the message is passed to that system's SSPU and processed. This function is described in Sections 4.4.3 and 4.4.4.

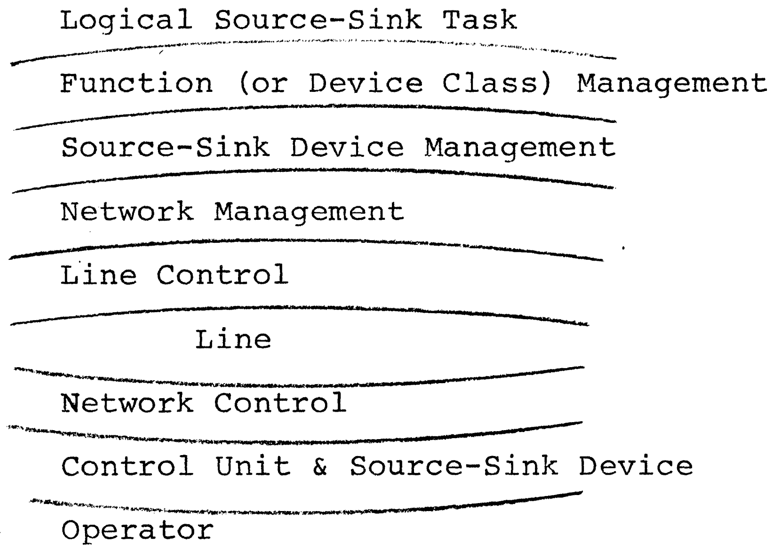


Figure 4.4.1

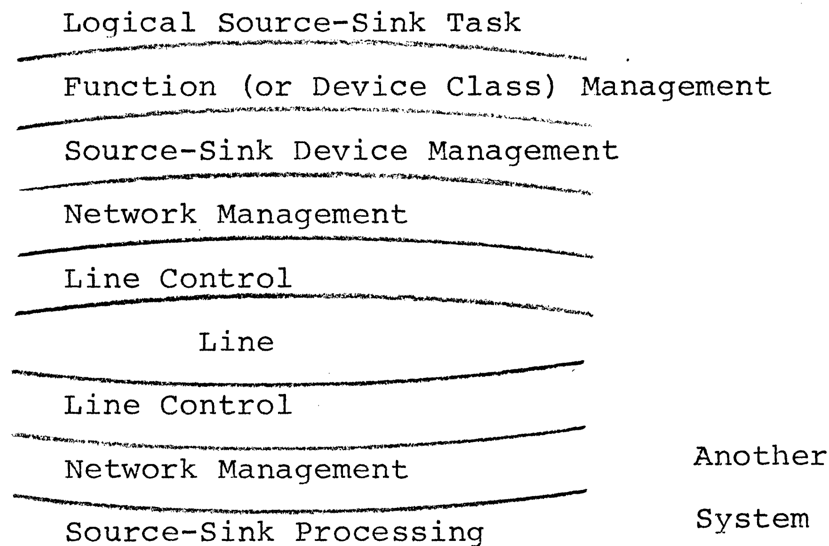


Figure 4.4.2

4.4.3 SSS Interrupts

Interrupts to the SSS may come from two sources, from the PPS or from source-sink devices. The subject of this section is the various causes of these interrupts.

The only communication that takes place between the PPS and SSS is in the form of "wake-up" signals which contain priority class, PU class, and queue designation information. These signals are passed to all PU's (SSPU's and PPU's) of a system on a line bundle which chains through all PU's. The PPS will signal the SSS whenever an output Task has been placed in the appropriate queue for the SSS.

Interrupts from source-sink devices to the SSS occur for the following reasons:

Incoming message with priority class
"Attention" signal to interrupt execution
of current task
Sign-on/Sign-off
Disconnect
Error detection on input
Error detection on output

The action of the SSS in response to these interrupts is discussed in Section 4.4.4 in conjunction with a possible implementation of the SSS.

4.4.4 SSS Functions and Implementation

The SSPU's of the SSS function in a manner very similar to the operation of the PU's of the PPS. In particular, with the exception of the floating point arithmetic capability, the description of the PU in Section 4.3 along with Figures 4.3.1 and 4.3.2 apply as well to SSPU design and operation.

The functions of the SSPU in addition to program execution required of all PU's, include line and network control, control of the "wake up" signals passing between the SSS and PPS, appropriate response to source-sink device interrupts, and maintenance of the system clock. Reference to Figure 4.4.3 will help the reader to visualize the operations taking place in the following discussion.

