

CONTROL DATA®

Advanced Design Laboratory

STAR

SOFTWARE SYSTEM

CONTROL DATA[®]

Advanced Design Laboratory

STAR

SOFTWARE SYSTEM

CONTROL DATA

CORPORATION

REFERENCE MANUAL

RECORD OF REVISIONS	
Revision	Notes
A	Released January 18, 1971
B	Released March 24, 1972. Complete revision. Change Order No. 19

Address comments concerning
this manual to:

STAR Software System Reference Manual
 Publication Number 59156400
 Copyright © Control Data Corp., 1972
 Printed in the United States of America

Control Data Corporation
 Advanced Systems Laboratory
 Documentation Group
 4201 North Lexington Avenue
 St. Paul, Minnesota 55112

PREFACE

The STAR computer system was developed at the Advanced Design Laboratory of Control Data Corporation. The system design, an evolution from the CONTROL DATA® 6000 and 7600 computer systems, is intended to provide solutions for the diverse computing and data processing requirements of the next decade.

The STAR central computer is a high performance general purpose processor that includes the well established features of conventional computers. It also has many original features which make new computational methods available. The most important of these is vector processing where the machine operates on pairs of consecutive elements from specified lists. When used in this way, for example, the most powerful STAR central processor (STAR-100) can produce floating-point multiply results at the rate of one hundred million per second.

Input and output of data in the STAR system is managed by completely separate specialized processors. Peripheral devices, grouped by type, are associated with controlling processors and data buffering storage to form stations. These self-operating stations allow extremely flexible configurations of peripherals in the STAR computer system.

The development of software to fully use STAR has led to some new concepts of data processing. This manual attempts to communicate these concepts by describing the software currently implemented or under development at the Advanced Design Laboratory.

The following Control Data documents contain further information about the STAR computing system:

<u>Title</u>	<u>Publication Number</u>
STAR-100 Hardware Reference Manual	60256000
STAR-1B Hardware Reference Manual	60326501
STAR Peripheral Stations Preliminary Reference Manual	59156100
PL/STAR Compiler/Assembler Preliminary Reference Manual (Published as Appendix J to this manual)	60324800

'And having thus endeavoured to discharge our duties in this weighty affair...and to approve our sincerity therein (so far as lay in us) to the consciences of all men; although we know it impossible (in such variety of apprehensions, humours and interests, as are in the world) to please all; nor can expect that men of factious, peevish, and perverse spirits should be satisfied with anything that can be done in this kind by any other than themselves: Yet we have good hope, that what is here presented, and hath been...with great diligence examined and approved, will be also well accepted and approved by all sober, peaceable and truly conscientious...sons.'

Book of Common Prayer

1662 Preface

CONTENTS

<u>Section</u>	<u>Title</u>	<u>Page</u>
1	INTRODUCTION	1-1
	Distributed System	1-1
	Virtual Memory	1-4
	String-Array Processing	1-4
2	CENTRAL OPERATING SYSTEM	2-1
	System Structure	2-1
	Virtual Memory Layout	2-5
	Job Sequencing	2-7
	Job Activation	2-14
	System Tasks	2-15
	External Channel Control	2-17
	Dynamic Storage Allocation	2-19
	System Call Processing	2-23
	Error Detection and Processing	2-27
	Summary of Central Monitor	2-31
3	PERIPHERAL OPERATING SYSTEM	3-1
	STAR Stations and System Functions	3-2
	Messages	3-3
	System Structure	3-4
	Procedures	3-5
	System Loaders	3-12
	Buffer Controller Memory Layout	3-14
	Software Development	3-15
	Maintenance Information System – MIS	3-15
	Customer Engineering Manipulative Language – AID	3-15
4	FILE SYSTEM	4-1
	Descriptor	4-1
	Layout of Descriptor File on the 841 Exchangeable Disk Pack	4-3
	Header	4-4

<u>Section</u>	<u>Title</u>	<u>Page</u>
	Characteristics	4-4
	Name	4-7
	Storage Map	4-7
	Access List	4-9
	Messages	4-10
	Future Features	4-12
5	USER RECORD MANAGEMENT	5-1
	Accessing Files by Mapping	5-1
	Record Access	5-1
	Record Map File	5-2
	File Record Management Table	5-3
	Organization of a User's File Tables	5-7
	Input/Output Flow	5-9
	Input/Output Functions	5-10
6	LANGUAGE SYSTEM	6-1
	PL/*	6-2
	FORTRAN	6-4
	Current Development	6-19
	ADL FORTRAN Syntax Language	6-20
	The Language	6-21
	Precedence Numbers	6-21
	Type Definition	6-21
	Operand Type Definitions	6-22
	Code Skeletons	6-22
	FORTRAN Extensions	6-24
	Multiple Valued (Subscripted) Variables	6-24
	Conditionally Selected Subarray References	6-26
	New Operators	6-30
	Parameter Statement	6-30
	Use of Subarray References in Data Statements	6-31
	Procedure Identification	6-32
	Intrinsic Statement	6-34
	Dynamic Space Management	6-35

<u>Section</u>	<u>Title</u>	<u>Page</u>
7	STRUCTURE OF PROGRAMS	7-1
	Structure	7-1
	Regblocks	7-3
	Static Storage	7-4
	Dynamic Storage	7-4
	Pointers	7-6
	Register File Conventions	7-9
	Machine Registers	7-9
	Temporary Registers	7-9
	Mixed-Use Registers	7-9
	Environment Registers	7-10
	Working Registers	7-12
	Parameter Registers	7-13
	Relocation	7-13
	Module Tables	7-14
	Module Header Table	7-16
	Code Block Table	7-20
	External/Entry Table	7-20
	Code Relocation Table	7-23
	Interpretive Data Initialization Table	7-24
	Executable Data Initialization Table	7-29
	External Data Initialization Table	7-29
	Interpretive Relocation Table	7-30
	Executable Relocation Table	7-32
	External Relocation Table	7-33
	Job Control	7-33
A	LIBRARY PROGRAMS	A-1
B	CARD FORMATS	B-1
C	SYSTEM COMMUNICATION MECHANISM	C-1
D	SYSTEM MESSAGES	D-1
E	STATION OVERLAY STRUCTURE	E-1
F	STATION MAINTENANCE INFORMATION SYSTEM AND AID	F-1
G	JOB CONTROL LANGUAGE – JCL1	G-1
H	EDIT	H-1
I	BUFFALO	I-1
J	PL/*	J-1

FIGURES

Figure	Title	Page
1-1	STAR System Showing Component Connections	1-3
2-1	Peripheral Station Network	2-3
2-2	Allocation of Virtual Memory	2-6
2-3	Control Point Layout	2-9
2-4	Periodic Table Entry	2-13
2-5	Format of an Entry in the User Directory	2-15
2-6	System Table Directory	2-17
2-7	Message Boat Format	2-18
2-8	Access Interrupt Processing	2-20
2-9	Key Allocation	2-21
2-10	System Library Table and Active File Table Entries	2-22
2-11	File Input/Output Message Formats	2-26
2-12	Overview of Monitor	2-32
3-1	Example of a Station Subroutine Specification	3-10
3-2	Layout of a Typical Control Package	3-11
3-3	Flow Diagram for Read Page Task Program (function code 200)	3-13
3-4	Typical Layout of Buffer Controller Core Storage	3-14
4-1	Component Parts of a File Descriptor	4-2
4-2	Format of Descriptor Header in 16-bit Words	4-4
4-3	Format of Characteristics Section of Descriptor	4-4
4-4	Format of Name Field of File Descriptor in 8-bit Bytes	4-7
4-5	Format of Storage Map Section of the File Descriptor in 16-bit Words	4-8
4-6	Layout of Access List Section of File Descriptor in 8-bit Bytes	4-9
4-7	Format of the Active File Table	4-11
4-8	Format of Activity Record Section of the File Descriptor, in 32-bit Words	4-14
5-1	Layout of the File Record Management Table	5-4
5-2	Pointers and Tables for Locating File Record Management Tables	5-8
6-1	Card Layouts	6-5
6-2	Calculating EOL Positions	6-6

<u>Figure</u>	<u>Title</u>	<u>Page</u>
6-3	Establishing Comment and Continuation Lines	6-7
6-4	Expansion of Control Vectors for Comment and Continuation Lines	6-8
6-5	Removal of Comments, Editing of Continuation Lines, and Extraction of Alphanumeric Data	6-10
6-6	Extraction of Operators and Punctuation Symbols	6-11
6-7	Syntactical and Semantic Analysis	6-14
6-8	Example of Operator Precedence Modification	6-17
7-1	Layout of a Library Program	7-2
7-2	Program Linkages and Data Access Paths in a Shared Library Environment	7-3
7-3	Outline Structure of User's Virtual Space	7-6
7-4	Static Space Pointers	7-7
7-5	Register File Assignments	7-8
7-6	PL/* Object Module Format	7-18
7-7	Program Assembled Under PL/*	7-19
7-8	Job Control Flow Diagram	7-34
D-1	System Messages	D-1

TABLES

<u>Table</u>	<u>Title</u>	<u>Page</u>
2-1	Control Point Fields	2-10
2-2	System Tasks	2-16
2-3	Acceptable System Call Messages	2-24
2-4	Call Message Formats	2-25
2-5	File Input/Output System Messages	2-27
2-6	System Error Codes	2-28

The STAR System contains a number of significantly new concepts in its architecture. The most important of these are the logical and physical distribution of operating system functions, the virtual storage system, and the string-array processing capability. Other important features include the large register file, the powerful instruction repertoire, the bit addressing structure, the large storage bandwidth, and the high input/output channel capacity.

All of these features affect the software of the system, but the first three mentioned are of fundamental importance. The operating system, languages, and application packages, and even user programs are, or should be, regarded in a different light because of the architecture of the system.

DISTRIBUTED SYSTEM

The STAR information processing system is a distributed system in that many of the different functions of data processing have been separated from one another. Each is then treated in its own right in what seems to be an optimal way.

The STAR computer itself is a central processor which knows nothing about the outside world other than that it has input/output channels to it. The file system consists of one or more input/output units called stations having channel connections to the STAR central processor. Similarly, slow input/output, interactive terminals, magnetic tape, and so on are organized into stations. A station consists primarily of a small processor specially designed for data handling capability rather than for data processing capability. Each has its own storage system and channels and, of course, the particular set of peripherals which give it its name. One important station, called a service station, consists essentially of a large storage buffer unit with its controlling processor. The buffer unit has channels to other stations, to peripherals and to the STAR central machine. The service station provides the necessary fan-out from the STAR processor to a host of peripheral devices.

The operating system is distributed in a manner which closely follows the distribution of the hardware. Thus, there are operating functions in each station as well as within the central processor. The connecting links between the distributed components of the operating system are controlled by a set of system messages, so message handling becomes of great importance in a distributed system.

The choice of where each operating function should be located is often self-evident, although a few functions are assumed to be movable from one element to another. Any final decision regarding function locations may depend on experience with particular work loads. In general, each operating function is located closest to the resource being used, providing modularity of both hardware and software and such advantages as:

- independence from other units, particularly in the areas of nonpropagation of errors throughout the system and more immediate action on fault conditions
- capability to be independently maintained
- easier replacement of future new hardware or software parts
- easier addition of new types of stations

Figure 1-1 illustrates the layout of a large STAR system, showing the connections between the various functional units.

A STAR central processor with its immediate storage is simply another station within the system — a data processing station — and in no way does it have any extra authority. It does, however, have two stations fairly intimately connected, the paging station and the maintenance/monitoring station. The paging station, under control of the hardware virtual page mechanism and the operating system, provides temporary storage for programs exceeding the available core space. The maintenance station, besides its functions of off-line and on-line fault diagnosis/repair and preventive checking, has the capacity to collect detailed information about STAR's performance.

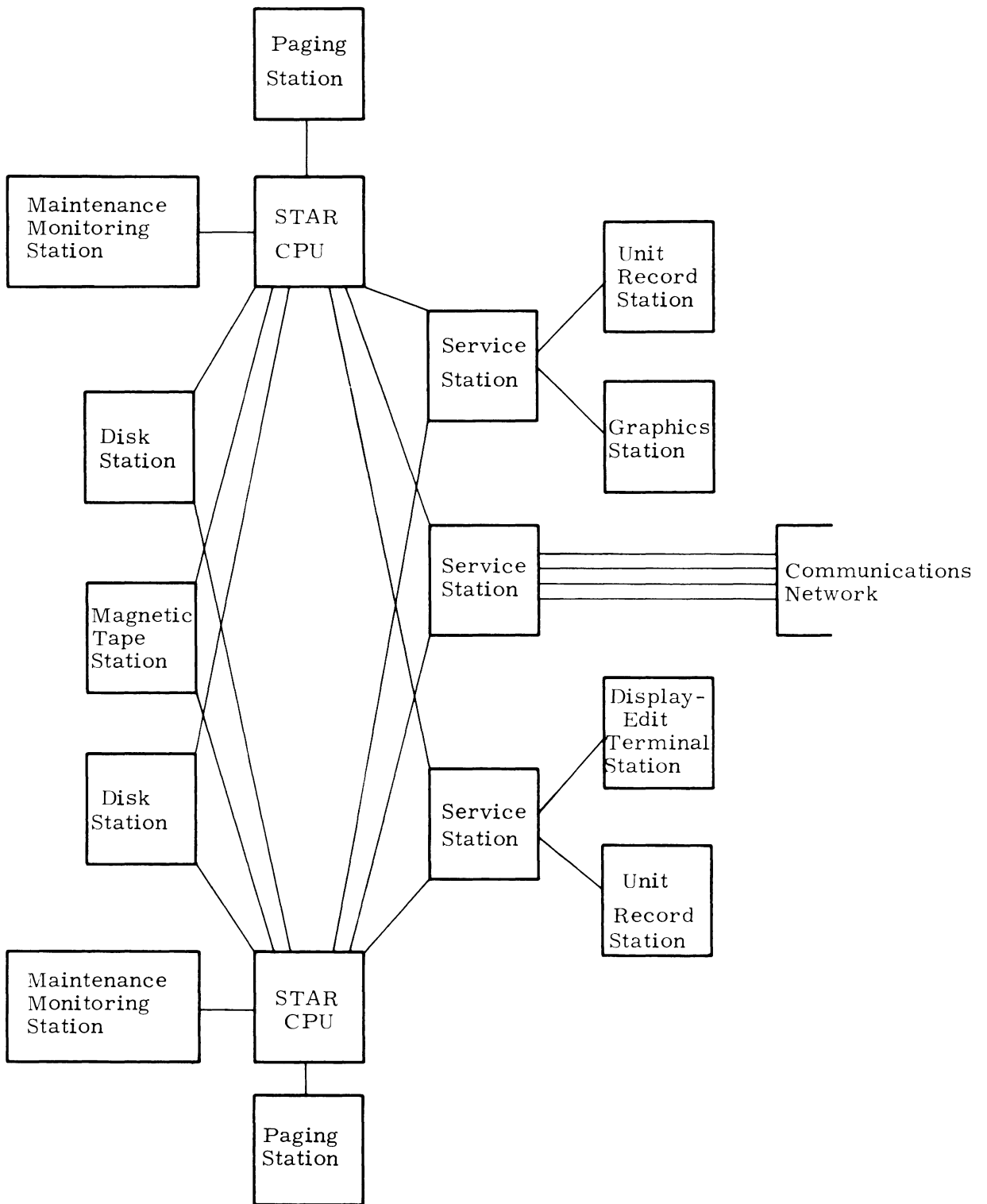


Figure 1-1. STAR System Showing Component Connections

VIRTUAL MEMORY

A very important hardware function in the STAR CPU is the virtual memory mechanism which provides a way of handling a potentially unlimited number of levels of storage media as if they were all one level. The mechanism, handling information in units of 32K-bit pages, ensures that the most frequently accessed pages exist in core storage. Unused pages drift out to slower backing media as necessary.

Each user can address his virtual space with bit addresses in the range 0 to $2^{48}-1$; of these bits, 33 are for the virtual page address. This uncommonly large amount of virtual space available to each user may significantly affect his style of programming. Because unused virtual space imposes no burden on the system, he may organize his program addresses in almost any manner which suits his convenience.

The virtual address mechanism maps the user's virtual address into an absolute physical address for central core storage in a hardware translator called a page table. Each entry in the page table contains the virtual page address and the corresponding absolute storage location together with an access mode lock and other control information. A successful association between a virtual address and an entry in the page table causes that entry to go to the head of the table, those in between being all moved down by one place. In STAR-100 the first sixteen entries in the table are kept in high-speed registers; these are examined in parallel with a simultaneous associative compare. If this compare is unsuccessful, a sequential search is made through the remainder of the table which is held in core storage. Hence, the addresses of infrequently used pages automatically float to the end of the table.

If an address has no entry in the page table, various hardware sequences are initiated, and the program requesting this address is interrupted. Monitor then normally provides the space which was addressed by transferring the desired block to central core storage from a special back-up storage station called the paging station. The program may then be restarted to continue processing from the point at which it was interrupted.

STRING-ARRAY PROCESSING

The STAR central processor includes several classes of instructions which can be used for either conventional computing or STring-ARray processing. Conventional scientific and business data processing is performed by major high-performance facilities. These facilities operate upon floating point operands, which may be 64

or 32 bits in size, and upon single bytes and bits. Some of the floating point instructions are of the register-to-register type, while the operations on single bits or bytes are of the storage-to-storage type.

STAR is also a vector processor. A vector in STAR is defined as a contiguous set of bits, bytes, half words, or full words in virtual memory. The definition depends upon how the contents of the virtual space may be treated by the vector processing instructions; not upon the nature of the content of this virtual space.

Pipeline units are provided which operate on such strings of operands; that is, on 64 or 32-bit arrays, byte strings, and bit strings. Information to specify the addresses of source and destination streams is usually held in the register file. The core storage system is designed in such a manner that two source operand streams and one destination stream can be simultaneously handled at logic speeds.

Many user functions are provided by the string and array mechanism to perform more complex operations on streamed data. Such functions amount to hardware macros and include, for example, polynomial evaluation, byte editing, scalar product of two vectors, sorting by merging byte string records, and vector arithmetic on sparse vectors.

The string instructions can operate on up to 65,536 operands in one pass. Although the function is executed serially in a pipeline, it is conceptually useful to think of it as being carried out in parallel on the data set. Thus, one can imagine that the user has between one and 65,536 parallel processors at his disposal, depending upon how he elects to achieve the processing through his selection of code. The processing facilities provided allow for efficient computing in the conventional sense and also make available a fundamentally different approach to programming via the string and array processing. To exploit efficiently the hardware provided means to "think parallel". Although conventional languages will be much used at first, it is inevitable that languages which allow the expression of parallelism and operations on structured data — without specifying in detail how the operations are to be done — will supersede the present ones.

Here, then, is a description of the software system which has evolved (or is evolving) from experimenting with the novel hardware features designed into the Control Data STAR processor.

The operating system is divided into a central part and a peripheral part. The central part controls the immediate operation of the STAR central processor. It resides in the central memory and executes in the central processor. The peripheral operating system resides in the peripheral stations. Each station has a common basic nucleus with the local task programs needed to perform its particular functions. All of these functions are concerned with input/output and the control of peripheral devices. The objective of the total operating system is to make the computing resources available to user programs in a controlled, convenient, optimal manner.

In the network of connected computers that make up a STAR system, information is exchanged by means of messages and files. Messages are used to communicate system requests and to acknowledge the receipt, progress, and completion of requests. Data in the system is always held in the form of files; that is, as collections of information with certain described properties such as names, access modes, physical layouts, and types. Data flow within the network consists of messages, files, and pages of files, where a page is 4096 bytes of information.

The central part of the operating system is called the monitor. It occupies about 2048 words of code and up to 4K words of tables, the latter depending on the number of jobs in the central machine.

SYSTEM STRUCTURE

There are three levels of programs in the STAR central machine:

- executive monitor
- system tasks
- user programs

The central operating system consists of programs written in the first two levels. The executive monitor is distinct from the other levels in that it operates in a privileged mode called monitor mode. When the processor is in monitor mode, interrupts are inhibited, a few extra instructions are enabled, and core storage addresses do not go through the virtual page mechanism, but are instead absolute addresses.

System tasks and user programs run in job mode. System tasks have some privileges denied to user programs. In particular, they can communicate with the monitor using some reserved messages, and they have access to some of the central operating system tables.

For illustration purposes, Figure 2-1 shows a network of peripheral stations connected to a STAR. This CPU model shows the functions performed by the central monitor.

The monitor provides:

- job sequencing and control
- channel control
- central memory allocation
- message handling
- central machine error handling

Control is returned to the operating system when an interrupt occurs causing monitor mode to be entered. Interrupts are not the normal manner in which peripheral stations communicate with monitor. The streaming nature of STAR suggests that polling techniques be used for peripheral stations.

There are four classes of interrupts from job mode:

- program illegal instruction
- program request to monitor
- external channel
- storage access

The first two of these are entries to the monitor from the currently running program.

The only external channel interrupt used is the monitor interval timer. This timer allows for the monitor to be re-entered after a user program time slice.

The storage access interrupt occurs when a program attempts to reference a page that is not in central storage, or when an access is attempted with illegal mode.

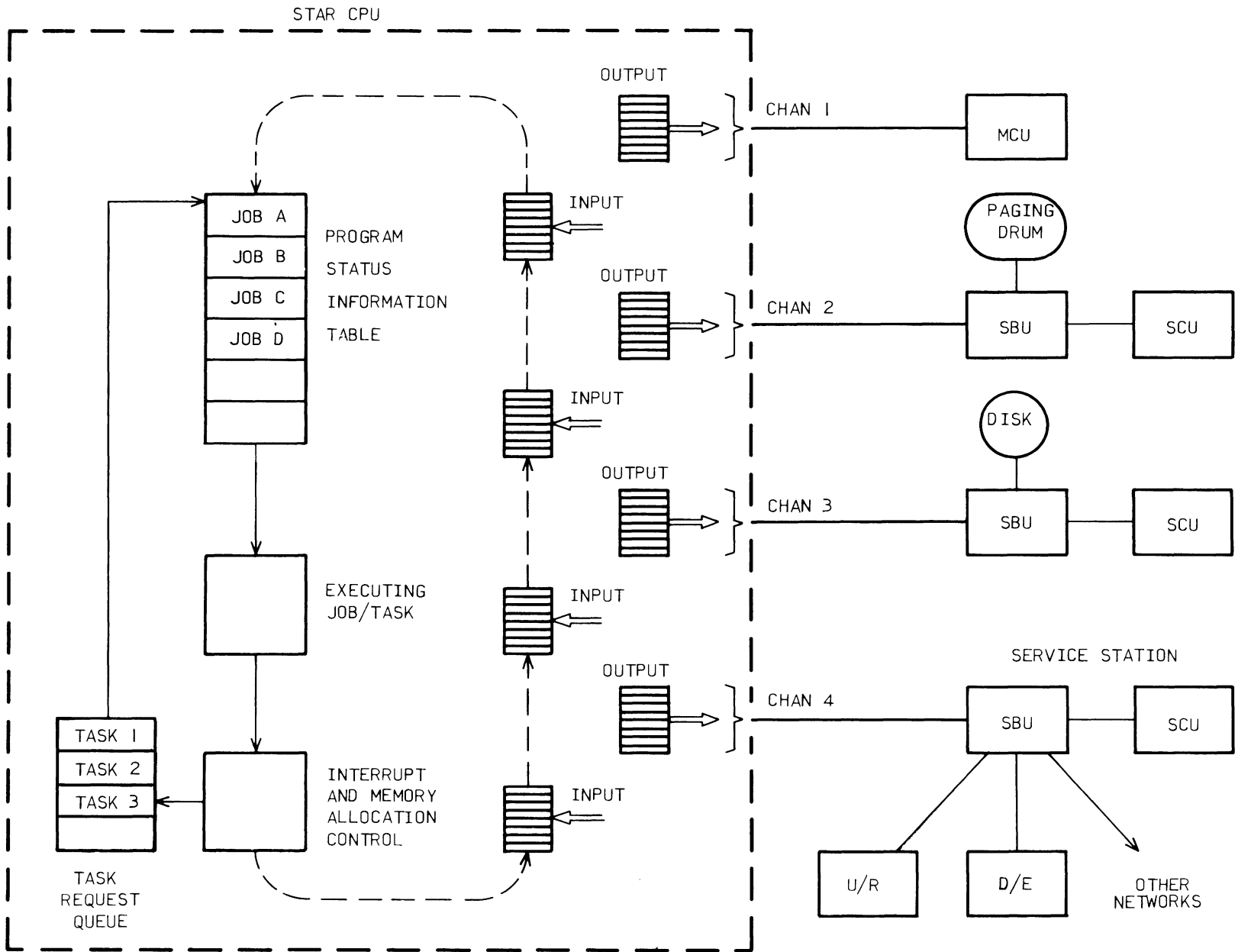


Figure 2-1. Peripheral Station Network

Some interrupts are serviced completely in monitor mode, but most result in the calling of system tasks. System tasks make the majority of operating system decisions, and these programs range from small slave programs of the monitor to much larger suboperating systems. About 2048 words of system task programs are resident in core storage. Those that can be shared by a number of users are placed in the user library.

The sequencing of jobs and system tasks is controlled by a basic table called the program status table. Entries in this table are called control points, and each entry corresponds to a program at some stage of execution. New entries are created by the monitor as jobs are entered into the machine and as system tasks are initiated to process required system functions. The table is scanned by the monitor, in a central scan loop, and the control point with highest priority is selected for execution.

Each channel has input and output communication areas to hold messages and responses and a ready/resume flag mechanism.

The ready/resume flags are inspected in the monitor scan loop. New incoming messages result in the appropriate system tasks being initiated. Responses from stations are also placed in the input communication areas and result in changes to the status information of control points. Although, there is a hardware interrupt associated with each channel, they are not used, but are reserved for real-time application suboperating systems.

Memory allocation is performed when the monitor is entered as the result of a storage access interrupt. This is caused by a program referencing a page of information that is not in core storage. Such an entry to the monitor provides implicit input in that the user program is not aware of the distribution of his program and data between the core storage and any other backing storage device. Programs can also explicitly pass messages to the monitor regarding storage allocation. Some of these are commands, and others are advisory in nature. Most memory allocation processing involves the monitor in sending messages to the drum station or file stations. The control point responsible for the monitor entry is set to a waiting status while such messages are being sent and while the information they require is being transferred into core storage. During this time the monitor selects another control point and allocates the central processor to it.

Requests to monitor from executing programs are processed similarly to the paging and timer interrupts. A processing subroutine is activated depending on the message's function code. Some requests are processed by using system tasks.

VIRTUAL MEMORY LAYOUT

Addresses in the central machine are bit addresses in the range 0 to $2^{48}-1$; that is, 0 to $1\ 000\ 000\ 000\ 000_{16}-1$. Virtual space is divided into two parts, 0 to $2^{47}-1$ for free use and 2^{47} to $2^{48}-1$ for restricted use.

Users can freely use any addresses in the range 0 to $2^{47}-1$, though for efficient use of the storage system it should be remembered that storage is allocated in units of 512-word pages. Random access over address fields larger than the physical amount of core storage can cause low utilization of the central processor. Note that the concept of paging makes overlays unnecessary and the large address field makes variable size segments possible. Each program always has a page zero, the first 256 words of which are in a register file while the program is active and stored in core storage when it is not.

The access mode within the user's region is normally read/write/execute, but a request message to monitor can allow write lockout to be applied over any specified range. There can be more than one such range, they need not be contiguous, and they need not all be requested at the same time. There are also messages that allow specified ranges to be removed from write lockout.

Figure 2-2 shows the layout of the virtual memory.

The user library uses virtual addresses greater than $800\ 000\ 000\ 000_{16}$.

Another division of virtual storage is made at the address $FF8\ 000\ 000\ 000_{16}$. Provision is made for local installation system programs to use the address space less than this address. The virtual space at addresses greater than $FF8\ 000\ 000\ 000_{16}$ is concerned with the sharing of data between users. The space is divided into 1000_{10} 16K-page segments. The first of these is for public sharing, and all users have read/write/execute access to this virtual area.

Private sharing is arranged by the monitor allocating one or more of these segments on request, with a specially assigned access key. Other legal users of the shared data base, who have the access key, can then find out from the monitor into which virtual address it is mapped, and hence, share access to it. Note that this is sharing the same copy and is different from sharing files on the disk station. The last of these virtual segments is reserved for system tasks.

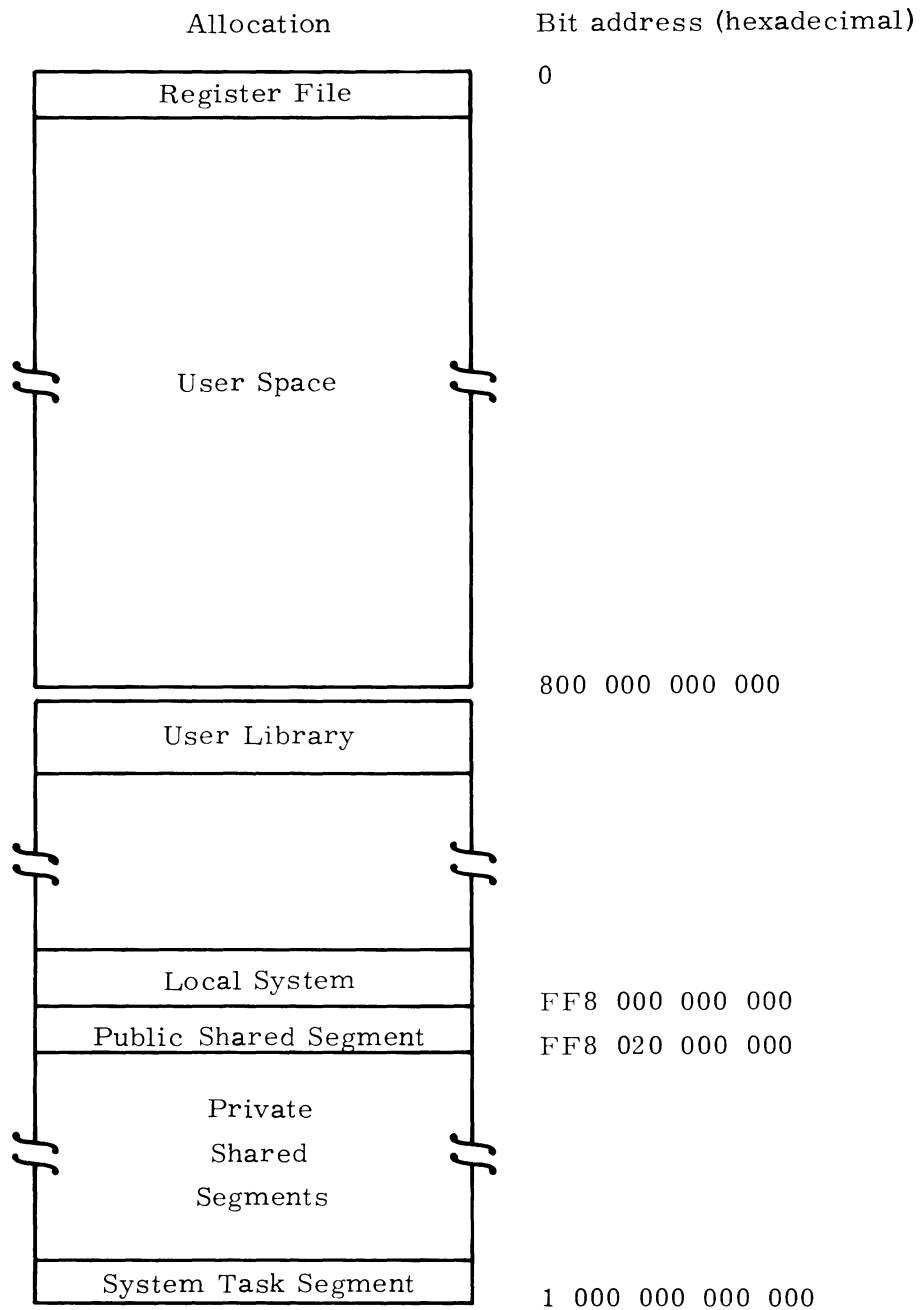


Figure 2-2. Allocation of Virtual Memory

JOB SEQUENCING

The monitor allocates the central processor to jobs in a sequence determined by a system task program called the scheduler.

Each active program has associated with it a control point (CP) and an invisible package (IP). The control point is maintained by the scheduler and contains information needed by the system about the program. The invisible package contains information needed to restart a program from where it was last halted. The IP area is initially set up by the scheduler; thereafter the hardware uses the area for storing and loading the invisible registers whenever the program is interrupted and restarted.

Job sequencing is controlled by means of a table of pointers to these control points. This table is called the alternator. An entry in the alternator stack is a 64-bit word. The format is shown below:

STATUS BITS (40)	CP INDEX (8)	IP INDEX (16)
------------------	-----------------	---------------

The CP and IP indices are 8 and 16 bits, respectively; they are pointers used for locating the program control point and invisible package.

The status bits are used by the monitor and systems tasks to show the control point status.

The following list gives the bit number, the name of the bit and its function. Bits are numbered with the most significant as bit 0.

<u>Bit Number</u>	<u>Bit Name</u>	<u>Used For</u>
0	On	This alternator word is active.
1	MTR	When the wait condition (bit 4) is removed, execute a monitor subroutine that has its entry address set in the control point at MTRTN.
2	--	Not used.
3	Pause	Suspend execution of the CP when all of its input/output messages are processed.
4	Wait	CP is waiting for drum or disk input/output.

<u>Bit Number</u>	<u>Bit Name</u>	<u>Used For</u>
5	Block	CP is waiting for keyboard input or remote card/printer input/output.
6	Recall	CP is in recall mode, that is, suspended for a fixed time.
7	Slice	This CP was stopped at the end of its time slice. It was executing when the monitor interval timer interrupted.
8	Rent	Flag for memory allocation routine.
9	--	Not used.
10	Protection	CP is using write lockout on some pages.
11	Advise	CP is performing disk or drum transfers and is not in wait state.
12	Interactive	CP is connected to an interactive terminal.
13	VPZ	CP page 0 is locked down in core storage.
14	Task	CP is running a system task.
15	Batch	CP is a batch program.
16	Display	The station controlling the interactive terminal has requested an output line for display.
17	--	Not used.
18	--	Not used.
19	Off	Nonrecoverable program error condition exists — this flag causes the CP to be deleted and all its core storage and drum space freed.
20	Buffin	These flags inform the memory allocation routines of the CP address to use on completion of their task.
21	Buffout	
22	Buffin-busy	Data pages are moving in/out of a CP virtual memory and the program is not in the wait state.
23	Buffout-busy	

Figure 2-3 shows the layout of a program control point. A control point occupies sixteen 64-bit words. The various fields show the names of the entries and the number of bits in the fields. These names are listed below with descriptions of the information contained in the fields.


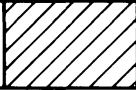
0	ID (32)			MTRTN (16)		VPZ (16)		
1	UT (16)	SFX (8)	ALT (8)	STAT (8)	TYPE (8)	CNTLE (8)	CNTRLR (8)	
2	TICKS (16)		SLOT (16)		QUANTA (16)		PRIORITY (16)	
3	LOG-ON TIME (32)				CPU TIME (32)			
4	RECALL TIME (32)				TIME LIMIT (32)			
5	KS0	SHARED KEYS KS1			KS2		KS3	
6	FAULTS (16)		CM (16)		FL (16)		DK (16)	
7		BLK (11)	USE (3)	KEY (12)	VIRTUAL NUMBER (33)			
8	R/W KEY (16)		TEST KEY (16)		RENTRTN (16)		ACCRTN (16)	
9	FNT (16)	FNT CONT			CHANPT (32)			
A	MESBK (16)		MESPT (48)					
B	OBJBK (16)		OBJPT (48)					
C	BUFFIN WORD							
D	BI_KEY	BI_RENT		BIRTN		MBIRTN		
E	BUFFOUT WORD							
F	BO_KEY	BO_RENT		BORTN		MBORTN		

Figure 2-3. Control Point Layout

Table 2-1 below provides the contents of the various fields of the control point area shown in Figure 2-3.

TABLE 2-1. CONTROL POINT FIELDS

<u>Field</u>	<u>Content</u>
ID	user identification number
MTRTN	a pointer showing the monitor activity when the CP was last active
VPZ	identifies the absolute location of the user page zero
UT	location of user terminal
SFX	suffix of this control point. It is possible for a user at a terminal to maintain up to four independent processes and thus up to four control points at one time. These are designated suffixes A, B, C, and D if they exist.
ALT	this control point's alternator slot.
STAT	state of the control point. The control point can be active, blocked, or have entered an error state (see listing).
CNTLE	control point of controllee (see CNTLR).
CNTLR	control point of controller. It is possible for a control point to initiate another program to take some action on its behalf. When this happens, the original controlling control point is called a controller and its slave control point a controllee. This controllee can in turn initiate another control point and be a controller of this control point. Forward and backward links between controllers and controllees are maintained using CNTLE and CNTLR. In such a chain of control points, only one at any time is actively linked into the alternator stack; that is, only one of the set runs at a time.
QUANTA	is the number of time units or ticks the user is allocated for his time slice. A tick is an installation parameter, the basic allocatable unit of central processor time to a user (1-250 milliseconds).
TICKS	is the count of time units remaining in the users time slice. When the count is reduced to zero it is reset to the value of QUANTA for the next time slice, and the next ready job is run.
SLOT	total number of ticks since the last input/output request made by this user. This count is used by the scheduler to adjust priority.

TABLE 2-1. (Cont'd)

<u>Field</u>	<u>Content</u>
PRIORITY	privilege level of this control point — dictated by the scheduler
LOG-ON TIME	control point start (time of day)
CPU TIME	actual central processor time used
RECALL TIME	time spent in suspension
TIME LIMIT	maximum central processor time allowed
SHARED KEYS	four keys are for shared areas in addition to the four keys for private areas that are in the user invisible package. Provision is made for more keys to be stored in an expanded control point space.
FAULTS	number of storage access interrupts
CM	number of core blocks
FL	maximum number of pages
DK	number of pages currently active on drum
BLK, USE, KEY, VIRTUAL NUM- BER	access interrupt word used in the control point
R/W KEY	user's read/write key
TEST KEY	key used for access interrupt processing
RENTRTN	space allocator state
ACCRTN	fault processor state. These status fields are concerned with the servicing of an access interrupt that involves the bringing in of a page from the paging station and finding space for it. The latter can involve writing out a page to the paging station.
FNT	pointer to user's file chain.
FNT CONT	count of file chain entries
CHANPT	pointer to input channel message area
MESBK	block number of active message page
MESPT	absolute address of active message page
OBJBK	block number of active data page

TABLE 2-1. (Cont'd)

<u>Field</u>	<u>Content</u>
OBJPT	absolute address of active data page
BUFFIN WORD	used for memory allocation for advise input operations
BI_KEY	key used for buffered input page
BI_RENT BIRTN MBIRTN	return addresses for memory allocation for buffered input
BUFFOUT WORD	used for memory allocation for advise output operations
BO_KEY	key used for buffered output page
BO_RENT BORTN MBORTN	return addresses for memory allocation for buffered output

Two pointers are used to process the alternator stack. The major pointer identifies the program currently in execution. When this program is halted, awaiting input/output for example, the minor pointer moves down the stack and activates the next available program. When the program identified by the major pointer comes out of wait status, the minor pointer is reset to it and the program continues to the end of its time quanta. The major pointer then proceeds on to the next available program in the stack.

When a control point is selected for execution, the program page zero is brought into core storage if not already there. This page resides in core storage, and is said to be locked down while the program is allocated the central processor. Before switching to job mode, the monitor sets the monitor interval timer equal to the time slice quanta that the control point will be allowed to execute.

In addition to processing the alternator stack and the input/output control section, the basic monitor loop processes a list called the periodic table.

Monitor tasks that run with a regular period are activated using the periodic table. Such tasks include the accounting summary, preservation of system tables for back-up recovery on error conditions, updating of the dayfile to the file station and activation of the scheduler. Batch jobs are requested from the service station

on a periodic basis, the period being an installation parameter. The periodic table can be accessed by system tasks that can alter the period dynamically, and also allow for their own reactivation.

An entry in the periodic table consists of two 64-bit words. The first of these words has four fields, namely the state, the last run, control and period.

The state field contains status information about the program. The last run field contains the system clock time in milliseconds when the program was last executed. The control field defines the conditions under which the program is run. For example this field determines which system tables the program is allowed to access and whether the program has an exchange package. The period field contains the number of milliseconds between successive runs of the program.

The second word is either a file name or a virtual address. In the latter case, a 1-bit flag indicates that the program is to execute in job or monitor mode.

Figure 2-4 shows the format of periodic table entries and the number of bits in each field.

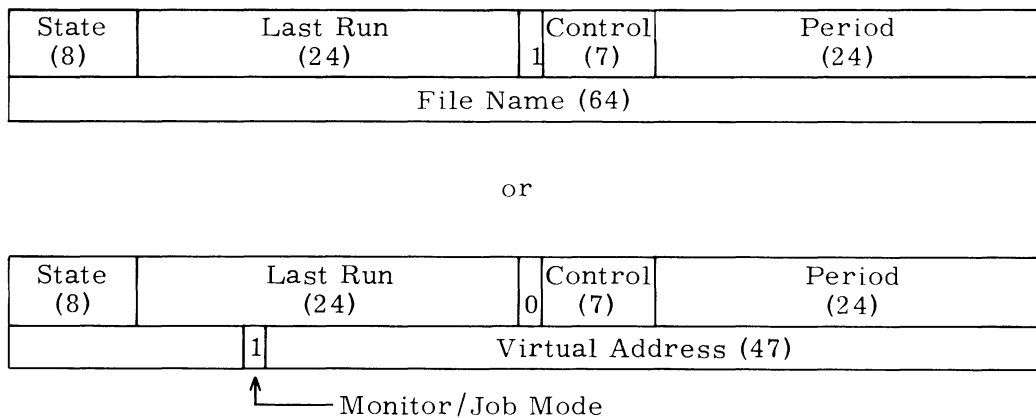


Figure 2-4. Periodic Table Entry

Programs running on STAR are either under the control of a user at a terminal or consist of files of program and data with at least one file of operating system commands. These two types of programs are termed interactive and batch respectively. In the first case each step in the execution of a job is requested by typed in commands.

For batch jobs, a file of these commands is made. The job control commands available to interactive programs are all available to batch programs.

JOB ACTIVATION

A new program is recognized by the central operating system on receipt of a log-on message from a communication station for an interactive job or a service station for a batch job. A batch job is treated exactly like an interactive job except that it does not have a terminal and is usually of lower priority. On receipt of a log-on message from a communication station, the monitor allocates and initializes some tables, allocates memory space for the program page zero and sets the program entry point to the start address of the suboperating system that will process the job control statements.

The tables that are allocated are the control point and exchange package. An entry is made in the alternator table and in the terminal connect table. The terminal connect table establishes a correspondence between an interactive user and a control point. A user can have up to four independent control points controlled from a terminal, termed A, B, C, and D. The format of an entry is as shown below:

Not Used (24)	S (8)	A (8)	B (8)	C (8)	D (8)
---------------	-------	-------	-------	-------	-------

S is the active suffix and is an 8-bit pointer to A, B, C, or D. These are also 8-bit pointers, and point to entries in the alternator stack.

The entry point for the program, set in the exchange package, is normally for Job Control. Job control is a system task in the user library that provides program command processing and the linkage management for using library utilities.

There is a user directory kept in the permanent file system for every legal user. When a user logs on and off, the appropriate entry from this directory is updated.

The entries contain the user password, identification number and real name along with a department name and account number. A user can take special action on the detection of certain errors. A control status and error address entry in the table allow the user to specify which errors and an address to transfer to if any of them occur. Once these provisions are made, they can remain in operation for all future

runs. The other entries in the table record the history of input/output, central processor activity and terminal time statistics. Periodically the entire directory is processed by an accounting program. Figure 2-5 shows the format of an entry in the user directory.

Password (64)			
Not Used		ID number (32)	
Control Status (16)	Error Entry Address (48)		
Department (32)		Account (32)	
Input/output activity (16)	CPU Activity (16)	Input/output batch (16)	CPU Batch (16)
Total Terminal Time (32)		CPU History (16)	Terminal History (16)
User Name (64)			
Spare (64)			

Figure 2-5. Format of an Entry in the User Directory

SYSTEM TASKS

System tasks arising from user monitor calls, the periodic table, or from input channel queues are controlled by the task dispatchers.

There are three types of system tasks:

1. Those sharing the user page zero and normally requested by him.
2. Those having their own page zero and running either for the system itself, such as the job mode scheduler, or for a user request that is processed in job mode.
3. Queued work waiting for an available alternator slot.

For a user's task processed in job mode, the user can be active or inactive while the task is executing, the activity of the user being a function of the requested task.

Tasks of type 3 can include background batch jobs if there are not enough entries in the alternator stack. In this group, tasks are not always connected to control points, and can involve reactivating suspended alternator slots or reinstating drop files. A

drop file is a job's virtual storage with control information taken by the system and stored in the file store temporarily. It can result from a user request for a re-start point, or from the scheduler deciding to suspend a job for a significant enough period of time to make it worthwhile to remove it from central storage. This process is also called "Roll Out."

The task dispatcher mechanism consists of three queues, one for each type of system task with the necessary control tables. The number of such tasks processed, however, is under the control of the alternator mechanism. The task dispatcher inserts entries into the alternator stack when there are free slots. This whole queuing system can be supervised dynamically by a high-priority job mode system task. The task dispatchers are basically tables of one word for each entry. Each entry contains the identity of the calling program, a bit that indicates whether the requested function executes in job or monitor mode, and an entry point for the requested function. The first word is an in/out pointer that allows the queue to operate circularly. The length of the queue is fixed at system autoloading time.

Parameters are passed to system tasks in the first sixteen registers, that is in the first 16 words of page zero. If the task is sharing the user's page zero, the calling program has these 16 registers saved in a monitor buffer and restored at the completion of the task.

Some tasks access and modify a set of system tables. These tasks are provided by the monitor with the access key to the system task private segment and have register 15 set to the start of the system table directory. This table gives the base addresses of all tables maintained by the monitor, as shown in Figure 2-6.

Table 2-2 shows examples of system tasks.

TABLE 2-2. SYSTEM TASKS

Description	Page Zero	Caller State	System Table Access
System Autoload	Yes	-	Yes
System Status Display	No	Blocked	Yes
Job Mode Message Processor	No	Blocked	No
Termination Accounting	Yes	Blocked	Yes
Periodic Accounting Update	Yes	Periodic	Yes
Password Validation	Yes	Blocked	Yes

	Terminal Table
	System Library Table
	Active File Table
	Not Used
	Control Point Table
	Drum Page Count Table
	Job Task Table
	Periodic Table

All entries are 48-bit addresses.

Figure 2-6. System Table Directory

EXTERNAL CHANNEL CONTROL

The channel control routines maintain communication between the CPU and peripheral stations. Message boats move messages and responses across the input/output channels. A boat may consist of a single message or a group of messages.

The boat core storage locations are listed in the system boat directory which lists the base address and length of each input/output channel boat area. The first boat after autoload is read from the base address in the directory and the length is 16-64-bit words. Each boat contains a "next boat pointer" and a "boat response pointer." The message transmitter (CPU or station) allocates the boat and generates the "next boat pointer" so the receiver will know where the next message can be found. The message receiver, after processing the message, returns the response code along with the message at the "response pointer" core address. This response can overlay the original message or, if the sender wishes, in a separate area. Thus the message boat position can be reused even though the actual message is still waiting for completion.

A message count field shows how many messages are in the boat. The length of the next boat in 64-bit words is next to the message count field as shown in Figure 2-7. Note that the boat length field is for the next boat so the receiver can read the next boat in one transfer from the "next boat pointer" base address.

The ready/resume flag mechanism consists of two 16-bit checksum fields at the end of the boat. Both fields are initially zero. When the message boat is ready, the message flag is set to the boat checksum. This action flags the receiver signifying that the boat is ready for processing. After processing by the receiver, the response flag is made non-zero by entering the response flag checksum. This action flags the sender indicating that the boat has been processed. When the sender is the CPU, the response flag is detected by the monitor in its basic scan loop. The checksum flags are the sums of 8-bit bytes in the boat (modulo 2^{15}) and do not include the checksum fields. The formats of peripheral station messages are in the appendix. The first word has the standard form shown in Figure 2-7. The message length is the number of 64-bit words in the message, each message having a separate length field. The left-most bit of the response code is set when the response is returned. The next bit of the response code is set if parameters are being returned in addition to the response code.

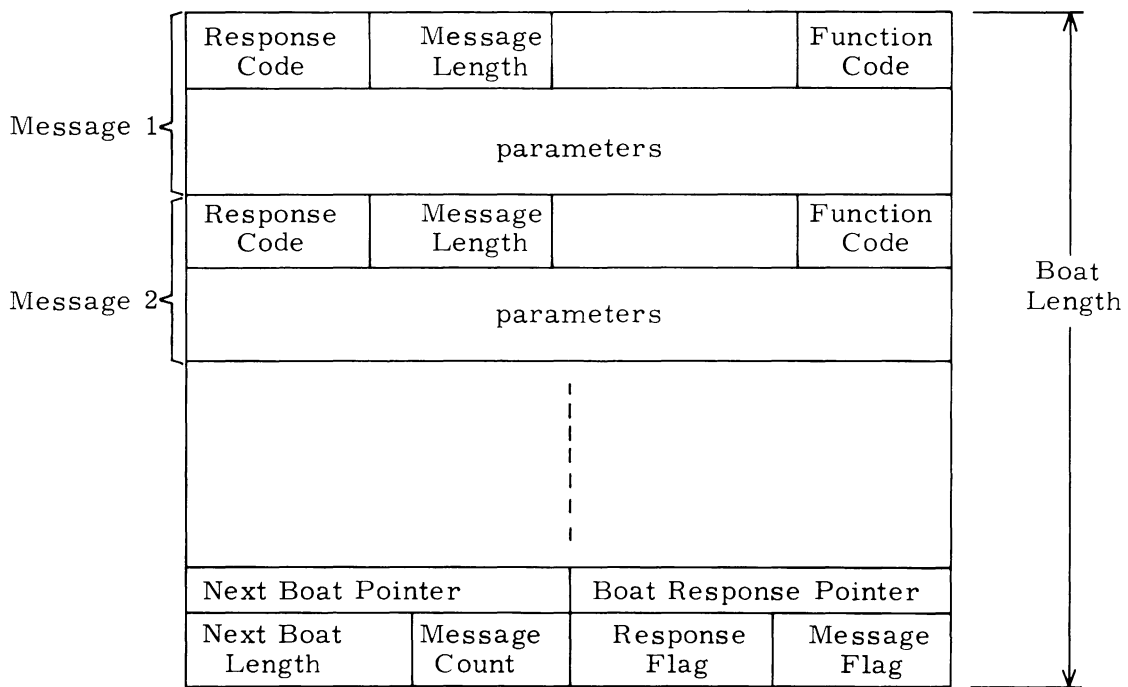


Figure 2-7. Message Boat Format

DYNAMIC STORAGE ALLOCATION

Central memory is occupied at any given instant by the monitor, the resident system tasks, the buffer areas involved with input/output channel transfers, and user jobs.

The monitor maintains a page table that identifies each page of core storage in use, giving the mapping between the virtual address and the physical location, the mode of access permitted, and information about usage. (For a full description of the page table and paging mechanism, see the STAR hardware manual.) The table is ordered in such a manner that the most recently accessed pages are at the top of the table and the least frequently accessed are at the bottom. This ordering is automatically updated by the hardware. At every core storage access, the entry for the page containing the required address moves to the top of the table.

Each user assumes that all of his program is in core storage. It is the responsibility of the monitor to arrange for as much of each user's virtual space to be in core storage as is necessary to ensure system efficiency. Stated another way, when several programs are competing for the available core storage, the monitor decides how much each one shall be allocated.

Storage management is primarily under the control of the access interrupt processor routines. These routines are entered when an access interrupt occurs. Such an interrupt is caused by a user trying to access a page not currently in core storage or by attempting to access a page in the wrong mode; for example, trying to write to a page that is marked as read-only. If the interrupt is caused by a user accessing a page not currently in core storage, the access interrupt processor first marks the requesting job's alternator slot as blocked so that the alternator can allocate the central processor to another job, and then arranges to obtain the requested page. This process consists of passing messages to those stations that might be able to provide the page via the channel input/output control routines. Usually it is to the paging drum station that acts as a large extension to the core storage. When an external station provides the data, it flags the monitor with a message. The page table is then updated to include the new entry, and the job alternator slot is marked as ready. In most cases, the alternator immediately restarts this job that now gets the storage access requested.

The peripheral operating system rents space in central storage for data input and response buffers. Hardware and software lockouts prevent such areas of storage from interfering with each other.

Figure 2-8 shows the general flow diagram for the processing of an access interrupt.

The 12-bit lock field in the page table prevents translations of virtual address which are not associated with the running program. A 12-bit key in the exchange package must also match a page table lock in order for access permission to be granted.