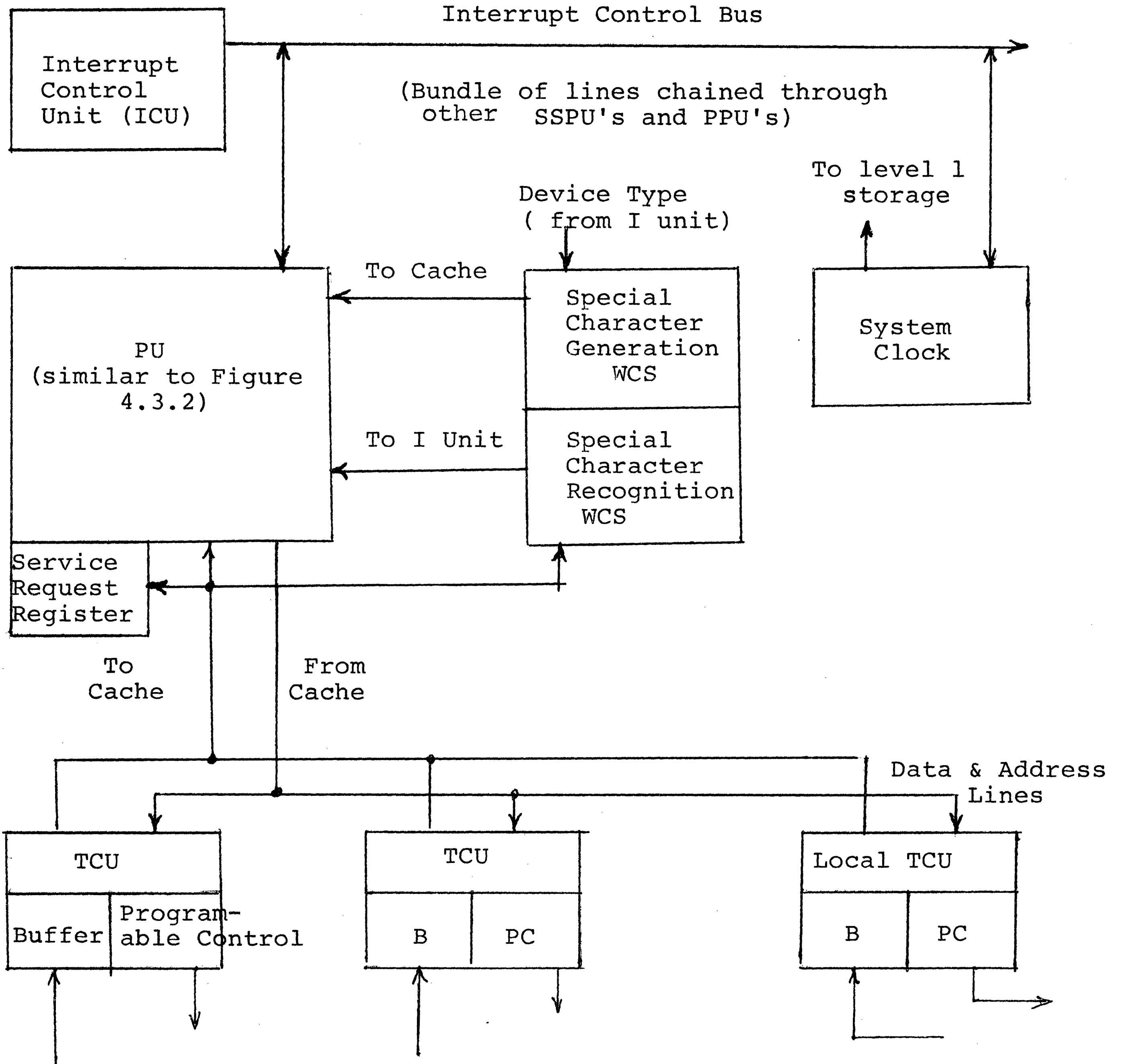


Figure 4.4.3

SSPU

4.4.4.1 Line and Network Control

The line control function is contained within a special piece of hardware called the Transmission Control Unit (TCU). Each TCU has its own buffer which is of sufficent size to contain one block of transmitted data received from each active source-sink device that it services. The TCU's may each be personalized by means of a programmable control for specific source-sink device classes. One or more TCU's exist for local source-sink devices not requiring transmission line control characters.

The TCU buffers incoming data; monitors the source-sink devices for requests for service, "message received" acknowledgements, and error responses; and makes connection to target devices for output transmission. Output data is transmitted directly through the TCU without buffering in the TCU since the data is already buffered in the CACHE. The TCU brackets outgoing message blocks with the appropriate line control characters. On input, a source-sink device requesting service simply puts its input message on the line, and the TCU buffers the input message. In order for further input transmission from that particular source-sink device to be allowed, the input message must first be analyzed under control of the Logical Machine for which the message is destined, and the appropriate response returned via the SSS.

Network control is the function of the SSPU. Output addresses are appended to output message blocks by the SSPU. The device's address is contained in the header of both input and output message blocks. When one or more input message blocks, buffered in the TCU, are waiting for service from the SSPU, a signal is sent by the TCU to set the Service Request register of the SSPU. When the SSPU completes an Output Task, it will next check to see if the Service Request register is set. If it is, it will poll the TCU's, take in the waiting input messages by buffering them in its CACHE, look in the appropriate Task queue in storage, and go to work on the Tasks to be accomplished. If Service Request is not set, the SSPU will go straight to the Task queue to look for work. If the storage Task queue is empty, the SSPU will continue to monitor the Service Request register, and poll the TCU's whenever this register is set, until the SSPU receives an interrupt from the PPS indicating that a Task has been placed in the queue. This scheme is consistent with the requirement to handle priority output messages, since a priority interrupt from the PPS can cause the SSPU to switch Tasks (e.g., to stop polling and look in the storage queue).

4.4.4.2 The Interrupt Control Unit

The Interrupt Control Unit of the SSS is a special

piece of hardware that services all PU's. It effectively controls the line bundle (Interrupt Control Bus) chained through all PU's. A PU that has put a Task in a particular queue puts on the ICB a signal containing the priority class of the Task and the PU class for the type of PU required to service the Task. The designation of the queue in which the Task may be found is implied by the PU class. These signals will chain themselves around the loop and cause the first PU in the chain (of the appropriate class), which is executing a lower priority Task (or is idle) to cleanly interrupt the lower priority Task, put that incomplete lower priority Task in a Task queue, issue an interrupt on the ICB, and switch to the higher priority Task. If an interrupt is issued for a Task which has a priority lower than any Task being executed, the interrupt will disappear at the end of the chain (the signal propagates past all PU's in a loop up to, but not including, the PU that issued the interrupt). Such a case implies that all PU's of the appropriate class were busy. When one of them completes its present Task, it will automatically look in the appropriate Task queues.

4.4.4.3 Special Character Generation and Recognition

A writeable Control Store exists as part of the SSS to provide a means of generating and recognizing special network control characters. On output the appropriate characters are added to an outgoing message under control of the executing code. The input data bus which takes data from the TCU into the CACHE is monitored by the character recognition WCS for special characters such as "Attention", "Sign-on", "Sign-off", "Device Termination", etc.

4.4.4.4 The System Clock

The System Clock resides in the SSS. Its function is to measure real time and provide the time to the rest of the system when requested. In performance of this function, the Clock may be considered as a highly specialized PU. It is connected to the Interrupt Control Bus and to Level 1 of storage. There are three kinds of Tasks the System Clock may be asked to perform by a PU. These are to give the time of day, to provide an interrupt after a specific time interval, or to provide an interrupt at a particular time of day. To help accomplish these Tasks, the System Clock has a pair of registers and utilizes storage to enqueue the required times of interrupt. When a PU needs service from the System Clock, the PU puts the Task in the appropriate queue and sends a "wake-up" signal on the Interrupt Control Bus consisting of Priority and PU Class (in this case CLOCK). The System Clock, when it processes

the Task will either put the time in the appropriate Task queue for the requesting PU and signal on the ICB, or it will calculate the time at which interrupt is desired by that Task and place it in a storage queue. The pair of registers will always contain the time of the next interrupt with its associated Task. The Clock will continually monitor the interrupt storage queue to ensure that a Clock register always contains the next timed interrupt to be given to a PU. The interrupt to the PU will be given when the time of day is coincident with the time placed in the register. The interrupt is given by putting the time of day, along with the Task for which the interrupt is intended, in the Task queue and sending Priority and PU Class on the ICB. Any PU of that class that is expecting an interrupt may look in the Task queue to determine if that interrupt applies to it.

4.4.4.5 Responses of the SSS to Interrupts from Source-Sink Devices

Incoming message interrupts have been discussed throughout the preceding paragraphs in the context of SSS Functions and Implementation. Some special handling is required, however, for certain interrupts such as Attention, Sign-on/Sign-off, Disconnect, and Error Detection.

Attention - Attention is sent as an input message from a source-sink device to abnormally terminate a Task being executed. The Task to be terminated may be an output Task on an SSPU or a problem Task on a PPU. The SSS will handle this message as an ordinary input message, place the "Attention" Task in a Task queue and place a "wake-up" signal on the Interrupt Control Bus. The "Attention" message must have a Task ID appended to it by the SSPU since it knows which source-sink device is requesting the action. Thus, the PPU that takes on the Task will know which PU to terminate. A Task is also initiated by the LMS to unlock the input device.

Sign-on/Sign-off - These messages also are handled as ordinary input messages as far as queuing and interrupt control is concerned. However, since there may be more devices on a TCU than input buffers, the TCU must decrement or increment its available buffers for each device that signs on or off so that it may refuse to sign on more than it can handle. Also,

an SSPU must process this Task to establish or drop connectivity to this device through the TCB and the Ownership Tree. Another physical Task established as a result of this message is the sending of the appropriate sign-on or sign-off output message to the device if one is required. A PPU will be assigned the Task of accounting for the total processing time and connected time used by the device.

Disconnect - A source-sink device that abnormally disconnects must indicate this to the TCU by dropping its line. The TCU automatically loads the appropriate buffer with a disconnect message containing normal header information (address of device). The SSS acts on this as with a sign-off and an Attention (of course no output message is sent), however the Logical Machine involved must be stored "as is" for future activation.

Error Detection - An interrupt, received by the SSS as the result of an error message from a source-sink device because of an incorrect output received by the device, is buffered in the usual way in the TCU. When passed to the SSPU, the Output Error character is immediately detected by the Special Character Recognition WCS, and the TCU is notified. Since the TCU receives an error message from a source-sink device in place of the expected acknowledgment, it initiates a retransmission of the erroneous last message block which has been retained in the SSPU CACHE. Output message blocks are retained in the CACHE for TCU transmission. The TCU clears the message block from the CACHE when acknowledgement of receipt is returned from the source-sink device to the TCU. A mechanism is needed to stop retransmissions and provide notification to the system operator, after a set number of retransmissions of a particular message block to a particular device.

Input messages are checked for errors by the TCU in two sections. The header containing the address is checked independently from the message body. In this way, an input error that is detected in the input message body will initiate an automatic "resend" signal from the TCU to the source-sink device that had sent that message. Obviously, an error in the input header (address) cannot initiate any system response as the sending device is unkown. Therefore unattended devices of a certain class should be buffered and should automatically resend after a suitable time out to allow for this situation. Attended devices will require operator intervention (e.g., Attention) should an input address error occur.

4.4.4.6 Other Hardware

Crosspoint switching of TCU's with SSPU's is required to provide system availability should an SSPU go down. The implementation of this feature will have an impact on performance if the switching is to be dynamic (under system control). If the TCU's (and possibly devices) are manually switched when required, implementation complexity diminishes considerably.

Disk drives and tape units that are part of the Storage Management Subsystem are not source-sink devices. It may be required at times (e.g., for use in emulation mode) to logically switch some of these devices out of the Storage Management Subsystem into the Source-Sink Subsystem as source-sink devices. This may only be done if the data on these devices are moved to another physical location in the SMS under control of the Physical Control Program, thus freeing these device for the SSS. Only devices may be switched in this way from the SMS to the SSS; data may not. Data may be transferred from the SMS to a source-sink device (or vice versa) only by going through the SSS and the associated protocols.

4.4.4.7 The "Physical" SSS

In the above discussion, various elements, such as the TCU, CLOCK, etc., were shown as independent boxes. While this physical independence may be true in a large system, it is also possible to treat these elements as logically separate but physically contained within one PU as may be the case in a small system. The ideas expressed throughout Section 4.4 should therefore be construed as an architectural definition rather than a truly physical one.