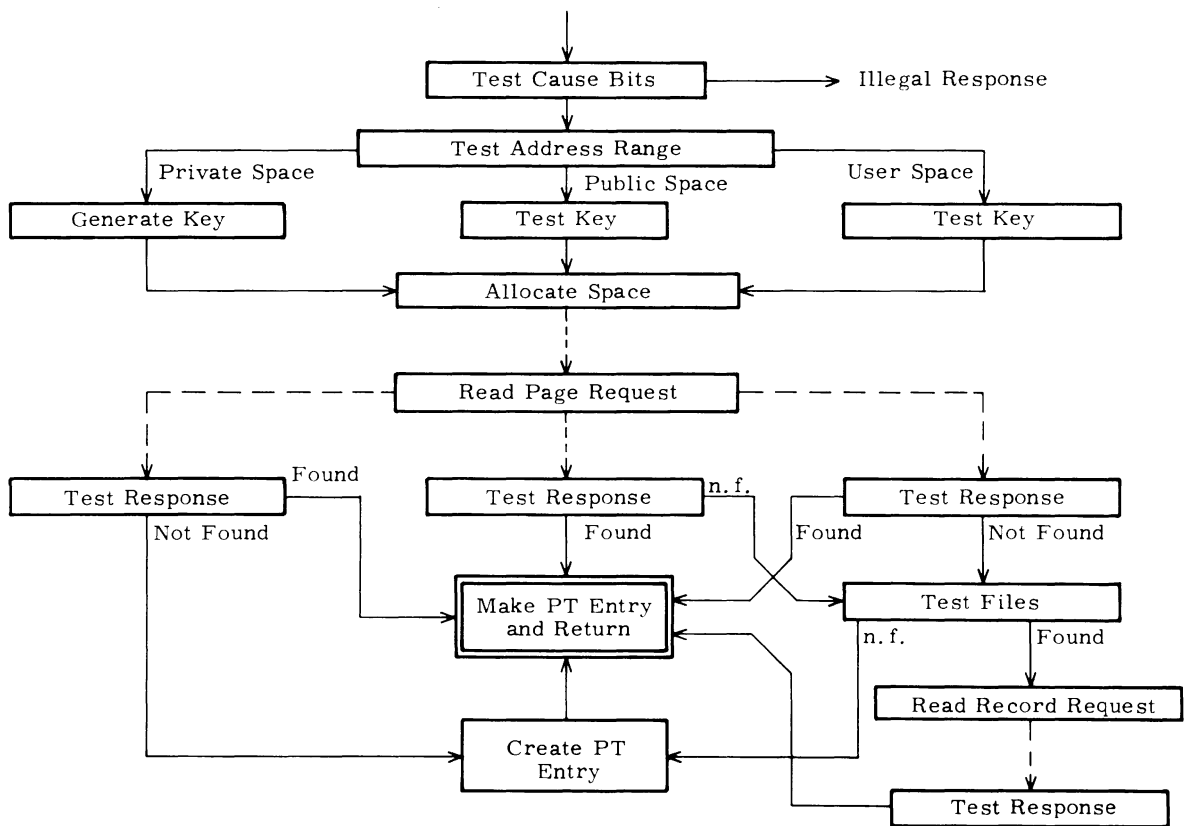


Figure 2-8. Access Interrupt Processing

Each user has a Read/Write and a Read Only key in addition to the public-shared library key. A user may also have one or more private shared segments (see Memory Layout). The shared segment keys are normally held in the control point and moved into the exchange package upon first translation request. There they remain until exchanged by a request desiring some key not in the exchange package. Note that if a user is not using shared segments other than the library, all his keys are loaded into the exchange package at activation time.

When space allocation is being performed and a central memory block is transferred to the paging station, the page table entry is marked with a "hold" key. The hold key allows the INSERT ENTRY routine to locate the block in which the data has been moved.

* * * * NOTE ON CONFLICTS * * * *

In the case of shared segments or shared public library, the HOLD key provides a means of testing on subsequent page faults if the transfer is in progress.

If the advise-type functions are being used to move data in/out of virtual memory, the hold key provides a means to check on succeeding faults wanting the same data which is already in progress.

Figure 2-9 shows the allocation scheme for the keys and locks.

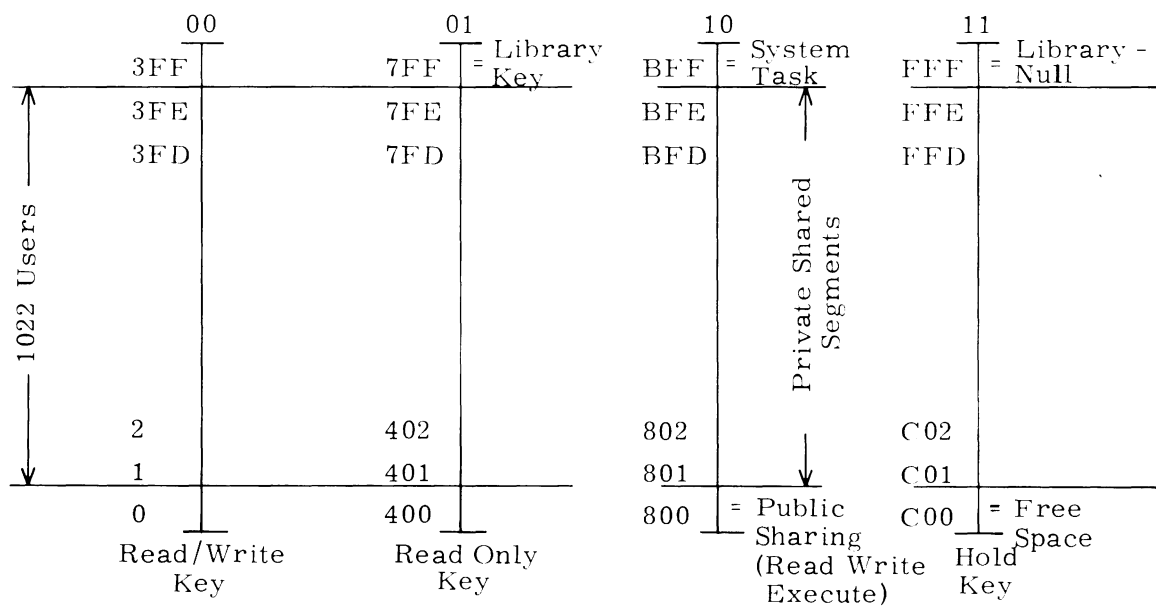


Figure 2-9. Key Allocation

The infrequently used library routines are kept on the disk, but they have a logical correspondence with a virtual memory region. This correspondence is accomplished with the System Library Table, an entry of which is shown in Figure 2-10. Each entry consists of two words. The first word contains the file identifier, and the second word contains the library virtual page and the length of the file in blocks. When searching for a library page which cannot be found either in core or on the paging drum, the System Library Table is searched. If the required address is within the table, a Read Record request is sent to the disk station.

Executing programs can also establish a logical correspondence between a file and virtual address range as long as the file is within the user space. This is the MAP-IN function described under CALL MESSAGES. This function builds a threaded list of active files and links them to a specific control point. Figure 2-10 shows the format for an Active File Table entry.

	file identifier	
length (16)	(15)	library virtual page (33)

System Library Table

	file identifier	
length (16)	next (14)	user virtual page (33)

Active File Table

Figure 2-10. System Library Table and Active File Table Entries

In addition to the use information in the page table, four bit strings are maintained by the access interrupt routine to show the condition of active blocks in central memory.

1. Memory Reservation String — shows which blocks in core are locked down and are not candidates for swap-out.
2. Input/output Lock String — shows which pages in core are in input/output wait state.
3. Drum Duplication String - shows which blocks in core are not on the drum.
4. Modification string — shows which pages in core have been modified at some point in the swapping process, even if not since the last swap-in.

In all four strings, the block number in the page table is an index into the string.

SYSTEM CALL PROCESSING

Job mode programs issue requests to the operating system by a mechanism of call messages. These requests are received by monitor through an exit force instruction. Requests are analyzed by monitor for function and location validity. That is, is there a function processor capable of handling the message, and is the entire message contained within a page boundary? The reason for the latter is efficiency of processing. Monitor does not use the virtual translating mechanism.

There are two types of Request/Call formats. First, a request (EXIT FORCE) followed by a direct pointer address to the message; and, second, a request followed by an indirect pointer to the message.

	<u>Type 1</u>		<u>Type 2</u>
EXIT FORCE	09000000	EXIT FORCE	09000000
Direct flag	00FFXXXX	Indirect flag	00EE00RR
Virtual address	XXXXXXXX	(RR is register containing message virtual address)	

Once the pointer has been found, it is translated via the page table to obtain the core address. If a fault occurs, monitor will try to find the page and move it into central memory. When the message is in central, the block is normally put in input/output lock state and processing begins.

Processing of call messages is generally accomplished by one of three mechanisms:

1. A request processor in monitor
2. Monitor calling and rerouting the request to a system task
3. Monitor queuing the request to a station for processing

Processing of 2 and 3 requires monitor, in most instances, to block the task from execution and be the interface between the task's request and a system task or station servicing the request.

The acceptable system call messages are shown in Table 2-3. Corresponding formats are shown in Table 2-4. Table 2-5 shows the file input/output messages.

The File Input/Output System Messages associated with monitor allow users to obtain disk space as files, attach files to programs, read and write files implicitly or explicitly, save files, and delete files. (A description of these messages is found in Appendix D.)

TABLE 2-3. ACCEPTABLE SYSTEM CALL MESSAGES

FC	Function Name	Parameters	Format
#000	Terminate_Request	FC	I
#003	Validate_Password	FC, User ID password de- scriptor	III
#030	Drum_Page_Count	FC	I
#031	Drop_Batch	FC	I
#03B	Status_Task	FC	I
#03C	Task_Response	FC	I
#050	Buffer_Input	FC, VA	II
#060	Private_Share	FC, Name	II
#061	Private_Access	FC, Name	II
#15A	Recall_Request	FC, usec	II
#220	Execute	FC, File Name	II
#222	Print_File	FC, File Name	II
#22A	Read_Disk_Record	FC, Drive (16), Record (16)	IV
#22C	Introduce_Disk_Pack	FC, Drive	II
#400	Type_In	FC	I
#401	Remote_Print	FC	I
#402	Remote_Card_Read	FC	I
#403	Display_Data	FC	I
#430	Lock_Block	FC, VA	II

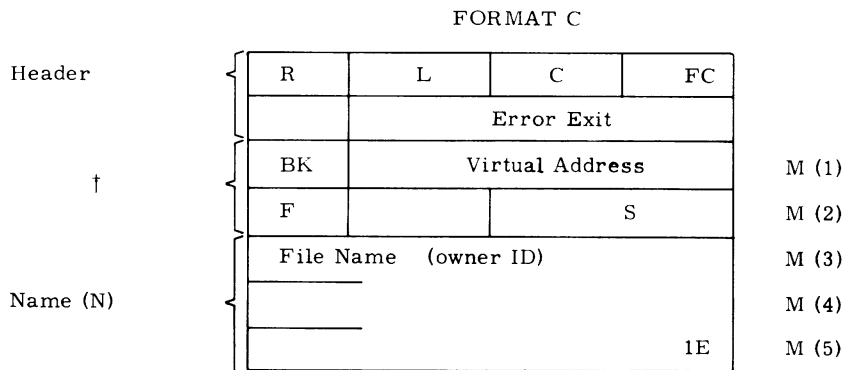
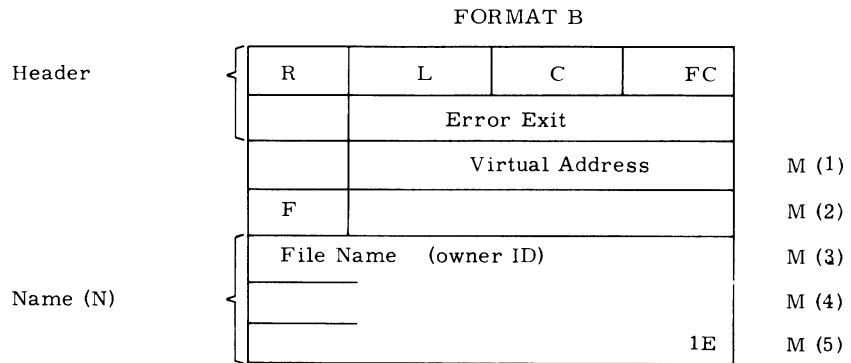
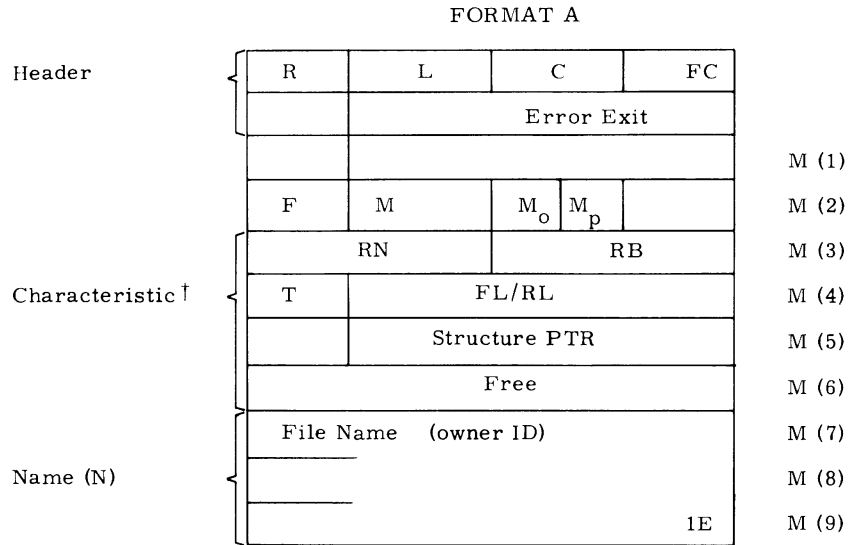
TABLE 2-4. CALL MESSAGE FORMATS

<u>FORMAT I</u>	RC			FC
<u>FORMAT II</u>	RC			FC
		PARAMETER 1		
<u>FORMAT III</u>	RC			FC
		PARAMETER 1		
		PARAMETER 2		
<u>FORMAT IV</u>	RC			FC
			DRIVE	RECORD
		V. ADDRESS		
	FILE IDENTIFIER			

FC = Function Code

RC = Response Code

Note: These formats are in the process of being changed to agree with those used for file input/output (Figure 2-11).



† See Appendix D for specific message fields.

- FC function code
- R response code
- L message length
- C sub control code
- F active file index number
- M mode

Figure 2-11. File Input/Output Message Formats

TABLE 2-5. FILE INPUT/OUTPUT SYSTEM MESSAGES

FC	Function Name	Parameters	Format
150	Buffer Input	(VA)	B
154	Buffer Output	(VA)	B
302	Set File Disposition	(D, NC, T)	D
616	Map-In	(C, F, VA, N)	B
640	Create and Open	(Char, N, M)	A
641	Open	(N, M)	A
642	Close	(F, N)	B
644	Close and Delete	(F, N)	B
645	Keep File	(F, N)	B
646	Set Characteristics	(Char, F, M, N)	A
648	Is File Open	(N)	A
64A	Read File Pages	(F, VA, S, N)	C
64B	Write File Pages	(F, BK, VA, S, N)	C
684	Release	(C, BK, VA)	B

ERROR DETECTION AND PROCESSING

Monitor tests for two types of errors:

- those affecting a specific user, and
- those affecting the total system

Errors which stop a user activate the system Job Control routine which then informs the interactive user of the problem. Batch jobs are terminated or suspended, and messages are sent to the operator and/or system dayfile. The type of error is passed to Job Control in the exponent of register 1. Following is a list of such error messages:

0. ILLEGAL INSTRUCTION
1. PAUSE
2. ILLEGAL REFERENCE MODE
3. MESSAGE FORMAT ERROR - OBSOLETE
4. STORAGE DRIVE FAULT
5. TIME LIMIT
6. STORAGE LIMIT

7. MAPPED FILE NOT FOUND
8. REFERENCE TO NON-EXISTENT LIBRARY PAGE
9. SHARED SEGMENT NOT ACTIVE
10. MESSAGE FORMAT ERROR - POINTER CROSSES PAGE
11. MESSAGE FORMAT ERROR - NO POINTER
12. MESSAGE FORMAT ERROR - MESSAGE CROSSES PAGE
13. DRUM FULL WARNING
14. PAGING STATION INACTIVE
15. MAP-IN TABLE FULL
16. ILLEGAL DISK NUMBER
17. TOO MANY MAP TABLE ENTRIES

Errors which stop the system are hardware failures which have been detected by monitor. These errors, shown in Table 2-6, are sent to the operator display station and the maintenance station.

TABLE 2-6. SYSTEM ERROR CODES

Error Code	Explanation
01	access interrupt with no cause bits set. Hardware error: An access interrupt occurred but none of the cause bits (bits 12-15) of word E of the Invisible Package for this job are set.
02	page table search for allocate space is in associative registers. The access interrupt processing code is unable to find a page table entry to swap to the drum to free up some space; all pages are locked down or are large pages.
03	access interrupt null market not found. Hardware error: 0F instruction failed.
04	private sharing segment is negative. Hardware error: 26 instruction or the 27 instruction failed giving conflicting results.
05	Free Space Block not found in page table. Hardware error: 0F instruction failed.
06	response pointer outside terminal buffer area. In a queue entry received from the Display/Edit station, it was found that the pointer to the response points to an area of central memory that the Display/Edit station was not to use.

TABLE 2-6. (Cont'd)

Error Code	Explanation
07	response pointer outside drum buffer area. Same discussion as for 06 but concerning paging message response pointer received from the service station.
08	response pointer outside disk buffer area. Same discussion as for 06 but concerning a file message response pointer received from the service station.
09	monitor mode illegal instruction. An illegal instruction interrupt occurred in Monitor Mode.
10	Get Message page not found in page table. Hardware error: 0F instruction failed.
11	input from nonexistent terminal. In a message or response received from the Display/Edit station, the rightmost eight bits of the FROM zip code was greater than 7; this is the terminal number and must be in the range 1-7.
12	output to nonexistent terminal or no control point for terminal. Either someone is attempting to use the multi-control-points-per-terminal capability which has not yet been implemented or a message or response is being sent to a terminal that is not logged on.
13	task queue overflow. The task queue appears to be in error as indicated by the IN/OUT pointers being equal when they should not be.
14	communication output queue overflow. The CPU to Display/Edit queue appears to be in error by the IN/OUT pointers being equal when they should not be.
15	communication input queue overflow. Obsolete error code — should never occur.
16	drum output queue overflow. The CPU to service station queue for paging messages appears to be in error by the IN/OUT pointers being equal when they should not be.
17	disk output queue overflow. The CPU to service station queue for file messages appears to be in error by the IN/OUT pointers being equal when they should not be.
18	virtual page zero found while allocating new control point. While initializing for a new job, a search of the page table produced a find when it should not have; a virtual page zero would not yet have been assigned for this job so that there should not be such an entry in the page table.

TABLE 2-6. (Cont'd)

Error Code	Explanation
19	interface DRUM/DISK device error. Obsolete error code should never occur.
20	stack word or control point space unavailable. Either an available word was not found in the alternator stack or an available 16-word control point area was not found when trying to initialize for a new job.
21	virtual page zero not found after log-on rent space. Hardware error: 0F instruction failed.
22	key out of range for drum count update. The key appeared to be incorrect; the user key and control point number are identical and neither can exceed 12 ₁₀ .
23	virtual page zero not found after "ROLL IN" connect. Hardware error: 0F instruction failed.
24	request pointer outside terminal buffer area. A queue entry in the Display/Edit to CPU queue was found to be in error in that it points to a message situated outside of the area legitimately available for such messages.
25	read record page not found in page table. Hardware error: 0F instruction failed.
26	an invalid response code was received from the service station when the CPU sent an OPEN FILE message.
27	delete R/W block error (MAP IN). An invalid response code was received from the service station when the CPU sent a paging message.
28	delete read only block error (MAP IN). An invalid response code was received from the service station when the CPU sent a read only message.
29	read disk record page not found in page table. Hardware: 0F instruction failed.
30	virtual page zero not found after batch activate. Hardware error: 0F instruction failed.
31	S/S message pointer outside buffer. Obsolete error code should never occur.
32	S/S output queue overflow. Obsolete error code should never occur.
33	system task virtual page zero not found. Hardware error: 0F instruction failed.

TABLE 2-6. (Cont'd)

Error Code	Explanation
34	response from nonexistent message. A response was received from the Display/Edit station to a message that the CPU did not originate.
35	undefined drum cell response. This error code should never occur — it is intended for future implementation of the MAP OUT/BUFFER OUT philosophy of data movement. Such does not exist in its entirety in the current version of Monitor.
36	undefined disk cell response. Same as error code 35.
37	Data Ready Message but control point not blocked. A TYPE-IN READY message was received from the Display/Edit station but the CPU is not ready for it; specifically, the control point is not blocked, i. e., awaiting input.
38	drum table full. A "device full" response code was received from the service station when the CPU sent a WRITE PAGE message to the service station. This condition should never legitimately occur if proper corrective action is taken by one or more users (like log-off) when the DRUM FULL WARNING message is issued by Monitor to all terminals.
39	drum write error. An invalid response code was received from the service station when the CPU sent a WRITE PAGE or a REWRITE PAGE message.
40	drum read error. Same as error code 39 but message was READ PAGE.
41	disk read error. An invalid response code was received from the service station when the CPU sent a READ FILE PAGE message.
42	illegal message sent by Display/Edit station to CPU. A message other than one that the CPU expects to receive was sent by the Display/Edit station to the CPU.

SUMMARY OF CENTRAL MONITOR

Figure 2-12 gives a diagram of the main components of the central operating system.

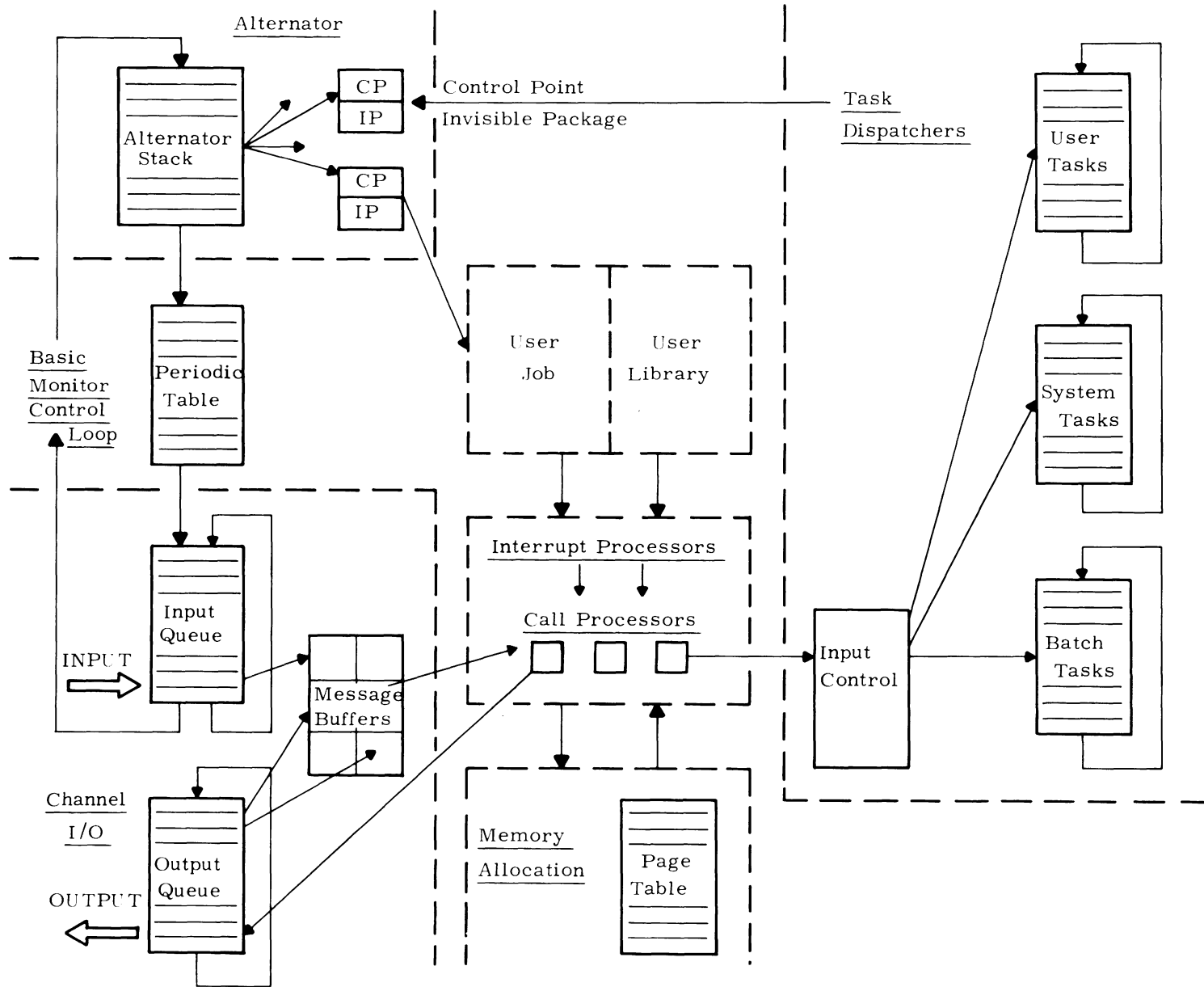


Figure 2-12. Overview of Monitor

The basic interrupt processors pass control to the Call Processors which do the call processing, input message processing, system service processing, etc. A request table contains the function codes and entry points of valid routines. The system task dispatcher, as explained earlier, controls queues of tasks for insertion into the alternator mechanism.

NOTE

The next level of documentation of the central monitor is the program listing itself. This is written in PL/* with copious explanations and details of all tables used.

The peripherals involved in the STAR computer system are organized into stations that provide one or more utilities or services of some nature. A station consists of a data handling computer with appropriate backing storage, extra data buffering storage, and channels both for the management of the resources that it controls and for connections to other stations. The basic building blocks of the STAR stations are the station control unit (SCU) and the station buffer unit (SBU).

The SCU consists of a buffer controller computer having either 8K or 16K bytes of core storage, a microdrum of approximately 80K bytes capacity, and a character display terminal with a keyboard.

The buffer controller computer is a processor specifically designed for data control functions. It is a 16-bit word processor that provides bit, byte and word manipulation, and controls one high-speed parallel block transfer channel and a number of individually driven channels for slower devices.

The SBU consists of 64K bytes of core storage, arranged in eight separately phased banks with storage control logic providing for 12 independent channel access. The SBU is always associated with a controlling SCU. Its function is to provide intermediate buffering of data and the logical control at a network nodal point of one STAR channel with many other station channels.

The stations of the STAR peripheral system do not execute user code. They perform assigned system tasks associated with input/output and peripheral activity. The station software is a distributed part of the total operating system.

The communication of required tasks to and from the stations is achieved by messages between stations. Each station knows the messages with which it can deal and the responses it can make.

Any computer system that is suitably hardware and software interfaced can be connected into a STAR system in a straightforward way.

STAR STATIONS AND SYSTEM FUNCTIONS

Every station manages its own resources and controls its own devices. Within the framework of an operating system, a station multiplexes and drives devices, buffers data involved in transfers, maintains communication paths to other stations, initiates messages, processes and responds to messages, handles error conditions, provides maintenance access, collects performance statistics, and interacts with machine operators. A station is named and defined by the specific functions it performs rather than by the equipment to which it is connected. There are presently eight different, functionally-oriented stations.

The maintenance station consists of an SCU with a card reader, a line printer, and a magnetic tape drive. These peripheral units are controlled directly from their associated I/O channels without any intermediate hardware controllers. The maintenance station also has some additional channels connecting it directly to the STAR central processor hardware. The station provides a diagnostic and maintenance service and a measurement and monitoring function for the STAR central processor. It can also function as a small unit record station.

The paging station provides virtual memory extension for the STAR memory system. It consists of drum storage units, one or two SBUs, and one controlling SCU. Each SBU contains a hardware search mechanism to assist in virtual address mapping. The station provides such functions as maintaining a virtual map of the drum system, searching this map, driving the drum units, processing queues of drum transfer requests and buffering blocks associated with these, detecting drum error conditions, and performing access checks.

The disk storage station provides file storage and file management. It consists of a large disk file transferring blocks of data in and out of an SBU under control of an SCU. The system functions performed by the station include driving the disk unit, maintaining the file directories, allocating disk space, providing file access protection, keeping file recovery data, executing the standard file functions such as read, write, create, and map, and providing rerouting for back-up systems.

The service station provides a focal point for the input/output system. It consists of an SCU and an SBU with one drum unit. Of the 12 SBU channels, one is connected to the SCU, one to the drum, eight to other stations, and two to the STAR central processor. The station maintains message and data paths for the network, provides

temporary storage for files, validates users, controls communication between STAR or other processors and the input/output network, manages the accounting for users, and maintains the print, punch, and batch queues.

The disk pack station and magnetic tape station both consist of an SCU and an SBU, with the SBU channels connecting to removable disk pack devices or magnetic tape drives. The stations provide the functions listed under the disk storage station and also control device mounting and checking and data translations. The media station is a combination disk pack and magnetic tape station.

The unit record station provides an interface between unit record devices and the system. It consists of an SCU and the peripheral devices that may or may not be connected via hardware controllers. Normally, this station is attached to a service station. The station drives the devices, compresses and expands data, and does a small amount of processing - for example, of job control cards.

The display station consists of a modified SBU called a station display unit (SDU), a controlling SCU, and up to 28 local display terminals. The system functions performed in it include driving the displays and keyboards and providing character editing systems.

The communication station drives remote terminals such as displays, teletype-writers, and card readers, usually using the telephone network. Typically there are 64 terminals per station.

A service station with display stations and unit record stations attached to it forms a file/edit subsystem. This provides input/output, storage, and editing of data independent of any STAR central processor activity. Figure 1-1, in Section 1, shows typical connections of these stations with STAR processors.

MESSAGES

Messages are sent between stations to communicate control information. Each message contains a header and a message body. The header contains the destination and the address of the sender, specifies the function being sent, and contains some checking information. The message body contains the parameters associated with the particular function. Because different functions require different numbers and types of parameters, there are a number of different message formats. These formats are kept to a minimum to make the message processing programs small and efficient.

Each station has a table of messages that it is capable of sending and servicing. Responses to messages are also messages, and these are always sent on the completion of requested tasks.

The addresses of stations and units in the system can be likened to postal zip codes and are often referred to by that name. The message directories and associated tables reflect the system components and configurations. For example, a task might be allocatable to more than one station, or alternative routes to a station might well be possible. In such cases, a station originating a message might have to decide which destination to choose or which rerouting to use if the normal routing is currently unserviceable.

In Appendix C, the message directories, buffers and formats are described. Appendix D gives the list of messages, showing which stations can process which messages. Appendix D also traces the steps involved in card reader input, showing the interaction between the unit record station and the service station.

SYSTEM STRUCTURE

Each type of station has a different set of tasks to perform. The hardware and software associated with each type of station is that which will enable the station to perform its tasks.

The software for any station is a portion of a complete set of station software files. A simple resident operating system is common to all stations, and many overlays are used by several types of stations. A single, complete set of station software that the stations can share is a structure that is economical to produce, maintain, and improve.

The set of software for any type of station is divided into as many as five different systems which are appropriate to different types of operation. These systems are:

1. microdrum loader system
2. run system
3. diagnostic system
4. off-line utility system
5. checkout system

The run system is the system used in normal system operation.

A primary and backup set of all the software associated with a station is kept on the microdrum. At autoload time, any one of the systems can be loaded. After loading the basic operating system, pointers to the remainder of the operating system are established and conditions under which these overlays will be called are defined.

PROCEDURES

Station software consists of a set of procedures which are identified by residency and type. Procedures are the bounded contiguous code sets seen by the system loader, the microdrum loader, and the overlay driver. Procedures are variable length, single or multiple functional code sets, or data sets.

Associated with each procedure is a header that contains an identifying name, procedure definition, and system assignments. The system assignments refer to the specific systems in which the procedure is used. The layout of this header information is given in Appendix F. The seven procedure types defined are:

- Declaratives
- System Autoloader
- Direct Core Overlays
- High Core Overlays
- Permanent Overlays
- Temporary Overlays
- Data Buffers

DECLARATIVES

Declaratives are pseudo instructions used for procedural definitions.

SYSTEM AUTOLOADER

The System Autoloader is a switch selectable procedure read in at autoload time under hardware control. A system autoloader is limited to one track (1152 words). In addition to the system autoloader program, this procedure contains the initial direct and high core code sets for the nucleus.

DIRECT CORE OVERLAYS

Direct Core Overlays are loaded by the system autoloader into the 256 directly addressable buffer controller memory words. A maximum of four direct core overlays will be sequentially loaded (following the nucleus direct core) as required by the system assignments. Assembly time locations and run time locations are identical for direct core overlays.

HIGH CORE OVERLAYS

High Core Overlays are loaded by the system autoloader following the initial nucleus high core. A maximum of four high core overlays may be assigned to a system. Assembly time locations and run time locations are identical for high core overlays.

PERMANENT OVERLAYS

Permanent Overlays are loaded by the system autoloader following the last high core overlay. In addition, each permanent overlay is mapped into the Virtual Residency Table. Although these procedures remain permanently in core (thereby allowing local dynamic modifications), location independence is assumed by the system loader.

TEMPORARY OVERLAYS

Temporary Overlays are procedures loaded by the overlay driver. These overlays are one pass, re-entrant, and location independent. In addition, they are assigned a release priority (0 or 1). As with permanent overlays, they are mapped into the Virtual Residency Table upon arrival in core. Temporary overlays remain in core until their space is required for another temporary overlay.

DATA BUFFERS

Data Buffers are procedures which generally do not contain execution code. They are called directly (via the microdrum driver) by tasks. Some examples of data buffers are code conversion tables, directive statements or format tables, and canned displays and printouts.

The nucleus, which includes the system autoloader, one high core, several permanent, and many temporary overlays, can be considered the station equivalent of

the STAR central monitor. The nucleus allocates and manages station resources (space manager, overlay driver, scanner), drives the local SCU devices (microdrum, keyboard/display, couplers), and schedules the system functions (message switch).

As part of the autoloader procedure, memory is partitioned into permanent space, temporary space, and buffer space. Permanent space includes all of direct core and the necessary high core required by high core and permanent overlays. The remainder of core is divided into temporary space and buffer space. Temporary space is an area set aside for exclusive use by temporary overlays while buffer space is used exclusively for buffers.

Station execution code consists of virtually and physically addressable routines. Virtual routines include all permanent overlays, all temporary overlays, and certain routines found in the system autoloader and high core overlay procedures. Physically addressable routines are code sets or subcode sets found in the system autoloader and in high core overlays. Physically addressable routines cannot be temporary overlays.

The Virtual Residency Table defines the residency of all virtual routines. The length of the VRT, which varies from station to station, is set by the number of virtual routines required. Each entry consists of three words; one word for memory physical address, the base address of the routine, and two words for the microdrum (or SBU) address of the routine.

Virtual routines are identified by their index value (called the Program Number) in the virtual residency table. A virtual routine is present in core when its base address is nonzero, absent when its base address is zero.

Virtual routines are called by program number and relative address. Their residency is at all times invisible to the caller. Virtual routines always return to the caller via a virtual return address.

Virtual routines can be initiated by three different mechanisms: the scanner, the virtual queue, and directly by another routine.

Physically addressable routines are permanently resident in core. These routines return to the caller via a physical or virtual return address, dependent on the synchronous/asynchronous nature of the called routine (the microdrum driver is asynchronous, the active overlay routine is synchronous).

The scanner is the idle loop of the nucleus. The primary purpose of the scanner is to map normal channel data signals to the Virtual Residency Table based on priority and logical selection, thereby providing a low overhead mechanism for handling asynchronous external events. The external events (such as channel flags, microdrum busy, or input ready signals) are presented to the scanner program via one or more normal channels. Associated with each channel are two logical selection words, the ENABLE mask, and the STATE mask. The channel data is exclusive or'ed with the state mask in order to select the appropriate signal polarities, and then matched against the enable mask. Any bits that are now set represent selected channel events in the desired state. These bits are scanned from left to right and the first bit found set is used to index the Virtual Residency Table. If all bits are zero, the scanner moves on to the next channel and repeats the procedure. One or more memory words are used to initiate internal events via the scanner. In this case, the memory words rather than the channels represent the raw input to the scanner. In a typical station, the scanner cycles through two normal channels and two memory words.

Some stations have a critical time within which the scanner must be re-entered so that time-critical dependent devices can be correctly handled. This time is at least a few hundred microseconds; however, tasks have to be written with this in mind and subdivided accordingly so that returns to the scanner can be made.

The Virtual Queue is a mechanism for calling virtual routines on a first come first served basis. Each one word entry in this circular queue points to a control package or a parameter package, or is null. In each pass of the virtual queue program, one entry is scanned; an N entry virtual queue requires, therefore, N passes to scan all entries. The virtual queue program is called by the scanner, generally as the lowest priority "bit" of the scan.

The nucleus idle loop then is: scan the channel bits, scan a virtual queue entry, repeat.

Virtual routines can also be called directly by other routines. All such calls, whether directly, by the scanner, or by the virtual queue, are directed to the Virtual Connector, the function of which is to pass parameters to a logically addressed program.

The virtual connector is entered with a flag which identifies whether the request is via a control package or a parameter package. The program number found in this package indexes the virtual residency table, and if the routine is in core, the virtual connector exits to it with index registers B1 and B2 pointing to the caller's control package and/or parameter package. If the requested routine is absent from core, the virtual connector saves the package pointers and calls the overlay driver subroutine.

After the routine has been read into core, the overlay driver inserts the starting address of the routine into the virtual residency table, picks up the package pointers, and exits to the routine.

The only virtual routines which can be absent from core are temporary overlays, and these overlays all share the same space. The overlay driver, which manages this space, always finds room for an overlay. This section of core, set aside for the exclusive use of temporary overlays, is dynamically inhabited by free space, priority - 0 overlays and priority - 1 overlays. Overlays remain core resident as long as the overlay driver can find sufficient contiguous free space for new overlays. Free space is created by first deleting all priority - 0 temporary overlays from the virtual residency table. If the resulting free space is still insufficient, all priority - 1 overlays are deleted; that is, all space is then free space.

Although station software will run in 4K memory, an additional 4K increases performance by allowing more of the working set (temporary overlays) to remain core resident, thereby reducing overlay thrashing.

Station tasks are initiated by:

- receipt of system or operator messages
- initiation of a user device input
- internal maintenance or error recovery procedures

Associated with each task that a station can perform is a task overlay. This overlay executes the task by calling the appropriate subroutines. The tasks that a station performs are defined by the messages it processes. For each message there is a message overlay to control the execution of the requested tasks. A message overlay is one form of a task overlay.

Programs that are used in more than one station – for example, a disk pack driver – are written as subroutines obeying certain coding conventions. These subroutines are documented separately in a manual entitled Peripheral System Subroutines. A typical example of a subroutine specification is shown in Figure 3-1. A subroutine is normally entered using an indirect jump. The subroutine specifies which register contains the pointer to the parameter area. A driver can consist of all or part of a residency overlay or can be a nonresident overlay.

NAME: RENT_CORE
AUTHOR/PHONE: C. Berkey/3774
DATE: 12 Feb. 72
LENGTH:
DESCRIPTION: Rent message area or buffer space in SCU
USE: Enter by indirect jump to RENT_CORE with B2 = parameter address.
Parameters:
1 control package (CP) address
2 return address
3 length in 16-bit words
4 response places SCU address here
COMMENTS: Parameter word 4 is zero on return if not enough space available.
MODIFICATION RECORD:

Figure 3-1. Example of a Station Subroutine Specification

Associated with each task execution is a task overlay that contains all the code necessary to control the task to its completion with the help of station subroutines. Two similar tasks frequently have completely separate task overlays that call many common subroutines. These subroutines are the major way of minimizing the total amount of software needed. There are presently more than 100 subroutines contained in the STAR station subroutine file, including device and channel drivers, resource allocation routines, error handling routines, functional subroutines, message queue handling, and message decoding routines.

The task overlays are re-entrant programs, and a single overlay handles several simultaneous tasks by the use of short tables called control packages. The control package contains the parameters for each task and shows its status. Figure 3-2 shows a typical control package format. Parameters are passed from one routine to another using either the control package or words in low core.

Subroutines retry address	Status
Message buffer address	Master control package address
Subroutine return address	Subroutine function code
SCU address	SBU address
Storage access control	
Disk or drum address	
Not used	
Not used	Message address

Entries are 16-bit words, except for the 32-bit storage access control and disk addresses.

Figure 3-2. Layout of a Typical Control Package

A typical flow diagram for a task program is shown in Figure 3-3. The message handling subroutine, responsible for reading messages from the STAR central processor, initiates this overlay on receipt of the read page message. The tasks to be done are shown in the center and the subroutines called to do them on the right. These subroutines are entered from the scanner, as requested by the calling routine placing entries in the appropriate queues. A subroutine can sometimes be blocked, for example, because a queue is full. In this case it exits to the scanner after making arrangements to be recalled later. On such periodic re-entries, the routine checks to see if it can continue. This process is shown on the left of the diagram. The task ends with the release of the control package, and the buffer spaces used in the SBU for data transfers and messages. The final exit is back to the scanner.

SYSTEM LOADERS

There are five loaders used in the peripheral operating system: The autoloader, the system autoloader, the overlay driver, and the microdrum physical and logical loaders.

The autoloader is a hardware sequence that loads a preselected track of 2304 bytes from the microdrum into core storage, starting at address zero. After the load, control is transferred to address one. The autoloader procedure is initiated locally from a button on the SCU control panel or remotely from the maintenance station.

The system autoloader is loaded with the nucleus at autoloader time. It executes a set of short diagnostic programs, then transfers control to a routine that loads the run system only when the key actuated lock is on or else awaits selection of a particular system by the operator. The various system types are displayed and are initiated from the console function keys. The system autoloader sets up the scanner and overlay tables for the selection system, loads the resident portions into core, and transfers control to it. External autoloader, which can be initiated from the maintenance station, causes automatic autoloader of the run system if the key-actuated lock is on. All stations may be started up by this one external autoloader.

The overlay driver is contained in the nucleus. The driver transfers nonresident overlay routines from the microdrum or SBU into core when called during station operation. Because these overlays are address relocatable, the driver simply transfers the microdrum or SBU binary code image into core without any processing.

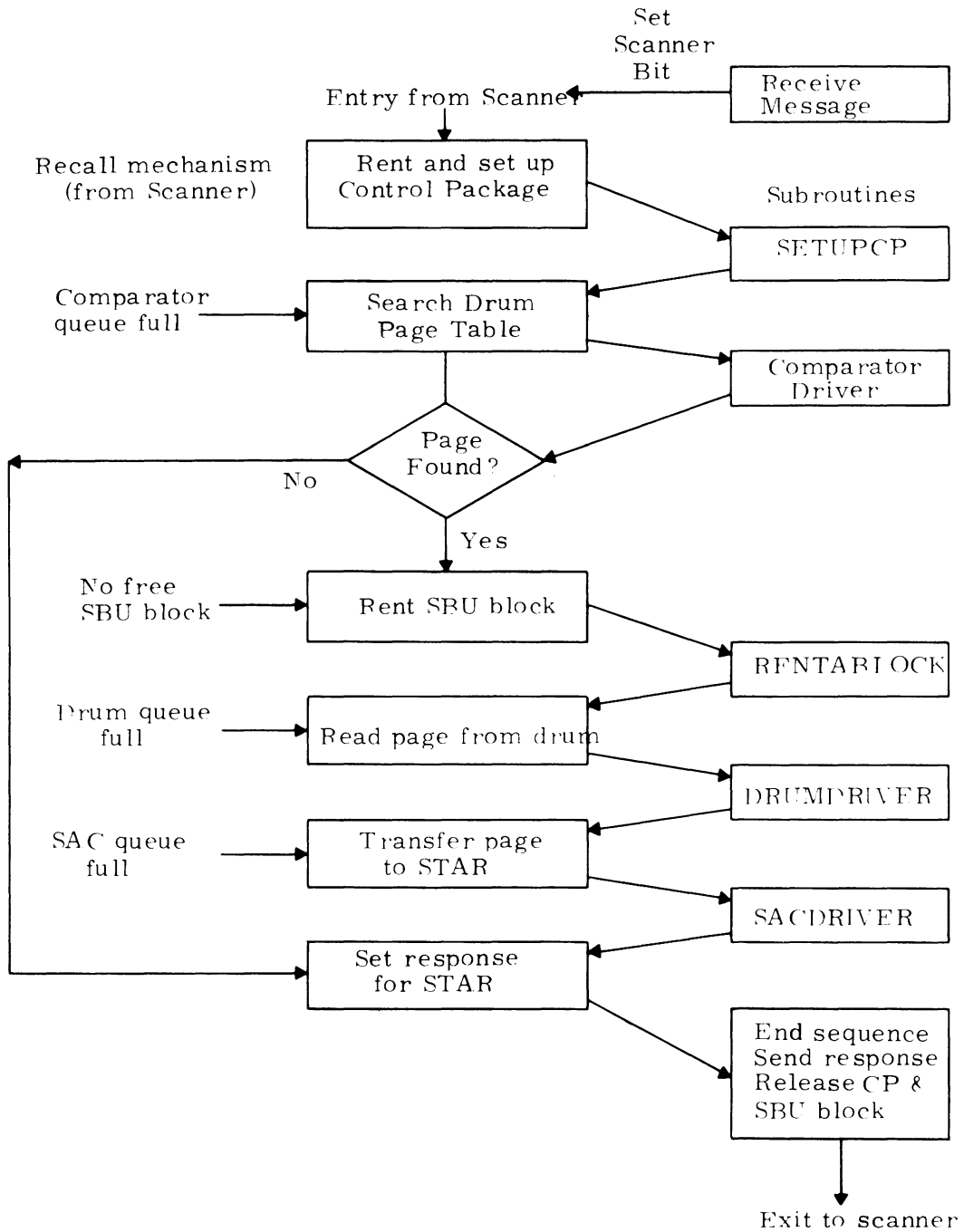


Figure 3-3. Flow Diagram for Read Page Task Program (function code 200)

The microdrum logical loader is a software system that accepts binary output from the assembler. It processes the header data associated with every assembled routine and then forms the scanner and overlay tables for each system and loads these and the overlays onto the microdrum. The binary data can be from any media. It is possible to add overlays to existing systems and to load complete new sets of software. Appendix F gives details of the layout of the microdrum and of the microdrum loader. The microdrum physical loader simply copies an image of the microdrum from SBU to central memory.

BUFFER CONTROLLER MEMORY LAYOUT

Figure 3-4 shows how the buffer core storage is organized in a typical station. The first 2048 words, approximately, are taken up by the basic common software that includes the nucleus. The remainder of the storage is divided between high-core overlays, permanent overlays, and space for temporary overlays and buffers.

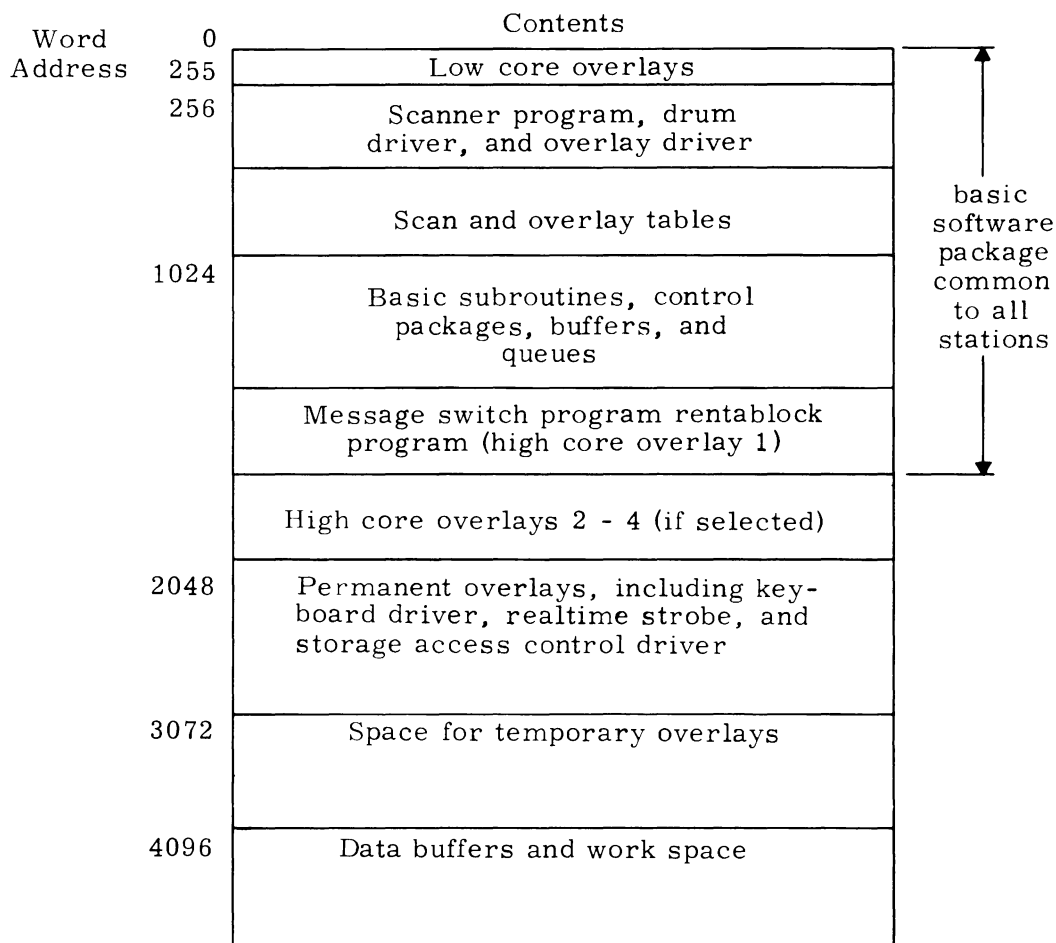


Figure 3-4. Typical Layout of Buffer Controller Core Storage

SOFTWARE DEVELOPMENT

The peripheral system software is written in the Buffalo assembly language. Appendix I describes Buffalo and includes a program example.

In adding a new task to a station, the task overlay and message overlay are designed and coded with any new subroutines or overlays needed. This software is then entered into STAR and assembled. The output is loaded onto the required system at the station and initially developed off-line using the operator's console. The keyboard/display system provides various software development aids and can be used to simulate sending the message for initiating the task.

One permanent symbolic file is used to hold the basic software systems for all stations; that is, the nucleus plus the variations in nucleus code required for different stations. The conditional assembly features in Buffalo allow the required variations for a particular station to be extracted. As well as the core nucleus file common to all stations, each station has another file which contains the special-purpose software necessary to the operational functions of that station.

MAINTENANCE INFORMATION SYSTEM – MIS

MIS is a standard method of error processing and maintenance access included in the peripheral operating system. It provides for:

- saving of station and device status on occurrence of errors
- automatic retry
- error and retry logging
- on-line isolation of faults
- an access mechanism to perform diagnostics on a unit while the remainder of the station is operating under system control
- background diagnostics
- visibility to customer engineer of maintenance operations

See Appendix F for a complete discussion.

CUSTOMER ENGINEERING MANIPULATIVE LANGUAGE – AID

AID is a manipulative language for use by the customer engineers. Its purpose is to provide the customer engineers with the capability of generating short test programs used to debug Input/Output hardware failures.

Programs are written using the SCU display and keyboard and stored on the SCU microdrum until execution. When requested, the generated program on the microdrum is read into SCU memory and executed. During execution, program control is not returned to the nucleus thereby permitting tight test loops for scoping purposes. Before execution, the generated AID program can be referenced on the microdrum. After execution, the program can be referenced both on the microdrum and in SCU memory. In both cases, the generated code is contiguous.

AID is part of the Diagnostic system and uses the resources of the nucleus; however, it does not use the system drivers. (See Appendix F for a complete description.)

A file is a collection of data items, together with a catalog entry called a descriptor that describes the collection to the system. A file is the basic unit in which information is handled within the computer, and can be manipulated by a set of file functions provided by the operating system.

This section describes the creation, maintenance, recovery, access, security, and storage layout of files. These capabilities, called the storage management of the file system, exist completely in the storage stations. The storage stations are independent of any other stations including central processor stations and system network connections.

The subject of record management, how the user accesses his files in the central machine, is described in Section 5.

DESCRIPTOR

Each file has a descriptor, or catalog entry that describes the file. The descriptor consists of eight sections as shown in Figure 4-1. The first six of these sections are the header, characteristics, name, storage map, access list, and activity record. The last two sections are free for future developments of the file system.

Any given storage unit contains a number of files with their descriptors. The set of all descriptors in a storage unit is itself a file which can be processed like any other file. It is called the descriptor file, or catalog, for the storage unit and has the name `DIRECTORY_FILE_xxx` (where xxx is the pack label for an exchangeable disk pack, for example). The catalog is not necessarily kept on the same storage device as the files it describes. Removable units contain their own catalog files, but these can be copied elsewhere on mounting.

The size of a descriptor is variable, but, for reasons of efficiency in the station control unit, it is always a multiple of 256 8-bit bytes. The basic descriptor is limited to 4096 contiguous bytes. Sections that must be variable in size contain overflow pointers to further areas; these overflow areas are not limited in size.

The space needed for the catalog is also variable, but a fixed area, 64 blocks of 4096 characters per block, is currently used to provide as many as 1024 files per storage unit.

Header
Characteristics
Name
Storage Map
Access List
Activity Record
Free
Free

Figure 4-1. Component Parts of a File Descriptor

When a file is created, a 256-byte descriptor space is allocated in the catalog area; the location of this space is a function of the file name. An address in the catalog area is formed by extracting part of the string of characters in the file name and altering its value with some logical operations. This transformation, commonly called a hash of the name, usually results in the address of an empty space in the catalog. However, it is possible for two or more names to hash to the same address. In such cases, the closest empty space is found and the descriptor placed there.

When a file is opened, the location of the descriptor associated with it is found in an identical manner. The same hash of the file name is made, and the resulting number used as an address of the descriptor. If the file name read from this descriptor does not correspond with the correct file name, then a search is made of the surrounding descriptors to find the wanted one. The descriptor is only referenced when a file is opened and closed.

LAYOUT OF DESCRIPTOR FILE ON THE 841 EXCHANGEABLE DISK PACK

The first two cylinders of the disk pack consists of 40 tracks of two blocks each, where a block is 4096 bytes. These blocks are used to hold the pack label, a map of free storage, and the descriptors.

<u>Blocks</u>	<u>Use</u>
0-1	pack label
2-3	free storage map
4-67	descriptors
68-79	spare space

The basic descriptor size is 256 bytes so there can be 1024 descriptors on the pack and, therefore, 1024 files. The capacity of the pack is 7200 blocks.