IBM CONFIDENTIAL

CHAPTER 5

MODELING

5.1 Description of Models

Two types of models are being used in the development of AFS. The first type, termed a Logical Model, is designed to simulate the logic of the system, with implementation considerations being minimized to as great a degree as possible, whereas the second type, termed a Timing Model, is designed to measure the performance of a specific implementation. Both hardware and software aspects of the architecture and design are being simulated. Execution of the Logical Model may be thought of as actually logically executing a program on a complete, but simple (sequentially processed, non-multiprogramming, non-multiprocessing) AFS machine with an infinitely large one-level storage. Output from the Logical Model includes not only the calculated answers of the source language program, but also a trace of the sequence of operations performed by the executing program and the storage addresses referenced. This trace is used as input to the Timing Model which measures the timing of a PPU and storage hierarchy complex with specific emphasis on implementation considerations. An introduction to these models is contained in ASP Memo 014.

5.1.1 The Logical Model

This section will describe the existing Logical Model (version 1), which embodies a great number of the concepts detailed in the earlier sections of this manual. A second version of the model, which will extend the current capabilities, is discussed in Section 5.4. The Logical Model, written in APL, currently accepts source language programs within a PL/I subset entered interactively. The subset selected for this initial implementation handles many of the complex addressing problems, but only provides scalar integers for computation purposes. The PL/I translator portion of the model produces a Module for each procedure entered and catalogs this Module in the system library. The Connector can then establish this procedure in the user's Logical Machine by duplicating those parts of the Module that are not read-only, and building the forward and back pointers that specify the PL/I static nesting structure. The Interpreter can then be called upon to interpretively execute the code, using actual data values such that branches and other operators behave as the programmer intended. The Interpreter will call upon the Linker to resolve the Symbolic Names within the existing environment. The Interpreter handles expression evaluation, based on data descriptors, as well as calls, returns and other scope changing operations. The Interpreter also utilizes the storage operations: READ, WRITE, CREATE, DESTROY, ADD, and DELETE. Explicit details for the Model are contained in ASP memos 046 and 047.

5.1.2 The Timing Model

The Timing Model is designed to measure the performance expected for a specific implementation of the architecture contained in this manual. Initially the model has been parameterized with timings comparable to the Model 85. It is anticipated that other similar models will be required for implementations satisfying other general price/performance markets, especially in the portions of the model which simulate the PU activity.

The Timing Model, written in APL, includes a storage hierarchy model and a very simple representation of PPU timing. The model is driven from trace output of the Logical Model. The N-level storage hierarchy portion of the model simulates reading or writing data, adding or deleting pages, and creating or destroying spaces. The simulated directories and storage devices operate asynchronously. The model includes the logic of moving space and page numbers through the directories of the hierarchy, as well as measuring the time required to do it.

Information is transmitted as words or multiples of words called pages. Word size is fixed throughout the system. Page size increases by integral factors (normally powers of 2) from level to level. A page is divided into lines, a line at one level being equal in size to a page in the preceding level. At level 0 a line is a single word. Pages are moved only between adjacent levels, local buffering being provided as necessary. Hierarchy traffic is not routed through the PPU memory. Level 0 contains a Lookaside Memory an associative search feature on Logical Names to bypass, if possible, one or more intermediate look-ups on repeated references to the same item.

The PPU is represented simply by a table of operation times comparable to the Model 85 for each operation code. Multiple PPU's, each with its own trace input, can be set up. Each PPU has its own level 0 memory. All PPU's currently use the same operation times (although this could easily be changed). Levels 1 on up are common to all PPU's. No provision has been made for logical interlocks required in multiprocessing. The model merely ensures that each gets back its own requests.

The Timing Model is documented in ASP 015 and 059 and timing assumptions are in a memo to file by W. Buchholz dated 10/21/70, "OLYMPUS Model Timing and Assumptions".

5.2 Model Usage Results

5.2.1 Logical Cases

This section contains several examples of programs that have been run on the Logical Model, and in one case, the OS 360 PL/I F equivalent. These cases have been included to demonstrate:

- The mechanisms that have been defined are capable of accommodating complex addressing problems.
- The advantage of the use of descriptors in catching errors.
- The rudiments of diagnostics

The Logical Model cases shown below begin with a "START 2" machine as it had been left at the conclusion of the previous run. The DISPLAY command prints the originally entered source text. Typing the name of a procedure causes execution of that procedure (connection into the user's workspace has been previously accomplished by a COPY command). Output is obtained by assigning to a dummy variable, ⎕. The line of output contains a descriptor and a value, where a descriptor equal to 16 denotes an integer variable, and equal to 24 denotes an integer constant.

Case 1: Computation with Described Data

Execution of this program with the Logical Model yields the expected results whereas the OS/360 PL/I F version of the program runs, but yields erroneous results, since PL/I does not have the advantage of descriptors. Later versions of PL/I may be better able to accommodate this problem.

Logical Model

```
    START 2
<TYPE COMMAND>
]DISPLAY TEST
<0>TEST:PROC
<1>[]=8
<2>CALL B(8)
<3>END
<TYPE COMMAND>
]DISPLAY B
<0>B:PROC(X)
<1>DCL Y INTEGER AUTOMATIC
<2>Y=X
<3>[]=Y
<4>END
<TYPE COMMAND>
TEST
24  8
16  8
<TYPE COMMAND>
]OFF
<OFF>
```