The format of the free storage map is as follows, in 16-bit words:

<u>Word</u>	<u>Use</u>
0-7	not used
8	storage map type
9	pointer to start of map
10	total number of blocks on unit
11	number of faulty blocks
12	number of used blocks
13	number of free blocks
14	number of entries in map table
15	number of active entries in map table
16-2047	storage map of free space

In the free space map, each entry consists of two 16-bit words. The first of these is the storage address and the second gives the number of contiguous blocks available, starting from that address.

The storage map is normally kept on the storage device, although the operating system can also process it from a station buffer if necessary.

The following sections discuss the component parts of a descriptor.

HEADER

The header portion of the descriptor consists of nine 16-bit words. It gives the byte length of the entire descriptor and the relative starting address of the other seven sections of the descriptor. The layout is shown in Figure 4-2.

<u>Word</u>	<u>Contents</u>
0	Descriptor byte length
1	Not used
2	Relative address of characteristics section
3	Relative address of name section
4	Relative address of storage map section
5	Relative address of access list section
6	Relative address of activity record section
7	Relative address of first free section
8	Relative address of second free section

Figure 4-2. Format of Descriptor Header in 16-bit Words

CHARACTERISTICS

The characteristics section of the descriptor occupies four 64-bit words. The first three of these give information about the type of file, its size, and its structure. The last word is currently not used. Figure 4-3 shows the layout of the characteristics section.

Number of records (32)		Number of blocks reserved for file (32)	
Type (16)	File length in bits (48) [†]		
Reserved for future types	Pointer to further structure information in file (48)		
Not currently used			

Figure 4-3. Format of Characteristics Section of Descriptor

[†] If the file type is binary fixed length, then this field contains the record length in bits and not the file length.

The 64-bit words are divided into 16-, 32-, and 48-bit fields as shown.

There are 16 bits in the type field, of which 15 are currently used; a further 16 are kept for the addition of new file types. The types of files are listed below with the corresponding bit address in the type field.

<u>Bit Position in Type Field</u>	<u>Type of File</u>
0	undefined
1	coded delimited
2	coded fixed
3	binary STAR
4	binary fixed
5	foreign delimited
6	foreign fixed
7	virtual memory
8	drop
9	labeled
10	multiple volume
11	incomplete
12	permanent
13	input
14	output

An undefined file is one with a name, but of unknown contents. It provides a convenient way for the system to apply file functions to data collections which are not in the form of user files.

Coded delimited, coded fixed, binary STAR, binary fixed, foreign delimited and foreign fixed are all files associated with an input device, usually the card reader. In such files, there is an internal structure. The files are divided into records. For coded delimited files from the card reader, a record consists of an ASCII card-image terminated with a record separator character. In such a record, multiple blanks are compressed. In coded fixed files, the records consist of ASCII card images with no record separators and no compression of blanks. For the other four types, the records are of variable lengths. In binary STAR, record separators determine the length of records. In binary fixed, each card is a fixed-length binary record. Foreign delimited and foreign fixed files allow for punched cards from any

different computer systems to be processed. (These six types of card files are described in detail in Appendix B, Card Formats.) On creation of a file with variable length records; that is, for all except coded delimited and coded fixed, a file is also created that records the type and length of each record. This is called the map file and it is of type binary fixed, with a record length of 64-bits.

It is possible for a file to consist of any mixture of these six types. Such a file has all the relevant bits set in its type field. The map file associated with it identifies which type any particular record within the file is.

Virtual memory files and drop files contain structural information within them, so they use the structure pointer defined in the file characteristic. A virtual memory file consists of units of STAR pages which are multiples of 512 64-bit words. With each unit is associated a page address. When the file is read into STAR storage, the page addresses are set from these addresses. The file is thus an image of a virtual memory region. A drop file is created by the central operating system from a job in the machine. It consists of the program virtual memory pages with all of the current program status information. It is used by the operating system for suspending programs and moving them completely out of the central machine, either at the user's request or for system reasons. A drop file can be loaded and continue execution from where it was suspended.

A labeled file is one that has a USASI label at the start. Any file can be labeled, so the type bit 9 can occur with any others.

A multiple volume file is a file that is spread over a number of storage units. The system controls the mounting of successive volumes.

An incomplete file is a file that can be processed in parts. Such a file might be too large to be mapped into the central machine all at one time.

Files can be temporary or permanent. Temporary files are automatically deleted after output and after batch execution and are intended for use on a short-time basis. Temporary files can be converted into permanent files if the user is entitled to the amount of storage space involved.

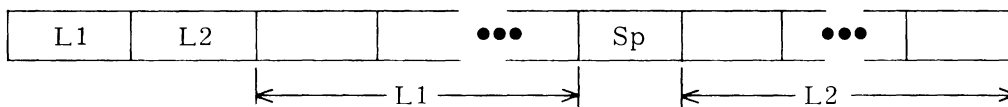
Input and output files are types internal to the system. When a file is being created from an input device and when a file is being sent to an output device, these type bits are set to allow interaction with the unit record station involved.

NAME

The name of a file, which uniquely identifies it, consists of two parts. Both are variable length strings of ASCII characters. The first is the name by which the file will be known and the second is the identification name of the owner. The strings are separated by the space character. The file name is not allowed to contain the characters for asterisk, slash, period, ampersand, vertical bar, or question mark (* / . & | ?) because these are reserved for special use. For example, the map file associated with files containing variable length records has the same name as the data file, except that it is followed by a reverse slash character. The period character is used to indicate a hierarchical structure within the name.

The name of the file is used to locate the file descriptor when the file is opened. The file has to be opened before it can be processed.

Figure 4-4 shows the layout of the name section of the descriptor.



- L1 number of bytes in file name
- L2 number of bytes in user identification name
- Sp ASCII space character, used as separator

Figure 4-4. Format of Name Field of File Descriptor in 8-bit Bytes

Example of file name with user identification:

MATRIX J249

In hexadecimal, the lengths and ASCII characters appear as:

06	04	4D	41	54	52	49	58	20	4A	32	34	39
----	----	----	----	----	----	----	----	----	----	----	----	----

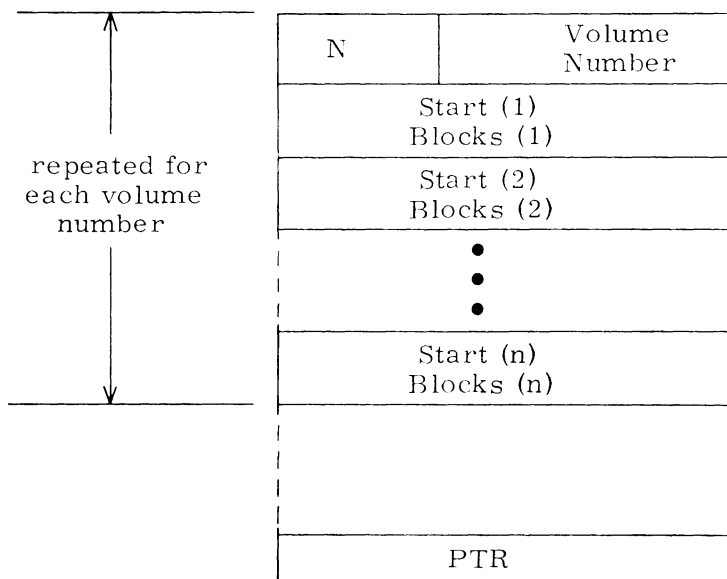
STORAGE MAP

The storage layout of a file varies with the particular storage device and is intended to optimize the performance of the devices. The current technique is to allocate the desired number of blocks in a contiguous fashion, or, when this is not possible, to allocate the total space in as few large sections as possible.

The unit of transfer is a block of 4096 bytes, and the current storage stations use either the 841 exchangeable pack disk drive or the large 817 disk drive. The layout of tracks on the 841 is a straightforward two blocks per track. The access time for an 817 disk file laid out in random blocks can take 100 times as long as for the same file laid out in contiguous blocks.

New storage devices requiring different layout techniques can later be introduced into the filing system. Because of this, the routine which allocates file space, known as RENT_STORE, is modularly replaceable and can be easily modified.

The storage map section of the file descriptor is a table kept to a fixed length. It is arranged as a number of 16-bit words, the last of which is used as a pointer to an overflow area if one is needed. Within the table, entries are grouped according to storage devices. Each group consists of a heading that gives the device identification and the number of sections of the file on the device. A section consists of one or more contiguous blocks. After the heading, there is a list of entries to give the starting block address and the number of contiguous blocks from that address for each section. The layout of the storage map is shown in Figure 4-5.



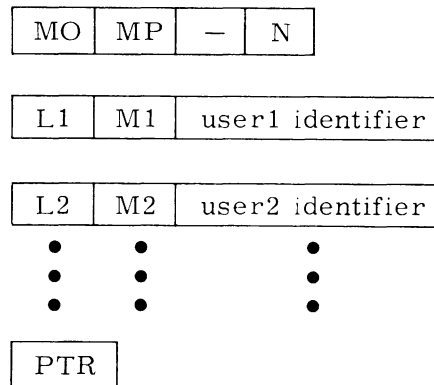
- N = number of sections of file on this device (8-bits)
- Volume Number = used with a table in the storage device label to identify the device holding these segments of the file.
- Start (i) = starting address of next section (16-bits)
- Blocks (i) = number of contiguous blocks (16-bits)
- PTR = pointer to overflow area for map, if required

Figure 4-5. Format of Storage Map Section of the File Descriptor in 16-bit Words

ACCESS LIST

The rights of a user to access a file are checked when the user initially opens the file. The open file function, like all file functions, involves sending a message to the storage station. The open file function passes the file name, the owner identification, the user identification, and the access mode required for this run as parameters. These are passed as ASCII character strings that are separated by a space character and terminated by the record separator character. If the owner is the user, the user identification field can be omitted. On receipt of this open file message, the user's access rights are checked against the access list in the file descriptor. If access is not permitted, an invalid access response is sent. If it is permitted, the run mode of access is recorded in a table called the active file table and the file is given an index number for use in future messages. Thereafter, for other file functions, the validity of the required operation is checked against the mode stored with the file entry in the active file table.

The layout of the access list is shown in Figure 4-6. It is a byte-organized table of fixed length, with the last two bytes used to give a 16-bit pointer to an overflow area, if an overflow area is required.



- MO = mode of access of owner
- MP = mode of access of public
- N = number of entries in the table
- M1, M2... = mode of access for user1, 2, ..., n
- L1, L2... = number of bytes in identifier for user1, 2, ..., n
- PTR = pointer to additional access list (final 16 bits in table)

Figure 4-6. Layout of Access List Section of File Descriptor in 8-bit Bytes

The first four bytes contain the owner access mode, the general public access mode, and the number of entries in the table. Each entry is in three parts, and contains a user identifier, the number of bytes in the identifier, and the access mode allowed for that user.

The access modes are:

- read only
- write only
- allowed to delete file
- allowed to alter access modes

These modes are set or modified by the access mode messages. The default option on creation of a file is that the owner has open access and the public has no access.

It is possible for a file to be open to more than one user at a time if all but one of these opens are in read only access mode.

MESSAGES

The file functions recognized by the storage station are as follows:

- create and open file
- open file
- close file
- close and delete temporary or permanent file
- close and delete temporary file
- keep file
- set file characteristics
- set file length†
- determine if file is open
- read file pages
- write file pages
- read file descriptor
- write access list entry†
- delete access list entry†
- modify owner and public access
- mount†
- dismount†

† Not yet implemented.

These file functions are all system functions issued by the monitor. When the user in the central machine makes a file request to the central monitor, a monitor task formats the message or messages for the storage stations. These messages each consist of a function code and a list of parameters passed in several different formats, and each causes a file function to be executed in the storage station. (Full details of the messages can be found in Appendix B.)

No file function is legal until the file has been opened. In the open file function, after the file has been identified and the user access validated, the file is given an active file index number. This number is used to identify the file in all further messages and is used by the storage station to index the active file table. The active file table is kept in the station buffer unit. In this table, each entry contains a pointer to the file descriptor, an access mode indicator, an identification of the file device, the starting address of the file, and the number of file blocks contiguous from this starting address. Each entry is 64 bits long, and the format is shown in Figure 4-7. The last file function is a close operation.

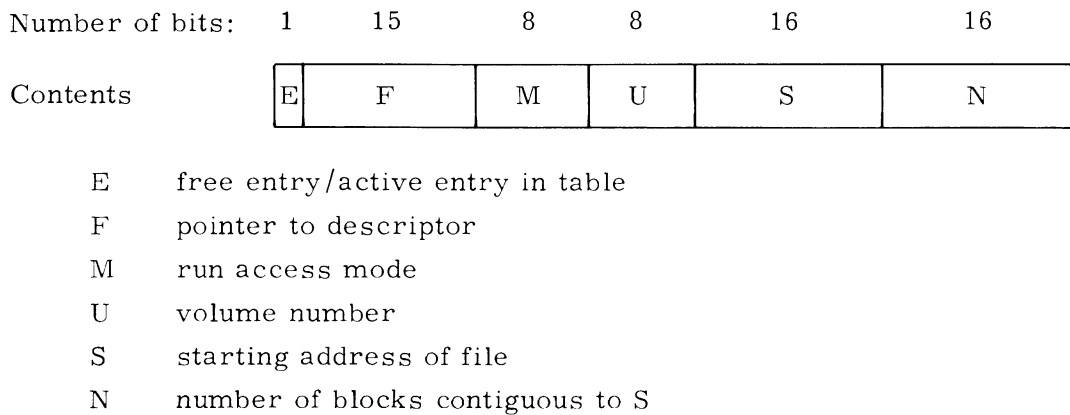


Figure 4-7. Format of the Active File Table

Using the active file index avoids sending variable length names in messages and the active file table allows rapid checking and execution of file functions.

Each message has a separate overlay program to process it. These are conditionally core resident in the station control unit, so, if any particular message is used frequently, it will remain in core storage. Otherwise, the overlays are loaded from the station microdrum as required.

FUTURE FEATURES

The following features are not yet implemented but they are listed here to show the way that the file system is developing. All the features here are planned for in the system design.

- Automatic mounting The allocation of drives, mounting and unloading of removable units such as disk packs and magnetic tapes, and the checking of labels are performed at the storage station. The standard label is the USASI standard.
- Multivolume files Multivolume files that occupy more than one storage unit are provided for by keeping the volume number in the file descriptor and by using the automatic mount/dismount facility.
- Archival file directory The present base supports one central file directory for all files, on-line and off-line, associated with a particular storage system. This directory can be kept on the storage station, processor station, or service station. The archival file directory is listed by owner identifier and provides a backup if the working directory is destroyed.
- Structured file name and owner/user identifiers The ASCII period character is reserved to indicate subsidiary files in a hierarchical file system, but the routines will continue to ignore this feature until the exact meanings and uses of the various parts are more clearly defined. One aspect of structured names and identifiers is the linking of files of a given class into a tree structure. An even more complex aspect is an access mechanism that involves grouping users into different access classes.

- **Shared-access security** The access mode conditions (no read, no write, no delete, no modifying access mode) are adequate at present. Shared access is only allowed on read-only files. When the record management subsystems define their shared access protection schemes, this system can be extended. One mechanism for allowing shared reading and writing of the same file is for updates to go to new file versions.
- **File editions** Different file editions involve mechanisms that allow the user to specify edition numbers or default to the latest edition, allow for these edition numbers in the message formats, and provide the ability to link the different editions in the storage station. Rather than have separate fields in the message formats for edition numbers or even new messages, a possible scheme is to append to the local name a slash character followed by a 2-digit edition number, thereby allowing 100 editions. Three slashes might indicate the latest edition, and, in this case, the edition number used could be returned with the response.
- **Error recovery and backup** The present basic system allows more sophisticated schemes to be superimposed on the present recovery and backup system. Error files are kept in the storage station itself, and the station software decides what to do on an error. Only when different recovery strategies might be invoked is the associated processor station asked for advice. When a sufficiently large backup store is available, the storage station will periodically dump archive files to it as backup.

- Accounting and performance statistics

The exact nature of extra accounting and performance data to be acquired has yet to be determined. The storage station only gathers and records the standard station accounting and usage statistics at present.

- File activity record

The activity record section of the file descriptor records the creation and expiry dates for the file and some information about its usage. The section occupies six 32-bit words, the last two of which are currently not used. The layout is shown in Figure 4-8. It is expected that different installations will require different information recorded about file usage. The activity recording routines have, therefore, been kept simple and modular.

Creation date/time	
Expiry date/time	
Last update date/time	
N	Not used
Not used	
Not used	

N = number of opens

Figure 4-8. Format of Activity Record Section of the File Descriptor, in 32-bit Words

In the STAR system, the file system consists of two separate parts; one that is internal to the central machine and one that is external. The external, or storage management part, is described in Section 4. It includes the control and organization of peripheral storage devices and the functions of file cataloging, reliability, recoverability, and protection. The internal, or file management part, is described in this section. It is concerned with how the user accesses his files within the central machine.

ACCESSING FILES BY MAPPING

One method of access has been described earlier, and that is the operation of mapping. A file can be mapped into the central machine storage by a system call that informs the monitor of the file name and a starting virtual address. The monitor sends messages to the storage station holding this file and, knowing the file size from the responses, associates a range of virtual addresses with the file from the given starting address. Future accesses to new blocks of memory in this range of virtual addresses cause new blocks of the file to be read into central storage by the normal demand paging mechanism coupled with the mapping function. This way of accessing files is sufficient for many cases. It has the advantages of simplicity and economy, both in time because there are no extra software overheads involved in obtaining new blocks from the external file, and in space because only file blocks referenced are brought into the central machine.

The operating system functions for renaming page addresses, deleting and writing pages out to external files, and the advice functions such as for the prereading of blocks, can all be used to support this type of file access. Inasmuch as the access is completely data independent, the user can employ any arbitrary data structures.

RECORD ACCESS

As well as the mapping form of file access, the operating system has to manage the usual file support and facilities required by high level languages. Conventionally, such input and output files consist of basic subdivisions called records interspersed

with record separator characters and, possibly, other control information. The name of unit record management is given to the software structure which controls these files and provides such functions as read, write, backspace, skip, and re-wind.

Consider the FORTRAN statement

```
READ TAPE3 A, B, C
```

where A, B, and C are real scalars.

The statement implies that there is a file known to the program as TAPE3, probably with a different external name, that contains at least one more record with a minimum of three elements in it. To satisfy the read, three consecutive words are extracted from the file and moved to become the values of A, B, and C. If there is any element left in the record, it is skipped over so the next read function is positioned to start at the next record. In some computer systems, such a statement causes the access of one record from whatever external media is being used to hold the file; for example, from magnetic tape, possibly with some buffering of transfers ahead of the current requests. The retrieval is assisted by some file structure information, such as end-of-file marks on magnetic tape.

In the STAR system, every file in use resides in virtual memory. For all but the simplest type of file, fixed-length binary-coded-decimal records, some means of representing the structure of the file data has to be provided. For fixed-length records, the record length can be passed to the input/output routines, and such functions as reading are implemented by moving data from one area of virtual memory to others, with advancement of the file index by one record and a check to see if the file is at its end.

RECORD MAP FILE

The problem of identifying record and subfile boundaries becomes more complex when files are allowed to contain variable length binary data or mixtures of different types of records. One header word is needed for each record to give its type and length. In some systems these headers precede each record in the file. This is not the case in STAR, for two reasons. First, such headers create data discontinuities that might prevent the efficient use of string or vector streaming instructions.

Second, backtracking over records on initial input to insert the lengths in the headers is unsatisfactory and, for long records, unnecessarily complex. The method adopted provides a separate file of header words, called a record map file. This file consists of 64-bit entries, one for each record in the associated data file. Each entry gives the type of record and its bit address relative to the first word of the file in virtual memory. The number of entries in the list is found in the appropriate entry of the file descriptor, as described in Section 4. The record map file has the same name as its associated data file except that it also has a reverse slash at the end. It appears the same as any other file to the file system and can be processed independently of its data file if necessary.

The map is generated or updated by the programs which act on the file. For example, in the unit record station, when a file is being created from a set of inter-mixed binary coded decimal and binary cards, the map file is created in parallel. The map is also used by the peripheral system on output to direct changes of mode and to place control characters on the output medium. Mode changes might include the change of binary-coded-decimal to ASCII-to-binary, and control characters include end-of-group and end-of-record marks or cards.

Note again that within the central machine, the map is only required when the file is complex. In the simple cases, such as fixed-length input from punched cards, the file descriptor provides all of the information needed to process the file.

Appendix B gives details of the different types of records that the system uses, shows the layout of map entries, and gives examples of card deck files.

FILE RECORD MANAGEMENT TABLE

A file record management table is kept for each active file used by a program. The table, referred to as an FRMT, contains all the information needed to control the file. It holds details about the file and record location, the file status, the last operation on the file, and actions to be taken on special conditions. Figure 5-1 shows the layout of the file record management table, which occupies 19 full words; the entries are described below.

word	0	15 16	31 32	63
0		Record address		
1		Record index		
2	Status	Number of records		
3	Active file index	Record length		
4	File access mode	Input/output index		
5	File type	File address		
6	Last operation	File length		
7	Active map file index	Map index		
8	Map file access mode	Map file address		
9	Initial number of records		Initial number of record blocks	
10	Initial type	Initial file length/record length		
11	Internal name			
12	External file name descriptor			
13	External map file name descriptor			
14	Structure descriptor			
15	Label descriptor			
16	On condition descriptor			
17	Forward descriptor			
18	Backward descriptor			

Figure 5-1. Layout of the File Record Management Table

The record address is the address in virtual memory of the next record to be processed. The record index is the number of the next record to be processed.

The status of the file is given by the lower seven bits of the status field as follows:

Bit Position	Value = 1	Value = 0
15	file is open	file is closed
14	file exists	file does not exist
13	end of file on last operation	no end of file encountered
12	file is mapped in	file not mapped in
11	file is mapped out	file not mapped out
10	file is released	file not released
9	file is stored	file not stored

If the file is mapped out, then it will be written back to the permanent file on completion of the program.

The number of records gives the total number of records in the file.

The active file index is an internal index to the file, given to it by the file system when the file is opened. It is used instead of the file name for all requests to the storage station.

The record length is the bit length of records if the file consists of fixed length records; it is zero if the records are not of fixed length.

The file access mode controls the protection given to the file. The modes are read only, write only, or read and write.

The input/output index is an internal index, based on the file type, that is used to select the appropriate routines to process file requests.

The file type gives the type, as specified in the characteristics section of the file descriptor.

The file address is the virtual memory starting address of the file.

The last operation field gives details about the last operation performed on the file. Bit 15 is set after successful completion, so the 16-bit field is even if the last operation is still in progress and odd if the file is ready to accept a new command.

Value	Last Operation
0	Open
2	Exit
4	Read
6	Write
8	Rewind
10	Skip
12	Backspace
14	Copy
16	Release
18	Close

The file length is the bit length of the entire file.

The active map file index is the internal index given to the map file by the file system when the file is opened.

The map index is the index within the map file of the next record to process.

The map file access mode controls the protection given to the map file; read only, write only, or read and write.

The map file address is the starting virtual address of the map file.

The initial number of records, record blocks, the initial file type, length and record length, are obtained from the file characteristics when the file is first opened. If the file did not previously exist, these entries are set to zeros.

The internal name is the name that is used by FORTRAN programs.

The external file name and external map file name descriptors point to the file names in the file name string table described below.

The structure descriptor gives the start and length of a table that defines the structure of subfiles.

The label descriptor gives the start and length of a field defining the file label.

The on condition descriptor gives the start and length of an address vector. Entries in this vector are for routines to be entered on special end case conditions.

The forward descriptor gives the FRMT of the next file, or zero if there are no more files. It is used for abnormal termination conditions to recover information in files.

The backward descriptor gives the FRMT for the previous file in the active list, and is used for the same purposes as the forward descriptor.

ORGANIZATION OF A USER'S FILE TABLES

The data space for a user consists of the virtual memory allocations he wishes to make and two other areas called static and dynamic space. Static space is used for the fixed-size data requirements of programs. The FORTRAN common space, for example, and for tables such as the file record management tables. A system of pointers at the beginning of static space points to such essential tables.

All FRMTs for a program are grouped together in static space. Three of the pointers at the start of the static space are used to keep track of the file information. The address fields of these pointers give the starting addresses of three tables, called the file pointer table (FPT), the file name table (FNT), and the file name string table (FNS). The length fields of the pointers give the number of files active, the maximum number of files that any single user can maintain, and the length of the file name string table. The maximum number of files is an installation parameter. The three tables are used in conjunction to establish the location of the file record management table for any particular active file.

The file name string table contains character strings giving the external name of every active file. Each string is preceded by an index number, contained within one byte, and terminated by the record separator byte. Scanning this list will result in finding the index to any active file. This active file index is then applied to the file pointer table. The file pointer table contains one full word entry for each active file. Each word is a descriptor giving the starting address and length of the file record management table for that file.

The file name table is used only for FORTRAN files. Each entry is one full word and consists of the internal file name. This is a character string of up to eight bytes, left-justified and blank filled. The list is scanned word by word, and the index derived from a successful search is again used as an increment to the file pointer table base address.

Figure 5-2 shows the layout of the three static space pointers and the three tables. The address of the pointers is determined from the main static space pointer. (Section 7 describes this process.)

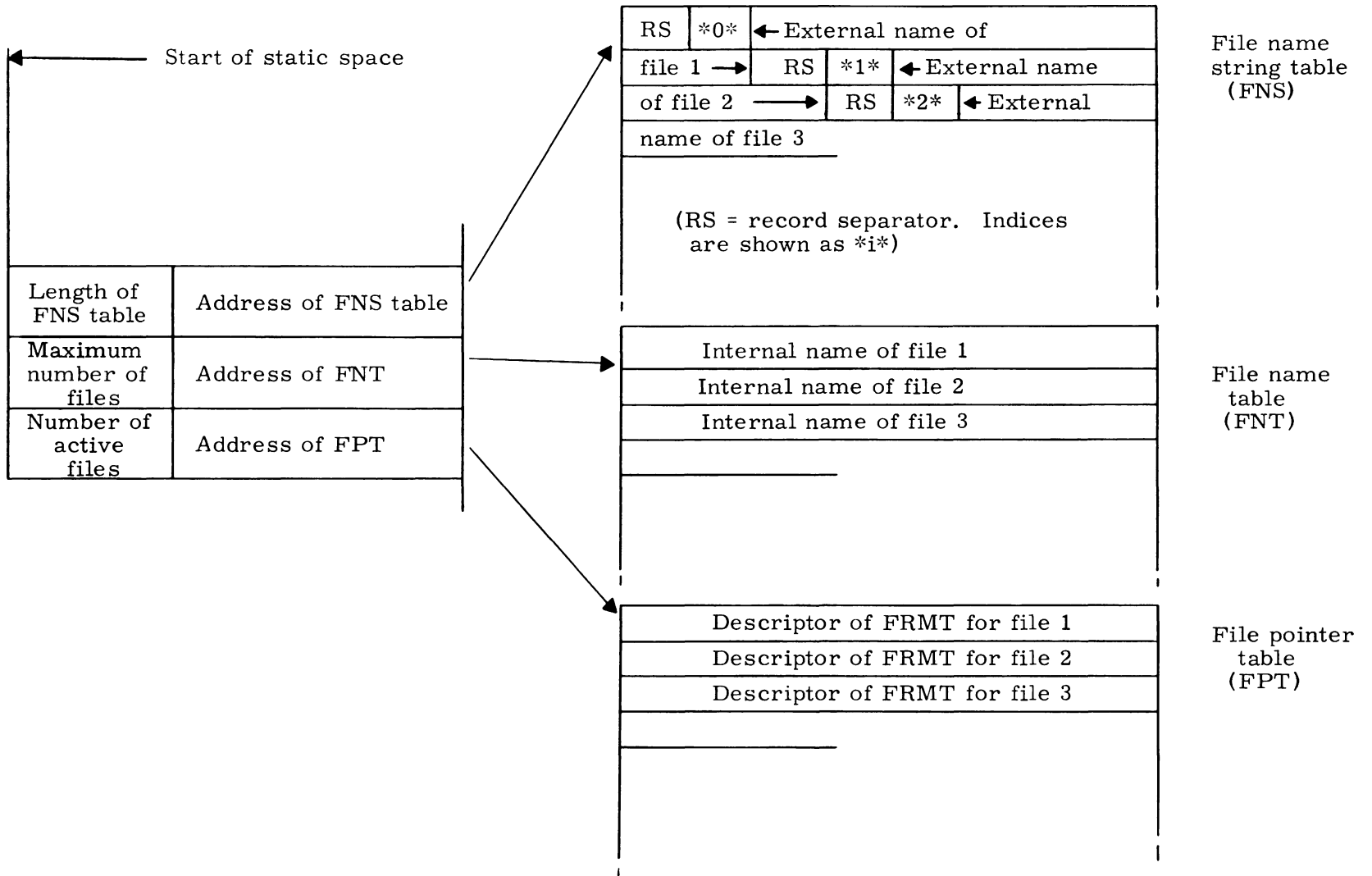


Figure 5-2. Pointers and Tables for Locating File Record Management Tables

INPUT/OUTPUT FLOW

At the beginning of a FORTRAN program execution, an input/output routine named Q8ENTRY is called to initialize the FRMT tables. Its first action is to resolve the correspondence between external and internal file names.

A typical program might have its first card:

```
PROGRAM TEST (INPUT, OUTPUT, TAPE3)
```

In which case the job control card might appear as:

```
TEST (FILEA, FILEB, FILEC)
```

Q8ENTRY builds in static space an FRMT table for each file. It then allocates twice the reserved file length in static space. If the file does not exist yet, a default length is allocated for it.

If a map file exists, determined by the file type shown in the characteristics, the corresponding map is attached and mapped into an allocated region of static space. The proper input/output jump address is inserted in the input/output index word. This process is repeated for all files required.

The same functions are performed at every appearance of an OPEN statement in a COBOL program.

At object time, the occurrence of a READ statement

```
READ TAPE3 A, B, C
```

causes entry to the module called READ with TAPE3 passed as the first parameter. READ searches the file tables for the corresponding internal file name. After some preliminary setup, READ then jumps to the address found in the input/output index word of the FRMT.

After moving the data as required by the read statement, a final routine updates the virtual memory index and checks for end-of-file conditions.

WRITE, SKIP, and BACKSPACE are implemented in a similar manner.

Random and random-sequential files are handled by maintaining a relationship between keys and the virtual addresses of the records. The structural information

defining this relationship is in the data file. The key generation and searching algorithms are executed in auxiliary modules to give compatibility with existing access methods.

At the end of the job, the files are all closed, and the data in them is stored on a mass storage device. The FRMT entries remain to allow further manipulation of the data.

INPUT/OUTPUT FUNCTIONS

This section gives a short description of the input/output programs in the library.

<u>Function</u>	<u>Purpose</u>
Q8ENTRY	initializes the FRMT tables for all requested FORTRAN files. It maps the files into either static space (default) or into a user requested area.
OPEN	sets the FRMT table to allow the user to manipulate the data to or from the file.
STORE	moves the data from virtual memory to a mass storage device for files with proper access mode.
CLOSE	sets the FRMT status so that no manipulation of the data is allowed.
READ	moves data from the file in virtual memory to areas specified by the user.
WRITE	moves data from user designated areas to the file in virtual memory.
REWIND	resets the FRMT entry for the file to beginning of information.
SKIP	skips requested number of records in the file or to end of information, whichever occurs first.
BACKSPACE	backspaces the requested number of records or to beginning of information, whichever occurs first.
FIL_STAT	displays FRMT entry for the file.
FILESTAT	returns the base address of the FRMT entry of the request file.
BAIL-OUT	closes the FRMT entry for all the files used by the job. It also calls STORE to save all the data generated by the job.

A large number of languages and compilers for STAR are under investigation. These include APL, PL-1, ALGOL-60, ALGOL-68, FORTRAN and COBOL, as well as some special-purpose languages.

A major part of this investigation has been into compilation techniques and programming concepts. This section provides a brief introduction to assembly languages for STAR and the buffer controller computer and describes some of the experimental work on FORTRAN.

One direction taken by the language investigation is toward having a single language processor capable of compiling a set of languages. The PL/* assembly language has been implemented within this framework, such that PL/* statements can be intermingled with higher level language statements. Part of the rationalization for such a language processor is that it is possible to reduce production time and maintenance needed, as well as the total amount of code, by taking advantage of commonality among different compilers.

Another direction of investigation has been into parallel techniques in compilation. It is possible to develop a FORTRAN compiler which uses vector instructions to linearly process the input stream. That is, it transforms the input stream, in parallel, into successive data sets until it finishes with optimized machine code.

A further topic is the subject of Decompiling. In many cases, an entire FORTRAN DO loop can be replaced by a single STAR instruction or by a simple, in-line series of instructions. The STAR instruction repertoire includes instructions which are at a higher level than any current language. The problem of recognizing such higher level language sequences and replacing them with simpler, in-line code is complex but must be approached as one way that STAR can be better exploited by existing FORTRAN programs.

Finally, BUFFALO, the algebraic assembler for the buffer controller, is described. There is more than one assembler for the buffer controller. BUFFALO is an attempt to provide a free-format, extendable system which can evolve as required. Appendix I gives a complete definition of the language.

PL/*

PL/* is a free-format, algebraic assembly language for the STAR central processor. PL/* is intended to form a language processing system which can process many languages and also allow the intermingling of assembly language statements with compiler language statements. There will be many individual language compilers in addition to the PL/* assembler. Within the framework of such a language system, PL-1 has been chosen as the primary language for the following reasons:

- PL-1 compilation includes solutions to most compiling problems of FORTRAN, COBOL, ALGOL-60, and other commonly used languages. Thus, dealing with PL-1 first essentially resolves the compiling difficulties of most other languages with the exception of ALGOL-68.
- Many of the data types and structures such as strings and arrays applicable to STAR can be easily handled by PL-1.
- In terms of standardization and coding investment, PL-1 is still flexible enough to allow further development of the language. In particular, APL¹ notation additions would allow experimentation in new coding techniques.

The following general points are highlighted as an introduction to the PL/* Assembly Language, described in detail in Appendix J.

- Wherever possible, the rules for PL-1 construction are followed. In particular, this holds for declarative statements and for reference to attributes, constants, and abbreviations.
- PL/* instructions can be intermingled in higher level languages. A dollar sign (\$) preceding a PL/* statement flags it to be an explicit machine code instruction for direct assembly and not for compilation.
- Symbol name lengths are unrestricted, and there are no reserved words.
- There are, however, special or unusual symbols limited to use in special or unusual operations.
- The extensive use of the full set of ASCII delimiters is intended to reduce the number of symbols to be coded, stored, and printed.

¹Iverson, Kenneth E. A Programming Language
John Wiley & Sons New York 1962

- Some delimiters are restricted in their use, so their function is always apparent. For example:

- . represents catenation
 - , represents a pairing of registers to describe a field
 - [] represents "address of"

- The conventions adopted for default cases are again intended to reduce the amount of coding. For example, a single name is used to represent both the storage address of a character string and the register holding the descriptor.
- The majority of the instructions are recognized by the natural placing of the operators. For example:

- \$ A =U B + C; is ADD UPPER.

The type of addition is then implied by the data types which have been previously declared.

- Register file management can be explicitly directed or left to the compiler. Several directives are provided to assist in register handling. These include ORGR, ORGW, FREE, FREEZE, REGBLOCK, and ENDBLOCK.
- Table construction is aided by the TABLE and STABLE directives.
- Compile time variables allow the programmer to manipulate and test origin counters and symbol type flags. Such statements are preceded by the percent sign (%) as in PL-1.
- There are several built-in macros, provided to mechanize the conventions of CALL, ENTRY, LOAD, FORMAT, etc.
- PL/* has macro processing capability based on the macro language ML-1.

A full description of PL/* is given in Appendix H.

FORTRAN

A major concern of the STAR development project has been the utilization of STAR's unique architecture in the execution of conventional FORTRAN object code. The large register file and wide variety of register instructions make a conventional implementation of FORTRAN on STAR quite simple. However, the major power of STAR is not realized until vector and other streaming operations are employed.

Two alternative methods of FORTRAN development have been investigated. The first, mentioned briefly under the discussion of PL/*, is the development of a full FORTRAN language facility within the PL/* system. Of necessity, such a compiler can do little optimization because the programmer is allowed complete freedom to manipulate compiler counters and to insert machine code wherever he likes. Thus, object code is limited to register-to-register operations, providing a system programming aid, but with string and vector instructions effected only through explicit coding of machine instructions. However, in such a FORTRAN environment, it is possible to locally optimize areas of code.

The other method of FORTRAN development under investigation is radically different and might be termed the parallel approach to compiling. It is based on the concepts and techniques described by Information Algebra and by APL.

This section discusses how such concepts are being applied to the compiling of FORTRAN for STAR. Although FORTRAN is used for the example, the techniques are applicable to any compiler.

The STAR FORTRAN compiler will implement a language as identical as possible to 6000 RUN FORTRAN. Additional language features will be implemented to make STAR more accessible to the programmer. The compiler is written in PL/* and generates relocatable STAR binary code, compatible with STAR-100 operating system.

FORTRAN source code is reduced by a series of vector operations to a stream of operators which is viewed as a numeric vector. A syntactical analysis is then performed using the numeric operator vector, to produce output code. The sequence number of the operator triggering object code is attached such that code may be merged in its proper sequence. The sequence number also plays an important part in code optimizing routines.

The front end of the compiler readies source code for syntactical analysis by producing an operator vector. Several of the key operations performed in producing this vector are: remove blanks, comments, continuation flags and card sequence numbers; interpret nonexecutable statements; resolve labels; identify keywords; convert constants; identify symbol types; assign array space; form symbols vector containing unique names; identify function calls; assign constants and variables with initial REGBLOCK location, etc.

The tail end of the compiler consists of syntax tables, code optimizing routines, and routines to build object modules and tables consistent with STAR-100 operating system. Code is produced by using the operator vector to search through syntax tables containing operator precedence numbers and instruction skeletons. Code is emitted in parallel in order of precedence and merged with the code stream. Code optimizing routines may be invoked during or at the end of the merging process.

To illustrate the use of vector instructions in the FORTRAN compiler, four card images will be followed from the middle of a FORTRAN program through several phases in detail.

```

    A = B + C*D
  C COMMENT
    D = E**F
  1 + E*C

```

ASCII conventions are used for format characters and the abbreviation EOL is used for end-of-line. We assume that the start of the first card is character position 325, and the card layouts appear as below (where the numbers give the character positions) in Figure 6-1.

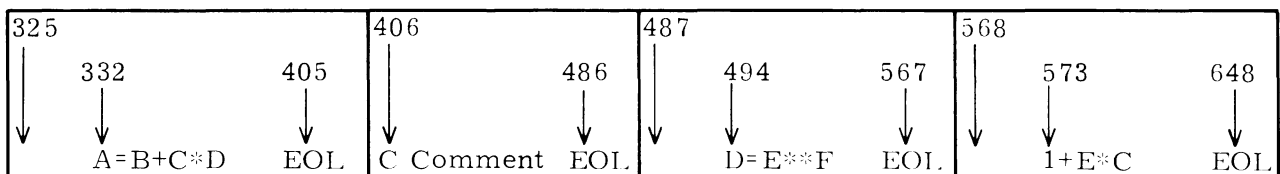


Figure 6-1. Card Layouts

The first step is to translate the original source text into a revised format with the following objectives in mind:

1. To remove all blanks and unnecessary format information (such as form feed characters).
2. Rearrange the collating sequence so that alphabetic and numeric characters are contiguous and include the period.
3. Attach to each converted character its original character position in the source stream. Any sequence numbering system is acceptable; however, attaching the character string position makes error processing and recovery easier.

Converted Source	EOls Selection Bits	EOls & Adjacent Select Bits	EOls Position and Adjacent Bits
A 332	0	0	
= 333	0	0	
B 334	0	0	
+ 335	0	0	
C 336	0	0	
* 337	0	0	
D 338	0	0	
EOL 405	1	1	405
C 406	0	1	406
C 410	0	0	486
O 411	0	0	494
M 412	0	0	567
M 413	0	0	573
E 414	0	0	648
N 415	0	0	000
T 416	0	0	
EOL 486	1	1	
D 494	0	1	
= 495	0	0	
E 496	0	0	
* 497	0	0	
* 498	0	0	
F 499	0	0	
EOL 567	1	1	
1 573	0	1	
+ 575	0	0	
E 576	0	0	
* 577	0	0	
C 578	0	0	
EOL 648	1	1	

Figure 6-2. Calculating EOL Positions

Once the string is converted and compacted, all comments must be removed so that no extraneous operators or symbols appear in the remaining intermediate

streams. This is accomplished with the function called "selection" wherein a bit vector is created containing a zero bit wherever a given comparison criteria is met. In this first case, the string of converted bytes is searched for a translated EOL character. The resulting selection vector appears in the second column with a one bit for each EOI found. See Figure 6-2

Next is performed the bit-by-bit logical OR of the selection vector with itself, offset by one bit position. The resulting string (in column 3) now contains two bits for each EOL representing the position of each EOL in the converted byte string and the next adjacent, significant (non blank, non format) byte. The desired data from the converted byte stream is compressed by use of the selection vector in column 3. Compression creates a stream containing only elements of the original stream corresponding to 1 bits in the selection vector. In this case, the bit string in column three is used to remove each EOL and adjacent byte (along with the character position sequence number) from the converted byte stream to form the more compact data stream in column 4.

The compressed stream is then arithmetically subtracted from itself offset by one place to produce a stream showing the relative displacements of the EOIs and adjacent bytes. See Figure 6-3.

<u>From column 4 of Figure 6-2</u>	<u>(Offset by one byte) +</u>	<u>Relative Displacement</u>	<u>Comments Selection Vector</u>	<u>Continuation Selection Vector</u>
405	406	1	1	0
406	486	80	0	0
486	494	8	0	0
494	567	73	0	0
567	573	6	0	1
573	648	75	0	0
648	000	-648	0	0
000				

Figure 6-3. Establishing Comment and Continuation Lines

To simplify what follows, assume that any non-blank data appearing in column one signifies the beginning of a comment and that any non-blank data appearing in column 6 signifies a continuation card. Again utilizing the selection function, the relative displacement stream is searched and a one bit is set in the selection vector wherever a one appears in the scanned stream. Likewise, the relative displacements are searched for all displacements of 6 (signifying a continuation card).

As can be seen from Column 3 of Figure 6-4, the comment bit appears now to correspond in relative position to the EOL in the converted source stream immediately preceding the comment card. The expansion of the continuation stream produces a similar result.

Removed next from the converted source are all bytes contained within the comment card, the EOL preceding a continuation card, and the byte in column 6 of that card. To remove all bytes in comments, a bit string is built to be used as a selection vector for compression between the EOL preceding the comment and the EOL terminating that card. To accomplish this, the expanded comment bit string and EOL bit string are viewed as arithmetic entities with the first of the string appearing to the far right.

EOL Bits	100	000	100	000	010	000	000	010	000	000
Comment Bits	000	000	000	000	000	000	000	010	000	000

Then is performed a "NAND" or inhibit operation -

RESULT = EOL bits AND NOT comment bits where AND and NOT are
BOOLEAN operations.

RESULT 1:	100	000	100	000	010	000	000	000	000	000
and subtracting the comment bits	-000	000	000	000	000	000	000	010	000	000
RESULT 2:	100	000	100	000	001	111	111	110	000	000

then perform a
NAND of RESULT 2 with RESULT 1 to eliminate extraneous EOI. bits
RESULT 3 - RESULT 2 AND NOT RESULT 1

RESULT 3:	000	000	000	000	001	111	111	110	000	000
-----------	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

A mask has been formed for the removal of the EOI and every significant byte on the comment card. To remove the EOL preceding the continuation card and the adjacent column 6 data, all that need be done is to perform the logical OR of the expanded continuation card selection vector with RESULT 3 and the OR of the continuation card stream offset by one bit;

RESULT 3:	000	000	000	000	001	111	111	110	000	000
<u>OR</u> expanded	000	000	100	000	000	000	000	000	000	000
continuation										
<u>OR</u> expanded	000	000	010	000	000	000	000	000	000	000
continuation										
offset by one										
bit										
RESULT 4:	000	000	110	000	001	111	111	110	000	000

The converted source stream is compressed using the NOT of bit string RESULT 4 taken from right to left as shown in Figure 6-5.

<u>Converted Source</u>	<u>NOT RESULT 4</u>	<u>Compressed Source</u>	<u>Alphanumeric Selection</u>	<u>Alphanumeric Stream</u>
A 332	1	A 332	1	A 332
= 333	1	= 333	0	B 334
B 334	1	B 334	1	C 336
+ 335	1	+ 335	0	D 338
C 336	1	C 336	1	D 494
* 337	1	* 337	0	E 496
D 338	1	D 338	1	F 499
EOL 405	0	EOL 486	0	E 576
C 406	0	D 494	1	C 578
C 410	0	= 495	0	
O 411	0	E 496	1	
M 412	0	* 497	0	
M 413	0	* 498	0	
E 414	0	F 499	1	
N 415	0	+ 575	0	
T 416	0	E 576	1	
EOL 486	1	* 577	0	
D 494	1	C 578	1	
= 495	1	EOL 648	0	
E 496	1			
* 497	1			
* 498	1			
F 499	1			
EOL 567	0			
1 573	0			
+ 575	1			
E 576	1			
* 577	1			
C 578	1			
EOL 648	1			

Figure 6-5. Removal of Comments, Editing of Continuation Lines, and Extraction of Alphanumeric Data

At the outset, it was indicated that the original input stream was converted so that the collating sequence was reorganized. A possible scheme using the 8-bit byte (which can carry a binary value of from 0 to 256) could be:

$$\begin{array}{rcl}
 A \rightarrow Z & = & 200_{10} \rightarrow 226_{10} \quad * & = & 80_{10} \\
 0 \rightarrow 9 & = & 100_{10} \rightarrow 109_{10} & \text{other} & \\
 & & & \text{operators} & = & 1 \rightarrow 70_{10} \\
 . & = & 150_{10} & & &
 \end{array}$$

A selection vector can be constructed for alphanumeric data (including the period) by scanning the compressed source for all values greater than 100_{10} . This bit vector appears in column 4 of Figure 6-5. With this vector, all alphanumeric bytes and their corresponding sequence numbers can be extracted from the compressed source stream. If this selection vector is inverted by a BOOLEAN NOT operation, it can be used to extract all of the other information (operator and punctuation symbols) from the converted source stream. At this point, the converted source string may be abandoned.

In the operator vector, each element represents a unique operation or punctuation symbol except in the case of the double asterisk which represents exponentiation. These elements must be converted to a single element operator before proceeding.

Operator Stream	Asterisk Selection	Asterisk Stream	Subtracted Stream	Selected Exponent	Expanded Exponent	Updated Operators	Compressed Operators
= 332	0			0	0	= 332	= 332
+ 335	0			0	0	+ 335	+ 335
* 337	1	337	160	0	0	* 337	* 337
EOL 405	0	497	1	1	0	EOL 405	EOL 405
= 495	0	498	79	0	0	= 495	= 495
* 497	1	577	-577	0	1	↑ 497	↑ 497
* 498	1				0	* 498	EOL 567
EOL 567	0				0	EOL 567	+ 575
+ 575	0				0	+ 575	* 577
* 577	1				0	* 577	EOL 648
648	0				0	EOL 648	

Figure 6-6. Extraction of Operators and Punctuation Symbols

Figure 6-6 demonstrates the manner in which the operator vector could be updated with single element EXPONENT operators. The method is similar to that previously discussed -

1. Selection of asterisk values from the operator vector creating the bit stream in column 2.
2. Compression of those asterisk elements (particularly the sequence numbers) from the operator vector yielding column 3.
3. Mapping (subtraction of adjacent elements of column 3) to determine the relative displacements of the asterisks.
4. Selection of all displacements equal to one (signifying catenated asterisks).
5. Expansion of the selection vector in column 5 (exponentiation operators found) by the selection vector in column 2.
6. Merging of a constant (\uparrow) for all ones in the expanded selection vector into the operator vector.
7. Compression of the resulting operator vector by the selection vector in column 6 - offset by one bit.

There are a number of operations necessary to perform on the Alphanumeric Stream, which was created in Figure 6-6, before the symbol table can be constructed.

1. Isolation of all logical and selectional operators (EQ., NE., NOT., etc.) removed from the alphabetic string and merged into the operator vector.
2. Isolation of all constants.
3. Detection of all KEYWORDS such as DO, IF, and ASSIGN.

To facilitate this activity, it is desirable to identify the beginning of each symbol string. If the BOOLEAN NAND, or inhibit, function is applied to the alphanumeric selection vector from Figure 6-5:

RESULT 6 = ALPHANUMERICS AND NOT ALPHANUMERICS (+1)

The second operand is the first operand full offset by one bit. From left to right there obtains:

	1 0 1 0 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0
	0 1 0 1 0 1 0 1 0 1 0 1 0 0 1 0 1 0 1
RESULT 6:	1 0 1 0 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0

This result vector can be used to select from the alphanumeric stream all first symbols (of each string) for the determination of whether a numeric string or alphabetic string is present. Operations to isolate the logicals and relationals would use the previously described technique beginning with the selection of all periods in the alphanumeric stream.

The result of the initial steps outlined above is to produce four parallel vectors:

1. The OPS vector consists of a string of numeric values representing the presence and type of operator or operand in the executable portion of the source program.
2. The ATTRIBUTE vector contains the register assigned to a particular operand, a function code for built-in functions and special code bits for equivalent variables.
3. The INDEXES vector contains the sequence number of each element in the original source stream. Operator sequence numbers are the character position of the operator in the original source stream. Operand sequence numbers are the character position of the first letter of the operand name in the source stream.
4. The STATEMENT INDEXES vector contains the sequence number of the EOL (End of Line) that begins each statement. The expression:

E _O L		A	=	B	+	C	E _O L				
31	32	33	34	35	36	37	38	39	40	41	42

would cause the generation of

OPS		ATTRIBUTE		INDEXES	STATEMENT INDEXES
60	EOL	0		31	31
2A	Real Scalar	20	Register A	37	31
2	=	0		38	31
2A	Real Scalar	21	Register B	39	31
3	+	0		40	31
2A	Real Scalar	22	Register C	41	31
60	EOL	0		42	31

Object code generation proceeds from these four vectors. Syntactical and semantic analysis is performed by using a variation of the "Current-OP", "Next-OP" (CO-NO) technique employed in some compilers. The current OPS value and four succeeding OPS values are packed into a single word contained in a new vector RESULT1, and then compared to known quintuplets stored in a 32-bit vector, by using the SEARCH instruction. Another vector, RESULT2, is created by performing a TRANSMIT INDEX LIST using the vector of indices obtained from the VECTOR SEARCH. The right-hand number in RESULT2 is used to index a table of instruction skeletons and

pointers. The left-hand precedence number triggers code generation. Not all quintuplets formed from the OPS vector are found in the syntax tables. Only those quintuplets causing code generation or error checking are used. Figure 6-7 shows the process.

OPS	RESULT1					RESULT2	
EOL	EOL	A	=	1.5	EOL	PN20	N20
A	A	=	1.5	EOL	A	0	0
=	=	1.5	EOL	A	=	0	0
1.5	1.5	EOL	A	=	B	0	0
EOL	EOL	A	=	B	+	0	0
A	A	=	B	+	C	0	0
=	=	B	+	C	+	PN10	N10
B	B	+	C	+	D	0	0
+	+	C	+	D	+	PN7	N7
C	C	+	D	+	E	0	0
+	+	D	+	E	*	PN7	N7
D	D	+	E	*	F	0	0
+	+	E	*	F	-	PN7	N7
E	E	*	F	-	-	0	0
*	*	F	-	-	-	PN5	N5
F	F	-	-	-	-	0	0

Figure 6-7. Syntactical and Semantic Analysis

We could, of course, have packed fewer than five OPS values into RESULT1. However, to utilize the streaming properties of STAR to their maximum, we chose to extend the CO-NO technique to four levels. The following benefits are gained when going to four levels:

- Syntactical integrity is almost completely checked for all FORTRAN and ENRICHED FORTRAN statements.
- A large number of statements are uniquely identified by one number at the fourth level. For example:

EOL GOTO 10 EOL

results in a simple number which triggers generation of a jump instruction.

Precedence numbers are arranged into three groups:

PN = 1 → 199 Generate code directly with no further analysis.
Example:

EOL A = 1.5 EOL

PN = 200 - 399 Evaluate arithmetic expressions

EOL IF (A*B*C) N1,N2,N3 EOL

PN = 400-599 Perform additional code generation using results
of arithmetic expression evaluation.

EOL IF (A*B*C) N1,N2,N3 EOL

PN = 600 → Error condition in syntax.

The CODE GENERATOR receives the vectors OPS, ATTRIBUTES, INDEXES, INDEXES1, and RESULT2 and proceeds to trigger code generation based on quintuplet precedence.

The GENERATOR first processes numbers 1 through 199 removing all corresponding elements from the input vector. Arithmetic expressions are then processed by removing precedence numbers 200 through 399. Finally, statements dependent on expression evaluation are processed.

Evaluation of expressions is from left to right with the precedence of operators and parentheses controlling the sequence of code emission. Operator precedence for arithmetic expression evaluation is shown below:

Functions	PN1
Subscripted Arrays	PN2
Exponentiation	PN3
Division	PN4
Multiplication	PN5
Subtraction	PN6
Addition	PN7
Relationals	PN8
Logicals	PN9
Assignment	PN10

PN1 > PN2 > PN3 - - - - > PN9 > PN10

In general each operator results in one precedence number and one code descriptor. Parentheses are not an operator and do not produce precedence numbers in RESULT2.

The problem of parenthetical expressions is resolved before code generation begins. Operator precedence is modified such that operators inside of parenthetical expressions have precedence over external operators. The following example illustrated in Figure 6-8 shows how STAR VECTOR instructions are used to attack this problem. The methods can be extended to error checking and processing of improper parenthetical expressions.

1. Build INTERVAL VECTOR with length of OPS vector (VECTOR INTERVAL instruction).
2. Form three bit strings marking right parenthesis, left parenthesis and precedence numbers greater than zero (VECTOR COMPARE).
3. Remove position numbers from INTERVAL VECTOR for right parenthesis, left parenthesis and precedence numbers greater than zero (VECTOR COMPRESS).
4. Reverse right parenthesis vector (TRANSMIT REVERSE).
5. Form indices vector by searching VECTOR3 until entries greater than or equal to VECTOR1 are found (VECTOR SEARCH).
6. Form indices vector by searching VECTOR2 until entries less than VECTOR1 are found (VECTOR SEARCH).
7. Add indices vectors found in steps 5 and 6 (VECTOR ADD).
8. Find min. entry in RESULT vector and use to obtain vector indicating parenthetical nesting (VECTOR MINIMUM, VECTOR SUBTRACT).
9. Adjust precedence (VECTOR MULTIPLY).

$$\text{EOL } A = \overset{1}{(} \overset{3}{(B+C)} \overset{2}{+ D)}$$

OPS	RESULT2		(1) INTERVAL	(2) RGT PAREN LFT PAREN PN > 0		
	EOL	0	0	0	0	0
A	0	0	1	0	0	0
=	PN10	N10	2	0	0	1
(0	0	3	0	1	0
(0	0	4	0	1	0
B	0	0	5	0	0	0
+	PN7	N7	6	0	0	1
C	0	0	7	0	0	0
)	0	0	8	1	0	0
+	PN7	N7	9	0	0	1
D	0	0	A	0	0	0
)	0	0	B	1	0	0

(3) OPERATOR (VECTOR1) LFT-PAREN (VECTOR2) RGT PAREN			(4) REVERSE RGT PAREN (VECTOR3)
=	2	3	B
+	6	4	8
+	9		

(5) VECTOR1 GE VECTOR3	(7) +	(6) VECTOR1 LT VECTOR2	(8) RESULT =	NORMALIZED RESULT
2 2 1		0 2 2	2 4 3	-(2-1) = 1 3 2

(9) ADJUSTED PRECEDENCE		
PN10 PN7 PN7	*	1 3 2
		=
		PN10 3*PN7 2*PN7

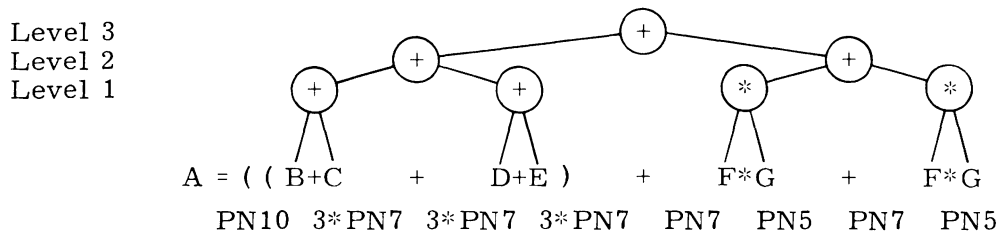
Figure 6-8. Example of Operator Precedence Modification

The modified precedence number vector is now used to trigger code emission. Operator precedence for all expressions is compared in parallel with operator precedence to the right and left. All operators having precedence greater than neighboring operators are emitted. In addition, operators having equal precedence (A+B+C+D) are emitted alternately from left to right. (Of course, a string of exponentiation operators are emitted from left to right, one at a time.) Since subtraction and division are neither commutative or associative, they require special treatment. Subtraction may be handled by converting - to +. For example, A=B-C-D-E becomes A=B-(C+D+E). Similarly, division can be handled by transforming / to *. For example, a*b/c/d becomes (a*b) / (c*d).

Consider the expression:

$$\text{EOL } A = ((B + C + D + E) + F * G + F * G)$$

The GENERATOR determines by the use of some boolean operations that two multiplies can be emitted simultaneously since multiply precedence is greater than neighboring add precedence. Also emitted are two adds of equal precedence. Thus, two multiplies and two adds are issued to the pipeline at the first code emission level.



At the second level, two adds are emitted, and, at the third level, an add is emitted and the result assigned to A. In essence, the parallel construction of a tree structure for each arithmetic expression contained in the program has occurred. The building of a tree structure and emission of code at each level has a number of advantages over conventional Polish string analysis.

- The emission of as many independent instructions as possible to the STAR pipeline reduces register and pipeline conflicts and maximizes issue rates.
- At each level of emission redundant instructions are removed within expressions. For example, one multiply is issued in the above example.

- Tree height reduction algorithms can be used to increase code emission at each level and improve optimization.
- Tree structures contained within nested DO loops can be analyzed, to either reduce register instructions or generate vector instructions.

Generated object code is emitted in packets of 96 bits for every half word of actual code. The first 32 bits contain the sequence number assigned to the object code. The second 32 bits contain the actual code, and the third 32 bits contain DOPE information used to compute relative addresses and to plug data into ENTER IMMEDIATE instructions. The code is merged with other code using the CHARACTER STRING MERGE instruction and the sequence numbers as keys.

Working registers are not allocated cyclically; rather, they are allocated on the basis of operations being performed. Thus, if the computation A+B is performed five times in a program, the result will always appear in the same working register, provided it is issued at the first level. This makes the elimination of "common subexpressions" much easier.

CURRENT DEVELOPMENT

The actual development of object code is proceeding along in three different phases in order of priority.

1. Generation of register to register code utilizing the register file and virtual space. Concepts being tested are:
 - a. Assignment of constants, scalars, and memory addresses to the register file.
 - b. Retention in register file of intermediate computations for large spans of code to aid in common subexpression analysis.
 - c. Retention of call linkages in register file.
 - d. Parameter passing by address and value through the register file.
2. Optimization of register instructions is occurring in conjunction with the generation of these instructions and will include the following:
 - a. Common subexpression analysis.
 - b. Subscript loop analysis.
 - c. Removal of invariant expressions from loops.

3. Analysis of source code programs for the purpose of generating vector and string instruction code to improve object module performance. The problem of recognizing such high level language sequences and replacing them with simpler in-line code is complex but must be approached. The analysis should consist of:
 - a. Local, Global Analysis – collection of arithmetic processing within a statement or group of statements on the basis of common operators. For example, gather local scalars such that $(A+B) \dots (C+D) \dots (E+F)$ could be evaluated with a single vector instruction.
 - b. DO Loop Analysis – single DO loop analysis consisting of
 - NO branch statements
 - Monotonic variation of DO variables
 - DO variables appearing only as left-most subscript in statements with arithmetic operators
 - Destination data not overlapped into source data. Example, $A(I) = A(I-1)$
 - c. Complex DO Loops – reduce complex DO loops by bringing generated code into analysis. Branch points within DOs would be permitted as long as there is no premature exit from inner loops.

ADL FORTRAN SYNTAX LANGUAGE

The ADL FORTRAN compiler is intended as a test bed for:

- Compiling techniques
- Register-register object code optimization
- Vectorization of standard FORTRAN statements
- FORTRAN extensions

To provide some flexibility for this test bed it was decided that the syntactical and semantical analysis as well as the actual object code generation should be "table driven." Hence, a special processor (BLD_SYNT) and a syntax defining language were created to allow reasonably arbitrary changes and/or additions to be made both to the actual language syntax as well as the object code stream for a given input statement. The language (which we will call ADLSYNT) is processed and formatted into a set of tables which become an integral part of the FORTRAN compiler. These tables are structured as sets of parallel vectors to permit streaming access by STAR instructions.

THE LANGUAGE

The language consists of four major parts of which the first two are the minimum required to form a statement.

```
SYNTAX DEFINITION = PRECEDENCE NUMBER { :TYPE DEFINITION /  
CODE SKELETONS / } ;
```

The syntax definition consists of a string of 2 to 5 fields (representing 2 to 5 levels of syntax processing), each field containing one or more symbolic name which stands for a FORTRAN operator or operand. For example, PLUS would represent the "+" sign, MULT the "*" symbol.

For convenience in writing, a number of additional symbols were defined to represent composites or groups of operator/operand types. For example, ARITH would stand for the composite group containing MINUS, PLUS, MULT, DIV, and EXPON. A single field may contain any combination of single operator/operand and/or composite names. For example, the SYNTAX DEFINITION:

```
EOL, RGT_PAREN/GOTO/IS/EOL = N96;
```

means that any structure beginning with either an end of line or right parenthesis followed by a GOTO which is followed by an integer scalar followed by an end of line is a legal syntax and is assigned the precedence number N96.

PRECEDENCE NUMBERS

The precedence number is required in all statements (other than COMMENTS) and is a symbolic name defining a hex number which determines in some cases the exact code sequence to be generated, in other cases the specific "special compiler action" to be taken and in all cases the order of processing.

TYPE DEFINITION

This field has several options:

OPTIONAL LABEL FIELD

Precedence Number: label. followed by a comma. To reduce the table size it is possible for any syntax statement to refer to a particular type definition and object code string by use of the "label." For example,

```
EOL, RGT_PAREN/GOTO/IS/EOL = N96:JUMPS:INT.INT=!INT/33400000/;
```


defines the following type definitions and code string as JUMPS. A subsequent definition could then refer back to it as follows:

```
EOL,RGT_PAREN/GOTO/LFT_PAREN/IS/RGT_PAREN = N96:JUMPS;
```

Thus, the compiler would end up with the same code string for two different legal syntax structures.

OPTIONAL OPERATOR FIELD

label: operator — operand type . operand type = operand type/. For example:

```
ESIGN/S/ARITH/S/EOL = N44:ARITHS:PLUS * REAL,REAL = REAL/
```

means that an equal sign (ESIGN) followed by a scalar followed by any arithmetic operator followed by a scalar, followed by an end of line is a legal syntax with precedence N44, which generates a code string labelled ARITHS. The first operational code string will be for the operator PLUS involving operand types REAL + REAL with a REAL result. A special case made for Built In Functions (BIFs) would appear as:

```
Precedence number:ARITHS:BIFS*SQRT.REAL,REAL=REAL/
```

where the BIF name (in this case for square root) immediately follows the word BIFS (with an intervening period).

OPERAND TYPE DEFINITIONS

All structures are assured to have two source operands "REAL,REAL" and one destination operand "=REAL". Where a structure such as "X=Y" has only one source the left hand operand can be written as NULL or the same type as the destination operand.

CODE SKELETONS

Code skeletons are written as a string of 32-bit packets and "DOPE" information is separated by slashes.

```
/62000000,IA+1,IA+2,IA+3/6B000000,IAI3,IA+2,IA+3/
```

The first field is the actual code skeletons; the three fields following correspond to the R, S, and T register fields in the instruction. If a fourth field, corresponding to F is required it would appear last. The information in the three DOPE fields

is a combination of symbolic and numeric data related to the source and destination register pointer vectors available in the compilers generators.

IA stands for Indirect from Assigned locs
IC stands for Indirect from Constants
DW stands for Direct from the Working vector

The numbers +1, +2, +3 correspond to the left, right and destination vectors respectively. Thus IA+1 in the above example would direct that the R field of the 62 instruction is to be filled in with the register appearing in the assigned locs vector at the position specified by the pointer in the left-hand register vector (LFTR) at compile time.

The code skeleton may be followed by a reverse slash and a code having special significance to the particular generator. For example:

```
/98000000\1, 0, IA+1/0, IA+2, 0, IA+3/
```

The \ 1 indicates to the arithmetic generator that there is one 32-bit packet following which is inseparable from the 98000000 for the purposes of work register assignment and code optimization.

A \ 100 indicates that the instruction is not to be moved or removed from its place in the code stream by the code optimizers.

A \ 1000 → \ 7000 reflects vector code skeletons being generated in place of register to register instructions.

A full scale example:

```
ARITH/SC/ARITH/SC/ALL=N44:  
SMATH:PLUS* REAL, REAL=REAL/62000000, IA+1, IA+2, IA+3/  
INT, INT=INT/62000000, IA+1, IA+2, IA+3/  
REAL, INT=REAL/62000000, IA+2, 0, IA+3/  
/62000000, IA+1, IA+3, IA+3/  
MINUSxREAL, REAL=REAL/66000000, IA+1, IA+2, IA+3/  
INT, INT=INT/64000000, IA+1, IA+2, IA+3/  
REAL, INT=REAL/62000000, IA+2, 0, IA+3/  
/66000000, IA+1, IA+3, IA+3/;
```

Comment lines are preceded by an asterisk in the first character of an input line.

FORTRAN EXTENSIONS

Extensions are added to FORTRAN to ensure more effective utilization of the computer. Effective STAR utilization includes: vector data bases; control vectors; single bit, eight-bit character, 32-bit halfword, and 64-bit full word data elements; large instruction set usage; and core and register management.

The generalization of the FORTRAN subscript to allow the selection of multiple array elements on any array reference, the ability of functions to return multiple result values, and the development of multi-value expressions and replacements provide a complete vector data base capability. Control vectors are added to the language through the new subscription forms to permit multiple array element selections.

The data types of FORTRAN are generalized to include type character. Logical data is assigned one bit per element; character data, multiples of eight bits of memory per element; integer data, real data, and each half of complex data, thirty-two or sixty-four bits per element; each half of a double precision datum, sixty-four or one hundred twenty-eight bits per element. The type character extensions to be used by will be those defined by the Canadian Development Division or the ANSI FORTRAN standard.

MULTIPLE VALUED (SUBSCRIPTED) VARIABLES

A multiple valued (subarray) reference identifies one or more array elements simultaneously. A subarray reference is an array function reference, an array name qualified by one of the multi-element subscript forms defined below, or an unqualified array name reference.

The array function reference and the array name reference can be subarrays of more than one dimension; the qualified array name is restricted to one multi-element subscript expression per reference (the other dimensions must be specified by scalar subscript expressions) — a qualified array name yields a one-dimensional subarray. Subarray references provide the means of extracting (compressing) elements of arrays or expanding small arrays to fill parts of larger arrays. The subscript expressions qualifying an array name are separated by commas.

The simplest form of subarray reference is the unqualified array name. This reference identifies all elements of the array. The dimensionality of an array name reference is the same as defined on the DIMENSION statement. As an example, if A is DIMENSIONED as A(5), then A=1.0 initializes array A to 1.0.

A regular sequence of scalar subscript values may be specified through the implied-DO subscript form. This form is patterned after the DO statement and input/output list implied-DO constructions. The additional capability of automatically providing the "maximum-end-value" is possible because this DO-construction can apply to only one identifier. The basic forms of the implied-DO multi-element subscript form are as follows:

1. $m_1 : m_2 : m_3$
2. $m_1 : m_2$
3. $*$
4. $m_1 : * : m_3$
5. $m_1 : *$
6. $* : m_3$

\underline{m}_1 are indexing parameters; \underline{m}_1 is the initial value; \underline{m}_2 is the final value; \underline{m}_3 is the step value. If \underline{m}_1 or \underline{m}_3 is not specified, a value of 1 is assumed. "*" may be used in place of \underline{m}_2 to specify the maximum scalar subscript value \underline{m}_2 can correctly assume. The length of the subarray dimension is defined as the number of index values the implied-DO defines; this length is always less than or equal to the length of the particular dimension of the array.

Examples:

```

DIMENSION X(10), Y(10,3)

X(2:9:3)    represents the array elements
             X(2), X(5) and X(8)

Y(8,*)      represents the array elements
             Y(8,1), Y(8,2) and Y(8,3)

X(*:3)      represents X(1), X(4), X(7) and X(10)

Y(7,2:*)    represents Y(7,2) and Y(7,3)

```

A list of "random" scalar subscript values can be specified with a multi-value integer expression. The referenced subarray is constructed by ordering the elements of the original array in the order specified by the index list. The length of the subarray is the length of the indexing array; the subarray length can exceed the length of the original array.

Examples:

```
DIMENSION I(6), A(4), B(2,3)
DATA I/1,3,1,3,2,3/
"A(I)" represents A(1), A(3), A(1), A(3), A(2) and A(3) - in that order
"A(I(5:6))" represents A(2) and A(3)
"B(2,I(1:2))" represents B(2,1) and B(2,3)
```

CONDITIONALLY SELECTED SUBARRAY REFERENCES

Array elements which are to be manipulated only when a test condition has been satisfied are conveniently manipulated with logical array or multi-element logical expression subscripts. The subarray is composed of original array elements for which the corresponding logical array element has a "TRUE" value. The length of the subarray is less than or equal to the length of the logical and the original arrays, being defined as the count of "TRUE" values in the logical array. (It is not permissible to use a logical array subscript that is longer than the dimension of the original array.)

Examples:

```
LOGICAL L(6)
REAL X(6), Y(10,10)
DATA L/.T., .F., .F., .T., .F., .T. /
"X(L)" represents X(1), X(4), and X(6).
"Y(7,L)" represents Y(7,1), Y(7,4), and Y(7,6).
```

SCALAR EXPRESSIONS

A scalar expression produces one result and is a scalar reference, scalar expression and/or array expressions reduced to a scalar by one of the operators .ALL., .ANY., or .NONE..

Examples:

REAL X, Y, Z, A(100), B(100)

LOGICAL LS, LA(100)

"X" is a scalar expression.

"X+Y" is a scalar expression.

"SUM(A(1:*:4))" is a scalar expression which computes

$$\sum_{i=1}^{25} A(4*i).$$

"Y*Z*PROD(B)" is a scalar expression which computes

$$\prod_{i=1}^{100} B_i * Y * Z.$$

"LS.AND. .ALL. LA(*:3)" which computes the logical product of every third element of LA and the scalar, LS.

SUBARRAY EXPRESSIONS

A subarray expression produces one or more results and is one or more unreduced multi-element expressions and may also contain scalar expressions. An array expression must have conformable subarray references (same number of dimension per array reference, same lengths per dimension). An array expression is evaluated by performing the stated operation on corresponding elements of the subarrays referenced. Scalar references are considered to be arrays of the proper dimensionality with all elements containing the scalar value.

Examples:

REAL X, A(100), B(100, 4)

1. A

This array expression has a 1 by 100 dimensional result

2. B

This array expression is a two-dimensional array with lengths 100 and 4 per dimension.

3. A + B(*, 4)

This array expression yields a one dimensional array of length 100. The 100 results are produced by adding pairs of values as follows:

$A(1) + B(1, 4) \rightarrow \text{result } 1$
 $A(2) + B(2, 4) \rightarrow \text{result } 2$
 $A(3) + B(3, 4) \rightarrow \text{result } 3$
 \vdots

5. $B*B$

This array expression yields a two-dimensional array with length of 100 and 4 per dimension. The 400 results are produced by multiplying pairs of operands as follows:

$B(1, 1) * B(1, 1) \rightarrow \text{result } 1$
 $B(2, 1) * B(2, 1) \rightarrow \text{result } 2$
 \vdots
 $B(100, 1)*B(100, 1) \rightarrow \text{result } 100$
 \vdots
 $B(100, 4)*B(100, 4) \rightarrow \text{result } 400$

SCALAR ASSIGNMENT STATEMENTS

A scalar assignment statement assigns the value of a scalar expression to a scalar variable or scalar array reference.

Examples:

REAL X, Y, Z, A(100), B(2, 2)
 LOGICAL LS, LA(100)

Z=X regular FORTRAN assignment statement

Y=SUM(A(2*:4) computes the sum,
 $A(2) + A(6) + A(10) + \dots$

Y=Y*Z* PROD(B) computes the product,
 $Y*Z*B(1, 1)*B(2, 1)*B(1, 2)*B(2, 2)$

LS= .ALL. LA(*:3) computes the logical product,
 $LA(1).AND. LA(4).AND. LA(7).AND. \dots$

ARRAY ASSIGNMENT STATEMENTS

An array assignment statement assigns the result of a scalar expression to every element of the subarray or transfers elements of an array expression result to corresponding elements of a subarray. The assignee and the array expression must be conformable.

Examples:

```
REAL  A(4), B(4,2), C(4,2)
DATA A(*)/1.,2.,3.,4./
DATA B(*,*)/.11,.21,.31,.41,.12,.22,.32,.42/
DATA C(*,*)/8*0.0/
```

1. $C(*, 1) = A + B(*, 1)$

Replaces elements of C as follows:

$$C(1, 1) = A(1) + B(1, 1) \quad (C(1, 1) = 1.11)$$

$$C(2, 1) = A(2) + B(2, 1) \quad (C(2, 1) = 2.21)$$

$$C(3, 1) = A(3) + B(3, 1) \quad (C(3, 1) = 3.31)$$

$$C(4, 1) = A(4) + B(4, 1) \quad (C(4, 1) = 4.41)$$

Elements $C(*, 2)$ remain unchanged.

2. $C = C + B$

(Assume original data values.) Each element of C is added to the corresponding element of B and replaces the C element as follows:

$$.11 = C(1, 1) = C(1, 1) + B(1, 1)$$

$$.21 = C(2, 1) = C(2, 1) + B(2, 1)$$

$$.31 = C(3, 1) = C(3, 1) + B(3, 1)$$

$$\begin{array}{cccc} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{array}$$

$$.42 = C(4, 2) = C(4, 2) + B(4, 2)$$

3. $A = A + .375$

The constant .375 is added to each element of A to produce the new A values:

$$A(1) = 1.375$$

$$A(2) = 2.375$$

$$A(3) = 3.375$$

$$A(4) = 4.375$$

NEW OPERATORS

New Operators have been added to allow for the transition between scalar and array (vector) data. The new operators are:

<u>Operator</u>	<u>Example</u>	<u>Explanation</u>
.ALL.	.ALL. LA(*:3)	The result of this expression is true if all designated values of LA are true.
.ANY.	.ANY. LA(*:3)	The result of this expression is true if at least one designated value of LA is true.
.NONE.	.NONE. LA(*:3)	The result of this expression is true if all designated values of LA are false.
.CAT.	A, CAT, B	The result of this expression is the value(s) of A followed by the value(s) of B.
.XOR.	LSA, XOR, LSB SA, XOR, SB	The result of this expression is the exclusive OR of the operands.

PARAMETER STATEMENT

The PARAMETER statement provides a method of identifying constants by symbolic names. Symbolically identified constants (parameters) can occur anywhere a constant can be used. Their use in DIMENSION statements, DO-loop index parameter positions, etc., provide a level of coordination between various uses of the same value within a program unit (e. g., ensuring conformable arrays).

The PARAMETER statement follows the form of the DATA statement as follows:

```
PARAMETER  $i_1/d_1/$ ,  $i_2/d_2/$ , ...,  $i_n/d_n/$ 
```

where i is a list of symbolic names separated by commas.

d is a list of constants as defined for the DATA statement or is a combination of constants and previously defined parameters separated by +, -, *, /.

The parameter name must be unique.

Examples:

1. PARAMETER PI, E/3.1417, 2.71/
Y = X + PI + Z * E

The symbolic name PI is equated to the value 3.1417 and E to the value 2.71. The second line is equivalent to the following:

$$Y = X + 3.1417 + Z * 2.71$$

2. PARAMETER NBODY/20/, NBODY7, NBODY9/NBODY*7, NBODY*9/
DIMENSION A(NBODY), B(NBODY), C(NBODY7,NBODY)
DO 1 I = 1, NBODY

The second and third lines are equivalent to:

```
DIMENSION A(20), B(20), C(140,20)
DO 1 I = 1,20
```

USE OF SUBARRAY REFERENCES IN DATA STATEMENTS

The DATA (data initialization) statement is used to define initial values of variable or array elements not located in blank COMMON. The statement form is:

```
DATA  $i_1/d_1/$ ,  $i_2/d_2/$ , ...,  $i_n/d_n/$ 
```

where i is a list containing names of variables and/or array elements and/or subarray references, but not containing dummy argument names.

d is a list of constants, optionally signed, which designate the values to be assigned to the list elements. Parts of the list may be grouped by parentheses, optionally preceded by a repetition factor "j*".

Examples:

```
DIMENSION AMASS (10, 10, 10), A(10), B(5)
DATA AMASS (6, *, 3)/4*(-2., 5.139), 6.9, 10. /
DATA A(5:7)/2*(4.1), 5.0/
DATA B/5*0.0/
```

ARRAY - AMASS:

```
AMASS(6, 1, 3) = -2.
AMASS(6, 2, 3) = 5.139
AMASS(6, 3, 3) = -2.
AMASS(6, 4, 3) = 5.139
```

ARRAY - A:

```
A(5) = 4.1
A(6) = 4.1
A(7) = 5.0
```

AMASS(6, 5, 3) = -2.	ARRAY - B:
AMASS(6, 6, 3) = 5.139	B(1) = 0.0
AMASS(6, 7, 3) = -2.	B(2) = 0.0
AMASS(6, 8, 3) = 5.139	B(3) = 0.0
AMASS(6, 9, 3) = 6.9	B(4) = 0.0
AMASS(6, 10, 3) = 10.	B(5) = 0.0

The array, AMASS, could also have been initialized as follows:

```
DATA AMASS(6, 1:7:2, 3), AMASS(6, 2:8:2, 3), AMASS(6, 9, 3), AMASS(6, 10, 3)
+ /4*(-2.) , 4*(5.139), 6.9, 10. /
```

PROCEDURE IDENTIFICATION

SUBROUTINE and FUNCTION identification is normally done by implications, a name that occurs in a CALL statement must be a SUBROUTINE name, and a symbolic name followed by a left parenthesis is either an array reference (and occurs in a DIMENSION statement or its equivalent) or it is a function name. Procedure name usage may require their explicit definition under two circumstances: the procedure name is used only as an argument in a procedure call or a function may return a multi-element result and, therefore, requires dimensioning. In addition, the user can force a restricted set of procedure names to produce external procedure linkages or inline code.

EXTERNAL STATEMENT

The EXTERNAL statement defines variable names to be external procedure names. This feature permits external procedure names to be passed as arguments to another external procedure; the names must be defined in an EXTERNAL statement in the program unit in which it is used.

```
EXTERNAL v1, v2, ..., vn
```

v_i are declared to be external procedure names.

Example:

```
EXTERNAL NAME1, NAME2, NAME3
.
.
.
CALL SUB(A, B, NAME2)
SUBROUTINE SUB(X, Y, IFUNC)
```

The user is also allowed to define an Intrinsic function name in an EXTERNAL declaration. This redefinition of an intrinsic function name causes the processor to consider any subsequent reference as an external function reference; the user must supply the procedure.

An EXTERNAL statement can be used in combination with or in place of a DIMENSION statement to identify multi-element valued functions. Any of the following statement groups would define a real, multi-element valued function, RF.

1. DIMENSION RF (100)
REAL RF
EXTERNAL RF
2. REAL RF (100)
EXTERNAL RF
3. REAL RF
EXTERNAL RF (100)

An array valued function is required to have a well-defined result dimensionality. The dimensioning information may appear in two forms, an explicit size specification in the same form used on type and DIMENSION statements or an implied size based on the dimensionality of the first argument. The second form of procedure dimensioning is identified by an "*" for the dimension constant list.

Example:

```
Given:  EXTERNAL FUNC1(10, 10), FUNC2(47)
        EXTERNAL FUNCA(*)
        DIMENSION A(10, 10), B(50)
```

- Then:
1. A = FUNC1(expressions)
This function call produces a 10 x 10 array result.
 2. B(1:47) = FUNC2(expressions)
The 47 element result from function FUNC2 is stored into the first 47 elements of B.
 3. A(5:10, 2) = FUNCA(B(1:6))
Six result values are provided by FUNCA and stored into array A.

4. $A = A + \text{FUNCA}(A)$

FUNCA returns a 10 x 10 result array which is added to the original A and the result of the expression defines new A values.

INTRINSIC STATEMENT

The INTRINSIC statement permits the programmer to force additional procedure references to produce inline code (i.e., the procedure's code is built into the statements code sequence).

Example:

A system subroutine, DISPLAY, which places a Hollerith coded message in the operator display window might appear as follows:

```
SUBROUTINE XYZ
INTRINSIC DISPLAY
.
.
CALL DISPLAY ('SAMPLE MESSAGE')
.
.
END
```

PROCEDURE CLASSIFICATIONS

FORTRAN subprograms are placed in four categories: intrinsic, basic external, library external and user external. An intrinsic procedure (or subprogram) reference results in code being placed in the referencing subprogram at the required point. A library external procedure reference results in a jump to a separate (external) code module. A basic external procedure is normally treated like a library external procedure but, if named in an INTRINSIC statement, may be treated as an intrinsic procedure. A user external procedure is defined by a FORTRAN source deck supplied by the user. An EXTERNAL statement changes an intrinsic procedure to a basic external procedure or user procedure. (A recent ANSI change allows an EXTERNAL statement to differentiate between a change of classification and a declaration of information. This latter change allows an intrinsic/basic - external procedure name to be treated intrinsically in the referencing procedure and passed as a procedure name to another referenced procedure.)

Examples:

```
SUBROUTINE XYZ(X)
REAL X(100),
X = ABS(X) + ALOG(X) + SIN(Y)
RETURN
END
```

The subroutine XYZ is a user procedure.

The function ABS is an intrinsic function.

The function ALOG is a library function.†

The function SIN is a basic external function.†

Two external calls are required to execute XYZ as above.

In the following version, only one external call is required.

```
SUBROUTINE XYZ(X)
REAL X(100)
INTRINSIC SIN
X = ABS(X) + ALOG(X) + SIN(Y)
RETURN
END
```

DYNAMIC SPACE MANAGEMENT

FORTTRAN intermediate array storage will be taken from the dynamic space area. One of the following extensions allows the user to assign the dynamic space base address. Two additional statements allow the user to request storage from dynamic space.

DYNAMIC SPACE BASE ADDRESS ASSIGNMENT

The addition of a COMMON block name to the PROGRAM, SUBROUTINE and FUNCTION statements is taken as the signal to change the dynamic space pointer.

The change is instituted as follows:

1. Save the current DSP (Dynamic Space Pointer).
2. Enter the base address of the COMMON block.
3. Before exiting the procedure, restore the saved DSP.

† This is for illustration only, no classification of FORTRAN procedures is intended as part of this proposal.

All external procedures called will allocate their storage from the newly established DSP. This feature is useful for programs which have two or more independent paths which are initiated from time-to-time from interrupts. Some applications require this feature because the independent paths may occasionally be restarted and not re-entered in the normal CALL-RETURN manner.

Examples:

```
BLOCK DATA
COMMON /PATH0/...
COMMON /PATH1/...
COMMON /PATH2/...
END
PROGRAM MAIN/PATH0/
.
.
.
END
SUBROUTINE SUB1/PATH1/(a1, a2, ..., an)
.
.
.
END
```

The statement "PROGRAM MAIN/PATH0/" changes the FORTRAN DSP from the system DSP to the COMMON block PATH0. Before returning to the system, the value of DSP before entry must be re-established (if the system environment register was changed). The statement "SUBROUTINE SUB1/PATH1/(a₁, ..., a_n)" changes the FORTRAN DSP from PATH0 to PATH1.

TEMPORARY STATEMENT

The TEMPORARY statement provides the means of dynamically defining the base addresses of data areas. The basic form of the statement follows the COMMON statement and appears as:

```
TEMPORARY /t1/i1/t2/i2 ...
```

i is a list of variable names and array declarators. t may be either blanks (or omitted) or a symbolic name. The value of t determines the exact processing of the statement.

If t is blank, the "TEMPORARY block" is allocated immediately upon entry to the procedure. If t is a unique name, the storage is not allocated until the occurrence of an ALLOCATE statement (next section). The final possibility is that t is the same as one of the arguments to the procedure; in this case, the base addresses of the list items are determined relative to the address of the named argument.

Examples:

1. PROGRAM MAIN/PATH0/
TEMPORARY//X(100), COEFF(10,10)
CALL XYZ(COEFF, LTH)
END

This program establishes the DSP as the COMMON block "PATH0". Before executing XYZ, a two hundred word block of "PATH0" is allocated and the descriptors for X and COEFF are constructed.

2. SUBROUTIN XYZ(AREA, LTH)
TEMPORARY/AREA/A(LTH), B(LTH)
A = A + B * FUNC(A)
RETURN
END

The argument, AREA, is used as a data base which is divided in pieces. In this example, the pieces are defined as variably dimensioned arrays A and B. Before executing the replacement statement, the descriptors for A and B are prepared.

ALLOCATE STATEMENT

The ALLOCATE statement is used to cause the allocation of dynamic space and the building of variable descriptors. The form of the statement is:

```
ALLOCATE t1, t2, ...
```

where t_i is the name of a TEMPORARY block ($i \geq 1$).

Example:

```
FUNCTION FUNC(ARG)  
TEMPORARY/T1/.../T2/.../T3/...  
.  
.  
.
```



```

ALLOCATE T1
.
.
.
ALLOCATE T2, T3
.
.
.
RETURN
END

```

Except for the ALLOCATE statements, execution of FUNC follows a normal FORTRAN course. At each ALLOCATE statement, a block of dynamic space is taken and the appropriate descriptors are computed.

HEXADECIMAL VALUES

A hexadecimal constant of the form:

$$\# h_1 h_2 \dots h_n$$

where h_i is a hexadecimal digit and $1 \leq n \leq 8$ or $1 \leq n \leq 16$ for 32- or 64-bit data, respectively.

Input/output of hexadecimal constants use the format declarator:

$$r \# w$$

where r is the repetition and
 w is the field width

TYPE SPECIFICATIONS

The type specifications include the FORTRAN standard specifications, the word TYPE before any of the FORTRAN type specifications, the IMPLICIT statement and the "*" s" byte designation in any of the above (except LOGICAL array specifications which always are allocated to bits; in this case, the "*" s" indicates boundary alignment requirements).

Examples:

```

INTEGER I, X
TYPE INTEGER J, Y
IMPLICIT INTEGER (I, J, X, )
INTEGER*4 K, W
TYPE INTEGER*8 L, Z
IMPLICIT*4 INTEGER(A, M), INTEGER*4(K, W), INTEGER*8(L, Z)

```

This section deals with the rather complex problems of the linkages between and among programs, procedures, blocks, and subprograms. It illustrates the development approach to be taken for all compiling systems. Standards imposed upon the object modules produced by compilers and assemblers ensure that routines and programs can be freely interchanged regardless of the original source language.

Most of the discussion in this section uses PLSTAR, PL-1, and FORTRAN language features for illustration, often in an interchangeable way, because the implementation of PL-1 linkages satisfies the requirements of FORTRAN.

STRUCTURE

The basic system philosophy involves, insofar as the implementation of linkage object code for compilers or assemblers is concerned:

1. Implicit I/O
2. Resource management distributed among stations
3. A global, re-entrant, absolute, shared library
4. Special sharing of regions of virtual space
5. Optimizing CPU utilization by maximizing the streaming functions and minimizing random fetches and jumps in code.
6. Making the memory and time penalties involved in the access of routines proportional to the complexity and frequency of access of the routine.

Items 3, 4, 5, and 6 impose some constraints on object code structures if the assumption is made that all programs are treated uniformly by the compiler including those in the library or shared virtual memory. Because executable code in shared space must be re-entrant and is most likely protected by write lockout, all potentially modifiable data or code must be given space in each user's virtual memory. The classes of data that must be treated in this manner are:

1. External label references (that are not linked until execution of the object code).
2. Local scalars and arrays (in PL-1 this would include "automatic" variables).
3. Pointers to blank-common and labeled common blocks (in PL-1 pointers to STATIC variables or storage and/or based variable pointers).

The object code structures need to be discussed in the light of the foregoing classes.

Figure 7-1 shows the layout of a library program or module. The module header table includes pointers to relocation data, entry and external lists, and object code. This is followed by a prologue, optionally, and then the body of the program, which may have multiple entry points. The last two areas are register-file load blocks, called Regblocks, and the linkage data. A more complete description of the tables comprising a module is given in a following subsection entitled Module Tables.

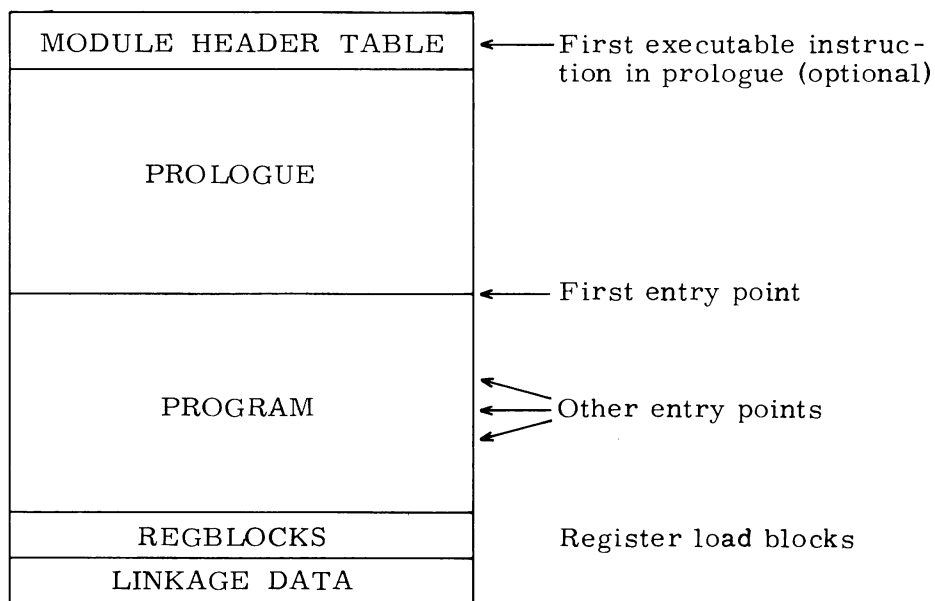


Figure 7-1. Layout of a Library Program

Execution of such a module usually requires the use of work space for calculation and input/output. Such workspace is normally used for both the reading and writing of data by the user program. In the general case, executable code is write protected; therefore, the work area must be separated in virtual memory from the accessing program.

Figure 7-2 shows a typical structure during the execution of user and library programs. The pointers denote possible program linkages or data access paths.

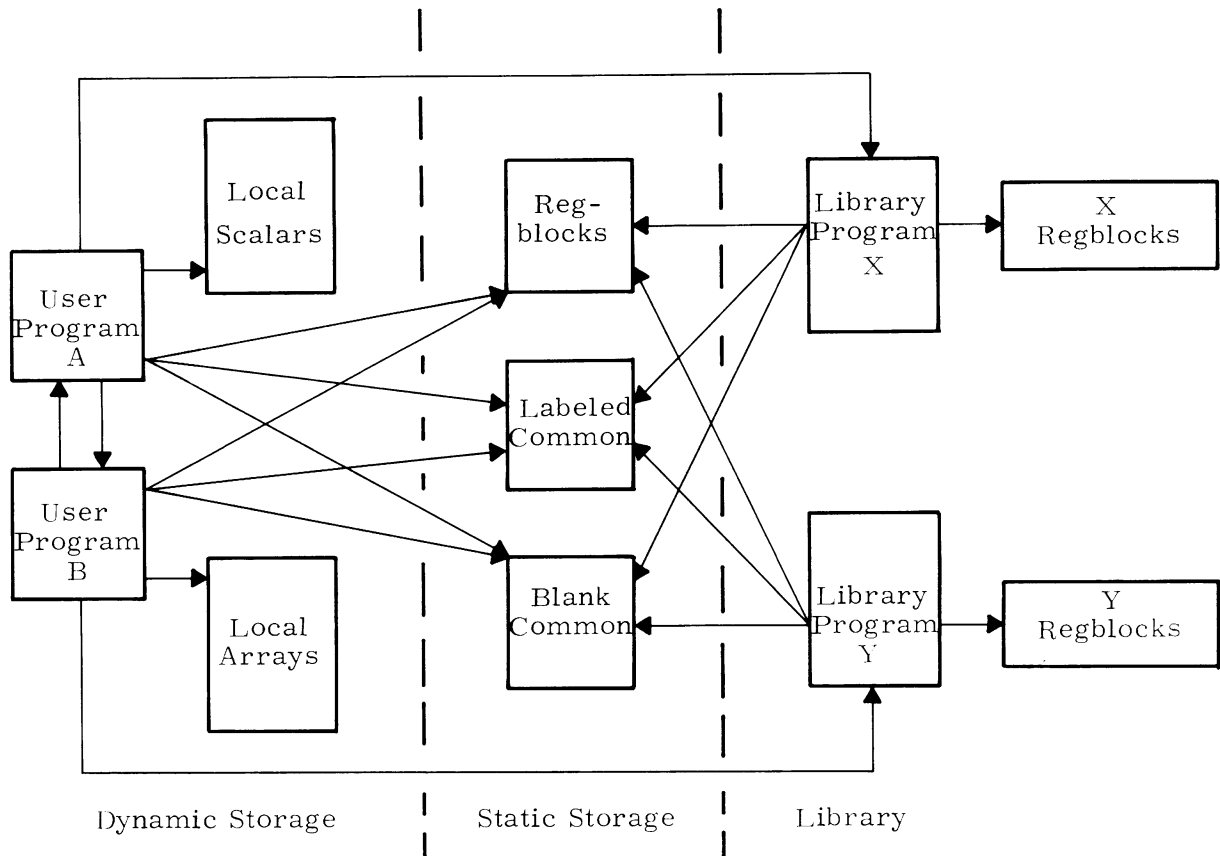


Figure 7-2. Program Linkages and Data Access Paths in a Shared Library Environment

REGBLOCKS

When a program begins execution, the contents of its register file are normally undefined. This is true for all programs except those that have been compiled with their data in place; that is, in the first half of their page zero.

Because nearly all instructions involve the use of the register file, a newly started program must get its data into the registers. This can be done not only by direct, in-line loading which sets the registers one by one, but usually much more conveniently by block loading an area of the register file with a vector instruction.

Thus, associated with each program there should be a block of data intended to be loaded into the register file. As long as none of the data in these blocks requires modification during execution nor needs to be carried over between executions of the subprograms, these blocks, called Regblocks, can be retained in the read-only area along with the code. Regblocks normally contain addresses or data pointers for local arrays, as well as for externals and common blocks. One or more such addresses appearing in a Regblock is called an address vector.

STATIC STORAGE

In many instances, a subprogram will reference COMMON blocks or an external which was not defined at either compile or load time and which will be linked at the beginning of execution. Such cases require that the contents of a given regblock be modified before execution of a subprogram (all external and common references must be register variables). These situations require that one or more regblocks be located in modifiable virtual space each time the program is called from there. Since these regblocks have been set up by updating the contained address vector at execution time for a particular job and remain constant throughout the job, they need to be allocated to a region of Static Storage in virtual memory.

Blank Common and Labeled Common blocks are allocated on the first occurrence of a reference to them. Once allocated, these common areas may remain fixed in address and length for the duration of execution and hence may reside in static storage. Note that allocation of various elements of static storage can occur at any time during execution (although they are retained for its duration). This implies that static storage is allocated and grows dynamically in its general case and is not de-allocated until the user logs off.

DYNAMIC STORAGE

From the point of view of virtual memory utilization and possible impacts on paging, static storage might be considered wasteful of resources although it is essential to support compiler languages. There are other kinds of data, however, that can be assigned to virtual memory areas which can be overlaid by other data, thereby re-using the same virtual region and reducing paging. The most prominent example of these are local scalars and arrays or "automatic variables." These include all data declared by a given subprogram which does not appear in COMMON (as either STATIC or BASED variables). Some language specifications demand that, for each entry into

a subprogram, the value of such data is either undefined or in a canonically initialized state. The normal mechanism for initialization in such cases is the DATA or INIT statement in FORTRAN or PL-1. Because local scalars and arrays are normally modified by the declaring procedures, they must exist in the user's virtual memory. The present compiler approach is to assign all "simple" scalars to the register file, along with scalar constants and array pointers. Simple scalars are those that do not appear in COMMON or EQUIVALENCE statements (in PL-1 they do not appear in STRUCTURES). Therefore, at each entry to a given subprogram, a region of virtual memory must be assigned to the subprogram's local complex scalars and arrays, and those quantities which are initialized must be loaded, either by enter immediate instructions or by a vector load of a data block.

The inherent overhead of such an initialization operation permits a given region of virtual memory to be collapsed and expanded with little additional time penalty as program execution progresses. Another major use for the Dynamic Storage region is for stacking data and pointers in recursive systems such as ALGOL or PL-1 and for stacking portions of the register file as job execution descends and ascends through levels of subprograms.

Further extensions of dynamic storage allow it to contain executable code which may be loaded during a particular job sequence and, in fact, to encompass all object code extant in the user's unique virtual space; that is, all that is other than shareable or library space.

Figure 7-3 shows the structure of a user's virtual space that has evolved from the above arguments. Two pointers, the Static Space Pointer (SSP) and the Dynamic Space Pointer (DSP) are introduced in the figure. These are discussed in the next section.

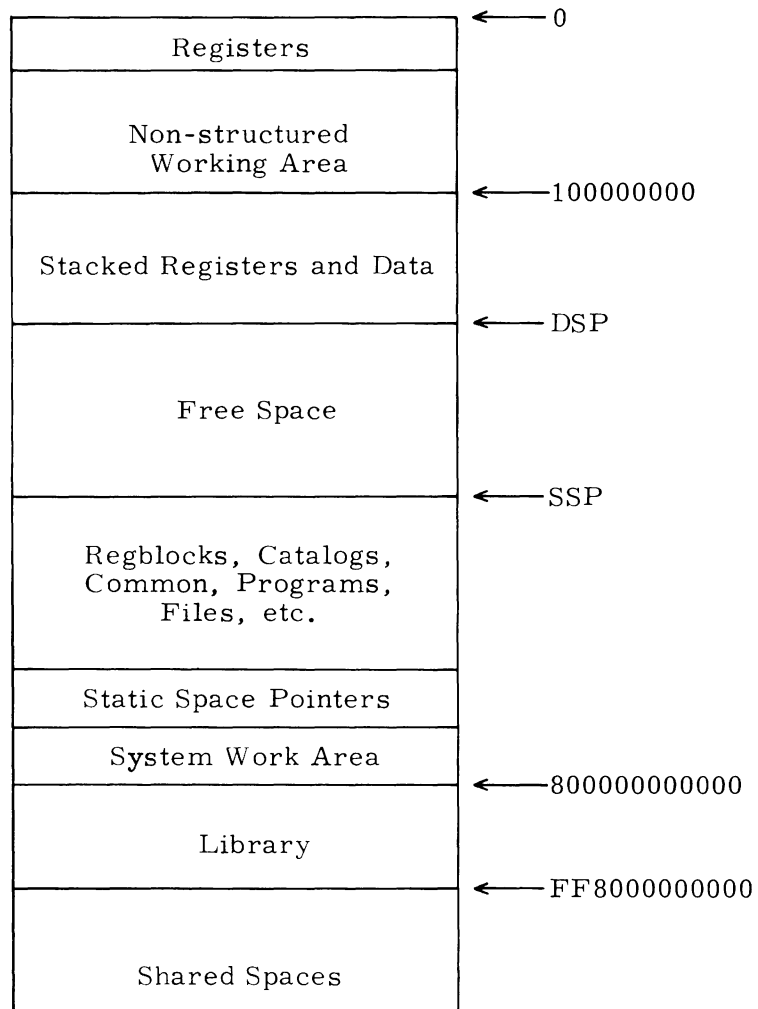


Figure 7-3. Outline Structure of User's Virtual Space

POINTERS

A group of eleven pointers describing static space is located at the address known as "ADDR_SSP." They are shown in Figure 7-4.

	Static Space Pointer (SSP)	←ADDR_SSP
Previous Track Ordinal	Current Track Pointer	
	Module Catalog Pointer	
	Link Catalog Pointer	
	FRMT Pointer	
	Internal File Name Pointer	
	File Name String Pointer	
	User ID Descriptor	
Sense Switches	JCL Pointer	
	Highest DSP	
Number of Tracks	Track Chain Pointer	

Figure 7-4. Static Space Pointers

Dynamic space begins at the location indicated by DSP, which is also known as the Next Stack Pointer. During the course of execution of a subprogram, DSP is continuously updated to always give the next available location in dynamic space. Another pointer, the Current Stack Pointer, gives the most recent DSP setting. If a subprogram must stack the register file or allocate local data space in dynamic storage, the Current Stack Pointer can be used to backtrack program linkage in case of error.

To support the structure just outlined, it is necessary to allocate some fixed region of the name space for the basic pointers indicating where the dynamic and static spaces are. There are also several other items which are required for program execution, such as return addresses and parameter lists. While it would be possible to allocate these items to some arbitrary region of virtual memory, the high frequency of use of these items dictates that they be placed in the register file. Thus, the basic scheme of register file division and usage shown in Figure 7-5 was derived.

0	Machine Zero		Machine Registers
1	Data Flag Return		
2			Temporary Registers
12			
13			
14			Mixed-use Registers
15	Vital Pointer		
16	Constant One	ONE	
17	Parameter Descriptor	PD	
18	Function Value	FV1	
19		FV2	Environment Registers
1A	Return	RETURN	
1B	Dynamic Space Pointer	DSP	
1C	Current Stack Pointer	STACK	
1D	Previous Stack Pointer	OLD_STACK	
1E	Callee Data Base	LINK	
1F	On Unit Stack Pointer	ON	Working Registers
FC	2nd Parameter Pair		Parameter Registers
FD			
FE	1st Parameter Pair		
FF			

Figure 7-5. Register File Assignments

REGISTER FILE CONVENTIONS

For the purpose of supporting the requirements of dynamic linking and loading of modules, the Register File has been somewhat arbitrarily divided into six definable regions:

- Machine Registers
- Temporary Registers
- Mixed-use Registers
- Environment Registers
- Working Registers
- Parameter Registers

MACHINE REGISTERS

These registers include only registers 0 and 1. Register 0, by convention, contains the machine representation of the number zero. Register 1 is used as the Data Flag Branch return.

TEMPORARY REGISTERS

Registers 2 through 11 are temporary registers, the contents of which are not saved across calls. This space is chosen large enough to permit execution of many lowest level subroutines, such as SIN, COS, etc., using registers only within the temporary space, obviating the need for saving and restoring any of the caller's permanent registers. The choice of low-numbered registers permits their use for both full- and half-word temporaries.

MIXED-USE REGISTERS

Registers 15 through 19 are used for various miscellaneous uses.

Register 15, called the Vital Pointer contains the bit address of the first register of the Environment Register group (Register #1A).

Register 16 (ONE) contains a one in its coefficient portion and a zero in its exponent. This register may, therefore, be accessed to obtain the fixed or floating-point (unnormalized) representation of the number one.

Register 17 (PD) contains the Parameter Descriptor. It contains the number of the parameters being passed during a call in the length portion of the register. The address portion contains zero if the parameters are in the register file, and the address of the parameter-list if the parameters are in virtual memory.

Registers 18 (FV1) and 19 (FV2) are used for storing function results obtained from some called subroutine. For example, the result of a trigonometric or exponential function would be placed in register 18. Register 19 is used when a result has two components; for example, the imaginary part of a complex number whose real part is returned to register 18.

ENVIRONMENT REGISTERS

These registers, six in number, are used to save and restore the status of the register file when executing separate callable routines.

Register - 1A - Return – holds the bit address of the caller to which the callee normally returns.

Register 1B - Dynamic Space Pointer – contains the bit base address of the next assignable location of dynamic space.

Register 1C - Current Stack Pointer – contains the bit base address of the region in the dynamic stack for storing the register file. The minimum length of that region is the maximum number of registers the caller needs to save. During a call sequence, the caller sets the length portion of the Current Stack Pointer to the number of registers to be saved by the callee. Although the Current Stack Pointer is set up by the caller, the vector transmission to save the caller's register file is done by the callee. The minimum number of registers that can be saved is the number of Environment Registers, six.

Register 1D - Previous Stack Pointer – contains the number of registers and the bit base address where the caller's registers have been saved. The callee's Previous Stack Pointer is an exact copy of the caller's Current Stack Pointer.

Register 1E - Callee Data Base – contains the bit base address of the static space which was allocated to the module by the loader. The caller passes the callee the address of the callee's static space in the Callee Data Base register. If, at the time of the call, the caller has not been linked to the callee by the loader, the value of the Callee Data Base will be the data base address of the loader. The exponent portion of the Callee Data Base register will contain an ordinal used by the loader to determine which module is making the call.

Register 1F - On Unit – contains the bit base address of a stack of data in dynamic space which defines the action to be taken by interrupt and error handling routines for a given set of pre-defined conditions for the active modules. This register must be in a fixed location and be stored at each call in order to support the execution requirements of condition handling in block structured languages such as PL/1, ALGOL, implementation languages, etc. If this register were not a canonical register, it would be very difficult to have one language communicate with another.

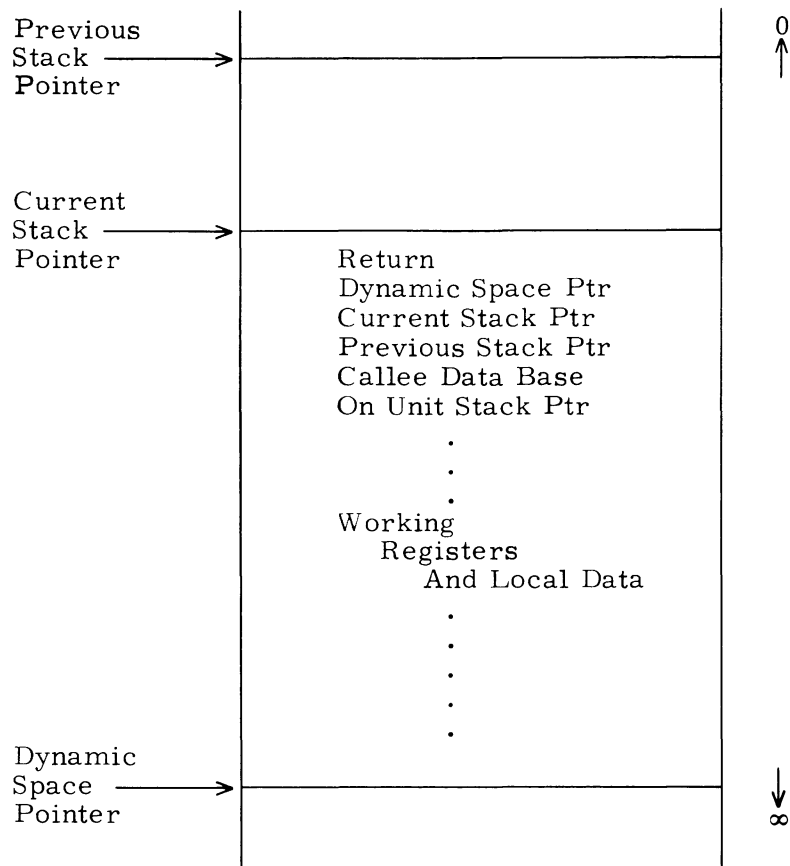
The Environment Registers are used in two areas of a Code Block Module called the prologue and epilogue. The instructions in the prologue and epilogue are inserted into the executable code by the assembler or compiler to ensure saving of the caller's register file when calling an external routine.