OS/360 PL/I F

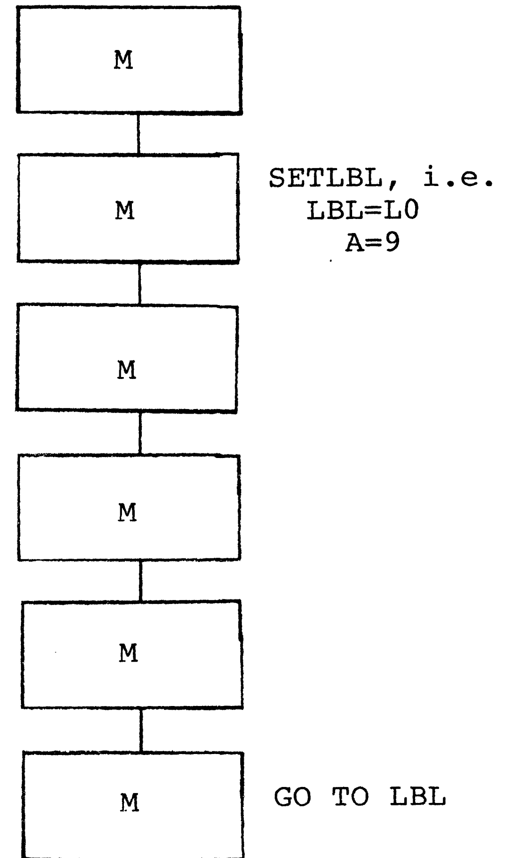
```
TEST:PROC OPTIONS(MAIN);
     CALL B ( 8);
      B:PROC(X);
        DCL (X,Y) FIXED BINARY (31,0);
        Y=X;
        PUT SKIP DATA(Y);
        RETURN;
        END B;
     END TEST;



Y=    -1945475136;
```

Case 1

Case 2: GO TO Label Variable

This program, M, calls itself recursively to create six activations. In the second activation, line 7 causes a label variable to be established in the then current environment and the current automatic variable, A, to be set to 9. Line 8 causes multiple recursions until the sixth activation. Then line 9 results in a GO TO to the environment existing at the time the label variable was established, with multiple RETURN's implied. The routine concludes by printing the value of the automatic variable, A, that in any other activation would have been equal to 1.

```
        START 2
    <TYPE COMMAND>
    ]DISPLAY M
    <0>M:PROC
    <1>DCL S STATIC INTERNAL INIT(0)
    <2>DCL A INTEGER INIT(1)
    <3>DCL LBL LABEL STATIC INTERNAL
    <4>⎕=A
    <5>S=S+1
    <6>⎕=S
    <7>IF S=2 THEN GOTO SETLBL
    <8>IF S≤5 THEN GOTO RECURS
    <9>GOTO LBL
    <10>SETLBL:⎕=555
    <11>LBL=L0
    <12>A=9
    <13>GOTO RECURS
    <14>RECURS:⎕=666
    <15>CALL M
    <16>RETURN
    <17>L0:⎕=00
    <18>⎕=A
    <19>⎕=S
    <20>RETURN
    <21>END
    <TYPE COMMAND>
    M
    16   1
S-  16   1
    24   666
    16   1
S-  16   2
    24   555
    24   666
    16   1
S-  16   3
    24   666
    16   1
S-  16   4
    24   666
    16   1
S-  16   5
    24   666
    16   1
S-  16   6
    24   0
A-  16   9
S-  16   6
    <TYPE COMMAND>
    ]OFF
    <OFF>
```

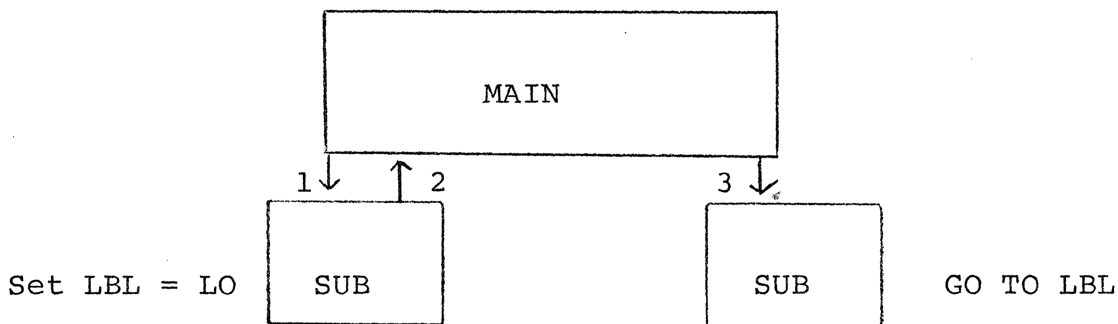


Case 2

IBM CONFIDENTIAL

Case 3: GO TO Label Variable - Erroneous Case

In this case, an erroneous GO TO is caught by the Logical Model, with a diagnostic. The activation chain as execution proceeds is shown below (the numbers denote the sequence of calls and returns). The first activation of SUB sets a label variable to a label constant in the then current environment. After returning to MAIN, a second activation of SUB attempts to branch to that label variable, but the environment, consisting of the first activation of MAIN and the first activation of SUB, no longer exists, resulting in a user diagnostic. Diagnostics in the Logical Model currently utilize a syntactic APL error to stop execution. Although not printed for the user in this version of the model, complete information exists which can advise the user of the current line number, environment, etc.