A program in process calls an external program. When this happens, the prologue of the called program executes code which saves the caller's register file in dynamic space and then transmits the Current Stack Pointer to the Previous Stack Pointer and the Dynamic Space Pointer to the Current Stack Pointer. Finally, the prologue loads the called program's register file from static space.

By this means, programs may call other programs to any desired depth. As one program calls another, dynamic space is built up with a stack of Regblocks, each containing the status of the register file when another program was invoked together with the linking information required for returning to the program. In the normal sequence, dynamic space increases until the lowest level called program has been executed; then the space contracts as the returns are encountered in the reverse order to the call.

Some programs are able to perform their tasks entirely within the temporary registers and do not invoke any other programs. Such routines need not contain a prologue and may be assembled or compiled to omit it.

Note that the registers are saved in a region of the user's Dynamic Space. Each user has his own Static and Dynamic Space. The organization of a region of dynamic space after several calls have been stacked appears as follows:



The epilogue or return from a called program is a macro compiled by the assembler or compiler. It normally performs two functions:

1. Restores the status of the Register File.
2. Transfers control to the RETURN location.

WORKING REGISTERS

These registers begin with register 20 (hex) and are available for general use by the programmer. Compilers and assemblers assign these registers in ascending order or, alternatively, keep track of register assignments made by the programmer. The working registers together with the environment registers constitute a contiguous register block which must be saved when a call is made to another program. In order to conserve time while saving the registers and restoring them later, only those registers which actually need to be saved are vector transmitted to dynamic space.

PARAMETER REGISTERS

The parameter registers are assigned by pairs beginning at register FF and working toward register 0. The even register of a pair contains the parameter descriptor of a parameter dossier which is intended to describe structured data such as arrays of arbitrary dimension or sparse arrays. Whether the parameter register contains a descriptor or a value is an option decided by convention between the caller and the callee.

Because the lower limit of the parameter registers is not defined, the programmer must take responsibility for preventing unwanted conflicts of register assignments. If the parameter list is long, an escape is provided; the Parameter Descriptor (PD) register may contain a descriptor to an extension of the parameter list. The parameter registers are not automatically saved between calls; hence, they may be used as temporary working registers. If a program contains calls to another program, the parameters in the calling program must, in general, be saved before the call is executed.

RELOCATION

All programs, whether compiled or assembled, are carried in the system in a self-relocating form. This means that a program ready for execution stored in virtual memory may be moved to any other portion of virtual memory for execution through a mechanism of pointers contained within the program itself. Such a structure permits a single image of any given program in the system which is at the same time relocatable and absolute. For object programs to retain the location and linking tables with each executing module might seem at first sight an unnecessary burden, but closer examination of system-wide storage requirements show this concept to be more efficient than any of the alternative approaches. For example, relocatable versions of programs appearing in the absolute libraries in large-scale environments must be available so that users may have private copies and so that the library may be repacked or restructured without recompiling.

It should be pointed out that it is possible and practical to write programs that are absolute. To do so merely requires that the addresses all be relative rather than absolute. Hence, absolute jumps typified by the B6 instruction and enter immediate instructions (specifying a label) such as the BE instruction would not be used.

MODULE TABLES

Previous paragraphs in this section have described the Register File conventions and the way in which programs may call other programs. In support of this facility, a number of tables are defined which enable the linker to load and link programs which may have been written by different persons using different compilers or assemblers. The actual linking of the programs is performed dynamically at execute time rather than during compile or assemble time.

The PL/* assembler and the compilers written for the STAR system produce groups of tables which collectively are called an "object module" or simply a "module." A module may simply be some elementary subroutine such as a conversion of a binary number to an ASCII string or it might be a complete assembler such as PLSTAR or BUFFALO containing calls to other modules. (Appendix A includes the names of all the modules current in the library; additional modules will be added as new facilities are developed.) The principal table in the module is called the code block table which contains, unsurprisingly, the object code; the remaining tables are used to facilitate relocation, linking, and loading.

Each table contains a standard two-word header of the following format:

NAME	
L	PTR

- NAME The literal name of the table in ASCII code.
- L The length of the table in 64-bit words including the two words in the header.
- PTR A 48-bit pointer which when added to the address of NAME gives the address of the Module Header Table.

The object module consists of several tables from the following list:

	<u>Header Name</u>
Module Header Table	MODULE
Code Block Table	CODE
External/Entry Table	EXT ENTR
Code Relocation Table	REL CODE
Interpretive Data Initialization Table	INT DATA
Executable Data Initialization Table	EXE DATA
External Data Initialization Table	EXN DATA
Interpretive Relocation Table	INT RELO
Executable Relocation Table	EXE RELO
External Relocation Table	EXN RELO

MODULE HEADER TABLE

The module header table provides information about the module such as its name, time of creation, the length of the code, the lengths of tables, and pointers to other tables associated with the module.

MODULE		} Standard two-word header
HL	PTR	
MODULE NAME		
DATE AND TIME CREATED		
T	PROCESSOR	
C	DATA BASE LEN	
TYPE	POINTER	
TYPE	POINTER	
TYPE	POINTER	

MODULE	Literal ASCII name "MODULE".
HL	Length of Module Header Table including the Standard Two-word Header.
PTR	0 for this table only.
MODULE NAME	ASCII name of the module.
DATE AND TIME CREATED	The date and time of the assembly/ compilation expressed as a 16-digit BCD number in the following order: year, year, month, month, day, day, hour, hour, minute, minute, second, second, millisecond, millisecond, millisecond, positive sign.
T	The length in 64-bit words of all tables in the module excluding the code-block table but including the module header table.
PROCESSOR	The STAR processing program used to generate the module; i.e., PLSTAR, FORTRAN, COBOL, etc.

C The length in 64-bit words of the code-block table including its standard two-word header and local data.

DATA BASE LEN The bit length of static space used by the module.

TYPE A 16-bit designator defining the type of table pointed to by POINTER. The table types are listed below:

<u>Hex Type</u>	<u>Literal ASCII Name</u>	<u>Description</u>
0000	MODULE	Module Header Table
0001	CODE	Code-Block Table
0002	EXT ENTR	External Entry Table
0003	REL CODE	Code Relocation Table
0101	INT DATA	Interpretive Data Initialization
0102	EXE DATA	Executable Data Initialization
0103	EXN DATA	External Data Initialization
0201	INT RELO	Interpretive Relocation Table
0202	EXE RELO	Executable Relocation Table
0203	EXN RELO	External Relocation Table

POINTER A 48-bit quantity which when added to the base address of the Module Header Table, points to the header of one of the above tables.

Before describing other tables associated with a module, the actual structure of a typical module should be examined. Figure 7-6 shows the structure of a typical program (module) assembled under PLSTAR. Six tables are used. Figure 7-7 shows an actual program assembled under PLSTAR. The same six tables are indicated.

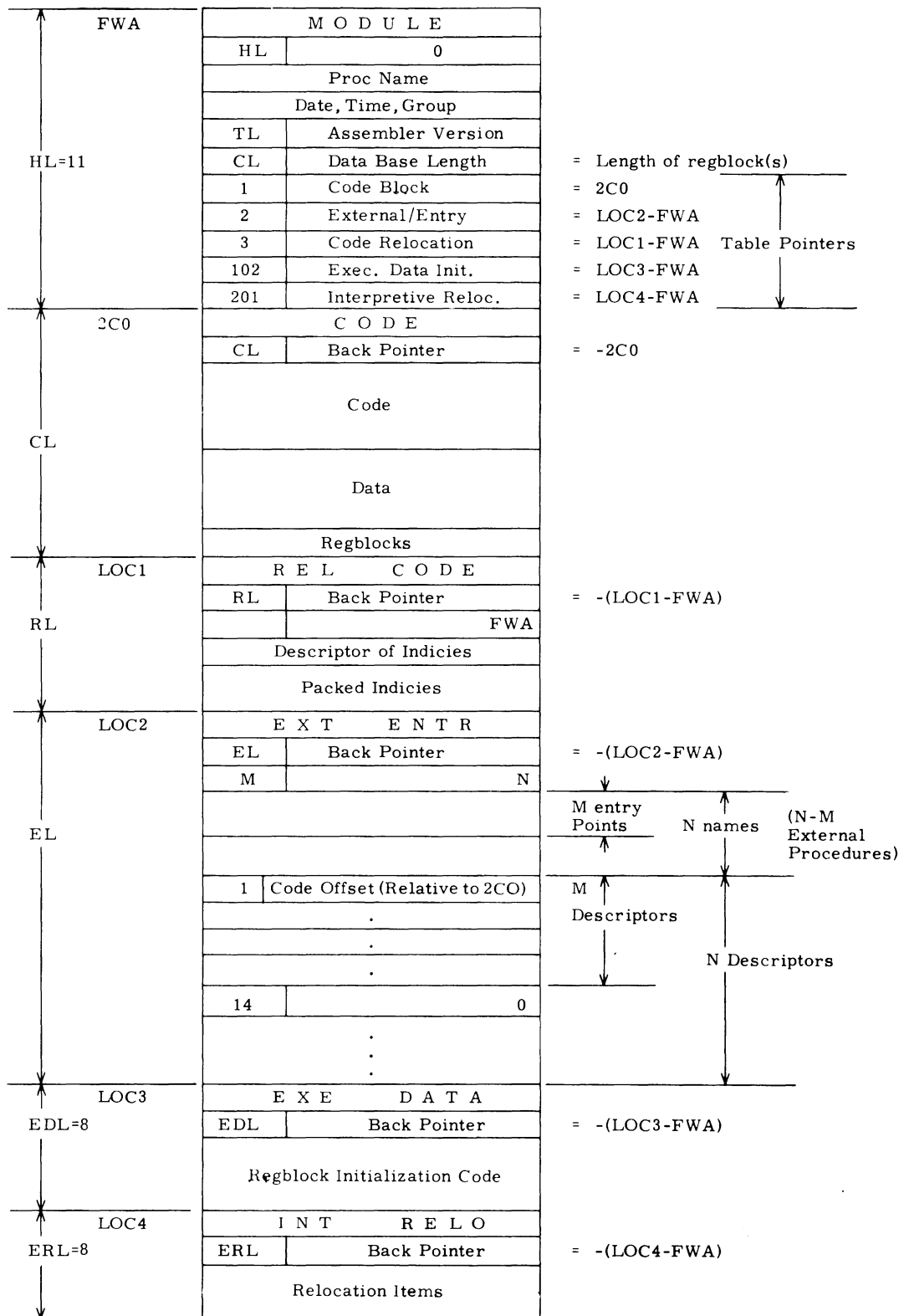


Figure 7-6. PL/* Object Module Format

TABLE TYPE	LOCATION	CONTENTS	COMMENTS
	8000	204D4F44554C4520	"M O D U L E" {Note 1}
	8040	0008000000000000	Length = 8 {Note 2}
	8080	4D4F444D41502020	"MODMAP"
	80C0	710831132027750A	Date and time {Note 3}
MODULE HEADER TABLE	8100	002C504C53545258	Tables = 2C words, "PLSTAR"
	8140	0066000000000880	Code = 66 words, static space
	8180	00010000000002C0	{Types and
	81C0	0002000000001D00	pointers to
	8200	0003000000001C40	other tables
	8240	0102000000002080	in the module.
	8280	0201000000002280	See note 4}
	82C0	2020434F44452020	"CODE"
CODE BLOCK TABLE	8300	0066FFFFFFFFFD40	Code = 66 words, pointer {Note 5}
	8340	3EFE0680381C00FE	First two instructions in the code block.
	9C00	0000000000000000	Last word in code block
	9C40	52454C20434F4445	"REL CODE"
CODE RELO TABLE	9C80	0006FFFFFFFFFE3C0	Length and pointer
	9CC0	000000000008000	Program base address
	9D00	0008000000000009	8 bits/index, 9 indices {Note 6}
	9D40	A2A6AAB2B6868A8C	values of the indices
	9D80	8E00000000000000	for code relocation
	9DC0	45585420454E5452	"EXT ENTR"
	9E00	0008FFFFFFFFFE240	Length and pointer
	9E40	0001000000000004	1 entry point, 4 names
	9E80	4D4F444D41502020	"MODMAP"
EXTERNAL ENTRY TABLE	9EC0	414444525F535350	"ADDR_SSP"
	9F00	43565F4454472020	"CV_PTG"
	9F40	4356482020202020	"CVH"
	9F80	0001000000000080	Type 1, Relative Address {Note 7}
	9FC0	0014000000000000	Type 14
	A000	0014000000000000	Type 14
	A040	0014000000000000	Type 14
EXE DATA INIT TABLE	A080	4558452044415441	EXE DATA
	A0C0	0008FFFFFFFFDF80	Length and pointer
	A100	3F0300803E040001	} Reg block initialization code
	A140	7E05040703070507	
	A180	8F07000000001440	
	A1C0	2A07002038070003	
	A200	9800000700000003	
	A240	3340006000000000	} INT RELO
INT RELO TABLE	A280	494E542052454C4F	
	A2C0	0008FFFFFFFFDD80	
	A300	00010000201010004	} 1st reloc. item
	A340	0000000000000002	
	A380	0001000401010006	
	A3C0	0000000000000004	} 2nd reloc. item
	A400	0001000601010000	
	A440	0000000000000005	} 3rd reloc. item

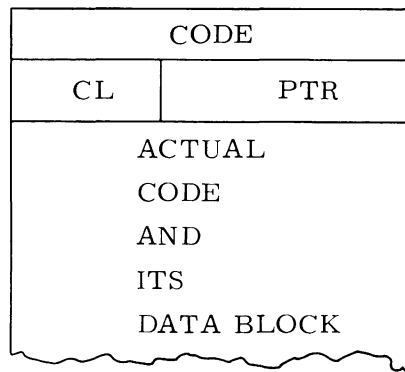
Note 1	Quote marks denote ASCII literal.
Note 2	All number representations are hexadecimal.
Note 3	This date-time representation is August 31, 1971 1:20:27.750 PM
Note 4	The pointers are indices relative to base address of the module (in this case 8000).
Note 5	All table header pointers are relative to the table name immediately preceding the pointer. The pointer added to the address of the table name immediately preceding the pointer always gives the base address of the module.
Note 6	To save space, the indices of locations which must be modified upon relocation are packed. The 16-bit exponent portion of this location specifies the length of the index in bits and the lower 48-bits of the location specify the number of indices in the list. In this example, the indices are 3 bits in length and there are 9 words to be modified. Each index is a half-word index relative to the beginning address of the module (#8000).
Note 7	This bit address is relative to the base address of the code block table. It points to the first executable instruction in the code block.

Figure 7-7. Program Assembled Under PL/*

The foregoing example shows the tables to be tightly packed around the code block table. However, the table structure with its pointers allows the tables to exist anywhere in virtual space. The example uses only six of the possible table types. At this point, only the Module Header Table has been fully described; descriptions of the remaining tables may now be undertaken.

CODE BLOCK TABLE

The code block table contains the executable code for the module and its local data such as translate tables, constants, etc. Modules are normally assembled or compiled with the module header at address 8000 (hex). The Code Relocation Table, to be described shortly, allows the absolute addresses to be modified (relocated) in the code block when the module is entered in the library.



CODE – Literal word CODE in ASCII

CL, PTR – See explanation of standard two-word header

EXTERNAL/ENTRY TABLE

The external/entry table contains the names of all entry points in the module, the names of external symbols, and common blocks. Associated with each name is an entry descriptor or external descriptor providing linking information for the linker/loader.

The following types of entry points and external symbols are defined:

- ENTRY POINTS

An entry point is a named value defined in the procedure and is intended to be referenced as an external by an external procedure. An entry pointer can have one of three types of values – an address in the code block, an address in the data section, or a constant value.

- COMMON BLOCKS

A common block is a named alterable space referenced by one or more procedures. A common block can be initialized with relocatable data. Blank common is a common block with a name of eight spaces.

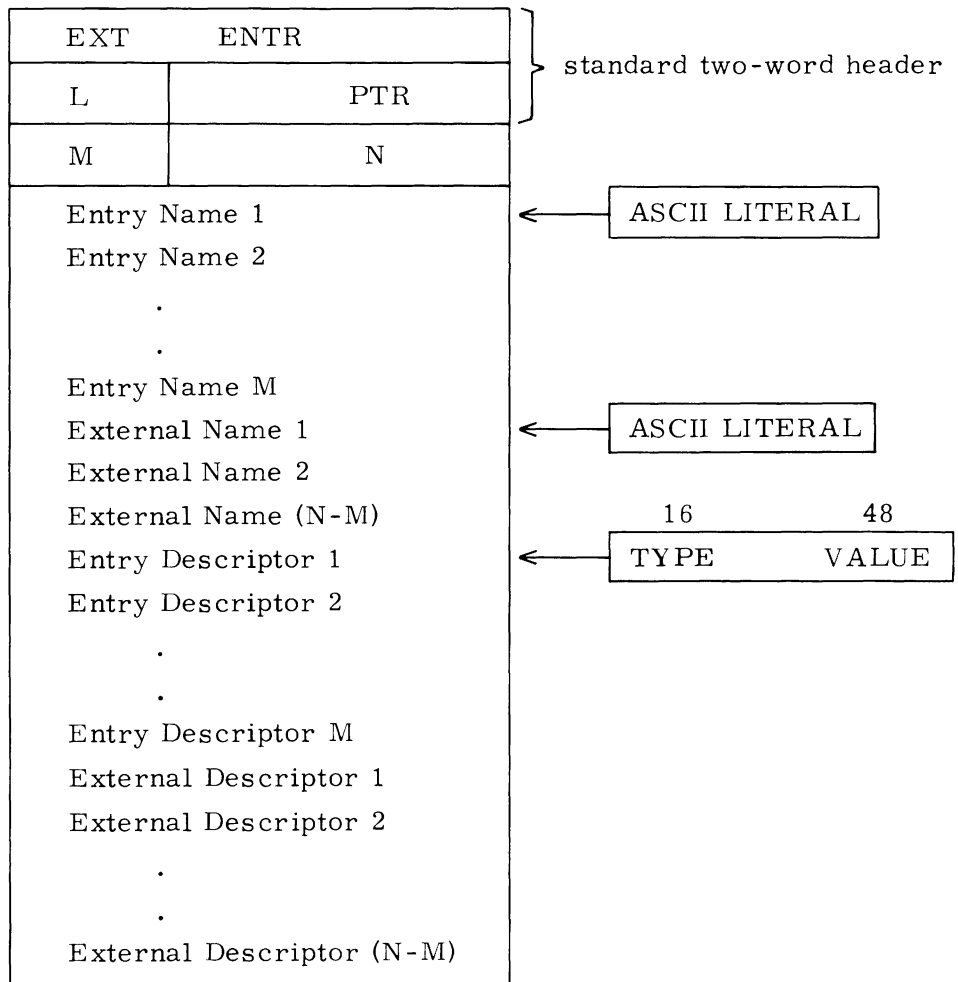
- EXTERNAL DATA

An external data is an external that is referenced by a method other than a procedure call.

- EXTERNAL PROCEDURE

A name defined as an external procedure should not be used as an external data. The standard method of using an external procedure reference is in the procedure call.

Having a symbol multiply defined (that is, as a common block, external procedure and as an external data) is specifically allowed.

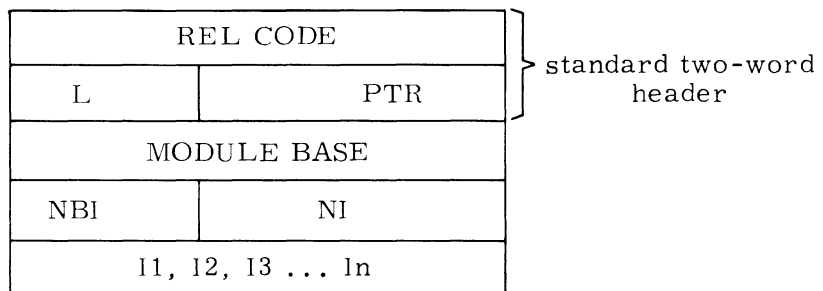


- | | |
|---|--|
| M | Number of entry points in the module |
| N | Total number of entry points plus external names |
| ENTRY NAME | A valid entry point name in the module |
| EXTERNAL NAME | The name of an entry point external to the module being run |
| ENTRY DESCRIPTOR
OR
EXTERNAL DESCRIPTOR | The descriptors each contain a 16-bit type designator in the exponent portion of the word and a 48-bit value in the coefficient. The type field defines the type of the value field. The value field contains information about the symbol name. The descriptors are paired one-for-one with the Entry Names and External Names. |

- Type = 1 Entry point in code. VALUE is a bit address relative to the Code Block header. It points to the first executable instruction in the code block.
- Type = 2 Entry point in data. VALUE is relative to data base bit address.
- Type = 3 Constant entry point. VALUE is a constant.
- Type = 14 External procedure. VALUE = 0.
- Type = 15 External data. VALUE = 0.
- Type = 16 Common block. VALUE is the bit length of the common block.

CODE RELOCATION TABLE

The code relocation table contains indices pointing to locations in the code block which are relocatable. If the code is location-independent, this table will not exist.

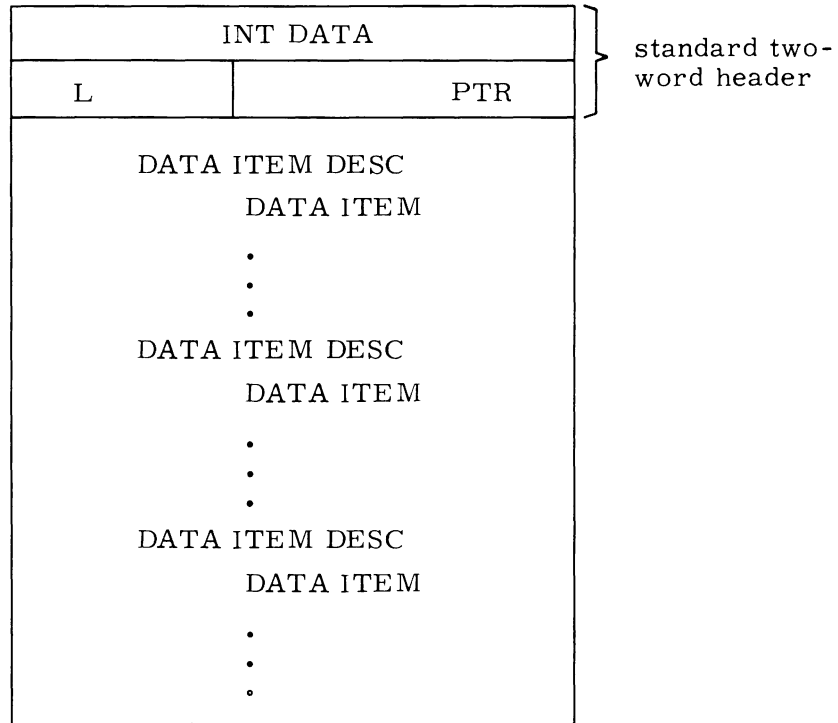


- MODULE BASE Base address of the module
- NBI Number of bits per index of the bit string starting in word 5.
- NI Number of indices in the string.
- I1, I2, ... In Full-word indices each NBI long. Each index references a full word in the code relative to the base address of the module.

As the result of processing this table, the bit base address of the module will be added to the 48-bit address fields of words pointed to by the indices in the index list.

INTERPRETIVE DATA INITIALIZATION TABLE

This table contains information which, when processed by the loader, results in the initialization of areas of static space with constant and relocatable data with the exception of procedure externals.



DATA ITEM DESC - A one-word descriptor of the data item:

0	15 16	31 32	39 40	47 48	63
ORD1	ORD2	TYPE	MODE	CHAIN	

ORD1 - pseudo-address vector ordinal of the static space to be initialized

ORD2 - pseudo-address vector ordinal relative to which relocation is to be done (relocation base)

TYPE - data item type

MODE - 00 values \longrightarrow destination

 01 values + relocation base \longrightarrow destination

 02 relocation base + destination \longrightarrow destination

When MODE=00, the values in the item are stored directly into the destination fields and ORD2 is ignored.

When MODE=01, the relocation base is added to the values before they are stored into the destination fields.

When MODE=02, the relocation base is added to the destination fields. Note that for MODE=2, the values are unnecessary and are, therefore, not present.

The pseudo-address vector is a table of addresses maintained by the linker for each user. The contents of the table are ordered such that ordinals ORD1 and ORD2 are full-word indices to this table. The organization of the pseudo-address vector is as shown below:

Pseudo-Address Vector

<u>Word</u>	<u>Contents</u>
0	Code table base address
1	Data base address
2	First external entry point
3	Data base of first external
4	Second external entry point
5	Data base of second external
6	Third external entry point
7	Data base of third external, etc.

The types of data to be initialized are listed below:

<u>Type</u>	<u>Description</u>	<u>Data Item Format</u>
1	Full-word broadcast	1
2	Half-word broadcast	1
3	Full-word vector transmit	1
4	Half-word vector transmit	1
5	Full-word sparse vector	2
6	Half-word sparse vector	2
7	Full-word index list	3
8	Half-word index list	3
9	Byte string	1
A	Bit string	1
B	Sparse structure	4
C	Character broadcast	1

The four data item formats are shown below:

L	INDEX
VALUE	
VALUE • • • VALUE	

- L Length in terms of the data type (full-word, half-word, byte or bit).
- INDEX Index relative to the address specified by ORD1. The index is dependent upon the data type. For example, it is expressed in bytes for byte strings and in full-words for full-word broadcast or vectors.
- VALUE Data dependent on type as below:
- Type 1 A full word to be broadcast in consecutive full-word locations starting at REL ADDR.
 - Type 2 A left adjusted half-word to be broadcast in consecutive half-word addresses starting at REL ADDR.
 - Type 3 A vector of full-words to be stored at consecutive addresses starting at REL ADDR.
 - Type 4 A vector of half-words to be stored at consecutive half-word addresses starting at REL ADDR.
 - Type 9 A left-adjusted byte string.
 - Type A A left-adjusted bit string.
 - Type C Left-most byte of value is broadcast.

DATA ITEM FORMAT 2

L	INDEX
VALUE · · · VALUE	
L2	BIT STRING
BIT STRING · · ·	

- L Number of values
- INDEX Index relative to the address specified by ORD1. It is descriptive of the data type employed; that is, for format 2, half-word or full-word index.
- VALUE The full- or half-word sparse vector.
- L2 Length of control vector in bits.
- BIT STRING The values of the control vector associated with the sparse vector.

DATA ITEM FORMAT 3

L	INDEX
VALUE · · · VALUE	
NBI	NI
STRING · · · STRING	

L	Number of full or half-word values.
INDEX	Index address relative to the address specified by ORD1. It is descriptive of the data type employed; that is, for format 3, full- or half-word index.
VALUE	Full- or half-word value.
NBI	Number of bits per index.
NI	Number of indices (=L).
STRING	Indices (connected end-to-end) associated with VALUES.

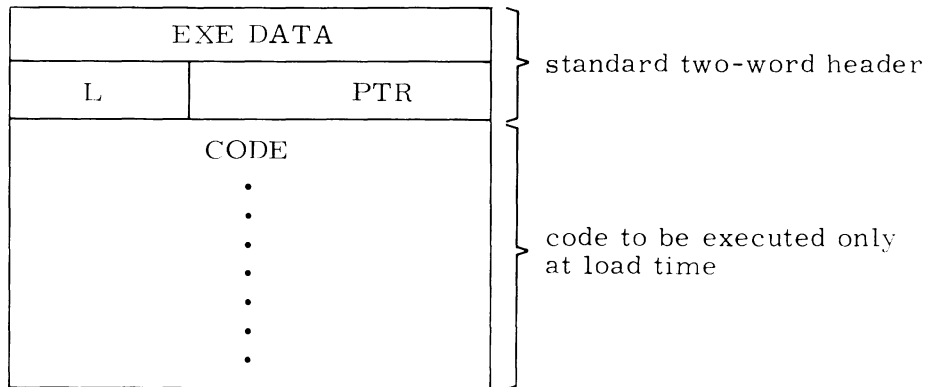
DATA ITEM FORMAT 4

L	INDEX
TYPE	ND
DESC 1	DESC 2
• • • •	DESC N
VALUE	
VALUE • • VALUE	

L	Number of items in value field.
INDEX	Index address relative to the address specified by ORD1. It is descriptive of the data type; that is, bit byte, half- or full-word.
TYPE	Type of value (word, half-word, byte string, bit string).
ND	Number of descriptors.
DESC1 DESC N	Half-word descriptors of data.
VALUES	Data.

EXECUTABLE DATA INITIALIZATION TABLE

The executable data initialization table allows a language processor to produce code that will be executed only at load time for the purpose of initialization of static space with data (excluding procedure externals). Thus, complex operations and non-standard initialization can be accomplished. The executable data table is processed after the interpretive data table.



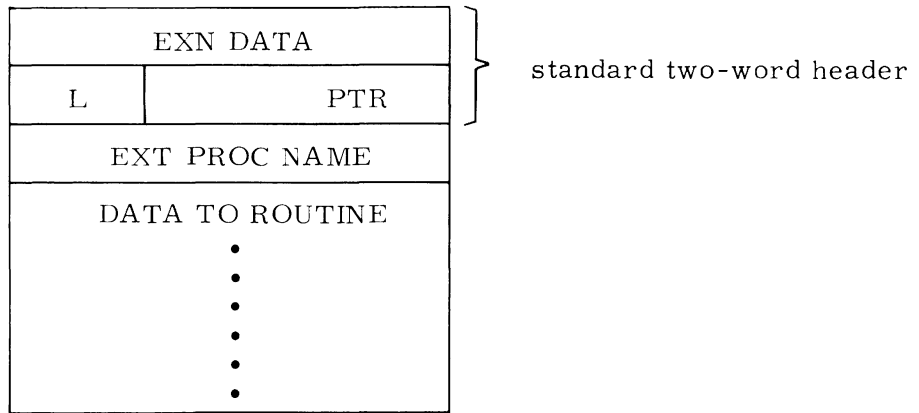
Note: The code expects the following parameters in registers 3 through 6:

Register 3	Data base address of the module
Register 4	Address of pseudo-address vector
Register 5	Address of executable data table
Register 6	Return address

If the code requires more registers than are available in the temporary area of the register file, then the register file must be saved and restored according to standard conventions.

EXTERNAL DATA INITIALIZATION TABLE

This table is similar to the EXE DATA table in that the code defined by the table is executed at load time. Instead of the code being a part of the module, however, it is contained in an external procedure.



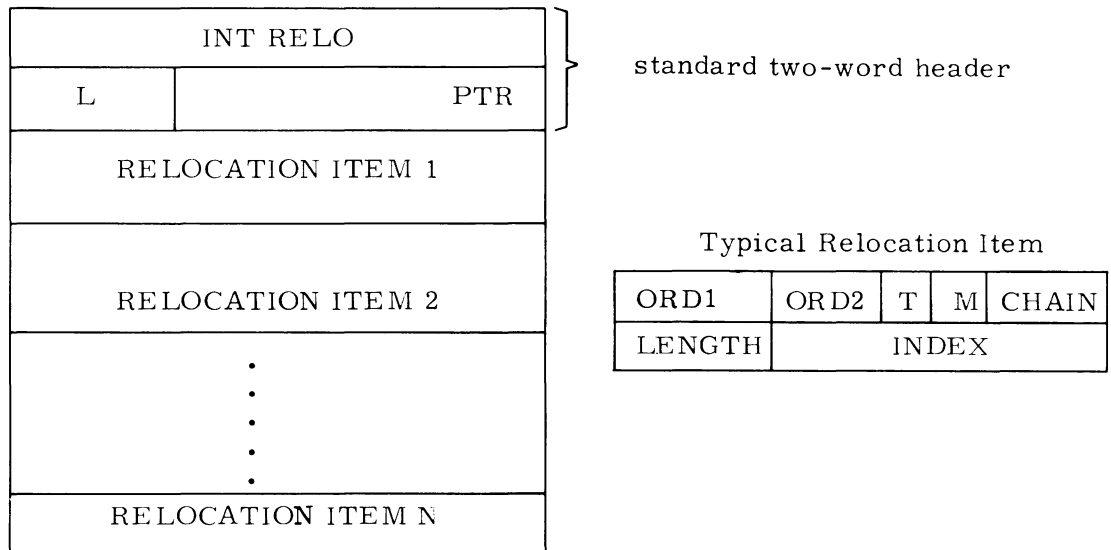
EXT PROC NAME The literal ASCII-coded name of the external procedure to be executed at load time.

DATA TO ROUTINE Data or parameters as required by the named external procedure.

Note: Registers 3 through 6 are defined and used in the same way as for EXE DATA.

INTERPRETIVE RELOCATION TABLE

This table contains information which, when processed by the loader, results in the initialization of areas of static space with procedure externals. This table is processed when the module of which it is a part is linked and also when other modules are linked as a result of a call within this module.



ORD1	pseudo-address vector ordinal destination	
ORD2	pseudo-address vector ordinal of relocation.	
TYPE	data item type. Only types 1, 3, 5, 7 are defined.	
MODE	00	64-bit field
	01	128-bit field

Item formats are similar to initialization table formats but do not contain VALUES.

TYPE=01. Full word broadcast

- mode=0. A single word from the pseudo-address vector is placed into one or more contiguous words in memory.
- mode=1. If the length is one, two contiguous words from the pseudo-address vector are placed into two contiguous words in memory. If the length is greater than one, the results are undefined.

TYPE=03. Full word vector transmit

- mode=0. Contiguous words from the pseudo-address vector are stored into memory.
- mode=1. Two words from the pseudo-address vector are stored into memory. The results are undefined if the length is greater than one.

TYPE=05. Full word sparse vector

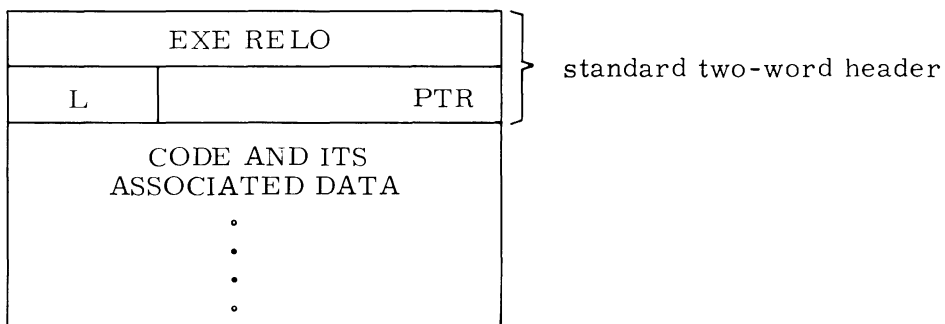
- mode=0. A single word from the pseudo-address vector is stored in memory.
- mode=1. A word pair from the pseudo-address vector is stored into word pairs in memory.

TYPE=07. Full word index list

- mode=0. See TYPE=05 mode=1.
- mode=1. See TYPE=05 mode=2.

EXECUTABLE RELOCATION TABLE

This table is similar in format and function to the executable data initialization table. Its purpose is to provide non-standard procedure externals in the code block by executing the code contained within this table. The code is executed at load time.

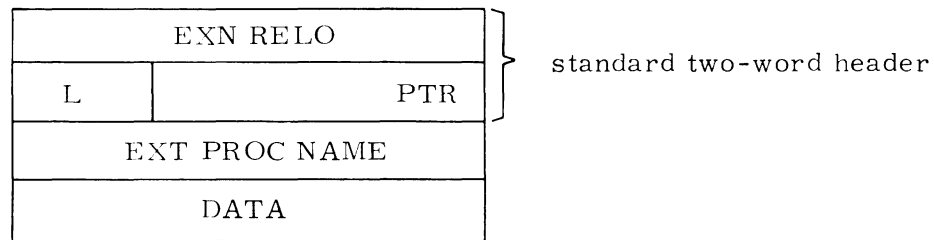


Note: The code expects the following parameters in registers 3 through 7:

Register 3	Data base address of the module
Register 4	Address of pseudo-address vector
Register 5	Address of the executable relocation table
Register 6	Return address
Register 7	Pseudo-address vector ordinal or the symbol that has been linked.

EXTERNAL RELOCATION TABLE

This table is similar in format and function to the external data initialization table. The purpose of this table is to provide for non-standard procedure externals in the code block by executing an external procedure whose name(s) is/are contained in the table.



EXT PROC NAME The literal ASCII-coded name of the external procedure.

DATA Contains data to be used by the external procedure.

Note: Registers 3 through 7 are used in the same way as for the executable relocation table - EXE RELO.

JOB CONTROL

Job Control consists of a collection of modules which initialize a user's virtual memory (as described in the preceding paragraphs) and control the sequencing of the user's job.

The Job Control program has the ability to interpret "primitive" command statements. It also handles exceptional conditions such as breakpoint, illegal instruction, pause, etc. A flow diagram of Job Control is shown in Figure 7-8.

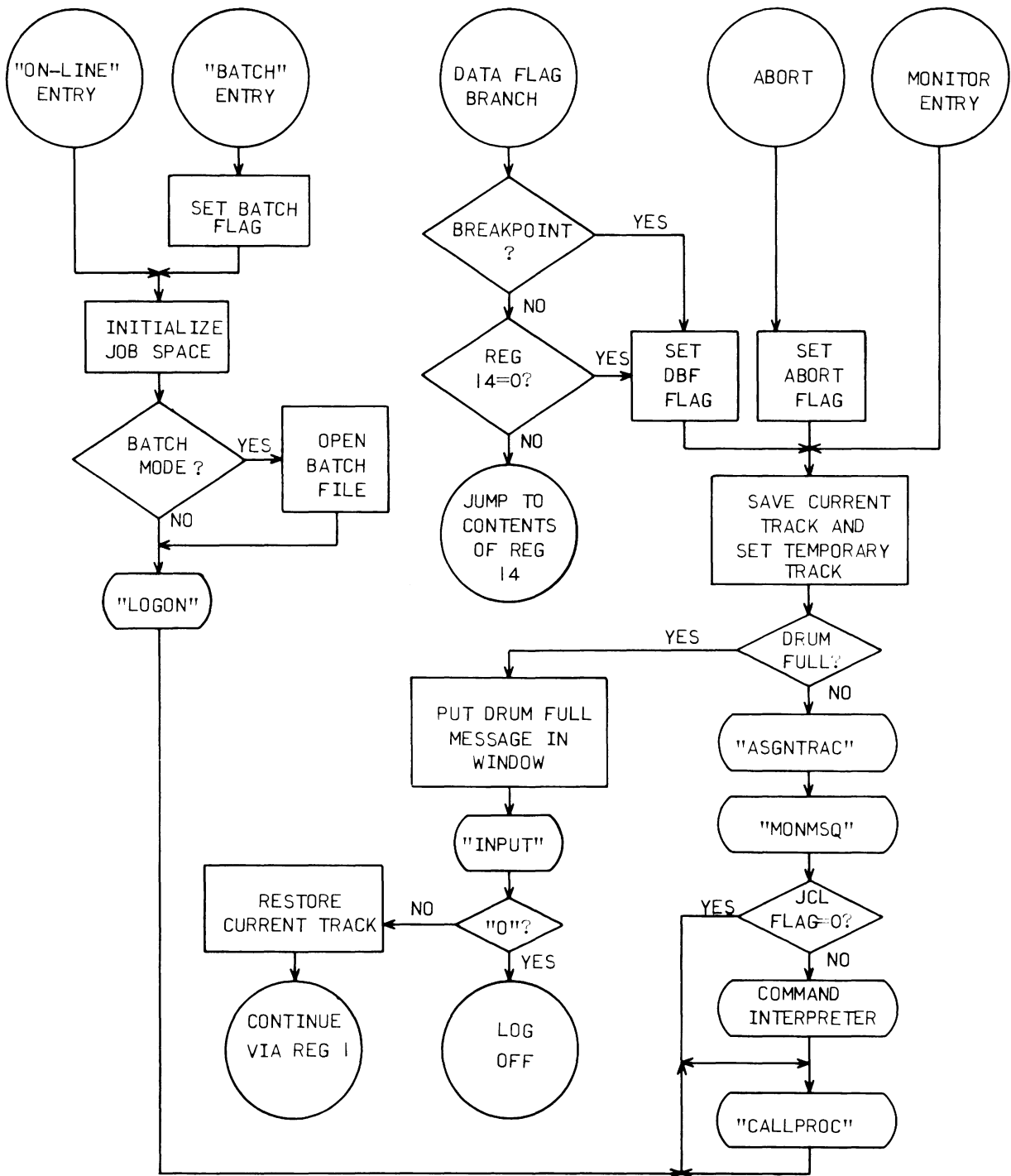


Figure 7-8. Job Control Flow Diagram

The system library consists of a number of callable subroutines. These subroutines are called modules and are assembled in relocatable form for eventual inclusion in the library.[†] Modules in library are write protected and shared by all users.

A module may contain multiple, and usually related, functions which are differentiated by separate entry points. A typical example is the module named COMPRESS that has an entry point named EXPAND to perform the inverse process.

The following pages in this section list the entry points for all modules which were cataloged at time of publication. It should be noted that some entry point names are identical to certain commands in JCL1 or EDIT. Such duplication of names is permissible where a "mode" is entered in which the commands are interpreted within the module.

[†] The large, and growing number of modules in the library necessitates the publication of a separate document containing their descriptions. This publication is entitled: STAR-100 LIBRARY SUBROUTINES.

<u>PROC NAME OR ENTRY POINT</u>	<u>PROGRAMMER</u>	<u>PURPOSE OF PROGRAM</u>
ABANDON	Hawley	Abandons designated track
ACCOUNT	Hawley	Formats accounting summary from accounting file
ACOS	Hartnett	Evaluates the arccosine of a real number
ACOSF	Hartnett	Same as ACOS
ADDMOD	Hawley	Adds and/or replaces module in a structured file
ADDTOCAT	Hawley	Inserts a name and corresponding value in user catalog
ALLOC_SS	Hawley	Allocates static space
ALOG	Hartnett	Evaluates the natural logarithm of a real number
ALOG10	Hartnett	Evaluates the base10 logarithm of a real number
ALOG10F	Hartnett	Same as ALOG10
ALOGF	Hartnett	Same as ALOG
ASGNTRAC	Hawley	Assigns a new track (a job control used module)
ASIN	Hartnett	Evaluates the arcsine function of a real number
ASINF	Hartnett	Same as ASIN
ATAN	Hartnett	Evaluates the arctan function of a real number
ATAN2	Hartnett	Evaluates the tangent function of a ratio of real numbers
ATAN2F	Hartnett	Same as ATAN2
ATANF	Hartnett	Same as ATAN
B	Hawley	Sets breakpoint in current track
BACKSPCE	Untulis	Backspaces records of a file
BACKTRAC	Hawley	Returns to JCL1 in track one - abandon all other tracks
BAIL OUT	Untulis	Stores and closes all active files
BATCH	Untulis	Runs a file as a batch job
BK	Hawley	Sets breakpoint in designated track
BUFFALO	Toth	Assembler for Buffer Controller
BUFFSYM	Van Hatten	Entry point in BUFFALO
C	Hawley	Continues the execution of a broken track
CALC	Hawley	Provides calculator capability from CRT terminal
CALLPROC	Hawley	Processes primitive call statements
CATALOG	Hawley	Puts entry point names in user catalog
CLEARFM	Untulis	Clears FRMT† pointers
CLOSE	Untulis	Closes a file to I/O
COMPARE	Hawley	Compares two byte strings for equality

†FRMT = File Record Management Table

<u>PROC NAME OR ENTRY POINT</u>	<u>PROGRAMMER</u>	<u>PURPOSE OF PROGRAM</u>
COMPAREC	Hawley	Continues comparison of two byte strings
COMPRESS	Toth	Conserves file space by compressing multiple spaces
COS	Hartnett	Evaluates the cosine function of a real number
COSF	Hartnett	Same as COS
COSH	Hartnett	Evaluates the hyperbolic cosine function of a real number
COSHF	Hartnett	Same as COSH
CSC	Hartnett	Evaluates the cosecant function of a real number
CSCF	Hartnett	Same as CSC
CTN	Hartnett	Evaluates the cotangent function of a real number
CTNF	Hartnett	Same as CTN
CUBE ROO	Hartnett	Same as CUBRT
CUBRT	Hartnett	Evaluates the cube root of a real number
CVH	Hawley	Converts a bit string to ASCII hex characters
CV_DTG	Hawley	Converts date-time group from packed to printable format
D	Hawley	Displays a region of virtual memory in hexadecimal
DACTFL	Untulis	Displays names of active files
DATE TIME	Hawley	Furnishes current date and time
DAYFILE	Hartnett	Places text in user dayfile
DELETE	Untulis	Deletes file from FRMT † and mass storage
DFILE	Untulis	Displays file
DFRMP	Untulis	Displays FRMT † descriptors for active files
DFRMT	Untulis	Displays FRMT † entry for file
DFSTRUCT	Untulis	Displays structure information for file
DISDAYFL	Hartnett	Displays the last ten user dayfile messages
DISPLA	Hartnett	Displays in user dayfile a variable name and its value
DM	Hawley	Displays a designated portion of a named module
DMPDAYFL	Hartnett	Dumps the user dayfile
DMPFT	Untulis	Dumps all FRMT † information for all files
DNMSTR	Untulis	Displays active file name string
DNMTBL	Untulis	Displays active file internal names

†FRMT = File Record Management Table

<u>PROC NAME OR ENTRY POINT</u>	<u>PROGRAMMER</u>	<u>PURPOSE OF PROGRAM</u>
DP_ADD	Hawley	EM-1 Prototype disc system index manipulation
DP_DEL	Hawley	
DP_DIR	Hawley	
DP_HASH	Hawley	
DP_STAT	Hawley	
DP_TMPD	Hawley	
DR	Hawley	Displays registers from designated track
DREC	Untulis	Displays specified record of file
DRP	Hawley	Displays returned parameters from previous call
DSENSE	Hartnett	Displays status of sense lights and sense switches
DT	Hawley	Displays portion of designated track
DUMP	Hawley	Provides listable dump of virtual memory
DUMPT	Hawley	Dumps portion of designated track
DYNALINK	Hartnett	Module name for dynamic linking routines
D_BIN	Hawley	Generates binary card format from virtual memory
D_HEX	Hawley	Produces hex card images (ZAP format) of virtual memory
D_STRUCT	Untulis	Displays structure descriptor for file
E	Hawley	Enters data in virtual memory and displays the new data
EDIT	Hawley	Provides a line edit facility for source files
EDIT_C	Hawley	Re-enters EDIT with indices as they were before leaving EDIT
EN_DT_TM	Hawley	Initializes date and time for system use
ER	Hawley	Enters register in designated track
ERROR	Untulis	Displays error message and abort program
ET	Hawley	Enters data in designated track
EXP	Hartnett	Evaluates the exponential function of a real number
EXPAND	Toth	Expands compressed spaces in source file
EXPF	Hartnett	Evaluates the exponential function of a real number
E_A	Hawley	Enters data in virtual memory with modified display
FADDMOD	Untulis	Adds a module to or changes a structured file
FILER	Hawley	Creates and maintains files of sym- bolic files
<u>FILESTAT</u>	Untulis	Finds FRMT [†] descriptor for file

[†]FRMT = File Record Management Table

<u>PROC NAME OR ENTRY POINT</u>	<u>PROGRAMMER</u>	<u>PURPOSE OF PROGRAM</u>
FIL_STAT	Untulis	Displays status of FRMT [†] entry for file
FINDTRAC	Hawley	Locates base address of stored track information
FORMAT C	Bolduc	Cracks FORTRAN format statements
FORTRAN	Lincoln	General FORTRAN Compiler
GETPAR	Hawley	Subprogram of CALLPROC for processing parameters
G_STRUCT	Untulis	Gets structure descriptor for file
HILB	Holeman	Generates an nxn Hilbert matrix
HYPER	Hartnett	Module name for hyperbolic function evaluation
IMPL IO	Untulis	FORTRAN I/O processor
INPUT	Hawley	Obtains next line of input from terminal or batch file
INPUTC	Bolduc	Converts ASCII string to internal values
INV	Holeman	Computes the inverse of an nxn matrix
JCL1	Lincoln	Job control language
JCL1EXIT	Lincoln	Exit from JCL1
JOBCON	Hawley	Job mode monitor program
LINENUM	Hawley	Puts page and line numbers on a text file
LINKER	Hartnett	Provides a dynamic linking capability
LNKMAP	Hartnett	Provides a printable map of the link catalog
LLGO	Hawley	Loads register file from virtual memory and starts execution
LOCMOD	Hawley	Locates the base address of a module
LOGON	Hawley	Processes ID and password for log-on
LOGRTHM	Hartnett	Module name for processing logarithmic functions
L_BIN	Hawley	Loads virtual memory from binary card formats
L_HEX	Hawley	Loads hex card images (ZAP format) into virtual memory
MCAT	Hawley	Adds and/or updates entry in master catalog
ML1	Hawley	ML1 macro processor
MODMAP	Hawley	Provides a printable map of user catalog and library catalog
MONMSG	Hawley	Displays translated error code from monitor
MOVE	Hawley	Moves data from one area of virtual memory to another
MOVED	Hawley	Same as MOVE but character delimited
MPIPE	Curtis	STAR multipurpose unit logic simulator
MRGPASS	Hawley	Subprogram of SORT-merges two sets of strings

[†]FRMT - File Record Management Table

<u>PROC NAME OR ENTRY POINT</u>	<u>PROGRAMMER</u>	<u>PURPOSE OF PROGRAM</u>
MULT	Holeman	Multiplies two nxn matrices
OFF	Hawley	Terminates job and disconnects
OFF_PACK	Hawley	Logically turns off a disc drive
OFF_SW	Hartnett	Turns off a sense switch
OLDMAP	Hawley	Maps in file using EM-1 prototype message format
OLDSTORE	Hawley	Stores file using EM-1 prototype message format
ON_PACK	Hawley	Logically turns on a disc drive
ON_SW	Hartnett	Turns on a sense switch
OPEN	Untulis	Prepares a file for I/O
OUT	Untulis	Moves data between data areas established by OUTI
OUTE	Untulis	Stores data assembled in the destination area by OUTI
OUTI	Untulis	Initializes and moves data from source to destination
OUTP	Untulis	Inserts page code in destination string
OUTRE	Untulis	Remote prints data assembled by OUTI
OUTPUTC	Bolduc	Generates ASCII string for printer
PLSTAR	Lincoln	STAR assembly language
PLSTARX	Lincoln	STAR assembly language
PRINT	Untulis	Prints a file
PRSTRUCT	Untulis	Prints structure information for a module
PUNCH	Untulis	Card punches a file
PURGE	Hawley	Deletes pages of a user's virtual memory/library from drum and core
P_STRUCT	Untulis	Stores structure descriptor for file in FRMT†
Q8NTRY	Untulis	Opens requested number of files for I/O
RANF	Hartnett	Repeated use generates a pseudo-random sequence of numbers
RBAIEX	Hartnett	Raises a real number by an integer exponent
RBAREX	Hartnett	Raises a real number by a real exponent
READ	Untulis	Moves data from file area to user area
READPACK	Hawley	Reads physical block(s) from disc pack into virtual memory
RELOC	Hawley	Moves and modifies program to execute at new virtual memory

†FRMT - File Record Management Table

<u>PROC NAME OR ENTRY POINT</u>	<u>PROGRAMMER</u>	<u>PURPOSE OF PROGRAM</u>
REMARK	Hawley	Synonym for DAYFILE
REMCARD	Hawley	Reads cards from 200-user terminal to virtual memory
REMPRF	Untulis	Remote prints a file
REMPRINT	Hawley	Prints on 200-user terminal printer from virtual memory
REWIND	Untulis	Rewinds file
S	Hawley	Steps the execution of a broken track
S2ED	Hawley	STAGE2 editor
SEC	Hartnett	Evaluates the secant function of a real number
SECF	Hartnett	Same as SEC
SECOND	Holeman	Returns value of real-time clock
SIN	Hartnett	Evaluates the sine function of a real number
SINCOS	Hartnett	Evaluates the sine/cosine function of a real number
SINF	Hartnett	Same as SIN
SINH	Hartnett	Evaluates the hyperbolic sine function of a real number
SINIIF	Hartnett	Same as SINH
SKIP	Untulis	Skips records on file
SLITE	Hartnett	Turns on/off sense lights
SLITET	Hartnett	Tests sense lights and turns them off
SOLO	Hawley	EM-1 prototype on-line SOI.O peripheral driver
SORT	Hawley	Sorts a file of variable length delimited records
SORTKEYS	Hawley	Subprogram of SORT
SQRT	Hartnett	Evaluates the square root of a real number
SQRTE	Hartnett	Same as SQRT
SQUEEZE	Toth	Compresses file by common digram substitution
SSWTCH	Hartnett	Tests sense switches
STATLINK	Hartnett	Processes module entry points prior to static linking
STATUS	Curtis	Displays control point status
STAT_LIN	Hartnett	Provides for static linking of modules
STG2	Hawley	STAGE2 macro processor
STORE	Untulis	Stores file on mass storage medium
S_ADDMOD	Hawley	Combines structured files
S_CAT	Hawley	Catalogs modules in a structured file
S_MCAT	Hawley	Adds and/or updates entries in master catalog
TACT	Bolton	File editor retaining history
TAN	Hartnett	Evaluates the tangent function of a real number
TANF	Hartnett	Same as TAN

<u>PROC NAME OR ENTRY POINT</u>	<u>PROGRAMMER</u>	<u>PURPOSE OF PROGRAM</u>
TANH	Hartnett	Evaluates the hyperbolic tangent function of a real number
TANHF	Hartnett	Same as TANH
TIME	Hartnett	Places a message in dayfile
TRACE	Untulis	Traces previous calling history
TRACETRK	Hawley	Displays previous calling history for designated track
TRACK	Hawley	Displays name of current track
UNIT	Untulis	Returns status of file
UNQUEUEZ	Toth	Expands a squeezed file
VEXP	Hartnett	Evaluates the exponential of a vector of real numbers
VEXPF	Hartnett	Same as VEXP
VLDEC	Hawley	Variable length decimal arithmetic interpreter program
VLDECADD	Hawley	Variable length decimal arithmetic and formatting subroutines
VLDECDIV	Hawley	
VLDECFMT	Hawley	
VLDECMUL	Hawley	
VLDECSCL	Hawley	
VLDECSUB	Hawley	
VLOG	Hartnett	
VLOG10	Hartnett	Evaluates the base10 logarithm of a vector of real numbers
VLOG10F	Hartnett	Same as VLOG10
VLOGF	Hartnett	Same as VLOG
VLOGRTHM	Hartnett	Module name for vector logarithm evaluation
WEOF	Untulis	Sends end-of-file bit in FRMT [†] status entry for file
WRITE	Untulis	Writes data from users area to file area
X	Hawley	Restarts execution of a broken track at a new address
XCUR	Hawley	Executes address in current track
XREF	Bolton	Provides cross-reference listing of a PLSTAR program

[†]FRMT - File Record Management Table

CARD FORMATS

B

CONTROL CARDS

There are essentially three separate types of control cards that can occur in a card deck. These are first card, last card, and record/group cards.

FIRST CARD

All card files must be preceded by this card that is processed by the input/output station but is not passed on as part of the file. The format is:

column 1	ASCII file separator	11-4-8-9 punch
column 2	action (ends with space)	
next columns	local file name (ends with space)	
next columns	user identifier (ends with space)	
next columns	user password (ends with space) - optional	
next columns	file length - two hex. digits (ends with space) - optional	
next columns	zip code - four hex. digits - optional	

Actions presently include STORE, PUNCH, PRINT, EXECUTE, STORE_EXECUTE.

LAST CARD

All card decks must end with this card that is transmitted with the file into the store. The format is:

column 1	ASCII file separator	11-4-8-9 punch
----------	----------------------	----------------

RECORD/GROUP CARD

This card divides the card deck into logical units, that is, records or groups. For coded files, the card is transmitted as part of the file. With binary or foreign file types, the card is processed and used to build the record map; then it is removed from the file. A record card must precede all records except the first record of a coded delimited file which is the default case. The card format is:

column 1	ASCII record separator	11-6-8-9 punch
	or ASCII group separator	11-5-8-9 punch
	or any other specified delimiter character	in foreign files

columns 2-3	record type
column 4	next record delimiter (FD only)
column 4-on	number of cards in next record (FF only)
column 4	code type E = EBCDIC, B = BCDIC, A or blank = ASCII (coded files only)
column 5-on	number of characters in unit record (CF only)

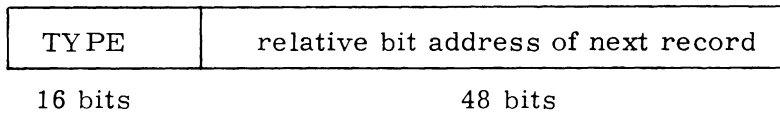
RECORD TYPES

CD	coded delimited
CF	coded fixed
BS	binary STAR
FD	foreign delimited
FF	foreign fixed

With FD records the next record delimiter punching must be specified, and with FF records the number of cards in the next record must be specified.

RECORD MAP

The record map for each binary/foreign record has a 64-bit entry of the following format:



The last entry of the map is zero. The record map is itself a file that bears the local name of the file to which a reverse slash is appended.

EM-1/STAR BINARY CARD FORMAT†

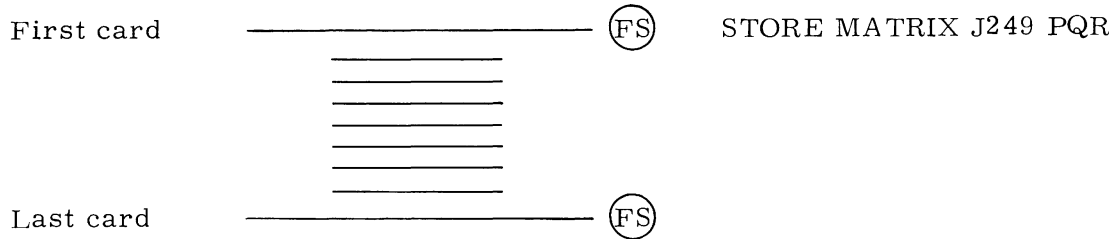
column 1 rows 12 through 5	count of characters on card
column 1 row 6	a 6 punch means ignore checksum
column 1 rows 7 through 9	7-9 punch
column 2	card sequence number (in ascending order with none missing)
columns 3, 4	checksum of characters on card
columns 5 through 80	data characters, three 8-bit characters in two columns. (if data does not fill card, extra characters are ignored)

† Binary common record type.

EXAMPLES OF CARD DECK FILES

Note that all control characters occur in column 1.

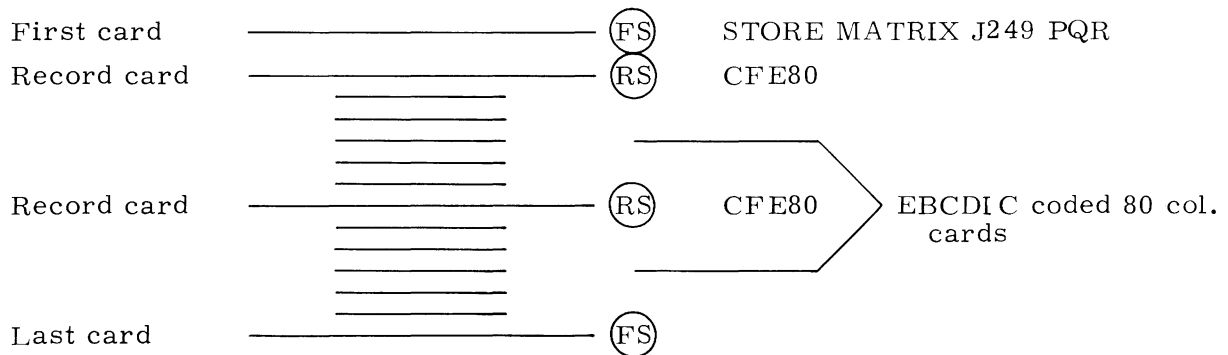
CODED DELIMITED



Characteristics:

- Unit separator inserted in file after each card
- Compressed blanks if ASCII code, ASCII ESC (1B) followed by count of blanks +30 hexadecimal
- No map
- Record/group control cards allowed and transmitted (compressed with unit separator)

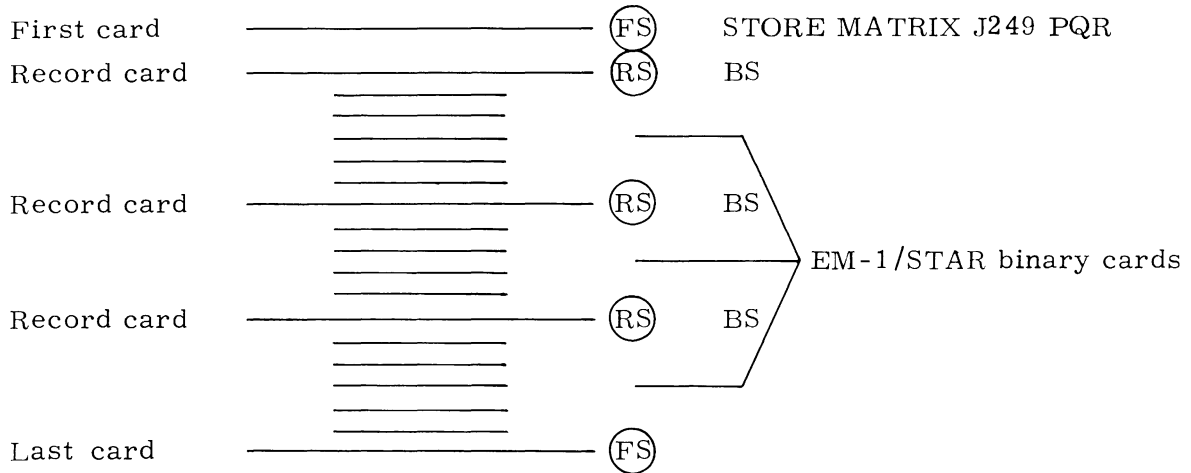
CODED FIXED



Characteristics:

- No unit separator inserted between cards
- No compression of blanks
- No map
- Record/group control cards allowed and transmitted (80 characters with no unit separator)

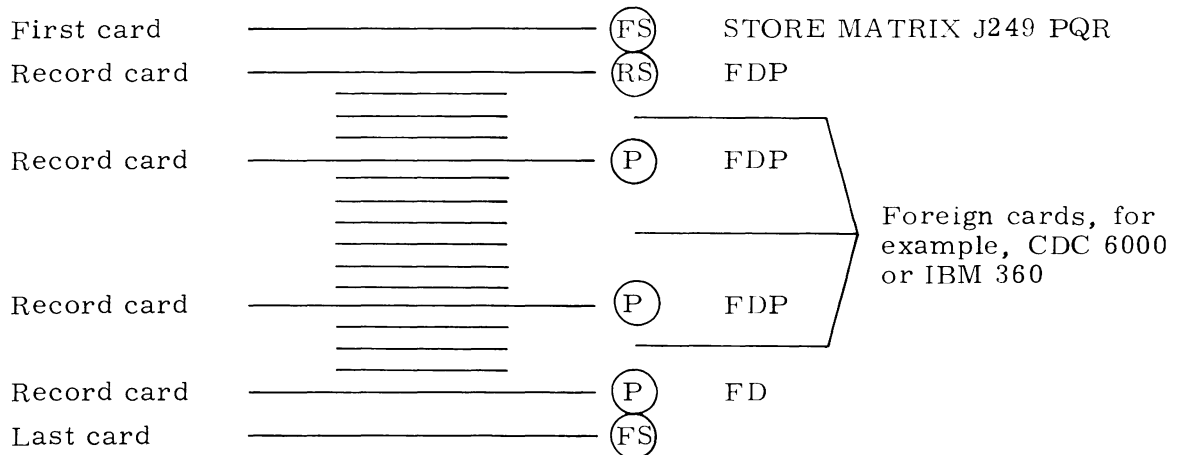
BINARY STAR



Characteristics:

- No unit separators
- Card processed to pure binary; checksum and sequence number checked and if illegal, reader stopped and operator informed
- Map created (type and record length entry for each record)
- All control cards except last card processed and removed

FOREIGN DELIMITED

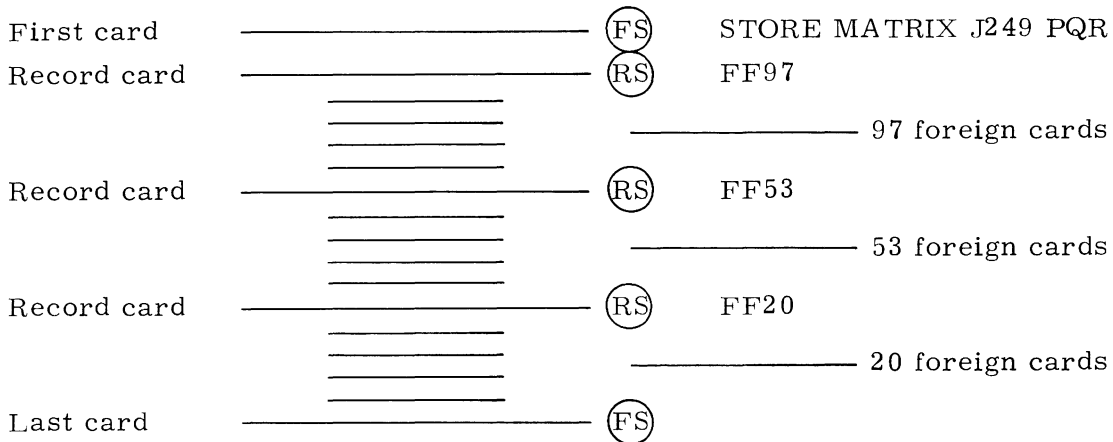


(P) is column punching of next delimiter

Characteristics:

- No unit separators
- Transmit whole card image, that is, 960 bits
- Map created, one entry for each record
- No control cards transmitted except last card

FOREIGN FIXED

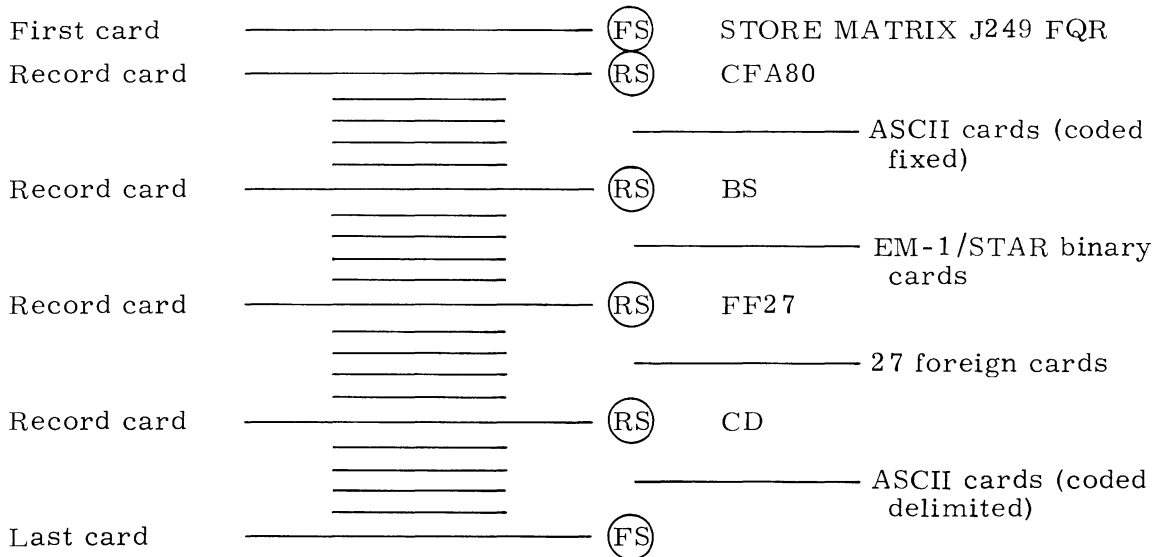


Characteristics:

- Same as for previous case; if count wrong, then reader stops with message for operator

MIXED DECK

Consists of any combination of any of the previous record types.



Characteristics:

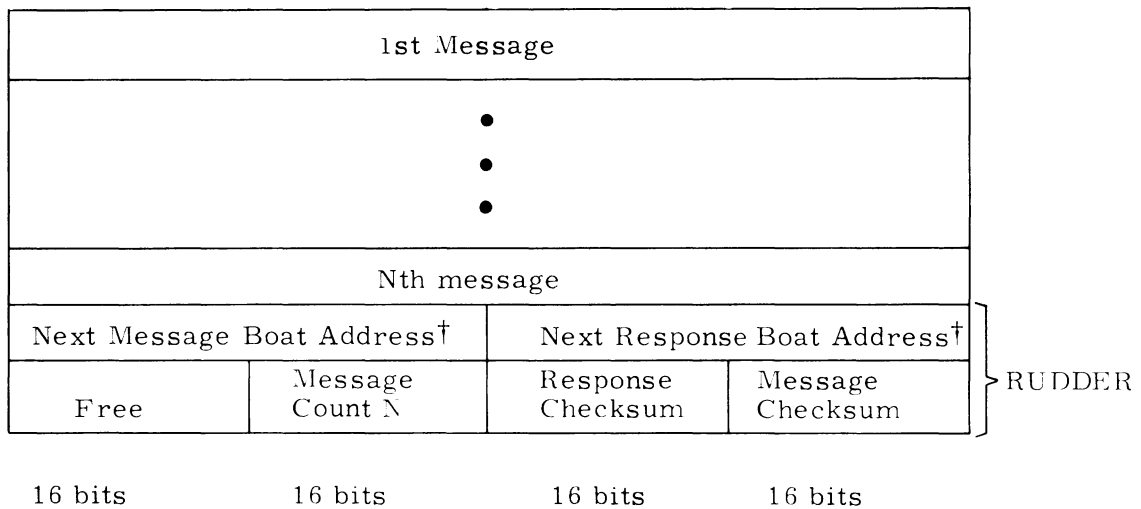
Characteristics of the individual sections apply, with the exception that the first control card of each section, whether coded, binary, or foreign, is transmitted with the file to delimit the new section.

SYSTEM COMMUNICATION MECHANISM

C

Stations communicate with the central operating system, and with each other via messages. Messages define certain tasks to be performed such as open or print a file. Each station has a list of messages that it will perform. Message queues are organized into message boats. A message boat may consist of several messages. Each message has a message header and a message body.

The format of the boat is:



† Bit address if boat originates in STAR, 16-bit word address if boat originates in station.

The format of each message is:

16 bits	4	4	8	16 bits	16 bits
Response Code	SB	HL-2	L	Used by Sender	Used by Sender
Used by Sender	To Zipcode			From Zipcode	Function Code
Message Parameters					

SB = Special bits

HL = Header length in 64-bit words

L = Total message length in 64-bit words

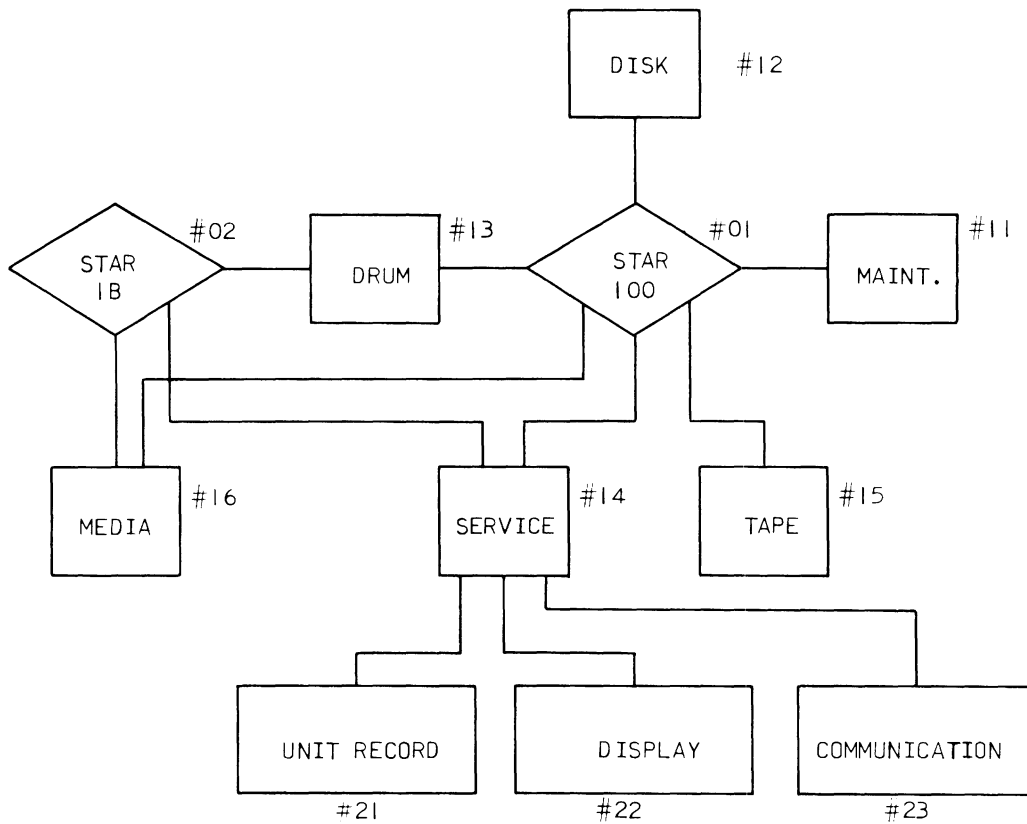
Interstation communication is managed by a message control routine which has four sections: receive message, send response, send message, receive response. Receive message on being flagged or at given time intervals reads the next boat from the "next message boat address" into SCU memory. The length of the boat is a preassigned station parameter. A test is then made of the message checksum to assure that it is nonzero and correct for the boat, then proceeds to activate the appropriate overlays, one for each function code, to process the messages. Each station may have up to N messages active concurrently, N being a station parameter. On completion of a message, the response is embedded back in the message area in the boat. When all messages are processed, the response checksum is set and the message checksum is cleared. The boat is then sent to the "next response boat address" to complete the cycle. A natural extension of this simple boat scheme is to provide for multiple active boats with responses returning in the first available boat. Note that there is an independent set of boats and docks for each channel.

Send message is a subroutine which has the following parameters:

16 bits	4	4	8	16 bits	16 bits
Immediate Return	SB	HL-2	L	Control Package Address	Return After Response
Message Address	To Zipcode			From Zipcode	Function Code
Message Parameters					

The message plus header is moved into a boat and when the boat is full or at regular intervals the boat is sent to its destination. In "receive message", the response is moved to the original message area and the "return after response" address is entered.

Sample Configuration:



Sample Message Boat:

0000	0003	0A60	1E07	}	Read Page Message
0B38	1301	0100	0200		
0070	0008	0005	8000		
0000	0003	0B20	1000		
0750	1301	0100	0202	}	Rewrite Page Message
0037	0120	0090	8000		
0	0	0	0		
0	0	0	0		
0	0	0	0	}	Rudder
0003	1400	0003	1800		
0000	0002	0000	0754		

Steps in Processing Message Boat:

1. The above message boat has been generated by the STAR monitor and resides at the "next message boat address" in STAR memory.
2. The message control routine in the paging station has been reading this boat area to the SCU memory; on finding that the message checksum is nonzero and correct, it proceeds to activate overlays to process the messages in the boat.
3. On finishing processing of a message, the overlay enters the "send response" subroutine which tells message control that this message is finished.
4. On finding that all messages in the boat have been responded to, message control sends the boat back to STAR at the "next response boat address," after setting the response checksum and clearing the message checksum.
5. On finding the response checksum nonzero, the STAR monitor proceeds to take action on the responded messages.

SYSTEM MESSAGES

D

System messages can be grouped into the two major categories of station messages and user messages. Station messages are messages sent either between one station and another or between a station and the central processor. User messages are internal to the central processor and cause monitor interaction with user programs and system task programs. Figure D-1 shows the two distinct types of messages.

The formats used for station messages and user messages differ only enough to accommodate the variations in information requirements. Each message consists of a 16-byte header followed by the message content.

In the descriptions that follow, a page is defined as a logical unit of data, and a block is a physical unit of storage. Pages, blocks, and records are of equal length; 512 64-bit words, 4096 8-bit characters, or 32,768 bits.

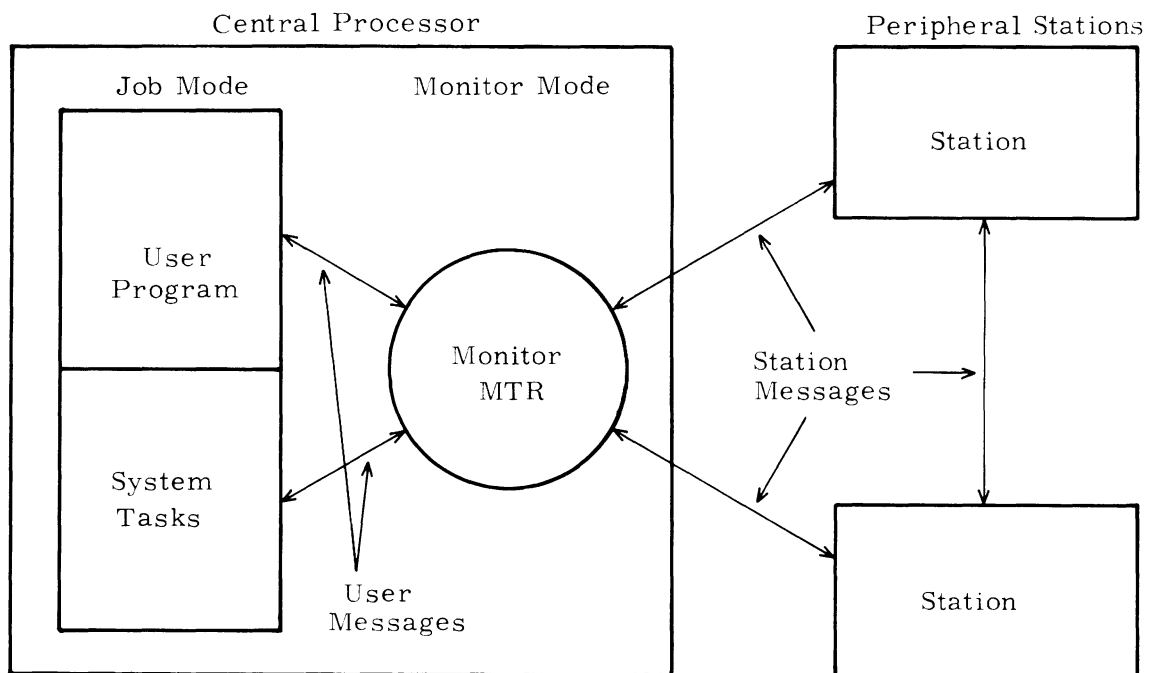


Figure D-1. System Messages

STATION MESSAGES

Station messages may be classified into the following general classes:

- Paging Messages
- Storage Messages
- File Messages
- Service Station Messages
- Communication Messages

All messages consist of a header followed by a message body. See Appendix C for message header format.

PAGING MESSAGES

<u>Function Code</u>	<u>Function Name</u>	<u>Parameters†</u>	<u>Format</u>
200	Read page	B, K, <u>U</u> , P	2A
201	Write page	B, K, U, P	2A
202	Rewrite page	B, K, U, P	2A
203	Delete N pages	<u>N</u> , K, P	2A
204	Delete key (N = no. of pages deleted)	<u>N</u> , K	2A
205	Read most-active block with given key, then delete. (Page name and usage bits returned)	B, K, <u>U</u> , <u>P</u>	2A
206	Read least-active block with given key, then delete. (Page name and usage bits returned.)	B, K, <u>U</u> , <u>P</u>	2A
207	Read and delete page	B, K, <u>U</u> , P	2A
208	Read drum page table	B, N, <u>S</u> , <u>E</u>	2F

† Parameters underlined are returned with the response.

STORAGE MESSAGES

<u>Function Code</u>	<u>Function Name</u>	<u>Parameters</u>	<u>Format</u>
220	Read N blocks from storage unit	B, N, UN, SBN	2E
221	Write N blocks to storage unit	B, N, UN, SBN	2E

UN = Unit Number
SBN = Starting block number

FILE MESSAGES

<u>Function Code</u>	<u>Function Name</u>	<u>Parameters</u>	<u>Format</u>
240	Create and open file	<u>F</u> , M, Mo, Mp, characteristics, name and user ID	2B
241	Open file	<u>F</u> , M, characteristics, name and user ID	2B
242	Close file	F, characteristics	2B
243	Close and delete file (temporary and permanent)	F	2C
244	Close and delete temporary file	F	2C
245	Keep file	F	2C
246	Set file characteristics	F, characteristics	2B
248	Is file open	F, characteristics, name and user ID	2B
24A	Read file pages	F, N, B, S	2C
24B	Write file pages	F, N, B, S	2C
250	Read file descriptor	F, B	2C
253	Modify owner and public access	F, Mo, Mp	2D

SERVICE STATION MESSAGES

In addition to processing the messages below, the Service Station acts as a message switching center for passing on other messages, some directly, others by interception, in order to handle the buffering and transmission of data. Also, all Storage Station messages are processed by the Service Station.

<u>Function Code</u>	<u>Function Name</u>	<u>Parameters</u>	<u>Format</u>
300	Rent SBU core	N, <u>B</u>	3A
301	Release SBU core	N, B	3A
302	Set file disposition	D, N, PR, T, name	3B
303	Request next file of given disposition	D, <u>name</u>	3B

COMMUNICATION MESSAGES

The communication messages define the linkages between STAR interactive programs and their users.

Central processor to station function codes.

<u>Function Code</u>	<u>Function Name</u>	<u>Parameters</u>	<u>Format</u>
400	Request type-in	None	None
406	Display Data	L, CA, PA	4C
410	Log-off	None	None
420	Central processor unavailable	None	None
421	Central processor available	None	None

Station to central processor function codes:

<u>Function Code</u>	<u>Function Name</u>	<u>Parameters</u>	<u>Format</u>
100	Log-on	None	None
101	Type-in data [†] /Buffer input	None/L, CA, PA [§]	4A/4C
102	Break	None	None
103	Read virtual data [‡]	VI, SN, RC, TL, SL, PA	4D
104	Release virtual segment [‡]	VI, SN	4E
106	Send data to display station	None	None

[†] Typein limited to 80 character buffer (plus #1F)

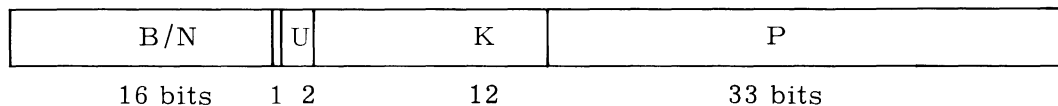
[‡] Not implemented initially.

[§] Used when typein is greater than 80 characters (up to full screen, 1152 characters).

MESSAGE FORMATS

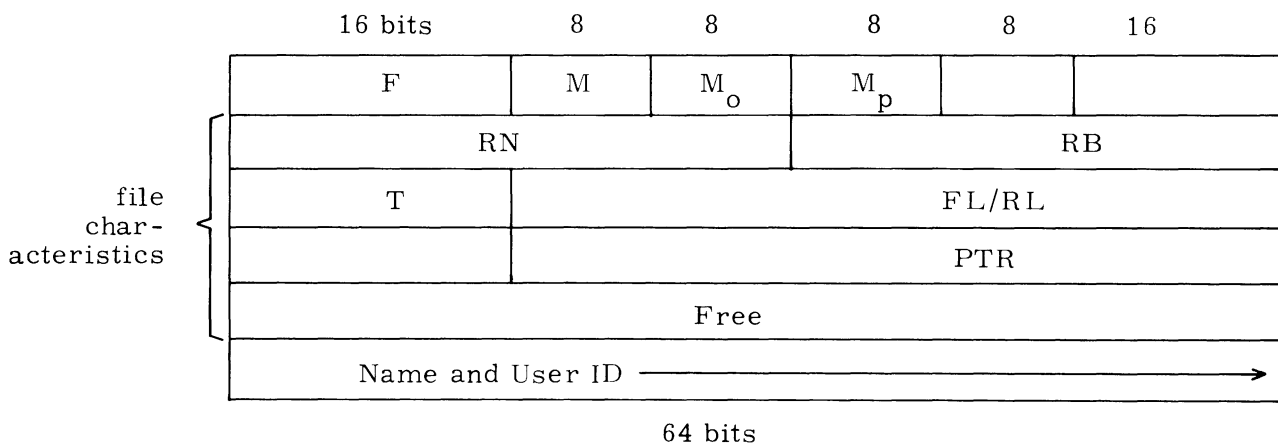
The following are the various system message formats.

FORMAT 2A.



- B/N Block address or number of pages
- U usage bits: These are stored in the drum page table on write and rewrite and returned in this position on read.
- U (bit 1) 0/1, unmodified/modified since initial access.
- K Key
- P Virtual page address

FORMAT 2B



- F active file index (given by storage station)
- M access mode for this run

{	bit 0 set	cannot delete
}	bit 1 set	cannot alter access modes
{	bit 2 set	cannot write
}	bit 3 set	cannot read

M_o, M_p access modes of owner and public respectively (used on creation)
 RN number of records in file
 RB number of reserved blocks for file
 T file type
 bit 0 set undefined
 bit 1 set coded delimited Initially only coded
 bit 2 set coded fixed delimited, coded fixed,
 bit 3 set binary STAR and binary STAR file
 bit 4 set binary fixed types are supported by
 bit 5 set foreign delimited the unit record station.
 bit 6 set foreign fixed
 bit 7 set virtual memory file
 bit 8 set drop file
 bit 9 set labeled file
 bit A set multiple volume
 bit B set incomplete
 bit C set permanent
 bit D set input
 bit E set output
 bit F set full

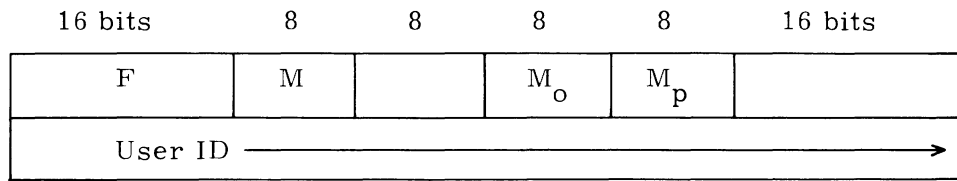
 FL/RL file length in bits/record length in bits, if fixed
 PTR pointer to structure definition within file (bit address)
 Name and User ID The file name has two fields — local name and owner identifier, separated by the space character (hexadecimal 20). Catenated to this by another space character is the user identifier which ends with the ASCII record separator character (hexadecimal 1E). If the owner and the user are the same person, then the user field can be omitted. (See file name section for definition of local name and owner identifier.)

FORMAT 2C

16	16	16	16
F			N
B		S	

F active file index
 N number of pages to be transferred
 B core block number; if bit 0 set, B=SBU address
 S starting file page number (starts with zero)

FORMAT 2D



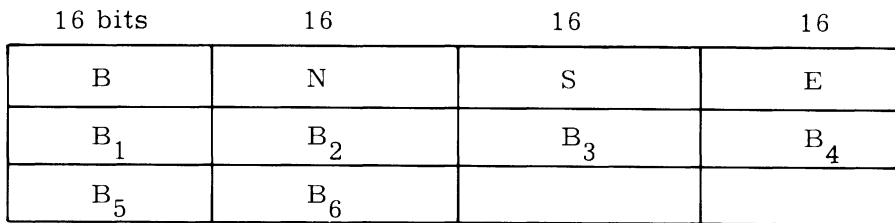
- F active file index
- M access mode
- M_o, M_p owner and public access modes
- User ID user access identifier, variable length string of characters which ends with the record separator character

FORMAT 2E



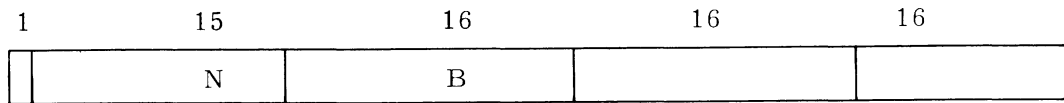
- B core block number
- N number of blocks to be transferred
- U unit number
- S starting block number

FORMAT 2F



- B starting block of contiguous set
If B=0 blocks not contiguous, then use B₁ through B₆
- N number of blocks to be read
- S word index (64 bit word) to first active entry
- E index (64 bit word) to last +1 active entry

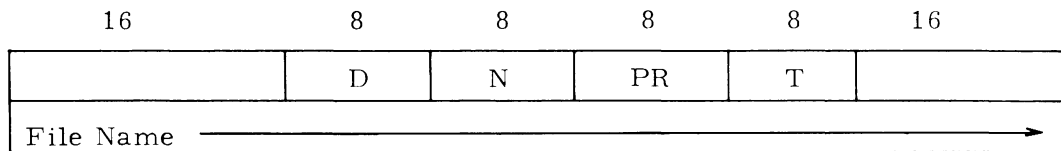
FORMAT 3A



Top bit set means quarter page

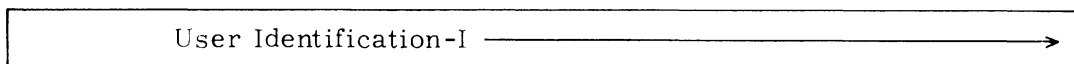
- N number of blocks (N=1 for the SBU case)
- B block address. In the SBU this is the core address; in central this is the block number.

FORMAT 3B



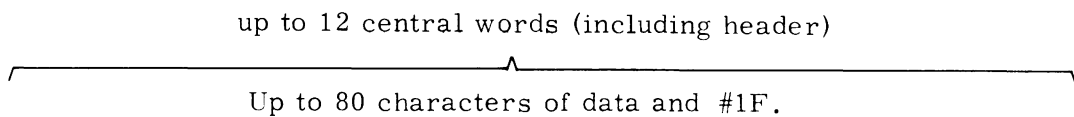
- D disposition code (01=print, 02=card punch, 03=batch job)
- N number of copies
- PR priority
- T card type (0=coded; 1=binary STAR)

FORMAT 3C

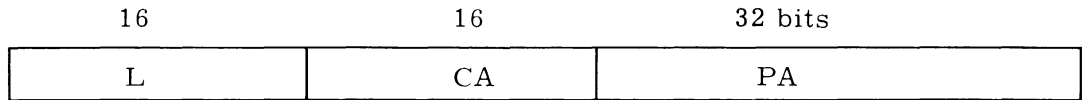


User identification consists of a variable length string of characters in two parts – user number and user password – separated by a record separator character and ending with a file separator character.

FORMAT 4A

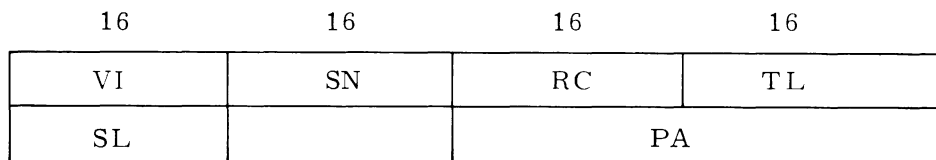


FORMAT 4C



L length (in 64-bit words)
 CA cursor address
 PA relative offset in 16-bit words from beginning of
 message boat. If FC = 101, then PA is the bit
 address of data in central memory

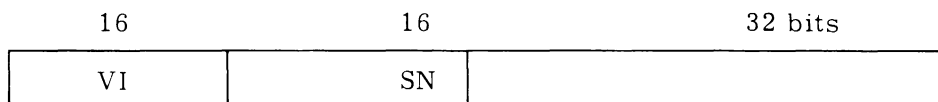
FORMAT 4D



VI virtual identifier
 SN segment number
 RC rate/class
 TL total length
 SL segment length
 PA physical address

TL and SL are in full word units

FORMAT 4E



VI, SN as defined for format 4D.

NOTES

1. The listed paging messages apply to normal pages (512 64-bit words). A similar set, the 120X (X=0-F) functions, apply to large pages (128 times small block size). As indicated, the function codes are the same as above but they apply to large pages; for example, FC-1200 means Read Large Page. The message formats are the same, but the lower 7 bits of both the block address and page name must be zero. Large page messages will apply initially only to the disk stations.
2. Not all parameters or their fields need exist when a message uses a standard format. For example, with the close file command, only F need exist, and only the first 64 bits of the message need be sent. The message must consist of an integral number of 64-bit words.
3. Initially on CREATE, if the file name already exists, and not opened for use, the existing file is deleted if it is temporary and the new file created; otherwise, the response indicates that a permanent file already exists by that name, or the file is in use, and no create occurs. At a later stage it may be appropriate to allow different editions/generations/versions of the same file.
4. On read file or read file page, the existing file or page is read to core and the copy left on the storage device. On write file or write file page, the existing file or page is simply overwritten in the initial implementation.
5. There are two types of close – short and long. In the short form the only parameter is F, and the entry is simply removed from the active file table. In the long form, the message includes the file characteristics which may have been changed since open to be written back to the descriptor; and if the file length was shortened, the excess space is released and the descriptor updated.
6. The zipcode destination of a message is also the destination of the data associated with the message.
7. Initially the same file can be opened by more than one user, provided the access modes are for read only.

SAMPLE MESSAGES

Read Page 47, key 8 into block 90

FC = 200, message =

0090	0010	0000	0047
------	------	------	------

Read file page number 1015 to core block 57 from file 123

FC = 24A, message =

0123	0000	0000	0001
0057	0000	0000	1015

EXAMPLE OF MESSAGE USE

The steps in card reader input messages from an input/output station to a service station are:

1. After reading first card, open and create input file (FC=240); this checks for duplicate file name.
2. Read SBU block (function code FC=300)
3. On filling SBU block, write file page (FC=24B)
4. On last card close file (FC=242) which releases unused file space.
5. Release SBU block (FC=301)

Step 3 is repeated as many times as necessary. During step 4 the service station transfers the file to its ultimate destination. If the file is very long, the service station spools the file to its ultimate destination in smaller subfiles.

The steps in file output are:

1. Request next output file (FC=303)
2. Open file (FC=241)
3. Rent SBU block (FC=300)
4. Read file page (FC=24A)
5. Delete and close temporary file (FC=244)
6. Release SBU block (FC=301)

Step 4 is repeated as many times as necessary. During step 1 the service station transfers the file from where it is stored to the service station; if it is a long file, it will be spooled over in smaller subfiles. On close output file the file is deleted from the service station and also from its storage station if it is only a temporary file.

The following messages clarify the log-on and type-in procedure:

<u>Step</u>	<u>Message to STAR-1B</u>	<u>Message from STAR-1B</u>
1		421 central processor available
2	response	
3	100 log-on	
4		response
5	106 ready for display	
6		response
7		406 here is display
8	response	

Type-In Request

9		400 type-in
10	response	
11	101 send data	
12		response

Cycle to Step 5.

USER MESSAGES

The File input/output system messages associated with Monitor VI allow users to obtain disk space as files, attach files to programs, read and write files implicitly or explicitly, save files, and delete files.

All functions performing an OPEN file require a NAME in ASCII, left justified with owner ID if needed. The initial File ID format is: file name followed by a space, owner ID followed by a space, and the ASCII 1E terminator. The size limit for this combination is 24 characters. It should also be noted that no user ID processing is available for the initial File ID format. With the response code, an active file index (F) is returned and any further reference to that file while open requires both the activity file index and the file name. The system records this index and performs implicit input/output using this identifier for the file name.

MESSAGE HEADER

All user messages are preceded by a two word header with the following format:

R	L	C	FC
Error Exit			

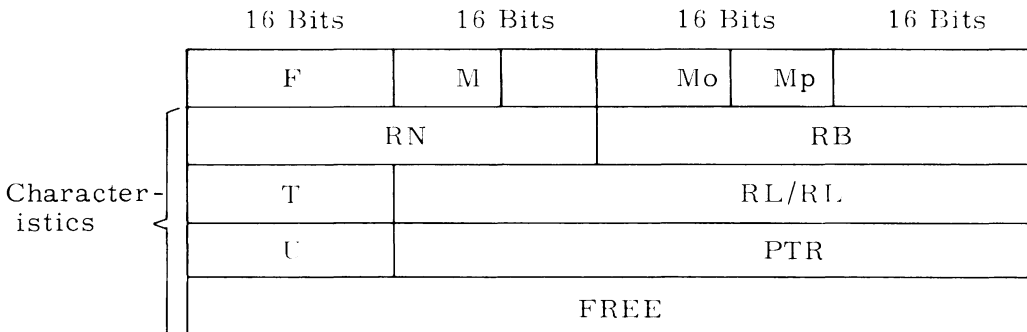
- R Response Code (see paragraph entitled Response Codes).
The error exit will be taken if R≠1 and the ERROR EXIT
field ≠0.
- L Length of message in 64-bit words (exclusive of header)
- C Control code, unused except where noted.
- FC Function code
- Error Exit Exit address in event of an error response

File Input/Output System Messages

<u>FC</u>	<u>Function Name</u>	<u>Parameters</u>	<u>Format</u>
150	Buffer Input	VA	B
154	Buffer Output	VA	B
302	Set File Disposition	D, NC, T	D
616	Map-In	C, F, VA, N	B
617	Map-Out	C, F, VA, N	B
640	Create & Open File	Char, N, M	A
641	Open File	N, M	A
642	Close File	F, N	B
644	Close & Delete File	F, N	B
645	Keep File	F, N	B
646	Set File Characteristics	Char, F, M, N	A
648	Is File Open	N	A
64A	Read File Pages	F, VA, S, N	C
64B	Write File Pages	F, BK, VA S, N	C
684	Release	C, BK, VA	B

CHARACTERISTICS AND F, M, M_o, M_p

Imbedded in each Format A message is a 4-word list of parameters called the characteristics. The fields of the characteristics are defined as follows:



- F active file index
- M access mode
- Mo access mode of owner (used on creation)

Mp	access mode of public (used on creation) bit 0 = cannot delete bit 1 = cannot alter access mode bit 2 = cannot write bit 3 = cannot read
RN	number of records in file
RB	number of reserved blocks for file
T	file type bit 0 = undefined bit 1 = coded delimited bit 2 = coded fixed bit 3 = binary STAR
FL/RL	file/record length in bits
U	model 841 unit number 1-4 (zero means create on any unit). This field is used only for CREATE and OPEN file
PTR	pointer to structure definition within file

MESSAGE DESCRIPTIONS

System Call 150 - BUFFER INPUT - Format B

System Call 154 - BUFFER OUTPUT - Format B (not available initially)

These messages cause the indicated page to be moved to/from central memory while the calling program remains active. This is an ADVISE function for files which have been mapped in/out.

- Notes:
1. BUFFERING a page already in core is a null request.
 2. BUFFERING a page not defined by a previous MAP or reference will cause the page to be created.
 3. BUFFERING pages is done one at a time, if a buffering request is in progress, further requests are not saved. (Response indication busy.)

System Call 302 - SET FILE DISPOSITION - Format D

This message is passed on to the storage station and causes the indicated file to be entered into the appropriate queue.

System Call 616 - MAP-IN - Format B

The MAP-IN message establishes a correspondence between a virtual address region and a disk file which is already opened. The caller

must supply both file index (F) and name used when the file was opened. No data is moved; pages will be read in when referenced, thus providing implicit input READ.

NOTE: C = control code in header - 0 Map-in starting address
- 1 Map-in ending address

System Call 617 - MAP-OUT - Format B (not available initially)

The MAP-OUT message provides implicit output by establishing a correspondence of virtual address region and a file which is already opened. The caller must supply both file index (F) and name used when the file was opened. No data is moved; when virtual space is RELEASED, the system stores pages into the file, thus providing implicit output WRITE. See System Call RELEASE.

NOTE: C = control code in header - 0 Map-out starting address
- 1 Map-out ending address

System Call 640 - CREATE and OPEN File - Format A

The CREATE and OPEN message reserves disk space under a symbolic name, sets the characteristics, and opens that file by making an entry in the Active File Table at the disk station. The file index (F) is returned with response.

System Call 641 - OPEN FILE - Format A

The OPEN FILE message makes an entry in the disk station Active File Table and returns the characteristics and the active file index (F), if the file already exists and access is valid.

System Call 642 - CLOSE FILE - Format B

The CLOSE FILE message removes an entry from the Active File Table for an open disk file and removes the correspondence with virtual memory by removing its MAP-IN and MAP-OUT entries from the system.

System Call 644 - CLOSE and DELETE FILE - Format B

The CLOSE and DELETE FILE message removes an active index and releases the disk space for re-assignment. Correspondence with virtual memory is removed by deleting MAP-IN and MAP-OUT entries from the system.

- NOTES:
1. The file index (F) and the file name (N) used at OPEN must both be supplied by the calling program.
 2. This call will cause a 243 message to be sent to the Service Station which will delete the file whether temporary or permanent.

System Call 645 - KEEP FILE - Format B

This message is passed on to the storage station and causes the indicated file to be kept (i.e. made permanent).

System Call 646 - SET FILE CHARACTERISTICS - Format A

The SET FILE CHARACTERISTICS message provides a means for a program to change the description of a file which exists and is open.

System Call 648 - IS FILE OPEN - Format A

This message verifies the indicated file name is open and returns the F number and characteristics if it is open.

System Call 64A - READ FILE PAGES - Format C

System Call 64B - WRITE FILE PAGES - Format C

These messages move a page from/to a disk file which is already open. The caller must supply both file index (F) and name used when the file was opened. They provide explicit record movement.

System Call 684 - RELEASE - Format B (not available initially)

The RELEASE message allows a program to output data to a disk file and/or delete a portion of virtual memory. Output files must be open and mapped out.

- NOTES:
1. Pages are stored only in files MAPPED OUT.
 2. If virtual memory is removed, the MAP-IN table is not changed so succeeding references to mapped-in files will bring that page to core from the disk.
 3. C = control code in header = XY
 - X = 0 Virtual memory remains.
 - 1 Virtual memory is deleted.
 - Y = 0 no - pages stored in files
 - 1 modified - pages stored in files
 - 2 all - pages stored in files

FORMAT A

Header	R	L	C	FC						
	ERROR EXIT									
Characteristics	M (1)									
	F	M		Mo	Mp					
	RN				RB					M (3)
	T	FL/RL								M (4)
	U	PTR								M (5)
	FREE									M (6)
	FILE ID									M (7)
Name (N)	M (8)									
	M (9)									

R, L, C, FC and
ERROR EXIT

As previously defined under message header

F

Active file index

M

Access mode

Mo, Mp

Access modes of owner and public, respectively

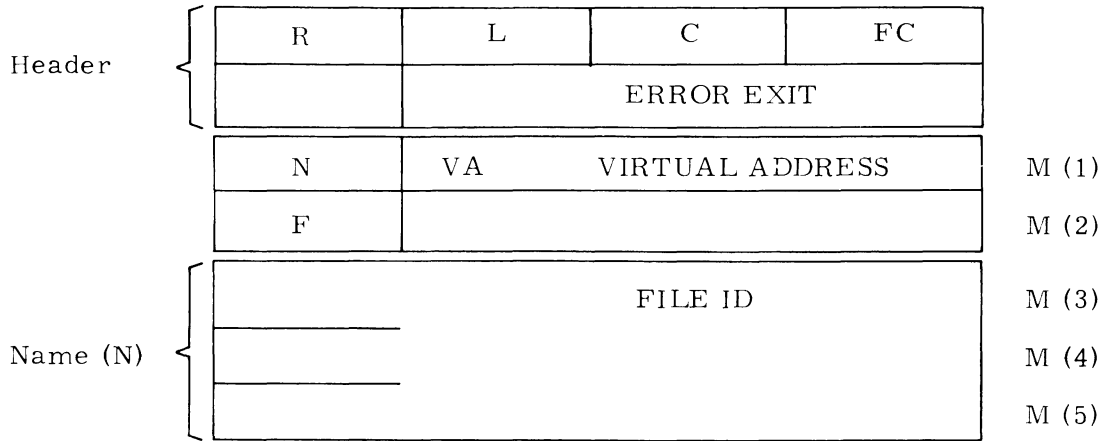
RN, RB, T, FL/RL,
U, PTR

As previously defined under characteristics

FILE ID

File name followed by a space, owner ID followed by a space followed by the ASCII record separator, #1E.

FORMAT B



R, L, B, FC and
ERROR EXIT

As previously defined under Message Header

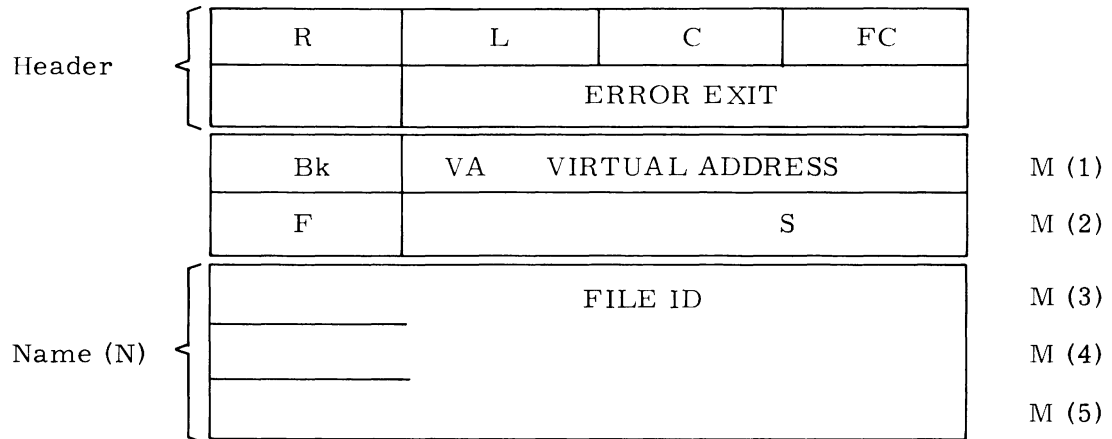
N Number of blocks

VA Virtual address

F Active file index

FILE ID File name followed by a space, owner ID followed by a space followed by the ASCII record separator, #1E.

FORMAT C



R, L, C, FC
and ERROR EXIT As previously defined under Message Header

Bk Number of pages to be moved (must be 1 initially for WRITE FILE PAGES)

VA Virtual address

F Active file index

S File page number (starts with zero)

FILE ID File name followed by a space, owner ID followed by a space followed by the ASCII record separator, #1E.

RESPONSE CODES

The following are the response codes issued by the system.

8001	message completed successfully
8002	illegal message (function code or parameter invalid)
8003	file/virtual page not found
8004	error on device
8005	device full
8006	message checksum error
8007	illegal access to file
8008	no shared access
8009	station saturated, try again
800A	end of file
800B	user identification invalid
800C	invalid password
800D	file already exists
800E	requested already
800F	file page out of range
8010	file not open

If bit 1 in the response code is set, it means there are parameters to be returned and the whole message area is returned.

MICRODRUM

The microdrum provides memory extension for the buffer controller. Its characteristics are:

- One head per track
- 3600 rpm (16.7 milliseconds/revolution)
- 36 data tracks plus four control tracks (addresses 0-B, 10-1B, and 20-2B)
- Capacity

Data Mode	18,432 bits per track, alterable in sectors of 64 16-bit words, 18 sectors per track
Display Mode	9,216 bits per track, alterable in multiples of 16-bit words up to 32 words (1 sector).

The microdrum is used in the stations for storage of performance statistics, program overlays, and display images; it also provides an autoload medium.

A specific overlay called the microdrum loader processes the binary program code and headers produced by the assembler, and stores them on the microdrum. All overlays start at a quarter-sector boundary and do not cross track boundaries.