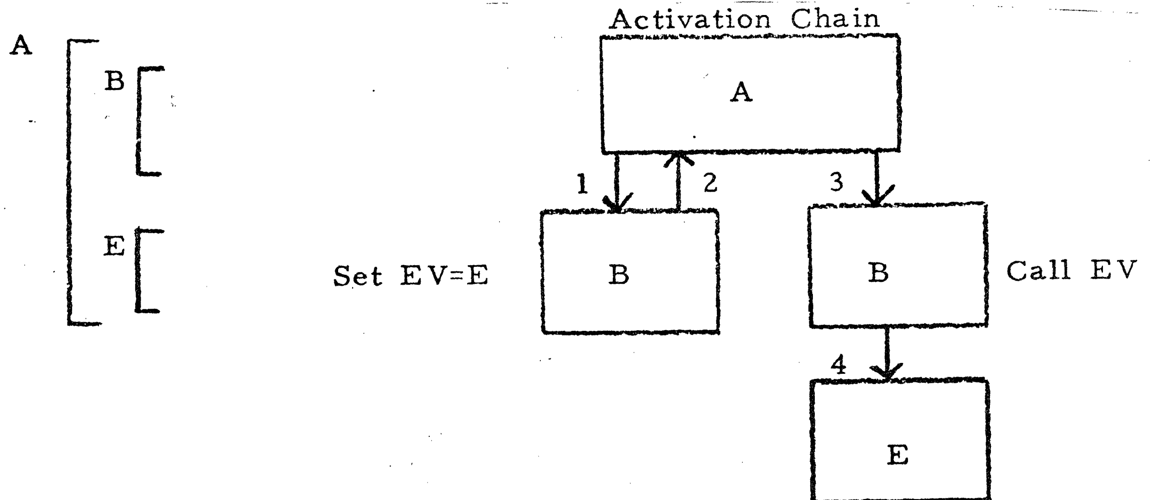


```
          START 2
     <TYPE COMMAND>
     ]DISPLAY MAIN
     <0>MAIN:PROC
     <1>⎕=1111
     <2>CALL SUB
     <3>⎕=1112
     <4>CALL SUB
     <5>⎕=1113
     <6>END
     <TYPE COMMAND>
     ]DISPLAY SUB
     <0>SUB:PROC
     <1>DCL A INTEGER AUTOMATIC INTERNAL INIT(1)
     <2>DCL S INTEGER STATIC INTERNAL INIT(0)
     <3>DCL LBL LABEL VARIABLE STATIC INTERNAL
     <4>⎕=2111
     <5>⎕=S
     <6>S=S+1
     <7>IF S=1 THEN GOTO SETLBL
     <8>⎕=2112
     <9>GOTO LBL
     <10>SETLBL:⎕=2113
     <11>LBL=L0
     <12>A=9
     <13>RETURN
     <14>L0:⎕=2114
     <15>⎕=S
     <16>⎕=A
     <17>RETURN
     <18>END
     <TYPE COMMAND>
     MAIN
     24   1111
     24   2111
     16   0
SUB<10> -  24   2113
     24   1112
     24   2111
     16   1
SUB <8> -  24   2112
     SYNTAX ERROR
     EXEC[111] <INVALID BRANCH TO LABEL VARIABLE ATTEMPTED>
                                                          ^
```

Case 3

Case 4: Entry Variable

This case portrays the Logical Model properly executing a program using an entry variable defined in the PL/I language specifications. The PL/I F compiler does not currently support this capability. The three procedures A, B, and E are displayed separately. The static nesting, developed by COPY commands, is shown below, along with the activation chain as execution proceeds (the numbers denote the sequence of calls and returns).



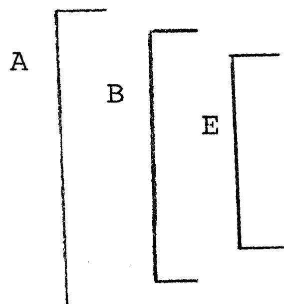
The first call of B sets the entry variable, EV, to E, in the current environment. Then B returns, and is called again by A, at which time B calls EV. In this case, the environment that existed when EV was set (the first activation of A is still active and the first activation of B is irrelevant since E is at the same lexical level as E) still exists, and the program runs properly to completion.

```
         START 2
    <TYPE COMMAND>
    ]DISPLAY A
    <0>A:PROC
    <1>DCL X AUTO INIT(1)
    <2>DCL S STATIC INIT(0)
    <3>DCL EV ENTRY VARIABLE STATIC
    <4>[]=1111
    <5>CALL B
    <6>[]=1112
    <7>CALL B
    <8>[]=1113
    <9>END
    <TYPE COMMAND>
    ]DISPLAY B
    <0>B:PROC
    <1>[]=2111
    <2>S=S+1
    <3>IF S=2 THEN GOTO CALLEV
    <4>[]=2112
    <5>X=9
    <6>EV=E
    <7>RETURN
    <8>CALLEV:[]=2113
    <9>CALL EV
    <10>[]=2114
    <11>END
    <TYPE COMMAND>
    ]DISPLAY E
    <0>E:PROC
    <1>[]=3111
    <2>[]=S
    <3>[]=X
    <4>END
    <TYPE COMMAND>
    ]COPY E IN A
    <COPY COMPLETED>
    <TYPE COMMAND>
    A
    24  1111
    24  2111
B<4> -  24  2112
    24  1112
    24  2111
B<8> -  24  2113
E<1> -  24  3111
    16  2
    16  9
    24  2114
    24  1113
    <TYPE COMMAND>
    ]OFF
    <OFF>
```

Case 4

Case 5: Entry Variable - Erroneous Case Caught

This case is identical to Case 4, except that the static nesting is changed to that shown below.



In this case, the environment that exists when EV is set, as far as E is concerned, consists of the first activation of A and the first activation of B. However, the first activation of B is destroyed immediately after B sets EV. Thus, when B calls EV in B's second activation, the original environment corresponding to that set in EV no longer exists, resulting in an error.

```
      &lt;TYPE COMMAND&gt;
      A
      24  1111
      24  2111
B 4 - 24  2112
      24  1112
      24  2111
B 8 - 24  2113
      SYNTAX ERROR
      EXEC[40] <ENTRY VARIABLE ENVIRONMENT NO LONGER EXISTS>
                                                          ^
```

Case 5

5.2.2 Performance Cases

Comparative runs have been made on the Timing Model and in PL/I on the Model 85 using two methods of computing factorial 4. Only the execution portion of the programs were measured. Translation, Connection, and Linking Times were excluded. Level 0 of the storage hierarchy was made large enough to hold all the pages required which corresponds to the program being contained entirely in the Model 85 buffer memory.

Case 1: Loop Method

The following loop was executed (in PL/I notation, with compiled 360 ops):

```
LOOP:  X = X * J;                    | L, L, M, SLDA, ST
       I = I + 1;                    | L, A, ST
       IF M >= I THEN GO TO LOOP;    | L, L, C, BC
```

For the Timing Model the loop was traversed 3 times. For the Model 85 the loop was traversed several hundred thousand times to get a measurable interval, but the result was normalized to 3 times. (The variable, J, was needed to permit a large number of repetitions on the Model 85. It was initialized to one.)

Two means of comparison are possible. First, we may consider the storage references made by each machine. Since the Timing Model for this AFS machine provides for the Lookaside Memory which reduces the number of storage references due to accessing data indirectly, three figures are shown.

	Instructions	Data	(In Words/Loop)	
Model 85	12	10		
AFS Machine	7	24 If no Lookaside	19 1st Loop with Lookaside	12 Other loops with Lookaside

Thus, based on storage references, the Model 85 requires 22 references per loop, and the AFS machine requires 19 (to 26 maximum) references per loop. The ratio is approximately 1:1.

Second, we may consider the times measured for each machine. The Model 85, normalized to 3 iterations through the loop, took 9.5 microseconds and the AFS machine took 6.1 microseconds. The fact that this time ratio is 9:6, rather than 1:1, is due to the

current level of detail in the Timing Model, especially with regard to storage reference and PPU times.

Case 2: Recursive Method

The recursive factorial function, FACR, shown below, was called 4 times. On the Model 85 this was repeated several hundred times.

```
<0>FACR:PROC(N,X)
<1>X=1
<2>IF N>1 THEN GOTO RECURS
<3>RETURN
<4>RECURS:CALL FACR(N-1,X)
<5>X=N*X
<6>RETURN
<7>END
```

The times measured for this case, normalized to one set of 4 calls, are:

Model 85	215 microseconds
AFS Machine	20.8 microseconds

The ratio of 10:1 for this case should be strongly tempered by the current level of detail in the Timing Model. But this is illustrative of the type of advantage that results from managing storing with commands such as CREATE, rather than GETMAIN as in System/360.

5.3 An Instruction-Level Machine Compared with a Higher-Level Language Machine

Concurrent with the Logical and Timing Model development, another pair of models have been implemented which yield supportive evidence to the performance potential of a higher level language machine. These models, developed in Palo Alto, have been used as a means of understanding techniques in the APL Machine, APLM, described by Phillip Abrams in his recent thesis* and of obtaining some crude estimates of the machine's performance vis-a-vis a present day von-Neuman machine. The APLM incorporates two fundamental new processes which Abrams has termed "drag-along" and "beating", where drag-along is defined as the process of deferring evaluation of operands and operators as long as possible, and beating is defined as the machine equivalent of calculating standard forms of selection expressions.

---

*Abrams, P.S. (1970) An APL Machine SLAC Report No. 114

The von-Neuman machine used for comparison is based on the MIX computer designed by Knuth and rather widely used in computer science courses at Stanford and elsewhere. It is intrinsically a simple, basic machine with an A-register, Q-register, and six index registers working with a word-oriented memory. The instruction repertoire somewhat resembles that of the IBM 7094. The APLM uses an instruction buffer for subscript offset calculations.

The CPU capabilities of the MIX machine have been augumented to have an instruction power comparable to that of the APLM. Thus, the performance of MIX versus APLM becomes a measure of the number of storage accesses made into the various types of storage media plus an estimate of the number of cycle required for each instruction over and above storage cycles.

Example 6 in Abrams thesis has been coded and run on both the APLM and MIX models. The APL statement is:

```
E[I;]←EP>|¯1+(+/(1 2 2⍉PT∘.-PT[I;])*2)*0.5
```

The statement has been hand coded for the MIX machine in two ways. It has been coded in a highly optimized manner, including unrolling of the loops, even though a compiler with such an ability does not exist. It has also been coded in the manner of a very good optimizing compiler. The number of references measured for this example are:

	Instructions	Data	Instruc. Buffer	Total
APLM	131	33	66	230
MIX (Highly optimized)	87	60		147
(Good optimizing)	140	74		214

Since the APLM is reasonably complete in accounting for CPU actions, and since the first set of values for the MIX machine represent extremely efficient coding, the ratio of 2:3 between MIX and APLM may be taken as an indication of a worst case bound. The ratio of 1:1 between the second set of values for MIX and APLM are in keeping with the results determined by the Timing Model.

5.4 Model Plans

Second versions of the Logical and Timing Models are being planned which will extend the current set of capabilities. Enhancements include:

- A more faithful representation of the PU, including tracing and timing of the stack manipulations required for the call and return mechanism and for expression evaluation.
- Byte addressing (rather than the current word addressing)
- A translator for an APL subset
- An edit/change/continue capability
- Implementation of the LDT as defined in this manual
- Implementation of vectors and fixed and floating point data types

IBM CONFIDENTIAL

Chapter 6

GLOSSARY

Numbers refer to sections of this document. Cross references to AFS Fundamental Concepts and System Language (SLM) are parenthesized.

Term	Definition
Activation Tree (2.3)	A structure in a Logical Machine containing information about which Modules of that Logical Machine are currently active, and the order in which they called or attached each other. (Activation Tree in SLM)
ADD (4.2)	A storage operation defined in the text.
Allocation (2)	The process of creating a Reference Table together with its contained list of DAPOVs. (Insert and Delete in SLM)
Attach (2.3)	The invocation of a Module to be executed as a separate Logical Task. (Create and Parallel in SLM)
BRING	A storage operation defined in the text.
Connect (2.2)	A command used to introduce a Module into a Logical Machine by the creation of a new node in the Program Tree.
CREATE (4.2)	A storage operation defined in the text.
Descriptor and pointer or value (DAPOV) (2.5.1)	A Descriptor together with its associated data or a system pointer which provides access to the data.
Data Object (2,5)	The set of a name, its descriptors, and value. (Object in SLM)
DELETE (4.2)	A storage operation defined in the text.
Descriptor (2.5.1)	Information specifying the type, aggregation, and/or representation of data.
DESTROY (4.2)	A storage operation defined in the text.