GENERAL LAYOUT

The drum layout is as follows:

<u>Track</u>	<u>Use</u>
00	nucleus 0
01	nucleus 1 (backup)
02	overlay tables for nucleus 0
03-0B	overlays for nucleus 0
10, Sector 0	microdrum map
10, Sectors 1-8 ₁₀	nucleus 0 system parameters
10, Sectors 9-16 ₁₀	nucleus 1 system parameters
11	reserved

<u>Track</u>	<u>Use</u>
12	overlay tables for nucleus 1
13-1B	overlays for nucleus 1
20†	physical microdrum loader
21†	free for temporary workspace
22-25†	diagnostics
26-27†	message tracks
28†	function track and AID
29†	error log
2A†	status display
2B†	memory display

OVERLAY TABLES

There is an overlay table associated with each word of the 8 scanner level words. An entry in the overlay table consists of two 16-bit words. Each entry in the table corresponds to the entry in the same position on the scanner jump table.

The format of an overlay table entry is:

Word 1	microdrum address of overlay (bits 0-7 = head address, bits 8-F = quarter sector address)
Word 2	bits 0-A = length in 16-bit words; bits B-F = residence condition (see overlay header section)

For example, suppose 0304 1401 is the second entry in the third level of the overlay table. This means that program number 32 resides on head 3 sector 1 of the microdrum, its length is A0 words and it has residence condition 1.

The layout of the overlay tables for nucleus 0 (8 scanner levels) is:

head and sector 0200	overlay tables 0-7 for system 1
head and sector 0204	----- 2
head and sector 0208	----- 3
head and sector 020C	----- 4

SYSTEM PARAMETERS

The system parameters are summarized in the following list.

† Unprotected tracks

<u>Head</u>	<u>Sector</u>	<u>Word Address</u>	<u>Description</u>
10	01	00, 02, 04, 06	microdrum address of low core overlays
		01, 03, 05, 07	length of low core overlays (bits 0-A)
		08, 0A, 0C, 0E	microdrum address of high core overlays
		09, 0B, 0D, 0F	length of high core overlays (bits 0-A)
		10-1F	list of scan bit assignments of permanent overlays (eight bits per overlay)
		20-3F	data block map (head/quarter sector address)
10	02	00-3F	system 2
10	03	00-3F	system 3
10	04	00-3F	system 4

MICRODRUM MAP

The microdrum map describes the use of each microdrum track. It is normally present on track 10₁₆, sector 0. Each word in the sector defines the status (type of information on the track), the nucleus to which it belongs, and the next available empty sector on the track. Each word in the map is associated with a corresponding drum track or a null track. The 36 tracks are structured in three groups of 12, and numbered 00-0B, 10-1B and 20-2B. The null tracks (0C-0F, 1C-1F, 2C-2F, 30-3F) are represented in the map by a null code, 0FFF. These codes in the map represent track addresses having no corresponding physical tracks.

The bit contents of each map entry consists of:

- 0-3 = nucleus to which overlay belongs
- 4-7 = track status
- 8-F = next available quarter sector number

The track status bits 4 through 7 are defined as:

- 0 = unused
- 1 = nucleus
- 2 = overlay
- 3 = display file message track

4 = overlay tables
 5-D = unassigned
 E = microdrum map - system parameters
 F = unavailable

When tracks 03-0B become full, nucleus 0 overflows overflow to the reserved track and to any free tracks within the nucleus 1 overlay area. If necessary, either nucleus can overflow to any of the unprotected tracks (20-2B) by setting the map entry to zero. Normally it is D00.

OVERLAY HEADERS

Each overlay is preceded by a header containing the following information:

word 0	= FFFF	header identifier [†]
1	= 00LL	header length [†] (length must not exceed 256 decimal)
2	= RRSB	RR = residence assignment [‡] SB = scanner bit assignment
3	= SSSS	system(s) assignment(s)
4 through N-1		
N	= XXXX	assembled program origin address [§]
N+1	= ZZZZ	program length ZZZZ = number of program words to the next PROC statement

The program header details can be summarized as:

word 0 is tested by the microdrum loader to identify a start of header.
 word 1 defines the length of header as word N minus word 2.

[†] The header identifier and the header length (words 0 and 1) are automatically generated using the Buffalo PROC statement.

[‡] Multiple assignments can be made when necessary. This is achieved by means of a flag in the residence assignment and a repeat of words 2 and 3 with the additional assignments (see header details).

[§] Word N and N+1 are automatically generated using the Buffalo ORG statement.

word 2 the leftmost byte of this word is used for the residence assignment of an overlay. The assignments are as follows:

RR = 00	priority 0, temporary overlay
= 01	priority 1, temporary overlay
= 02	permanent overlay
= 04	priority 0, temporary overlay - source is SBU
= 05	priority 1, temporary overlay - source is SBU
= 08-0F	fixed-area overlay
= 40-43	direct core overlays
= 44-47	high core overlays
= 6X	data block overlay

The foregoing RR codes, modified by setting the highest order bit, are also valid. When this bit is set, the microdrum loader looks for additional scanner assignments in words 4 and beyond.

The rightmost byte of word 2 defines the scanner level (priority) assigned to the overlay.

SB = scan level bit assignment for the routine.

word 3 defines the system or systems to which the overlay is assigned. The systems are defined by bit number in the word as follows:

SSSS = 0000	NUCLEUS
= 8000	(bit 0 set) LOADER system
= 4000	(bit 1 set) RUN system
= 2000	(bit 2 set) DIAGNOSTIC system
= 1000	(bit 3 set) OFF-Line system
= 0X00	(bits 4-7) experimental bits 8-F unavailable
= FFFF	display messages. Start on head HH, sector SS as defined in word 2.

word 4 second residence and scan level assignment, if multiple scanner bit assignments.

word 5 second system assignment(s), if multiple scanner bit assignments.

word 6 other uses as needed up to 256 decimal words.
through N

word N origin address of assembly
word N+1 program length – this word not only gives the length of the overlay, but length +1 defines the location of the next header identification tag.

LOADING THE MICRODRUM

STATION NUCLEUS 0

1. Address #FB = Data input medium code
 Input Code 0 = Paper Tape Input
 1 = SBU Memory Input
 2 = CPU Memory Input
2. Address #FC = Load function
 Function 0 = Initialize micro drum
 1 = Add new nucleus
 2 = Add overlays to nucleus
3. Address #FD = Nucleus number (0 or 1)
 NOTE: This number designates the nucleus for add overlays or the new nucleus to be created.
4. Address #FE = Number of overlay table levels required by the nucleus.
5. Address #FA = SBU or central starting address for loader input data.

The following call function has been added to the nucleus and is used by the micro drum loader as follows:

CALL PN/RA, P1, P2, P3, P4, P5

PN = Program number for the Micro Drum Loader
RA = Relative address - #00 for the loader
P1 = Data input medium
P2 = Load function
P3 = Nucleus number
P4 = Number of overlay levels
P5 = SBU or CPU address (32-bit word address)

NOTE: If the values of P1 through P5 are correct in core, only PN/RA need to be entered in the CALL command. Write lockout must be turned off before executing the CALL command.

STATION DEAD START LOADER

1. Address #E = Load function
2. Address #F = Nucleus number
3. Address #10 = Number of overlay levels

NOTE: Turn write lockout off after setting the above parameters. Depress function key F2 to start execution of the loader. Be sure the stop switch on the Maintenance Panel is on. A stop at an address other than address #900 is an error. See listing of Dead Start Loader for type of error.

If loading from paper tape, the following steps must be performed from the CE maintenance console:

1. Note starting address and place paper tape in reader. The selective stop switch should be on.
2. Press tape AUTOLOAD. Reader should begin reading and then stop at starting address plus 400. If the reader fails to stop, an error occurred either on tape or in the reader.
3. Press READER HALT, MASTER CLEAR, CLR-CHL and GO. After writing each routine to the microdrum, the Buffer Controller halts at the starting address.
4. Repeat steps 2 and 3 until the whole tape is read in after which the buffer controller goes into its idle loop.

In the event both Nucleus 0 and Nucleus 1 are destroyed, a bootstrap routine must be loaded from paper tape as follows:

1. Place Bootstrap Load paper tape in reader.
2. Push Autoload switch on C.E. maintenance console.
3. After paper tape has been read, master clear.
4. Set selective stop switch on C.E. maintenance panel.
5. Depress GO switch.
6. Turn off write lockout switch on SCU control panel.
7. Depress F2 key. This initializes map and parameter tables on track 10 and stops at address 900.

8. Put NUCLEUS tape in paper tape reader.
9. Depress Autoload switch. Stops at D00.
10. Master clear and GO. Stops at 900 (if no error).
11. Repeat Steps 9-11 until all tape has been read.
12. Turn on write lockout switch on SCU control panel.

STATION MAINTENANCE INFORMATION SYSTEM AND AID

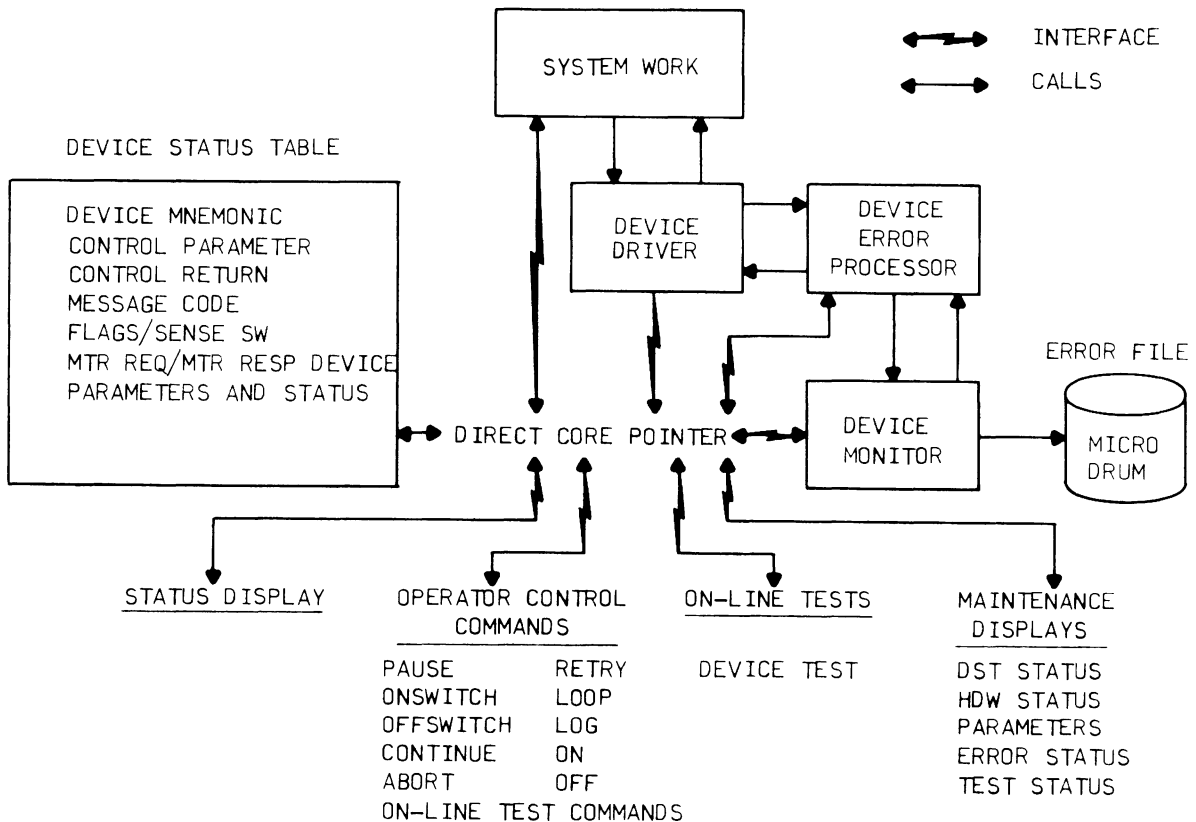
F

STATION MAINTENANCE INFORMATION SYSTEM

The Maintenance Information System (MIS) provides the Customer Engineers with a standard interface to the STAR 100 station device drivers. MIS features include:

1. Information displays of device, driver and user status.
2. Error logging.
3. Device driver breakpoint.
4. Error recovery control.

These features are facilitated through the interface described in this document.



DEVICE STATUS TABLE (DST)

Each system device has a DST. The DST contains pertinent information related to the specific device. Each DST has a standard 6-word header with a cell in direct core pointing to the header.

DST POINTER

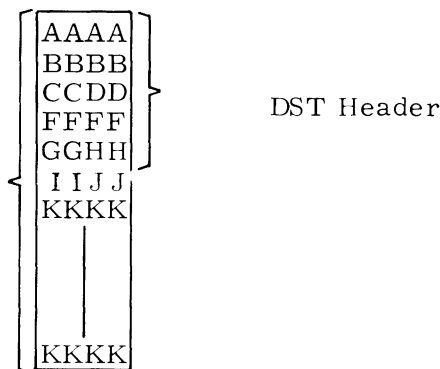
There are 2 labels defined in the nucleus between which all DST pointers must be located. These labels are:

NUC_DST_DIRECTORY

NUC_DST_DIRECTORY_LIMIT

The address of the DST would normally be plugged into the pointer via an initialization package.

DST FORMAT



Suggested Initial Header Setup

```
Device Mnemonic
Pointer to DST
Error Processor PN/RA
# 80
# C000
0
```

<u>Symbol</u>	<u>Contains</u>	<u>Description</u>
AAAA	Device Mnemonic	2 ASCII characters (upper bit is DST break point flag).
BBBB	Control Parameter	The contents of this call is loaded in B1 on return from the device monitor. Normally the DST address is placed here.
CC	Program Number	This is the program number which is called on return from the device monitor. Normally the error processor program number.
DD	Relative Address	This is the relative address in program number CC to be recalled by device monitor.
FFFF	Message Code	This code is set by the device error processor and defines the address of a 32-character message or is an index code to a message in the message file.
GG	Flags	Communication flags.
	<u>Bit</u>	<u>Flag Name</u>
	0	Device On
	1	Log Errors
	2	Loop
	3	Diagnostic Mode
	4	Step Mode Pause
		If reset, no more system requests are to be made to this device.
		If reset, the error log is turned off for this device.
		This flag must be monitored by the device error processor on return from a MTR request. If the flag is set, the last I/O request should be retried until the step mode pause flag sets. Error recovery algorithms should be suppressed.
		This flags the device driver that it has been called by a diagnostic. It may or may not have special meaning to the device driver.
		If set, the device driver should pause via device monitor after executing the current I/O request. This is the fundamental device driver break-point. The flag must be monitored by the device driver to be effective.

<u>Symbol</u>	<u>Contains</u>	<u>Description</u>
GG	Flags (cont'd)	
	<u>Bit</u> <u>Flag Name</u>	
	5 Pause on Error	This flags the device monitor to breakpoint the device driver on all errors.
	6 Pause on Error Recovered	This flags the device monitor to breakpoint the device driver after recovery from errors.
	7 Pause on Fatal Error	This flags the device monitor to breakpoint the device driver before aborting on fatal error.
HH	Sense Switches	Sense Switches 1-8 to facilitate interactive control by the operator. Definition is optional.
II	Monitor Request Code	This byte is set by the device error processor to the appropriate code. Codes are defined in Request Codes on the following page.
JJ	Monitor Response Code	This byte is set by the device monitor in response to monitor request byte II. Codes are defined in Response Codes on the following page.
KKKK	Optional Device Parameters and Status	

DEVICE MONITOR

When a monitor request is made, action is taken according to the monitor request code and a response code returned to the requesting device error processor. The device error processor requests the device monitor whenever:

1. An operator action is required; for example, printer out of paper.
2. The device "step mode pause" flag is detected in the DST.
3. An error has occurred.
4. An error recovery has occurred.

Normally, the device monitor logs the error and returns control to the error processor with a response code of 0. If, however, an operator action is required

or an operator breakpoint has been set, the alert light and buzzer are activated, an operator message is displayed, and the response is suspended until action is taken by the operator.

DEVICE MONITOR REQUEST

The following steps are necessary to request the device monitor:

1. Set control parameter in word 2 of DST.
2. Set program number and relative address in word 3 of DST.
3. Set monitor request code in left byte of word 6 of DST.
4. Call device monitor by setting B1 = DST address and jump indirect to `DEVICE_MONITOR_CALL`.

REQUEST CODES

- 1 - Pause for operator action
- 2 - Pause because "step mode pause" flag set in DST
- 3 - Pause after error recovery
- 4 - Pause before error recovery
- 5 - Pause before abort on fatal error

NOTE: If the upper bit in the MTR request code is set, an entry will be made in the error log.

DEVICE MONITOR RESPONSE

The device monitor sets the response code and returns control to the error processor per word 3 of the DST. On entry to the error processor, B1 will be set to the contents of DST word 2 and the response code will be in the right byte of word 6 of the DST.

RESPONSE CODES

- 0 - Continue error processing algorithm normally
- 1 - Retry the current I/O request
- 2 - Abort current I/O request and send fatal error response to caller (Bit 0 set in CP status word.)

DEVICE PAUSE MNEMONICS

<u>Mnemonic</u>	<u>Meaning</u>
O	Paused for operator action
S	Step mode pause
E	Paused on device error
R	Paused on error recovered
F	Paused on fatal error

DST STATUS DISPLAY

The DST Status Display is selected by keyboard command STATUS. Each DST is represented by a 1 line display. The status line shows the DST index number, the device mnemonic, 5 words of the DST beginning with the flags word in the DST header, and the last message.

MIS KEYBOARD COMMANDS

All commands are preceded by an n. where n is the DST pointer ordinal for the specified device. The ordinal is shown on the STATUS display.

<u>Command</u>	<u>Description</u>
ON	Sets "device on" flag in DST.
OFF	Resets "device on" flag in DST.
LOG I/O	Sets/resets "log Errors" flag in DST.
PAUSE	Toggles "step mode pause" flag.
PAUSE X	Where X may be S, E, R, F, or CS, CE, CR, CF or any combination of the above. S = Set "step mode pause" flag CS = Clear "step mode pause" flag E = Set "pause on error" flag CE = Clear "pause on error" flag R = Set "Pause on error recovered" flag CR = Clear "Pause on error recovered" flag F = Set "pause on fatal error" flag CF = Clear "pause on fatal error" flag

<u>Command</u>	<u>Description</u>
ONSWITCH #	Sets "Sense Switch #" in DST
OFFSWITCH #	Resets "Sense Switch # in DST
CONTINUE†	Sets response code 0 in byte JJ of DST, recalls device error processor per CC/DD in DST word 2 and clears DST message.
RETRY†	Sets response code 1 in byte JJ of DST, recalls device error processor per CC/DD in DST word 2 and clears DST message.
ABORT†	Sets response code 2 in byte JJ of DST, recalls device error processor per CC/DD in DST word 2 and clears DST message.
LOOP	Toggle "loop" flag – if result is true, send retry response to device driver.
LOOP I/O	Set/reset "loop" flag – if set, send retry response to device driver.

ERROR LOGGING

An error log entry is made on Device Monitor calls when the upper bit in the request code is set and the "log errors" flag in the DST is set. Each error log entry contains the time of the entry and the first 15 words of the DST. A one sector buffer is kept in SCU core to log the errors on head 29 of the local microdrum. In the future, the local error logs will be passed up to a central file where a statistical analysis can be made. At present, the error log will be end-around and will contain the 71 most current entries.

† These commands are legal only after the device has paused. If the "loop" flag is set, the response code is forced to 1 (retry).

MESSAGE FILE

Microdrum heads 26 and 27 contain the message file. There may be up to 71 32-character messages per head, four messages per sector. Message codes 0-71 flag the appropriate message as shown below:

Head 26				
Sector 0	//////////	Message 1	Message 2	Message 3
1	4	5	6	7
2				
.				
.				
.				
17	68	69	70	71

NOTE: Message code #80 will clear any previous message in the DST status display. To prevent an initial garbage message, this code should be assembled in DST.

AID

PURPOSE

The purpose of AID is to provide customer engineers with the capability of generating short test programs to be used in debugging hardware failures. In addition, some cases permit the generation of simple tests to be used as diagnostics.

OPERATION

In order to activate the AID program, two commands must be typed in at the console. Typing ONAID (off line system, function key 4) will turn on the AID commands. Typing RESTART will cause the initial AID display to be displayed and initialize the AID program. The basic memory commands for the nucleus will remain; however, others will not. To return to the original conditions, type in OFFAID. Once the AID display is up, it is possible to alternate between buffer controller and SBU displays as usual. To return to the AID display, type in AID. Briefly stated:

ON AID	Initialize AID commands
RESTART	Initialize AID display and program
AID	Return to the AID display
OFF AID	Return the normal systems commands

The AID display consists of 16 lines labeled A, B, C, . . . , P. The * sign indicates where the next AID entry will be entered on the display. If at any time an illegal entry is made, an error message will appear on the screen indicating the error.

While the generated AID program is executing, the cursor will indicate which statement is being executed.

NOTE: After typing in ONAID, an attempt is made to RECOVER any previous AID program that may have been generated by the customer engineer. It is possible that the following conditions may exist:

1. The AID display has been destroyed; however, the generated program is still intact.
2. The AID display is present; however, part of the generated program has been destroyed.
3. Both the AID display and generated program are intact.

In any case, if AID cannot attempt to recover, RESTART will be activated automatically.

ORGANIZATION

The generated program is written on the drum beginning at track 28, sector B so that at any time the code may be examined by the following method:

Type-in: READ 280B F00 F40

The above will display at F00 the first 40 hex words of the generated program.

Once the generated program is initiated (by typing START), the program on the drum (track 28, sector B) is read into SCU memory and executed. The first word address (FWA) of each subroutine located in core is then displayed opposite the statement on that line.

If the program hangs after execution begins, MASTER CLEAR and GO will return to the original condition before execution began.

The last FWA displayed is the FWA of the write buffer. The read buffer immediately follows the write buffer (the longest length specified in any statement determines the length of the write buffer).

All the code for the generated program is contiguous in memory and can be examined after execution has terminated. Execution may be terminated by depressing the alert key.

NOTE: Only the first four characters of each statement need be typed.

Example: CONNECT 5000 = CONN 5000

NOTE: The first word address (FWA) displayed after typing START includes the jump back to the AID monitor. To breakpoint at the actual sub-routine code, set the breakpoint to the FWA + 2.

NOTE: Certain statements will cause status to be dynamically displayed on the console. Since this requires a call to the microdrum, the program may be slowed considerably. To eliminate the dynamic display, depress MODE key 4.

The following definitions apply in the descriptions of the AID statements:

- L = word length
- J_n = jump line number
- n = normal channel number
- a_n = address
- X_n = 16-bit data word
- N = line numbers
- A = Input or output operation

BASIC AID STATEMENTS

- | | |
|----------|--|
| JUMP J X | J = A, B, C, . . . , P.
X = Optional loop count
Jump causes program execution to begin at line number J. |
| STOP | Stop causes program execution to terminate and returns control to the system. |

OUTPUT n x

Output causes value x to be output on normal channel n.

INPUT n X₁ X₂ J₁ J₂

Input causes the value on normal channel n to be input into LAST STATUS and displayed.

X₁ = Mask value = X₁ .(LAST STATUS) → LAST STATUS

X₂ = Compare value - value of LAST STATUS is compared with X₂.

J₁ = Compare good jump

J₂ = Compare bad jump

X₁, X₂, J₁ and J₂ are optional parameters.

If X₂ is present, J₁ and J₂ are required.

MASK X J₁ J₂

X = Compare value

J₁ = Compare good jump

J₂ = Compare bad jump

The compare value X is compared with the contents of LAST STATUS.

Example 1: A INPUT 6
B MASK 0200 C A
*C
.
.
.

Normal channel 6 is input to LAST STATUS and compared with 0200.

Example 2: A INPUT 6 FF00
B MASK 0200 CA
*C
.
.
.

Normal channel 6 is input, and with FF00, and compared with 0200.

Example 3: A INPUT 6 FF00 0200 B A
*B
.
.
.

Normal channel 6 is input, and with FF00, and compared with 0200.

COMPARE L J₁ J₂

L = 1, 2, ..., n

J₁ = compare good jump line number

J₂ = compare bad jump line number

L words of the SCU write buffer will be compared with L words of the SCU read buffer.

If a compare error is detected, the word number (relative to the write buffer FWA) is displayed. Word numbers = 0, 1, 2, ..., n.

DATA x₁ x₂ x₃ x₄ x₅ x₆

The parameters x₁, x₂, ..., x₆ are used to fill the write buffer with a data pattern. Any number of parameters can be entered with a maximum of six. The write buffer is set to zeros if no Data Statement appears.

Example: Data 5555 6666 write buffer: 5555
6666
5555
6666
.
.
.

Data 5555 6666 7777 write buffer: 5555
6666
7777
5555
6666
7777
.
.
.

INCREMENT x

Increment causes each word of the write buffer to be increased by the value x.

CODE x₁ x₂ x₃ x₄ x₅ x₆

x₁, x₂, ..., x₆ are machine language instructions. Any number of parameters can be entered with a maximum of six. Pass instructions are entered if trailing parameters are omitted.

Code with no parameters acts as a NO-OP (Pass) instruction.

BASIC AID COMMANDS

START J

Start causes execution of the generated program. Execution will continue until the alert key is depressed or until the program hangs. If the program hangs, Master Clear and GO will return the buffer controller to the condition prior to executing START.

J = First statement to be executed.

RESTART Restart will return AID to the initial conditions.

NORMAL n Normal will assign normal channel n for those subroutines which require a normal channel to be specified such as the SBU read and write subroutines.

DELETE N Delete will remove the statement at line N and move the remaining statements up.

INSERT N Insert will take the last statement entered and insert it in line N. The remaining statements will be moved down.

BKPT a Breakpoint will cause a jump to the line specified by the START J statement. The jump will be inserted in the generated code at address a. Address a is an absolute memory address.

Example: Typing START will cause a jump to line A.
 Typing START C will cause a jump to line C, etc.

MONITOR Typing monitor will toggle the monitor bit in the FLAG WORD. With the monitor bit set, the first two words of the generated code for each statement will consist of the following code:

```

1B02    A = F.#2;
BC10    /U(MONITOR);
      .
      .
      .
  
```

The purpose of these two instructions is to return to the AID monitor to check if the alert key has been depressed and to advance to cursor on the display.

PASS Pass will cause a pass count to be displayed each time the PASS statement is executed.

WAIT X x = 16 bit hex word.
 Wait will cause a delay of approximately $X \cdot 2 \times 10^{-6}$ sec.

SBU AID STATEMENTS

SBU A a L n I

SBU causes a transfer between SBU and SCU Memory.

A = O or I for Output or Input
a = SBU address
L = transfer length
n = block count if I is absent
= address increment if I is present
(n is optional)
I = address increment flag (optional)

Example: SBU O a L will cause L words to be written to address a.

SBU O a Ln will cause Lxn words to be written to address a.

SBU O a Ln I will cause L words to be written to a on the first pass, L words to be written to a+n on the second pass, etc.

SBUCOMPARE L a₁ a₂ J₁ J₂

L = compare length
a₁ = 1st SBU address
a₂ = 2nd SBU address
J₁ = compare good jump line number
J₂ = compare bad jump line number

SBUCOMPARE compares L words beginning at a₁ with L words beginning at address a₂.

If a compare error is detected, the word number (relative to either a₁ or a₂) is displayed. Word number = 0, 1, ..., n.

CONNECT X

X = normal channel six connect code for SAC_1, SAC_2, 7000, etc.

X in this case is an 8-bit code in order to select the SCANNER or wrap-around hardware.

Example: CONNECT 0011 will select the SCANNER and SAC_1 in most cases.

SAC AID STATEMENTS

SAC A a L

A = O or I for Output or Input
a = SAC channel address
L = transfer length

SAC causes a transfer of L words between the SCU memory and the SAC channel using single word transfers.

SACX A a₁ a₂ L n

A = O or I for Output of Input
a₁ = SAC channel address
a₂ = SBU memory address
L = transfer length
n = block count (optional)

SACX causes a block transfer from SBU memory to the SAC channel of length L. If n is present, n×L words will be transferred in L word blocks.

7000 AID STATEMENTS

ADDRESS a₁ a₂ a₃ a₄ a₅ a₆ a₁, a₃, a₅ = SBU starting address

a₂, a₄, a₆ = SBU terminating address

ADDRESS causes from 1-6 addresses to be output to the 7000 channel.

NOTE: The 7000 channel requires that bit zero of the first address be set and that bit zero of the last address be set.

FUNCTION x₁ x₂ x₃ x₄

x₁, x₂, x₃, x₄ = FUNCTIONS

FUNCTION causes from 1-4 functions to be output to the 7000 channel.

865 DRUM STATEMENTS

DRUM A x₁ x₂ x₃

A = O or I
x₁ = 0 = full page mode
x₁ = 1 = 1/4 page mode
x₂ = head increment value
x₃ = sector increment value

DRUM and FUNCTION provide the same basic capabilities except that DRUM allows the head and sectors to be incremented after each pass through the subroutine.

Example: DRUM O 0 1 1 will cause the sector to be incremented until track zero has been written; then the head will be incremented etc.

DRUM O 0 0 1 will cause only the sector to be incremented until track zero is written.

DRUM O 0 1 0 will cause only sector zero to be written on every track.

817 DISK STATEMENTS

DISK A X₁ X₂ X₃

A = O or I

X₁ = head number = 0 or 1

X₂ = position increment value, 0-1FF

X₃ = sector increment value, 0-26.

DISK and FUNCTION provide the same basic capabilities except the DISK allows the position and sector to be incremented after each pass through the subroutines.

DISK and DRUM provide for the same basic operations.

3000 AID COMMANDS

URSFUNCTION X

X = function code for a 3000 normal channel and must include the function, connect data and parity bits where required.

Example: URSFUNCTION 5000

Connect bit = 1

Parity bit = 1

Connect code = 000

URSOUTPUT L

L = transfer length

URSOUTPUT outputs L words from the write buffer on the 3000 normal channel.

URSINPUT L

L = transfer length.

URSINPUT inputs L words from the 3000 normal channel to the read buffer.

EXAMPLES:

1. Master Clear the Station Interface.

```
A OUTPUT 6 1012          (Select M.C., SCANNER, CONNECT)
B OUTPUT 7 00E0          (Select M.C. INTERFACE)
C OUTPUT 6 0012          (CLR, M.C.)
D OUTPUT 7 0000          (CLR M.C. INTERFACE)
E STOP
*F
.
.
.
```

2. Write an Address Pattern through the SBU and Check Results

```
A DATA 0 1 2 3
B SBU O 0 4 4 1
C SBU I 0 4 4 1
D COMPARE4 E G
E INCR 4
F JUMP B
G STOP
*H
.
.
.
```

3. Input Status from a 3000 Device

```
A URSFUNCTION 5000
B OUTPUT 7 8000
C INPUT 5
D STOP
*E
.
.
.
```

4. 865 Drum Operation

Step 1: Set up SBU core

```
A DATA OFFF FF00 000F
B SBU O 0 3 4
C DATA FFFF
D SBU O B 1              (set sync pattern)
E DATA 0000
F SBU O C 4              (set header pattern)
*G
H
I
.
.
.
```

Step 2: Write Head 4 Sector 6

```
A COMM 0012
B ADDRESS 8000 000B 000C 000F 0010 820F
C FUNCTION 0404 4206
D INPUT 6
E MASK 0200 A D
*F
.
.
.
```

Step 2: Read Head 4 Sector 6

```
A CONN 0012
B ADDR 880C 80F 810 8A0F
C FUNCTION 0404 4006
D INPUT 6
E MASK 0200 A D
*F
.
.
.
```

Step 2: Drum Test Increment Sectors and Heads

```
A SBU O 10 20 10
B CONN 0012
C ADDR 8000 B C F 10 820F
D DRUM O 0 1 1
E INPUT 6
F MASK 0200 G E
G ADDR 880C 80F 810 8A0F
H DRUM I 0 1 1
I INPUT 5
J MASK 0008 K I
K SBUC 200 10 810 L N
L INCR 1111
M JUMP B
N STOP
O
P
```

NOTE: Only the first four characters of each statement need be typed.

Example: CONNECT 5000 = CONN 5000

NOTE: The first word address (FWA) displayed after typing START includes the jump back to the AID monitor. To breakpoint at the actual sub-routine code, set the breakpoint to the FWA+2.

JOB CONTROL LANGUAGE – JCL1

G

JCL1 is a job control language which may be called as a primitive following system log-on. It allows, for example, the system user to display (directly or indirectly) his virtual space, to enter data in his virtual space, to breakpoint the execution of a program, to call and execute other routines such as EDIT, BUFFALO, PLSTAR and any of a large number of library routines.

<u>NAME</u>	JCL1	<u>MODULE</u>	JCL1
<u>PROGRAMMER</u>	N. R. Lincoln		
<u>PURPOSE</u>	To provide shorthand forms of frequently used system commands for interactive users.		
<u>OTHER ENTRY POINTS</u>	None		
<u>PARAMETERS</u>	None		
<u>FUNCTION VALUE</u>	None		
<u>DESCRIPTION</u>			

JCL1 provides shorthand forms of frequently used system commands for interactive users. Because JCL1 provides essentially a different mode of operation, its commands differ in format from the usual CALLPROC format:

XXX (param, param,)	CALLPROC format
XXX param, param, <u>EOL</u>	JCL1 format

Parameters in JCL1 may be numbers (without the # sign) which are interpreted as hex, or any unquoted symbol string which stands for a file name in STORE and MAP calls. Any address parameter (addr) may have an at (@) sign catenated at the end followed by a space, comma, or a single digit 0-9 signifying the corresponding offset to be used (see OFFSET in commands listed below) as part of the address. JCL1 commands and parameters may be separated by any non-blank, non-aphanumeric character(s). A line ends with an EOL (end of line).

OTHER SUBROUTINES CALLED

D	Display last address displayed by a D command
D addr	Display memory beginning at addr
D addr, X	Display memory beginning X 64-bit words from addr
D addr, X, C	Display memory beginning X characters from addr
D addr, X, B	Display memory beginning X bits from addr
D addr, X, H	Display memory beginning X half-words from addr
D addr, X, R	Display memory beginning at addr offset by the number of 64-bit words contained in register X
D addr, X, CR	Display memory beginning at addr offset by the number of characters in register X
D addr, X, BR	Display memory beginning at addr offset by the number of bits in register X
D addr, X, HR	Display memory beginning at addr offset by the number of half-words in register X
DI	Display last addresses displayed by DI command
DI R	Display contents of register R and twelve words of data at address pointed to by R
DI R, X	Display contents of register R and twelve words of data at address pointed to by R offset by X 64-bit words
DI R, X, P1	Display contents of register R and twelve words of data at address pointed to by R offset by X items where P1 defines the item types as in the D command P1 may be C, B, H, R, CR, BR, HR
DR	Display last registers displayed by a DR command
DR R	Display twelve registers in last track beginning at full-word register R
DR R, X, P1	Display register R offset by X items according to the value P1 as in the D command
DC	Display last addresses displayed by DC command

DC addr1, addr2	Display side-by-side two vectors of twelve 64-bit words; the left-hand vector begins at addr1
DC addr1, addr2, X	Display side-by-side vectors offset by X 64-bit words from addr
DC addr1, addr2, X1, X2	Display side-by-side vectors each offset from their corresponding addrs by X1 and X2
DC addr1, addr2, X1, X2, P1	Display side-by-side vectors offset by X1 and X2 from addr1 and addr2, respectively, by characters, half-words and bits according to the value of P1 as in the D command
DC addr1, addr2, X1, X2, P1, P2	Display side-by-side vectors offset by X1 and X2, respectively, from addr1 and addr2, the nature of the offsets are determined by the values of P1 and P2. Note that if one vector is full-word and the other is half-word, the half-word vector will be displayed one half-word per line to ensure positional correspondence with the full-word vector – full-word number five will be on the same line as half-word number five
DCI	Display memory last displayed by DCI command
DCI R1, R2	Functions similar to DC in displaying side-by-side vectors, however, R1 and R2 are register numbers containing pointers to the displayed regions
B addr	Breakpoint at address addr for READ, WRITE and EXECUTE
B addr, P1	Breakpoint at addr for cases specified by any combination of the catenated symbols R (for read), W (for write) and E (for execute)
B addr, P1, P2	Breakpoint at addr according to the mode specified by P1 and continue to breakpoint P2 times before stopping. Note that, at arriving at a breakpoint, the last display format will be used to display memory.
S	Step program one instruction
S P1	Step program P1 instructions
X addr	Execute program beginning at addr
C	Continue breakpointed program
C addr, P1, P2	Set breakpoint according to P1 and P2 and continue execution
E addr, num, num, ...	Enter hexadecimal data num, ... beginning at addr up to ten half-words of data

ER R num, ...	Enter hexadecimal data num, ... beginning at 64-bit register R up to ten half-words
EHR R num, ...	Enter hexadecimal data num, ... beginning at 32-bit register R up to ten half-words
ETX addr, string	Enter ASCII data string beginning at addr and ending with EOL
R	Roll current display forward one increment – an increment is determined by the display mode
R P1	Roll current display forward P1 increments
BACK	Roll current display backwards one increment
BACK P1	Roll current display backwards P1 increments
CAT addr	Catalog module at addr
T	Display last area displayed by T display
T addr, X, P1	Display ASCII text beginning at addr and offset according to the value of P1 as in the D command
TI R, X, P1	Display ASCII text beginning at location pointed to by register R and offset according to the value P1 as in the D command
OFFSET addr	Set offset register 0 to addr. (see note below)
OFFSET addr, n	Set offset register n to addr. (see note below)

NOTE

The OFFSET facility is used in the JCL1 commands which specify an address parameter. The specified address from the JCL1 command is added to the indicated offset register to give the true virtual address. For example, if offset register 0 is set to FF7F FFD8000 and offset register 3 is set to 10000000, then

```
D 8340 @
    will cause display of address
    FF7F FFE0 0340, and
D 500 @ 3
    will cause display of address
    10000500
```

STORE name, addr (, n)	Store the named file beginning at addr. See notes below
OPEN name (, addr)(, m)(, p)	OPEN the named file and map it into memory beginning at addr (if specified). See notes below
CLOSE name	Close the named file

NOTES:

1. File names must always be followed by a comma or end of line. File names may be 1-7 characters (for the EM-1 prototype file system). When using the STAR/EM-1B file system, file names may be in the form fname owner-ID, the total characters therein not to exceed 16. If "owner-ID is absent (i.e., form "fname,") JCL1 will append the logged on "USER-ID" as the "OWNER-ID".
2. (, n) is an optional parameter specifying the actual number of blocks to be stored. The default is the number of reserved blocks allocated for that file. (n) must be less than or equal to the number of reserved blocks.
3. (, m) specifies the creation of a file "m" reserved blocks long.
4. (, p) is used only with the STAR/EM-1B (and EM-1 prototype with STAR stations attached) to specify a specific unit on which a file is to be created. A "0" unit number or absent parameter indicates that the system is to assign the file as it sees fit.

COPY name1, name2

Copy file "name1" to file "name2". If the files are of unequal length, the shortest length will be used for the copy.

COPY name1, name2, unit1, unit2

Same as above except that the user may specify a particular unit number as source and destination. This permits copying the same named file from one unit to another. If the unit number is 100₁₆ or greater, the unit is assumed to reference one of the 4 disk packs on the EM-1 prototype system. If the unit number is 0, the first occurrence of the file in either the prototype or STAR/EM-1B system is used. The order of file search for the prototype is:

1. Prototype (854) disks - 0, 1, 2, 3
2. EM-1B/STAR disks

For prototype system only:

<u>Unit Number</u>	<u>Drive Number</u>	
100	0	(Normally ADL RED PACK)
101	1	(Normally ADL TEMP1 PACK)
102	2	(Normally ADL TEMP2 PACK)
103	3	(Normally Archive Pack)

If name2 is omitted, it is assumed to be the same as name1.

EDIT allows a system user to line edit his source files from a display terminal. It is not intended as a generalized text editor but instead finds its greatest use in preparing source files for assembly or compilation. Complete lines, of any length up to 80 characters, may be inserted or replaced; individual words or characters cannot, in general, be inserted or replaced.

The following pages describe the EDIT facility in detail.

<u>NAME</u>	EDIT	<u>MODULE</u>	EDIT
<u>PROGRAMMER</u>	C. L. Hawley		
<u>PURPOSE</u>	Line edit facility for source files.		
<u>OTHER ENTRY POINTS</u>	EDIT_C		
<u>PARAMETERS</u>			
1.) Description	Base address of file A		
Type	Value		
Default	#10000000		
2.) Description	Base address of file B		
Type	Value		
Default	#18000000		
3.) Description	Base address of file C		
Type	Value		
Default	#20000000		
4.) Description	Base address of file D		
Type	Value		
Default	#28000000		
<u>FUNCTION VALUE</u>	None		
<u>DESCRIPTION</u>			

EDIT operates on four virtual memory "files".

Old Files (Input)

A Default Base Address #10000000
 B Default Base Address #18000000

New Files (Output)

C Default Base Address #20000000
 D Default Base Address #28000000

At entry files A and C are selected.

EDIT-C allows the user to exit from EDIT and, upon return, find all files in the state at which they existed at exiting.

Available Commands

In the following commands, PG is a decimal page number
 LN is a decimal line number
 NUM is a decimal count

File Selection Commands

@ or

@1

Toggle the selection of "old file".
(If file A is selected, change to B; if file B, change to A).

@2

Toggle the selection of "new file".

@3

Toggle the selections of both "old file" and "new file".

Copy Commands

+

Copy from the current location of "old file" to "new file" until an ASCII FILE SEPARATOR is encountered.

+PG, LN

Copy from "old file" to "new file" up to but not including the designated page and line.

C NUM

Copy the designated number of lines from "old file" to "new file".

Skip (Delete) Commands

-

Skip the current line of "old file".

-PG, LN

Skip up to the designated page and line of "old file".

Insert Commands

I NUM

Insert the designated number of lines into "new file". A line beginning with an ASCII "group separator" will terminate the insert even though the count is not exhausted. If the last character of a line is an ASCII "NULL" (hex 00), then more characters may be input to the same line.

Insert a line consisting of an ASCII "form feed" (hex 0C).
(This causes a new page to be started).

Replace Commands

X

Replace current line with replacement line. This command has the effect of executing a delete (-) followed by an Insert (I).

X PG, LN

Copy up to specified page and line and ready terminal to accept insert. The new line replaces the current line.

Reset (Rewind) Commands

- R1 Move contents of "new file" to "old file" and reset "old index".
- R2 Reset "old index".
- S1 Reset "new index".

Exit Command

- E Exit from EDIT program.

OTHER SUBROUTINES CALLED INPUT

BUFFALO

I

INTRODUCTION

Buffalo is an algebraic field-free assembly language for the buffer controller computer, similar in form to the PL/* assembly language for the STAR central processor.

The Buffalo assembler is a STAR program that assembles overlay programs for the buffer controller. It is a two-pass assembler. The current version, Buffalo II, is upward compatible with the original Buffalo.

A program written in Buffalo consists of two types of statements, directives and instructions, that can be intermixed. It can also have an identifying program name preceding the initial PROC statement, written as a label and not used elsewhere.

DIRECTIVES

Directives are used to make data assignments or to direct the assembler in some way. Except for the ORG directive, they are reserved words and cannot be used as symbols in the program. The directives used are listed below and described in the following sections.

PROC	CON	OFF	ENDIF	MACRO	IFEQ
PROCOFF	EQU	TITLE	EJECT	ENDM	IFNE
ORG	BITSET	IFDEF	SPACE	IFT	IFGE
END	LIST	IFNDEF	XREF	IFF	IFLT
					IFLE
					IFGT

PROC

Keyword PROC is the first keyword used in a program and follows the optional program name.

The occurrence of keyword PROC signifies the beginning of a new program. A 2-word program start header is produced in the object code listing. The first word of this header is always #FFFF. (The hash symbol # is used here for hexadecimal

constants.) The second word is filled with the number of words to the next ORG, PROC, or PROCOFF statement. Special program identification or parameters can thus be placed in the object code listing merely by following the PROC statement with appropriate constants.

PROCOFF

This directive causes inhibition of further object code generation until a PROC statement is encountered. The print listing is not affected by the directive.

ORG

Keyword ORG defines the starting location of the program. An ORG statement must terminate with a semicolon. ORG can appear anywhere within a program. If not at the beginning, its effect is to stop allocating program space from a previously defined ORG to a new space starting with the newly defined ORG.

Examples: ORG #2A0;
 ORG JOHN + 20;
 ORG ORG + #2F;

Note that the address can be symbolic, decimal, or a hexadecimal constant. If symbolic, the symbol(s) used must have been previously defined. Note also that the current value of ORG can be used in an expression.

Because the assembler is designed to assemble overlay programs, the occurrence of an ORG within the program causes a special message of two 16-bit words to be placed in the object code file. The message consists of a starting address in the first word followed by a length in the second word. Every span of program or data has such a message header. Whenever an ORG is encountered, the length of the previous program is determined and inserted in its message header. Then a new message header is begun by inserting the starting address of the new origin, and assembly continues. Every program should contain an ORG definition before any program statements are written. If omitted, the default ORG is address 0000. Because all program start headers begin with #FFFF, no program should use address #FFFF as an origin.

END

Keyword END directs BUFFALO to terminate assembly, and a terminating two-word trailer consisting of #FFFF0000 is put in the object file. The keyword END must be followed by a semicolon.

The error file is appended to the listing after the END statement is processed.

CON

Keyword CON establishes a relationship between a symbolic address preceding keyword CON and an expression following the keyword CON. The format of the statement is as follows:

```
name CON (expression 1) expression 2;
```

The optional constant (expression 1) is a length specification defining the number of ASCII characters or buffer controller words in the constant. The length specification is an expression following the general rules for expressions as given below. Any symbolic constants used in length expressions must be defined before use.

Expressions can be decimal constants, hexadecimal constants, symbolic constants, or ASCII constants or strings. Any combination of the first three types can be connected by addition, subtraction, multiplication, or division operators (+-*/). Expressions are evaluated from left to right. Expression 1 is limited in size to four hexadecimal digits. If the arithmetic result produced is larger than can be represented by four hexadecimal digits ($2^{16}-1$), an error message is formatted. Expression 2 is limited in size by the value of expression 1.

Hexadecimal numbers are prefixed by the symbol #. Decimal numbers have no prefix or suffix. ASCII constants are delimited by pairs of double quotation marks; for example, "ASCII CONSTANT." Embedded quotes are not permitted between pairs of quotes.

A pure hexadecimal string or a pure ASCII string can be of any length, and each is limited only by the allowable length of a unit record (80 characters). ASCII constants are left-justified to a word boundary and filled to the specified length with the ASCII space symbol (#20). Hexadecimal constants and all other evaluations are right-justified.

If a string is more than one buffer controller word in length, the symbolic address points to the first word.

The length specification, *n*, and the expression are both optional for strings. If no length is specified, one word is assigned or the number of characters or digits in the string is counted by the assembler and the number of buffer controller words required to hold the string is calculated. If a length is specified but no literal constant follows, the length is interpreted as the number of buffer controller words that are initialized to zeros. If neither the length nor the expression is defined, one word of all zeros is allocated. If the defined length is zero, no space is allocated.

```

Examples:   ERROR_MESSAGE CON (14) "INVALID SYMBOL";           (a)
            ERROR_MESSAGE-ONE CON (#1A);                       (b)
            ZERO CON;                                           (c)
            BUFFER_ADDRESS CON BLUE+16-#3F;                    (d)
            LONG_ASCII_CONST CON "ABCDEFGHI";                   (e)
            LONG_HEX_CONST CON #1234567ABC;                    (f)
            CON (2) #123;                                       (g)

```

In case (a), the ASCII constant is 14 characters (seven words) long. In case (b), hexadecimal 1A words (26 decimal words) are allocated for the constant. The area is cleared to zeros. In case (c), one word is allocated and cleared. In case (d), the symbolic address BLUE is obtained and added to hexadecimal 10 (decimal 16) and, from this result, hexadecimal 3F is subtracted. The result is a one word constant. The symbolic address BLUE must have been previously defined. Cases (e) and (f) illustrate the definition of long strings the lengths of which are not specified. Five words are allocated and filled for case (e), and three words are allocated and used for case (f). Two words are allocated in case (g) and the contents are #00000123.

EQU

EQU equates a symbol to an expression. Its general format is:

```
name EQU expression;
```

Any symbols used in the expression must be previously defined. The result is always interpreted as a 16-bit quantity. If the arithmetic result is larger than $(2^{16} - 1)$, an error message is formatted.

EQU differs from CON in that no memory locations are required. EQU is, therefore, merely a directive to the assembler that equates a symbolic name to a constant or to an expression that evaluates to a constant.

IFDEF

IFDEF enables assembly if the name following the directive has been defined previously.

IFNDEF

IFNDEF enables assembly if the name following the directive has not been previously defined.

ENDIF

ENDIF, followed by a name, determines the span of the preceding IFDEF or IFNDEF statement having the same name.

If statement is correct, the notation ASSEMBLY ON/OFF in the error field defines previous assembly conditions.

These three directives may be nested in a manner analogous to nested DO loops in FORTRAN. The first IFDEF or IFNDEF statement which inhibits assembly causes all following IFDEF or IFNDEF to be inactive. Assembly is resumed upon recognizing an ENDIF statement having a name corresponding to one which inhibited assembly in an earlier IFDEF or IFNDEF statement. In default of any of these three directives, normal assembly is performed. Nesting is permitted to a level of eight.

Example:

	COMMENTS
NAME: PROC;	Assemble
CON;	
CON;	
ORG #1000;	
IFDEF NAME;	Name is defined so Assemble
IFNDEF NAME1;	Name1 is undefined so Assemble
IFDEF NAME2;	Name2 is undefined No assembly
ENDIF NAME2;	End of span for Name2 Assemble

ENDIF NAME1	End of span for Name1 Assemble
ENDIF NAME	End of span for Name Assemble
	To End

EJECT

The EJECT directive places a page eject control character in the output listing and puts a title (if one has been defined) at the top of the page.

SPACE

The SPACE directive is followed by an expression that must evaluate to a constant n ($0 \leq n < 64$). It causes n blank lines of print output to be generated.

Example: SPACE 10; causes 10 blank lines to be printed.

XREF

The XREF directive causes a cross-reference listing to be generated and formatted for printing. In default of the directive, no cross-reference listing is obtained.

MACRO

The MACRO directive flags the source of a set of statements which will be inserted into the program when called by a statement with the same label as the one on this statement. This label may have as many as eight characters.

The macro source is moved to the symbol table file to be used in macro expansions. The source is thus saved for reuse the same as symbols — see BUFFSYM.

ENDM

The ENDM directive signals the end of the macro source statements.

IFT

This directive is followed by three expressions. If the first two are identical (true), then assembly continues normally. If they are not, the third is evaluated and that many source statements are skipped. This directive is most useful within macro definitions where character string substitutions are used.

EXPRESSIONS

Expressions consist of any mix of hexadecimal constants, decimal constants, and symbolic constants connected by the operators +-*/. Hexadecimal constants are preceded by the symbol #. ASCII constants are not permitted in an arithmetic expression. A variable number of spaces can appear in the expression on either side of the operators and at the beginning or end. A beginning decimal or hexadecimal constant in an expression is positive if no algebraic sign is given. Parenthetical expressions are not allowed. All expressions are evaluated left to right modulo the size of the field to be filled. An error is indicated whenever the expression exceeds the intended modulus. Catenation of two or more arithmetic operators is not allowed.

FORMAT RULES

Statements are free field with a variable number of spaces permitted except as noted above or in the following. The symbol pairs or triplets =CC, =C, and /C (C being the character(s) immediately following the = or /) form compound operators. No spaces are permitted between characters comprising the compound operator.

OVERLAYS

In assembling overlays, the following rules apply.

- Each overlay must begin with a PROC statement.
- The PROC statement can be followed by any number of CON statements.
- Following PROC or CON must be an ORG statement giving the beginning address of the program.
- ORG can be changed at will within a program.
- Assembly continues until the END statement is reached.

Example:

<u>Statements and Directives</u>	<u>Object Code</u>	<u>Comments</u>
NAME_1: PROC;	FFFF }	PROGRAM START
DUMMY CON "NAME_ONE";	0004 }	HEADER
	4E41	
	4D45	
	5F4F	
	4E45	

<u>Statements and Directives</u>	<u>Object Code</u>	<u>Comments</u>
ORG # 100;	0100 } (LENGTH) } P R O G R A M	ORIGIN HEADER (PROGRAM AND DATA FOR PRO- GRAM NAME_1)
ORG # 1000;	1000 } (LENGTH) } P R O G R A M	ORIGIN HEADER (MORE PRO- GRAM FOR NAME_1)
	XXXX	CHECKSUM OF PROGRAM
NAME_2: PROC DUMMY_A # 123AB	FFFF } 0002 } 0001 23AB	PROGRAM START HEADER
ORG # 150	0150 } (LENGTH) } P R O G R A M	ORIGIN HEADER PROGRAM FOR NAME_2
	XXXX	CHECKSUM OF PROGRAM
END;	FFFF } 0000 }	END TRAILER

The program start header associated with the PROC statement consists of one word of all ones followed by a word containing the length of the PROC message in buffer controller words. The words following the header are the message itself.

In the example, the message associated with the first PROC is "NAME_ONE". It is copied into words 3 through 6 inclusive of the object code appearing there as hexadecimal expressed constants.

Note that object code locations are continuous and that the beginning of each program is preceded by a 2 word header giving the starting address and length of the program.

As shown for program NAME_1, the origin can be changed at will within a given overlay program. Each origin assignment causes a two-word header to be embedded in the object code file.

A checksum of the last segment is added to the end of each overlay. It consists of the modulo 2^{16} sum of the 16-bit words in the object code starting after the length specification of the last preceding ORG, PROC or PROCOFF statement. The length specification of the overlay includes the checksum word.

COMMENTS

Comments are delimited by /* and */ as in PL/*. Any representable ASCII characters can appear between these pairs of delimiters except the combination */ which is always recognized as the end of the comment field. Comments can follow, but never precede, a statement or directive on a given line. Comments are not formatted, and appear in the output listing in the same relative columns as they appear in the input. A comment occupying a line not having a statement or directive is also unformatted and begins in print column 21. Comments can be continued from one line to the next indefinitely, subject only to the rule that the comment starts with the pair /* and terminates with */.