IBM CONFIDENTIAL

Dynamic Storage Mechansim (2.4)	The collection of Storage Anchors and Reference Tables used to provide storage and execution time addressability for Data Objects.
External Node (2.2)	A special node in the Program Tree which contains the Local Symbol, Link and Declare Tables for all the external names of the program.
Generic Descriptor (2.5.1)	A descriptor associated with a name in a Local Declare Table.
Interpreter (2.6)	The logical executor of code.
Job (3.1)	Work performed during the time between activation and deactivation of a Logical Machine.
Link (2.5)	The process of resolving Symbolic Names by searching the Local Symbol Tables contained in the nodes of the Program Tree.
Local Declare Table (LDT) (2.5.2.3)	A table containing all the information that is known about each Symbolic Name declared or referenced in the Module. The Generic Descriptor is part of this information.
Local Link Table (LLT) (2.5.2.1)	A table containing execution time connectivity to the DAPOV for each Symbolic Name declared or referenced in the Module.
Local Symbol Table (LST) (2.5.2.1)	A table containing all the Symbolic Names declared or referenced in the Module.
Logical Input/Output System (3.3)	The facilities for the transfer of information to and from the Logical Machine.
Logical Machine (2.1)	That part of the system provided for the processing of each independent unit of work.

Logical Machine Supervisor (3.1)	A Logical Machine which is in control of all other Logical Machines in the system, and provides an interface to the physical processors on behalf of these Logical Machines.
Logical Name (2.5.2)	The internal form of a Symbolic Name.
Lookaside Memory (4.2 & 4.3.6)	A local name associative array that permits look-aside to minimize storage accesses on repeated references to the same information.
LREAD	A storage operation defined in the text.
Module (3.2)	The combination of the source code, executable code, line directory, LST, LLT, and LDT for a sequence of source statements in which all uses of the same Symbolic Name refer to the same object.
Offset (4.2)	An index to a particular byte of a space.
Ownership Tree (3.2)	A structure defining the ownership relationship between all objects of the system, and containing information about the access rights of each object.
Pointer	A generic term for a type of data whose value is the logical address of another data object, and for a System pointer.
System Pointer	A System Pointer is a type of system data whose value is the physical address of another space or of a byte within a space.
Processing Unit (PU) (4.3 & 4.4)	A generic term for, a Program Processing Unit or a Source-Sink Processing Unit. Also used to represent the common functional part of thest two units.

Program Processing Subsystem (PPS) (4.3)	A physical subsystem (comprised of one or more Program Processing Units) which processes all Physical Tasks not requiring Source-Sink I/O.
Program Processing Unit (PPU) (4.3)	A physical unit in the Program Processing Subsystem.
Program Tree (2.2)	A tree structure in a Logical Machine which defines the static nesting of Modules in the LM and is used to determine the static scope of name resolution. (Static Environment Tree in SLM)
READ (4.2)	A storage operation defined in the text.
Reference Table (2.5)	Storage for a list of one or more DAPOVs together with a back pointer to previous generations of this Reference Table.
Relative ID (2.5.3)	The second field in a Local Link Table which identifies the appropriate DAPOV in a Reference Table.
Semaphore (4.1.4)	A special integer variable used by Control to synchronise tasks and provide access to serially re-usable resources.
Source-Sink Processing Unit (SSPU) (4.4.4)	A physical unit in the Source-Sink Subsystem.
Source-Sink Subsystem (SSS) (4.4.1)	A physical subsystem (comprised of one or more Source-Sink Processing Units) which processes all physical I/O Tasks.
Space (4.2)	An independent portion of the storage capable of linear extension and contraction. Referenced in the Logical System by <u>Space Name</u> and the Physical System <u>Space Number</u>.
Storage Anchor (SA) (2.4)	Storage Anchors are a component of the Dynamic Storage Mechanism. System Storage Anchors address the Reference Tables directly associated with the Program Tree. User Storage Anchors address the DAPOVs for variables under user control. Storage Anchor Name and Storage Anchor Register are the logical and physical entities, respectively.

Section 6.0 GLOSSARY

Storage Management Subsystem (SMS) (4.2)	That portion of the physical system that contains addressable storage and the controls to allocate storage spaces, determine the physical location of stored information, and provide access to that information. The SMS communicates with the PPS and the SSS.
Symbolic Name (2.5.2)	The external, character string, form of the identifier of a Data Object or Logical Object. (Symbol in SLM)
System Language (SL) (1.1)	The logical description of the system contained in the SLM. also, loosely, the logical form of the execution Language in the Logical System.
System Node (2.2)	The root node of the Program Tree which contains the names of, and connectivity to the system functions which are available to the Logical Machine.
Task (3.1)	Each independent parallel activity within a logical machine is a logical task. The first task started is the master task. The others are subtasks. A physical task is the unit of work dispatched to a PPU by the Logical Machine Sypervisor through the Physical Control System. (Process in SLM)
Task Control Block (3.1)	A Logical Task Control Block contains the information which binds an Interpreter to the other mechanisms in the Logical Machine in which it is active. A Physical Task Control Block defines the status of the physical task as it is processed by a Processing Unit or is queued in Control.
Translator (3.2)	Takes user written code and builds a Module.

UNLK (4.2)	A storage operation defined in the text.
Value Descriptor (2.5.1)	The descriptor associated with the current value of a Data Object.
WRITE (4.2)	A storage operation defined in the text.