OUTPUT LISTINGS

The output listing containing the location, object code, the source statements, and the error messages is stored in virtual memory in compressed form beginning at a location specified by a parameter in the call to BUFFALO. If no parameter is specified in the call, the listing goes to virtual address # 20000000.

Object code location	columns	1 - 4
Object code contents	columns	6 - 9
Execution memory cycles	columns	11
Labeled statements	columns	13 - 92
Unlabeled statements	columns	17 - 96
Comments	columns	17 - 96
Error messages	columns	98 - 121
Page and line number	columns	122 - 128

The first page of the output listing contains a copy of all PROC statements with their page and line numbers.

OBJECT CODE

Object code, together with the generated headers needed for loading, appears in virtual memory beginning at a location specified by a parameter in the call to BUFFALO. In default of the parameter being specified, the object code goes to virtual address # 30000000.

ERROR FILE

A separate error file contains only those statements that contain errors. If the storage address of the error file is not specified, it is stored at virtual address #38000000 by default.

DISPLAY

When Buffalo II is called from a display console, information displayed for the operator includes:

- program name
- Buffalo II version number
- current label being assembled
- current page and line number
- last encountered error — if any
- current error count

Following assembly, the number of blocks of print output is displayed. This includes the cross reference listing, if any.

ERROR MESSAGES

The following is a list of the error messages printed on the line having the offending statement. The same message is displayed at the terminal from which the program was initiated.

MISC INVALID CHARACTER

A character has been encountered which is meaningless in its context.

INVALID DECIMAL DIGIT

A decimal number has an invalid character embedded in it.

INVALID HEX DIGIT

A hexadecimal number has an invalid character embedded in it.

ASCII INVALID IN EXP

An ASCII string exists as part of an arithmetic expression.

DIAGNOSTIC-LENGTH FLAG	This error should only occur if there is a machine malfunction.
ASCII GT DEF'D LENGTH HEX GT DEFINED LENGTH	The CON statement contains a length specification the value of which is less than the actual string length to be stored.
INVALID LENGTH SPEC	The CON statement contains a length specification which cannot be evaluated properly.
UNDEFINED SYMBOL	A symbolic expression contains a symbol which has not been defined in the program.
DIVISION BY ZERO	An arithmetic expression contains a division having a zero divisor.
LINE EXCEEDS 80 COLUMNS	Self-explanatory.
INVALID COMB OF CHAR'S	An invalid catenation of characters in source file.
MULTIPLY DEFINED LABEL	The label associated with the statement is defined in more than one location.
INVALID COMPOUND OP	The catenation of characters defining an instruction operator is invalid.
INVALID PAIR OF OPS	The pair of operators defining an instruction fail to do so.
NEGATIVE ADDRESS EXPR	The expression evaluates to a negative number.
OPERAND SIZE EXCESSIVE	The evaluated expression is too large to fill the instruction field.
UNDEFINED INDEX	The index register required to define the instruction is undefined.
INVALID INSTRUCTION	Self-explanatory.
INVALID DIRECTIVE	Self-explanatory.
NONSENSE STATEMENT	Insufficient meaning in the statement to determine the kind of error.
INVALID EXPRESSION	Some rule in forming expressions has been violated.
SYMBOL UNDEF'D BY IFDEF	The symbol following the ENDIF statement does not correspond to a like symbol on an IFDEF or IFNDEF directive.
IFDEF OR MACRO NEST ERR	The ENDIF statement is out of order.

LENGTH SPEC IS NEGATIVE

The CON statement contains a length specification which is negative.

LENGTH SPEC TOO LARGE

The CON statement contains a length specification which is too large

CALLING BUFFALO

Buffalo is invoked by the following call statement:

BUFFALO (Symbol file, Source file, Print listing, Object file, Error file)

where	Symbol file	is the virtual address of the location where the symbol table is desired. In default, #8000000.
	Source file	is the virtual address of the file to be assembled. In default, #10000000.
	Print listing	is the virtual address of the location where the print listing is desired. In default, #20000000.
	Object file	is the virtual address of the location where the object file is desired. In default, #30000000.
	Error file	is the virtual address of the location where the error file is desired. In default, #38000000.

All addresses must be expressed in hexadecimal and include the symbol #.

In addition, the SYMBOL FILE may be saved for reuse with another assembly.

To do this:

1. Assemble first time with BUFFALO
2. Save SYMBOL FILE in permanent file
3. Save other output as desired

To reuse SYMBOL FILE:

1. Map in symbol file
2. Map in new source file
3. Execute BUFFSYM. The symbol file will be reused up to but not including the original END; statement or END: (label) whichever is first.

CALLING BUFFSYM

BUFFSYM is invoked by the following call statement:

BUFFSYM (SYMBOL FILE, etc.)

where SYMBOL FILE is the virtual address of the symbol file to be used. In default, #8000000.

All other parameters are the same as the BUFFALO call.

INSTRUCTION LISTING

<u>Instruction</u>	<u>Code</u>	<u>Assembly Form</u>
Selective Stop	0000	/;
Selective Set Bit T of A	010T	A =S #; # is a 4-bit constant called S or T in the instruction.
Selective Clear Bit T of A	020T	A =R #;
Selective Toggle Bit T of A	030T	A =T #;
Count Leading Zeros in A to A	0400	A =Z;
Shift A Right T Places End-off	050T	A =E #;
Shift A Right T Places Circularly	058T	A =C #;
Shift A Left # Places Circularly	058T	A =CL #; T = 16 - #;
Enter Index 1 With (A) + Y	06 Y	B1 = A + Y; Y is an 8-bit constant
Enter Index 2 With (A) + Y	07 Y	B2 = A + Y; stant
Clear C	0800	C =R;
Load C With Bit # of A	081T	C = #; T = 15 - #
Set C if A Overflow	0820	C = O;
Bit # of A or Overflow	083T	C = O, #; T = 15 - #
Set C if A Odd Parity	0840	C = P;
Bit # of A or Odd Parity	085T	C = P, #; T = 15 - #
A Overflow or A Odd Parity	0860	C = P, O;
Bit # of A or Overflow or Odd Parity	087T	C = P, O, #; T = 15 - #
Load C With Bit T of Channel S	09ST	C = #, #; #T precedes #S
Selective Set Bit T of Channel S	0AST	#, # =S; "
Selective Clear Bit T of Channel S	0BST	#, # =R; "
Input To A From Channel S	0CS0	A =I #; OR A =< #;
Set Channel S Per Ones In A	0DS0	# =S A/O;
Set Channel S Per Zeros In A	0DS8	# =S A/Z;
Clear Channel S Per Ones In A	0ES8	# =R A/O;
Clear Channel S Per Zeros In A	0ES0	# =R A/Z;
Output A to Channel S	0FS0	# =O A; OR # =< A;
Output Complement A to Channel S	0FS8	# =ON A; OR # =< ^ A;
Add Immediate	10 Y	A =+ Y;
Subtract Immediate	11 Y	A =- Y;
Exclusive Or Immediate	12 Y	A =X Y; OR A =% Y;
Logical Product Immediate	13 Y	A =P Y; OR A =& Y;
Test Index 1 Immediate	14 Y	C = B1 ? Y;
Test Index 2 Immediate	15 Y	C = B2 ? Y;
Enter A Complement Immediate	16 Y	A =N Y; OR A =^ Y;
Load A With (A) + Y	17 Y	A =G (A + Y);

<u>Instruction</u>	<u>Code</u>	<u>Assembly Form</u>
Enter A With Address	18Y-1FY	A = M; M is an address mode
Enter Index 1 With Address	20Y-27Y	B1 = M; designator (see below)
Enter Index 2 With Address	28Y-2FY	B2 = M;
Test Index 1	30Y-37Y	C = B1 ? M;
Test Index 2	38Y-3FY	C = B2 ? M;
Load A	40Y-47Y	A =G M;
Load A Complement	48Y-4FY	A =GN M; OR A =G^ M;
Load Left Byte Into A	50Y-57Y	A =L M;
Load Right Byte Into A	58Y-5FY	A =R M;
Add	60Y-67Y	A =+ M;
Subtract	68Y-6FY	A =- M;
Exclusive OR	70Y-77Y	A =X M; OR A =% M;
Logical Product	78Y-7FY	A =P M; OR A =& M;
Replace Add	80Y-87Y	M =+ A;
Replace Add One	88Y-8FY	M =+ U;
Replace Left Byte	90Y-97Y	M =L;
Replace Right Byte	98Y-9FY	M =R;
Store	A0Y-A7Y	M =A;
Store Zeroes	A8Y-AFY	M =Z;
Load A and Clear	B0Y-B7Y	A =D M;
Unconditional Jump	B8Y-BFY	/U M;
A Zero Jump	C0Y-C7Y	A /Z M;
A Nonzero Jump	C8Y-CFY	A /N M;
A Positive Jump	D0Y-D7Y	A /+ M;
A Negative Jump	D8Y-DFY	A /- M;
Condition True Jump	E0Y-E7Y	C /T M;
Condition False Jump	E8Y-EFY	C /F M;
Input (A) Words	F0Y-F7Y	M =I; OR M < ;
Output (A) Words	F8Y-FFY	M =O; OR M > ;

Notes: The equal sign is always immediately followed by either a symbol or a blank. When a blank immediately follows the equal sign, it is to be construed as an ASCII space character (hexadecimal 20)

represents a 4-bit constant or an expression which equates to a 4-bit constant.

Y represents an 8-bit constant or an expression which equates to an 8-bit constant.

M is defined below.

A is the accumulator

B1 is index register 1

B2 is index register 2

C is the condition bit

Examples:

- If M = .Y Address = Y for Y < #100;
Otherwise, address is relative to current location if Y > #FF.
- If M = .(Y) Address = (Y)
- If M = B1.Y Address = Y + (B1)
- If M = B1.(Y) or B1(Y) Address = (Y) + (B1)
- If M = F.Y Address = (P) +Y
- If M = R.Y The assembler interprets Y to be an expression of address dimension. The assembler computes the forward or backward distance from current ORG to Y and assembles the proper instruction to correspond.
- If M = B2.Y Address = Y + (B2)
- If M = B2.(Y) or B2(Y) Address = (Y) + (B2)
- If M = B.Y Address = (P) - Y

This appendix consists of the Control Data PL/* Compiler/Assembler User Guide Reference Manual, Preliminary Revision 04, that is reprinted here in its entirety.

CONTROL DATA[®]

PL/STAR

COMPILER/ASSEMBLER

CONTROL DATA

C O R P O R A T I O N

**PRELIMINARY
USER GUIDE REFERENCE MANUAL**

PREFACE

PL/* is a free-format, algebraic assembly language especially developed to exploit the capabilities of the STAR central processor. PL/* was developed primarily as a tool to be used in the development of other languages, utility programs, and a monitor. Wherever possible, PL/* statements use PL-1 conventions.

Most PL/* statements are of the replacement type in which the operators define the instruction. For example, the instruction $A =U B+C;$ defines a floating-point add (the upper significance is determined by the compound operator =U). This identical statement could be a register-to-register add instruction or a vector add of either 32-bit or 64-bit operands. How the statement is interpreted is determined by the declarations defining the data types for A, B, and C. Default conditions covering options in an instruction give the programmer great freedom of expression. The language is surprisingly concise and easy to use.

CONTENTS

<u>Section</u>	<u>Title</u>	<u>Page</u>
1	SYNTAX NOTATION	1-1
2	CHARACTER SET	2-1
3	IDENTIFIERS	3-1
	Symbol	3-1
	Keywords	3-3
4	CONSTANTS	4-1
	Decimal Constant	4-1
	Floating-Point Constant	4-1
	Hexadecimal Constant	4-1
	Character Constant	4-1
	Address Constant	4-2
5	EXPRESSIONS	5-1
	Assembly Time Arithmetic	5-1
	Address Scaling	5-1
	Qualifiers	5-2
6	STATEMENTS	6-1
	Statement Types	6-1
	Spaces	6-1
	Labels	6-1
	Comments	6-2
	Alignment and Boundaries	6-2
	Format	6-3
7	DIRECTIVES	7-1
	List Control	7-1
	Location Control	7-1
	Register Counters	7-2
	Data Generating Directives	7-5

<u>Section</u>	<u>Title</u>	<u>Page</u>
8	PROGRAM CONSTRUCTION	8-1
	Program Structures	8-1
	Register Pairs	8-1
	Vector Instructions	8-2
9	COMPILE-TIME SYMBOLS AND STATEMENTS	9-1
	Compile-Time Replacement Statements	9-2
	Conditionals	9-2
A	INSTRUCTION SET	A-1

SYNTAX NOTATION

1

Whenever a PL/* statement or combination of elements is discussed, the manner of writing that statement or phase is illustrated using a uniform system of notation, which is as follows: (This notation is not part of the PL/* language.)

1. Unless otherwise stated, language elements must appear in the sequence given.
2. Any upper case letters or any delimiters other than †, ∫, or { } are part of PL/* syntax and must appear as written. The underlined portions of upper-case characters signify the shortened versions which the machine recognizes.
3. Lower case letters are general names for which specific information is supplied in the text.

DCL identifier;

DCL must appear followed by identifier (defined elsewhere) followed by a semicolon.

4. Braces are used to group and illustrate alternate forms. Elements vertically listed in braces are mutually exclusive and indicate that a choice is to be made.

SYNCH $\left\{ \begin{array}{c} P \\ H \\ F \\ C \end{array} \right\}$

The vertical stack indicates one of the letters, P, H, F, or C, must appear after SYNCH.

5. The character † inside braces indicates the group is optional and can be omitted.

% ORGR = $\left\{ \begin{array}{c} + \\ - \end{array} \right\}$ constant;

Either + or - can follow the equal sign, or the term might not appear at all.

6. The character \int inside the braces indicates the group can be repeated one or more times in succession.

DCL identifier attributes $\{, \text{identifier attributes } \dagger \int \}$;

In this statement the $, \text{identifier attributes}$ group can be repeated one or more times, or be omitted.

PL/* uses the ASCII character set listed in Table 2-1.

The term alphanumerics refers to the alphabetic set A through Z, the numeric set 0 through 9, and the underline character. The term delimiters refers to the set of ASCII characters other than the alphanumeric characters. The term syntax delimiters refers to the set of delimiters used in the infix notation of the assembler instructions to differentiate this set from delimiters used as arithmetic operators or separators (blanks). Some of the primary syntax delimiters used in PL/* are:

=A	absolute assignment operator
=C	ceiling assignment operator
=E	exponent assignment operator
=F	floor assignment operator
=G	get, register load operation
=L	lower assignment operator
=N	normalize assignment operator
=P	put, register store operator
=R	round assignment operator
=S	significant assignment or square root operation
=T	truncate assignment operator
=U	upper assignment operator
=X	index assignment operator
=+	increase operator
:	branch register or address follows
\	expansion
!	compression or reduction
.	merge or catenation
,	register pairs
<<	move left
>>	move right

Table 2-1. ASCII Character Set

Hexa-decimal	Character	Punch	Hexa-decimal	Character	Punch
20	space	no punch	41	A	12-1
21	!	12-8-7	42	B	12-2
22	"	8-7	43	C	12-3
23	#	8-3	44	D	12-4
24	\$	11-8-3	45	E	12-5
25	%	0-8-4	46	F	12-6
26	&	12	47	G	12-7
27	'(apostrophe)	8-5	48	H	12-8
28	(12-8-5	49	I	12-9
29)	11-8-5	4A	J	11-1
2A	*	11-8-4	4B	K	11-2
2B	+	12-8-6	4C	L	11-3
2C	,	0-8-3	4D	M	11-4
2D	-(minus)	11	4E	N	11-5
2E	.	12-8-3	4F	O	11-6
2F	/	0-1	50	P	11-7
30	0	0	51	Q	11-8
31	1	1	52	R	11-9
32	2	2	53	S	0-2
33	3	3	54	T	0-3
34	4	4	55	U	0-4
35	5	5	56	V	0-5
36	6	6	57	W	0-6
37	7	7	58	X	0-7
38	8	8	59	Y	0-8
39	9	9	5A	Z	0-9
3A	:	8-2	5B	[12-8-2
3B	;	11-8-6	5C	\	0-8-2
3C	<	12-8-4	5D]	11-8-2
3D	=	8-6	5E	^	11-8-7
3E	>	0-8-6	5F	_	0-8-5
3F	? @	0-8-7 8-4		(underline)	

An identifier is a string of alphanumeric and underline characters preceded and followed by a delimiter. The initial character must always be alphabetic. The maximum length of an identifier is 64K (K = 1024) characters.

SYMBOL

A symbol is a programmer-defined identifier. Symbols can represent registers, memory locations, and constants. Symbols carry properties of value, type, size, and length. If only one value is associated with a symbol, it is called the primary value.

The properties involved in a symbol are:

- value: the numeric value of the symbol; the bit address of an address symbol, the numbers associated with a set of registers.
- type: the basic kind of data represented: decimal string, vector, binary string, constant, etc.
- size: the basic unit of memory represented. The size of a character string is a byte. The size of a register can be either a full or half word.
- length: the number of size units implied by the symbol. A vector of 100 full words has a length of 100.

Symbols can represent both a memory location and a register value. These are called multivalued symbols. The primary value of a multivalued symbol is the memory value; the set of registers is the secondary value.

REGISTER NAME

A register name is a symbol with properties of: value – the register number; type – register; size – full or half word; length – 1.

ADDRESS SYMBOL

An address symbol contains the primary value of a memory location. The other properties vary with the type of the symbol.

SYMBOLIC CONSTANT

A symbolic constant is defined in a DCL directive using the attribute EQU. A symbolic constant has properties of value equals specified constant, type equals constant, and a length and size equal zero.

MULTIVALUED SYMBOLS

A multivalued symbol is defined with allocation of data fields, and has an address as a primary value. The other properties can vary. The secondary value is a full word register set of from one to three register numbers. These registers usually hold the base address and the index or offset of the data field defined.

USE OF SYMBOLS

When a multivalued symbol is used as an operand, it is usually obvious which value applies, but in an ambiguous case, the primary value is used. All instructions except Branch Immediate, Enter Immediate, and Increase Immediate use the secondary values. In the exceptions, the assembler uses the primary values. The secondary values can be forced with qualifiers. The REG attribute, %ORGW directive, and %ORGR directive all use the secondary value; other identifier occurrences use the primary value.

Some PL/* instructions have a unique arrangement of syntax delimiters which do not appear in any other instruction. These instructions use symbols as if they had the appropriate properties. For example, no other instruction has the same format as the Polynomial Evaluation.

The properties of a symbol become important when the symbol is used in an instruction which does not have a unique set of syntax delimiters, such as in the arithmetic instructions. The properties determine whether a sparse vector,

vector, or register instruction is assembled and whether the operands are full or half words. Qualifiers can be used to further define, modify, or override the interpretation of a symbol.

KEYWORDS

A keyword is an identifier which is part of the language. Keywords are not reserved words and can be used as symbols.

DIRECTIVE IDENTIFIERS

A directive identifier is a keyword used in the beginning of a statement to define the function of that statement. For example:

```
DCL
REGBLOCK
STABLE
```

ATTRIBUTES

Attributes are keywords that specify properties of data or symbols. For example:

```
FLOAT(7)
ARRAY (100)
```

SPECIAL KEYWORDS

A special keyword is a syntax delimiter. This word is preceded by an apostrophe (') and must be followed by a space. Special keywords are also known as quasi-operators. Examples of these keywords are:

'EQ	'DEL	'ABS	'BKPT
'NE	'MAX		'ALGO
'GE	'MIN	'ON	'EXIT
'LT	'REV		'MCLOCK
'LE	'XOR		'RCLOCK
'GT	'OR	'DFR	'STAR
'GE	'AND	'CHAN	'LDAR
'AVG	'NOT	'IDLE	'KEYS

RESERVED WORDS

The following words are reserved when assembling relocatable code:

<u>Name</u>	<u>Description</u>
ONE	Number One
PD	Parameter Descriptor
RETURN	Return Register
DSP	Dynamic Space Pointer
STACK	Current Stack Pointer
OLD-STACK	Previous Stack Pointer
LINK	Callee Data Base
ON	On unit stack pointer
FV1	Function value return
FV2	Function value return

For a further description of these registers see section entitled Program Structure.

The term constant refers to any one of the following definitions.

DECIMAL CONSTANT

A decimal constant is a string of numeric characters from the set 0 through 9, optionally preceded by a plus or minus sign. The number is converted to its binary equivalent, right-justified, and zero-filled. For example:

123 50 -477665

FLOATING - POINT CONSTANT

A floating-point constant is a string of numeric characters from the set 0 through 9, optionally preceded by a plus or minus sign, with a decimal point included.

3.1415926535898 666.

HEXADECIMAL CONSTANT

A hexadecimal constant consists of a number sign (#) followed by a string of characters from the set 0 through 9 and A through F. Each character is converted to its 4-bit hexadecimal equivalent, right-justified, and zero-filled. Optionally, a plus or minus sign can precede or follow the number sign. For example:

#12A #FF #EA40000 -#CAFE

CHARACTER CONSTANT

A character constant is an ASCII character string surrounded by double quotes ("). Double quotes cannot appear in the character string. Each character in the string is stored in its 8-bit ASCII code. For example:

"ERROR" "?" "SUPPLY; THEREFORE"

ADDRESS CONSTANT

An address constant is an expression delimited by square brackets. For example:

[LABEL] [JUMP_ADDRESS]

Explanation: [ABC]

1. If ABC is an address symbol, the value of the constant is the bit address of the address symbol.
2. If ABC is a register name, the value of the constant is the bit address of the register.
3. If ABC is both an address symbol and a register name, [ABC] is the bit address of the address symbol. The expression [ABC(B)] yields the bit address of the register.
4. If ABC is a symbolic constant, the value of the symbolic constant is used.

ASSEMBLY TIME ARITHMETIC

The general format for an expression is:

$$[\left\{ \pm \right\} \left\{ \begin{array}{l} \text{constant} \\ \text{address_symbol} \\ \text{register_name} \end{array} \right\} \left\{ \text{operator} \left\{ \begin{array}{l} \text{constant} \\ \text{address_symbol} \\ \text{register_name} \end{array} \right\} \right\}]$$

The operators allowed are plus, minus, divide, multiply, and exponentiation (**). The expression is evaluated from left to right. No parentheses are allowed for grouping, but they can be used with qualifiers.

A register in an expression is referenced as a bit address. If AAREG is a register name, it is referenced as the bit address of the register.

Given: DCL AAREG REG 1;

The expression [AAREG + 5] equals 45 because the bit address of AAREG is 40.

An expression can be used wherever an address constant can be used.

ADDRESS SCALING

The operator circumflex (^) followed by a P, F, H, or C, scales (truncates) to a page, full word, half word, or character quantity, respectively.

<u>Code Letter</u>	<u>Quantity</u>	<u>Shift Right</u>
P	page	8 bits
F	full word	6 bits
H	half word	5 bits
C	character	3 bits

For example: DCL ABC ARRAY PRESET;

[ABC^F]	yields the full word address of the variable ABC
[ABC^H]	yields the half word address
[ABC^C]	yields a byte address
[ABC(B)^F]	yields the full word address of base register ABC
[ABC + 5]	is a bit address result
[ABC + 5^F]	is a word address of ABC
[ABC^F+5]	is a word address five words from the start of ABC

QUALIFIERS

Qualifiers are used to force a secondary value to be used when referencing a multivalued symbol, to force a symbol to have a type different from its own, or to sample the length property. Qualifiers immediately follow the symbol they qualify and consist of characters surrounded by parentheses.

FORCING THE SECONDARY VALUE

Because symbols can be multivalued, and the primary properties are used when there is ambiguity, qualifiers provide a method of referencing the register values associated with the symbol. The following key letters are available as register qualifiers:

- B base address register or first register defined.
- X index register or second register defined.

For example:

```
DCL VECTOR ARRAY (100) REG 10/11;  
VECTOR(B) full word register 10  
VECTOR(X) full word register 11
```

If no such register exists, a value of 0 is returned.

FORCING AN INSTRUCTION

PL/* has many instructions with the same infix notation, and the instruction assembled is dependent on the properties of the operands used. Rather than require a separate declaration and numerous unique symbols, qualifiers can be used with a symbol to force properties other than those usually associated with the symbol.

The following key characters enclosed in parentheses can be used as qualifiers in instructions:

- A force full word vector
- S force sparse vector
- D decimal string
- Y binary string
- C character string
- < use upper half word
- > use lower half word
- B use base register
- X use index register

One or two qualifiers, surrounded by parentheses, can follow a symbol to give the appropriate conditions. The combination (A<) or (A>), and (S<) or (S>) force a half word vector and half word sparse vector, respectively. The <, > characters can be used interchangeably in this case (i. e., (A<) and (A>) yield the same results).

The last four qualifiers, besides forcing a type, force the appropriate register number. The combination B< gives the register number of the upper half word register of the base register.

```
DCL VECTOR ARRAY(100) REG 8/9;
DCL R1 REG 10;
DCL R2 REG 11;
$ R1 =U R1 + R2; full word register add upper
$ R1(A) =U R1(A) +R2; vector add upper, R2 broadcast
```


A PL/* program is constructed from basic program elements called statements. A statement is defined as:

$$\{ \text{statement_identifier} \} \quad \{ \text{statement_body} \} \ ;$$

STATEMENT TYPES

PL/* currently has four unique statement types: directives, instructions, compile-time, and macros. For directives, the statement identifier is a directive identifier and the body varies with the directive. For instructions, the statement identifier is a dollar sign (\$). For compile-time the statement identifier is a percent sign (%). All statements end with a semicolon.

SPACES

Spaces are generally ignored, but they:

1. Cannot appear in identifiers, in decimal or in hexadecimal constants.
2. Must appear between combinations of identifiers and constants if no other delimiter naturally appears between them.
3. Must follow the =Q (where Q represents an upper case letter of the alphabet) or the =+ operators.

LABELS

The general format for a label is:

$$\text{identifier: } \{ \text{identifier: } \updownarrow \}$$

A label is an identifier followed by a colon which precedes a statement. The appearance of a label defines it as an address symbol with a value from the program location counter. More than one label can precede a statement and the labels can be used interchangeably to reference the statement.

Labels are meaningful only when they precede instructions or %ORG and PROC directives. For all other directives, labels are essentially ignored.

COMMENTS

The general form of a comment is:

```
/ * {character string†} */
```

Comments are normally used for documentation and are not involved in the execution of the program. A comment can appear externally to any statement. The character string in the comment must not contain the character combination */ in sequence.

```
/ * THIS IS A COMMENT * /
```

ALIGNMENT AND BOUNDARIES

Instructions and data are aligned to appropriate boundaries. Instructions always are aligned on the next sequential half word while the alignment of data is dependent on its type. The general rule for alignment is to round up to the next boundary if not already on that boundary.

A given boundary is any address that fulfills the definition for that boundary. A full word boundary is any address whose lower six bits are zero, a half word boundary has five lower bits of zero, a byte boundary has three lower bits of zero. Therefore, a full word address or boundary is also a half word or byte address or boundary.

FORMAT

Statements are free form and can start and end anywhere on a physical record. More than one statement can appear in each physical record, and a statement can be continued from one physical record to the next.

To aid in program maintenance, the assembler assigns a sequence number to each physical record input. The assembler listing, however, has only one statement or comment per line, and it splits and aligns records as necessary.

Directives are used to control the listing, control location, control register assignment, and to generate data.

LIST CONTROL

LIST; a listing is printed. LIST is implied at the beginning of an assembly
OFF; listing is suppressed

LOCATION CONTROL

PL/* provides two location counters, one for executable code instructions called the program location counter, and one for data called the data location counter.

`%ORG=expression;`

ORG is a permanent, compile-time variable which controls the program location counter. The program originates at the bit address indicated by the expression. Any label on an ORG is given the value of the expression in the statement body.

Default origin if no ORG is given is the dynamic space pointer (DSP), for example:

```
%ORG = #8100
%ORG = [%ORG + #40];
```

`% ORGD = expression;`

ORGD controls the data location counter. Data is placed starting at the bit address specified by the expression.

ORGD is a permanent, compile-time variable; consequently, expressions like

```
%ORGD = [%ORGD + #100];
```

are allowed.

The default value is the bit address of the first full word following the instructions.

`%OBJECT` constant;

This directive is used to store the object code at a memory location other than where it is executed. The code is stored at the address specified by constant relative to the current program location counter.

```
%ORGD = #12000;
%ORG = #8000;
%OBJECT = #100000;
```

Executable code, which is assembled with respect to address #8000, is stored at #100000. The data is also offset by the same amount so data assembled with respect to address #12000 is stored at address #10A000. This directive is meaningful only when absolute code is being generated.

REGISTER COUNTERS

`PL/*` maintains two register counters, one for full word registers and one for half word registers.

`PL/*` automatically allocates registers in data generating declaratives if no register is specifically indicated. Two separate bit streams, one for full word registers and one for half word registers, provide maps of which registers are assigned and which are available. When a register is allocated or specifically assigned in a data statement, the corresponding bits in the full word stream and half word stream (if applicable) are set to indicate that these registers are in use or frozen. Therefore, if full word register 10 is allocated, bit 10 is set in the full word stream, and bit 20 and 21 are set in the half word stream. This prevents overlapping during automatic register assignment. In automatic assignment, if registers are encountered which are previously assigned, these registers are skipped and the next free register is used.

If, in using automatic assignment, a program requests more registers than are available, an error message is given when all registers are used. The assembler

starts assignment again at the value of the last applicable ORGW or ORGR, but an error message appears with every register assigned.

`%ORGR = expression;` default #40

`%ORGW = expression;` default #20

ORGR (ORGW) starts the assignment of half word (full word) registers at the value of the expression. For example:

`%ORGR = #40;`

`%ORGW = [%ORGW + 20];`

`%FULL_REG_AUGMENT = expression;` default -1

`%HALF_REG_AUGMENT = expression;` default +1

FULL_REG_AUGMENT (HALF_REG_AUGMENT) specifies the value to increment ORGW (ORGR) during full word (half word) register allocation.

REGBLOCK identifier { constant[†] };

This directive is used for storing preset register data in a memory block other than the register file. It is intended that during program execution this block is transmitted to the appropriate position in the register file. After a REGBLOCK directive, any register assigned has a word set aside for it in the block. If the register is PRESET or INIT, the preset value is stored in the appropriate place in the memory block not in the register file. The non-initialized registers are set to zero. The definition associated with the register symbol is unchanged. It is assumed a contiguous block of registers is represented.

The block starts at the address specified by the constant term, and identifier is the address symbol of the first word of the block. All REGBLOCKS, for which no memory locations are specified, are placed after the program's data and aligned to the next full word boundary.

REGBLOCK uses the current value of the full word register counter as a reference point for the registers stored. This counter must be in forward mode. If half word registers are defined in the range of a REGBLOCK, the half word counter must first be set to coordinate with the full word counter.

The block is properly set up for a LOAD macro. The length is determined after the ENDBLOCK.

```

%FULL_REG_AUGMENT = +1;
%ORGW = #20;
REGBLOCK REG_BLOCK #4000;
DCL REG1 REG ? INIT 0;
DCL REG2 REG ?/? PRESET(L=#15, B=[REG1]);
DCL VECTOR ARRAY (#100) PRESET;
ENDBLOCK REG_BLOCK;

```

Produces:

#4000	4	800
		0
	15	800
	100	[VECTOR]

```

ENDBLOCK { identifier +
           identifier FREE } ;

```

This directive terminates the effect of the preceding REGBLOCK identifier. If identifier FREE is used, all registers which were assigned under the REGBLOCK are made available for assignment (are freed). If FREE is omitted, the registers are considered assigned.

Immediately following the PROC directive a register block is automatically begun and ends with the first ENDBLOCK REGBLOCK or END directive. The register block is loaded by the prolog generated at a PROC or ENTRY directive. The first register loaded is register #20, and the length of the block is determined by the value of %ORGW when the register block is terminated. This same register block is loaded at subsequent entry points.

LOAD identifier;

This macro is necessary to load a previously defined REGBLOCK, at the location specified by identifier, to the appropriate place in the register file.

DATA GENERATING DIRECTIVES

DCL identifier attributes {, identifier attributes†};

Identifier is the symbol being defined. Attributes are combinations of the following keywords.

<u>F</u> LOAT (7)	<u>I</u> INIT	<u>R</u> EG	<u>L</u> ABEL
<u>F</u> LOAT (14)	<u>P</u> RESET	<u>A</u> RRAY	<u>E</u> QU
<u>F</u> IXED (7)		<u>S</u> PARSE	<u>M</u> EM
<u>F</u> IXED (14)		<u>C</u> HAR	<u>S</u> YNCH
		<u>B</u> YN	<u>D</u> OUBLE
		<u>D</u> EC	<u>T</u> RIPL <small>E</small>
		<u>B</u> IT	<u>E</u> XT <small>ERNAL</small>

Shortened forms of the attribute can replace the full word. The short form is underlined.

FLOAT $\left\{ \begin{array}{l} (7) \\ (14) \end{array} \right\}$

This attribute specifies that the identifier is to represent floating-point data items. The value in parentheses gives the effective number of decimal digits to be maintained in the fractional part, and therefore indicates whether a half word FLOAT(7), or a full word FLOAT(14) is being defined. In the absence of this attribute, FLOAT(14) is assumed.

```
DCL DOG  FLOAT  (7)  REG  10;
DCL CAT  FLOAT (14)  SPARSE(100);
```

FIXED $\left\{ \begin{array}{l} (7) \\ (14) \end{array} \right\}$

Same as FLOAT.

INIT

This attribute is used to initialize data into a memory or register defined field. Its use is further defined as it applies to different data types in the following attribute definitions.

PRESET $\left\{ \begin{array}{l} (ALL) \\ (L=y, B=yy, X=yyy) \end{array} \right\}^{\dagger} \quad y, yy, yyy = \text{constant}$

Preset is an attribute used to initialize full word registers and descriptors. It usually appears in a multivalued symbol definition to preset a length and base address in a register. The control vector register cannot be preset.

The form PRESET (L=y, B=yy, X=yyy) is used to specifically indicate a:

- L length value (upper 16 bits base register)
- B base address values (lower 48 bits)
- X index value (index, or offset, or second register)

Any or all of the values can be included within the parentheses. If a specific value is not mentioned, default conditions (the primary values and an index of zero) are used. Just PRESET or PRESET (ALL) use the default conditions.

If registers are to be automatically assigned, PRESET implies that a base register is assigned. Unless an index register is specifically mentioned in a PRESET (L=y, B=yy, X=yyy) or in a REG, none is assigned. A PRESET (ALL) or PRESET with no REG attribute, or with a REG attribute with no index register assigned, will assign only one register, the base address register.

```
DCL CHAR_STRING CHAR(10) PRESET;
```

One full word register is assigned containing a length of 10 and a base address of CHAR_STRING.

```
DCL VECTOR ARRAY (100) REG 5/11/12 PRESET (L=50);
```

Register 5 contains a length of 50 and a base address of VECTOR. Register 11 is initialized to zero. The contents of register 12 are undefined.

```
DCL C_VECTOR BIT(100) I 0 P (X=0);
```

Two full word registers are assigned, one containing a length of 100 and a base address of C_VECTOR, and the other register containing zero. In addition, the bit string C_VECTOR is initialized to zero.

$$\text{REG } \left\{ \begin{array}{l} n \\ n_1/n_2 \\ n_1/n_2/n_3 \end{array} \right\} \quad n = \text{constant or ? or register name}$$

The attribute REG is used to define a register symbol. If n is a constant, the register number(s) assigned is the value of that term. If n is a ?, a register is automatically assigned by the assembler. If n is a register name, it refers to its previously defined value. For example:

```
DCL X REG ?; DCL Y REG X; X & Y have the same register value.
```

A full word register is assigned unless REG is preceded by FLOAT (7).

n	defines one register, the base address register
n ₁ /n ₂	defines a register pair
n ₁	the base register
n ₂	the index register, order vector register, or offset register
n ₁ /n ₂ /n ₃	defines three registers
n ₁	the base register
n ₂	the offset register
n ₃	the control vector register

In automatic assignment, the registers n₁, n₂, and n₃ are not necessarily sequential. They are the first free registers encountered. The terms base register, index register, and control vector register are not necessarily indicative of the contents of these registers. They are just terms for referencing the first, second, and third registers defined.

An INIT of the form INIT constant can appear after the REG attribute to initialize the register.

Qualifiers are used to reference one specific register in the set of registers defined. The character < or > in parentheses give, respectively, the upper or lower half word of a given full word register.

```
DCL REG_SET ARRAY (10) REG 2/3/7;  
    REG_SET(B)      full word register 2  
    REG_SET(X)      full word register 3  
    REG_SET(B>)     half word register 5
```

Whenever a reference requiring a register is made to a symbol associated with a set of registers, an attempt is made to use all the registers possible. Given the above declaration, if REG_SET is used as follows:

```
$REG_SET = another array;
```

the base register, index, and control vector registers are all used in the instruction. To override the use of any register, a zero must be specifically stated.

```
$ REG_SET, 0 = another array;
```

In this instruction no index register is used.

```
$ 0 'ON REG_SET = another array;
```

This instruction has no control vector but the base address and index registers are used.

If the register is not applicable to the field in which it is used, it is ignored. For example, if REG_SET is used as an A field vector, the control vector is ignored because it does not apply to that particular field.

A control vector operating on ones is assumed. The register must be specifically mentioned to have it operate on zeros.

```
$ 'NOT REG_SET(Z) 'ON REG SET = another array;
```

$$\left\{ \begin{array}{l} \text{SPARSE} \\ \text{ARRAY} \end{array} \right\} \{ (\text{constant}) \dagger \};$$

This attribute is used to define and allocate memory for vector or sparse type symbols. The size of the symbol is full word unless ARRAY or SPARSE is preceded by FLOAT(7), in which case it is half word. Constant indicates the length of the symbol and the number of full words (half words) to be allocated, generally under the data location counter after alignment to a full word (half word) boundary. The value of the symbol is the bit address of the data location counter after alignment, unless another address is indicated by a MEM attribute.

If constant is omitted or equals zero, no memory is allocated and the symbol length equals zero. This is usually done just to give a symbol the appropriate properties for an instruction.

```

DCL VECTOR ARRAY(10);
DCL H_VECTOR FLOAT(7) A(100);
DCL S_VECTOR FLOAT(7) SPARSE(100);
DCL SPARSE2 SP(10) PRESET(X=0);

```

An INIT can appear after the ARRAY or SPARSE attribute to preset the vector. For example:

```
DCL B ARRAY (100) INIT 3.5; fills the entire B array with 3.5
```

```
DCL APPLE A(60) I (3,4,1.5,2); initializes the first four elements of the array
```

```
DCL B ARRAY (100) INIT (3.5); initializes only the first element of the B array with 3.5
```

CHAR { (constant)† }

This attribute is used to define and allocate memory for a character string type symbol. The size of the symbol is a byte. Constant indicates the length of the symbol and the number of bytes to be allocated, generally under the current data location counter after alignment to a byte boundary. Unless another address is stated by a MEM attribute, the value of the symbol is the bit address of the data location counter after alignment.

If constant is omitted or equals zero, no memory is allocated and the length of the symbol equals zero. This is usually done to give a symbol appropriate properties for an instruction. An exception to this is when an INIT value is given, in which case the length is implied to be the number of characters.

When using an INIT with a character string, the string is treated as a whole not as a series of bytes. If the data is shorter than the field, the data is adjusted and filled according to type of the constant.

```
DCL TEST_FIELD CHAR(5) INIT #1253149111;
```

```
DCL ERROR C(10) INIT "ERROR 5" PRESET;
```

DCL ERROR C INIT "ERROR 5" PRESET; a length of 7 is implied.

DCL CAT CHAR(5) I "HOUSE"; the ASCII characters for HOUSE are stored in the field CAT. Initialized characters are left-justified, space filled. Note that in the PL/* statement \$A=X "ABC", the characters are right-justified, zero filled.

DEC $\{(constant)\dagger\}$

This attribute is used to define and allocate memory for a right-aligned, packed decimal string type symbol, used primarily in BCD arithmetic instructions. The size of the symbol is a 4-bit unit. Constant indicates the number of 4-bit units to be allocated, generally under the control of the data location counter after alignment to a byte boundary. The constant must allow for the 4-bit sign field. The length of the symbol in bytes is constant divided by 2. If constant is an odd number, the length after division is incremented by 1. If constant is omitted or equals zero, no memory is allocated and the length of the symbol equals zero. Unless another address is specified by a MEM attribute, the value of the symbol is the bit address of the data location counter after alignment.

```
DCL DECTERM DEC(5);
```

```
DCL DEC_NUM DE(5) INIT -125;
```

The length is 3 bytes for both examples.

BYN $\{(constant)\dagger\}$

This attribute defines and allocates memory for a right-aligned binary string type symbol used primarily in byte aligned binary arithmetic instructions. The size

of the symbol is a bit. Constant indicates the number of bits to be allocated, generally under the control of the data location counter after alignment to a byte boundary. The length of the symbol is a constant divided by 8 to give length in bytes. If the constant is not a multiple of 8, the length after division is incremented by 1. If constant is omitted or equals zero, no memory is allocated and the length of the symbol equals zero.

```
DCL BIN_NUM BYN(10);
```

The length in the example is 2 bytes

```
BIT {(constant)†}
```

This attribute is used to define and allocate memory for a bit string type symbol. The size of the symbol is a bit. The length of the symbol and the number of bits allocated in memory is given by constant. Memory is generally allocated under the current data location counter. No alignment is necessary. Unless another address is stated by an MEM attribute, the value of the symbol is the bit address of the data location counter.

If constant is omitted or equals zero, no memory is allocated and the length of the symbol equals zero.

```
DCL CONTROL_VECTOR BIT (100);
```

LABEL

This attribute follows a register assignment, and indicates that the register symbol defined will appear later as a label on an instruction. This allows the register containing a branch address and the branch address to use the same symbol. The register assigned is preset with the bit address of the symbol as a label. LABEL automatically assigns a register if none is specified.

```
DCL JUMP2 LABEL;  
DCL A_JUMP REG #F LABEL;
```

In the first example, the register is automatically assigned. The register is pre-set with a length of zero and a base address of JUMP2 as a label.

EQU n n = any constant or symbol

EQU is an attribute used to equate a symbol to a constant. If the term following the EQU is other than a constant, the primary value of the term is handled as if it were a constant. No properties other than value are carried over by the EQU. No other attributes can appear with an EQU.

```
DCL DOG EQU #15;  
DCL CAT EQU DOG;
```

If DOG is full word register 5, CAT would have a value 5 but would have a type of constant not register. The statement DCL CAT REG DOG; would give a register type to CAT.

MEM constant

This attribute is used to assign a specific memory location value to a symbol. If MEM is used in conjunction with a data defining attribute, storage is also allocated at the address given. The MEM attribute must appear before the data defining attribute. If RELOC is specified, the value is relocatable only if it falls within the range of the initial and final values for %ORG or %ORGD.

```
DCL STRING1 MEM #41000 ARRAY(100) PRESET;
```

```
DCL STRING2 M [VECTOR] CHAR(800) REG 5/11 PRESET;
```

```
SYNCH { P  
       F  
       H  
       C  
       S }
```

This attribute is used to align a data field to other than its implied boundary; for example, to force a vector of half words to start on a full word boundary. SYNCH must appear before the data defining noun and rounds up to the appropriate boundary.

P align to page
F align to full word
H align to half word
C align to byte
S align to sword

DOUBLE

DOUBLE is an attribute used in lieu of the term REG ?/? and assures automatic assignment of an even/odd register pair. FLOAT(7) can precede DOUBLE if a half word even/odd pair is desired.

```
DCL FINAL_AVG_DIFF FLOAT(7) DOUBLE;
```

```
DCL ARRAY_RESULTS ARRAY(100) DOUBLE PRESET;
```

TRIPLE

TRIPLE is an attribute used in lieu of the term REG ?/?/? and assures automatic assignment of an even/odd register pair for A, C, C+1 (base, offset, and control vector register).

```
DCL RESULT_VECTOR A(100) I 0 TRIPLE PRESET;
```

```
DCL VECTOR A(10) T;
```

EXTERNAL

This attribute specifies that the identifier is used here but defined in some other program as a label on a PROC or ENTRY. LABEL must appear as one of the attributes. EXTERNAL can also be a separate statement.

```
DCL    GRASSLAND LABEL EXTERNAL;
```

```
DDCL attributes { , attributes† } ;
```

The dummy declaration is used to generate data without having to specify an identifier.

```
DDCL C(7) INIT "ERROR 1";
```

```
DDCL ARRAY(100) INIT 0;
```

```
DESCR identifier { (L=y, B=yy) }  
                symbol  
                y, yy = constant
```

This directive is used to initialize full words at the current ORGD aligned to a full word boundary. Identifier is an address symbol.

L length (upper 16 bits of word)

B base address (lower 48 bits of word)

If the symbol form is used, the length and the value for the symbol is used in the descriptor word.

NOTE: The qualifier L in parenthesis after a symbol gives the length of that symbol.

```
DESCR VECTOR_DES (L=500, B= [VECTOR]);
```

```
DESCR BIT_DES CONTROL_VECTOR;
```

DDESCR $\left\{ \begin{array}{l} (L=y, B=yy) \\ \text{symbol} \end{array} \right\}$

The dummy description is used to initialize full words at the current ORGD aligned to a full word boundary without having to specify an identifier.

DDESCR (L=3, B=[START]);

DDESCR TABLE;

STABLE identifier $\left\{ \text{attributes} \right\} \left\{ \text{modifiers} \right\} : \left\{ \text{char} = \text{constant} \right\}$;

This directive was initially designed to create a syntax cracking table for the translate instructions.

This directive allocates and initializes a 64-byte table corresponding to the 64 characters in the ASCII subset. Identifier is the address symbol for referencing the table. The value of the symbol is 32 bytes less than the start of the table because the ASCII codes are used as byte indices to the table. Care must be used that the set referencing the bytes in the table lie in the range #20 to #5F.

Attributes are used as in the DCL statement. The allowed ones are REG, PRESET, MEM, SYNCH, EXTERNAL.

Either FILL or BASED can be used as modifiers.

FILL $\left\{ \begin{array}{l} \text{constant} \\ \left(\left\{ \left\{ \pm \right\} m \right\} ! \left\{ \left\{ \pm \right\} n \right\} \right) \end{array} \right\}$

This modifier determines the fill character to be used in every byte except those specifically mentioned in the latter part of the STABLE statement. If FILL is not specified, 0 (zero) is used as the fill value. The option which uses the exclamation point (!) specifies a function to be evaluated at each fill position. Take exclamation point to represent the current byte position in the table (e.g., the 32nd position is a space). Then m is a multiplicative constant for the position, and n is added or subtracted to form the final value.

BASED constant

This modifier gives a base address when STABLE is used to create a relative jump table. The contents of every byte in the table is interpreted as an index of this address. For example:

```
STABLE GNU R ? FILL (-! +255) :^#^= 2;  
STABLE HORSE BASED [START/32] FILL 7 :^#3B^= #FF ^$^=  
[ENTRI/8];
```

The expression to the right of the colon is used to fill specific bytes in the table.

char – some ASCII character or its numeric value specifying the byte to be filled

constant – the value to be placed in the specified byte

For example:

```
^A^= 5
```

This expression is interpreted as:

in the character position A, #41 bytes from identifier, #21 bytes from the start of the table, put the constant #05.

If the constant is an address expression, this address is subtracted from the base address given and the difference in bytes stored in that character position.

Because of the characteristics of the syntax cracker in PL/*, the character ; (semicolon) cannot appear as a char in the byte fill field. Rather than specify the ASCII character, the hexadecimal constant for the ASCII code can be used. This alternate form is called the hex escape. In STABLE the range for the hex escape is #20 to #5F. For example:

```
^#3B^ =1
```

This expression uses #3B instead of a semicolon, and is interpreted as:

in the #3B character position, $3B_{16}$ bytes from identifier, $1B_{16}$ bytes from the start of the table, put the constant 01_{16} .

STABLE TRANS_TABLE FILL #11 :^,^ =0 ^#3B ^ =4 ^ ^ =2;

The following table is produced:

TRANS_TABLE+32 bytes

02	11	11	11	11	11	11	11
11	11	11	11	00	11	11	11
11	11	11	11	11	11	11	11
11	11	11	04	11	11	11	11
11	11	11	11	11	11	11	11
11	11	11	11	11	11	11	11
11	11	11	11	11	11	11	11
11	11	11	11	11	11	11	11

TABLE identifier {attributes†} {modifiers†} :{ ^ char ^ =constant† } ;

This directive is similar to STABLE. The only differences are that TABLE is 256 bytes long and identifier is not displaced but is the bit address of the first byte in the table.

The following sections are miscellaneous comments on program structure and use of symbols.

PROGRAM STRUCTURES

The usual sequence for a PL/* program is:

1. Labeled PROC directive
2. Counter orgs, data declarations, and symbol definitions
3. Executable code
4. RETURN directive returns control to system
5. END directive

All register names and data address symbols must be defined before use. Forward references to labels are allowed because PL/* is a 1-pass assembler and forward references to labels are linked after assembly; the object code on the listing contains zero. Instructions and directives can be mixed.

REGISTER PAIRS

Register pairs are two registers which are related by being a base address-index, base address-offset, base address-order vector, or even/odd pair. Register pairs can be defined by REG n/n, DOUBLE, TRIPLE or PRESET attributes, where one symbol represents the pair. They can be represented in an instruction by using that symbol or by two register names separated by commas. The base address register or even register appears first.

The assembler attempts to use as many registers defined with a symbol as possible. Therefore, if a symbol represents a register pair, both registers are used if the instruction where the symbol is used allows a pair. A qualifier or a specifically stated pair can be used to override the pair.


```

DCL R1 REG 10/11;
DCL R2 REG 13, R3 REG 14;
$ R3 =: R1;           R1 represents both 10 and 11
$ R3 =: R1(B);       R1 expression = register 10
$ R3 =: R1, 0;       R1 pair represents 10 with no index
$ R3 =: R1, R2       R1 pair uses base register 10, index register 13

```

If any qualifier is used with a register pair, it must follow the first register name.

VECTOR INSTRUCTIONS

CONTROL VECTOR

{'NOT†}register_name 'ON symbol

A vector or vector macro instruction can have a control vector specified in the arrayC field.

A control vector operating on ones uses the following format:

```
BitZ 'ON arrayC
```

where bitZ is the control vector register.

A control vector operating on zeros uses the following format:

```
'NOT bitZ 'ON arrayC
```

If a control vector was included in a DCL defining a symbol and the symbol is used in an arrayC field which allows a control vector, the control vector operates on ones. If it is desired to have the control vector operate on zeros, it must be specifically mentioned.

```
'NOT VECTOR    'ON VECTOR
```

To get a different control vector or no control vector, the control vector can be overridden.

```
0 'ON VECTOR  
C_V_3 'ON VECTOR
```

BROADCAST REGISTER

To use a broadcast register in a vector instruction, the term to be broadcast must be either defined with a primary type of register or forced to be a register name using qualifiers. In the following examples, VECTOR1 and VECTOR2 are arrays with secondary register pairs.

```
$VECTOR1 =U VECTOR1+VECTOR2;
```

The above is an add upper of two vectors.

```
$VECTOR1 =U VECTOR1(X) +VECTOR2;
```

The above is a vector add upper with the index register of VECTOR1 broadcast over the vector VECTOR2.

Compile-time statements produce no object code but are available to the user as another tool for assembly control. For reference purposes, all statements (symbols) which are not compile-time statements (symbols) are called assembly-time statements (symbols). The primary difference between the two classes is the value of a compile-time symbol can be changed during assembly, while the value of an assembly-time symbol is fixed once the symbol is defined.

Compile-time symbols can be used in either compile-time statements or assembly-time statements. Compile-time statements are preceded by a percent sign (%) and terminate with semicolon (;). One form of a compile-time statement is a simple replacement statement.

$$\%A = [\%A + 5];$$

Assembly-time symbols in the form of an address expression can also appear to the right of the equals sign in a compile-time statement.

$$\begin{aligned} \%A &= [\%A + LABEL]; \\ \%A &= [VECTOR (L)]; \end{aligned}$$

Compile-time symbols can be used in assembly-time statements in place of an assembly-time symbol. Each compile-time symbol must be preceded by a %. The value of the compile-time symbol is used as if it were a constant, and appropriate attributes are assumed according to the instruction.

The PL/* assembler has six predefined compile-time symbols %FULL_REG_AUGMENT, %HALF_REG_AUGMENT, %ORG, %ORGD, %ORGR, and %ORGW which can be referenced and manipulated like any compile-time symbols.

COMPILE-TIME REPLACEMENT STATEMENTS

`%compile_time_symbol = compile_time_expression;`

A `compile_time_expression` has the form:

$$\left[\left\{ \begin{array}{c} +^{\dagger} \\ - \end{array} \right\} \left\{ \begin{array}{c} \text{constant} \\ \text{compile_time_symbol} \\ \text{object_time_expression} \end{array} \right\} \left\{ \text{operator} \right\} \left\{ \begin{array}{c} \text{constant} \\ \text{compile_time_symbol} \\ \text{object_time_expression} \end{array} \right\} \left\{ \begin{array}{c} \dagger \\ \dagger \int \end{array} \right\} \right]$$

The operators are:

- + plus
- minus
- / divide
- * multiply
- ** exponentiation

Parentheses cannot be used for grouping, and evaluation precedes from left to right. Note that a compile-time statement which starts with a compile-time symbol needs only one leading % sign.

CONDITIONALS

$$\%IF (\text{term} \left\{ \begin{array}{c} .EQ. \\ .NE. \\ .GT. \\ .GE. \\ .LT. \\ .LE. \end{array} \right\} \text{term}) \text{statement};$$

This statement is used to conditionally alter the sequence in which statements are processed in the assembler. If the logical expression enclosed in parentheses is true, then the remainder of that conditional statement is used. If the expression is false, the next sequential statement is processed.

Term can be either a `compile_time_symbol` or an assembly-time expression.
 Statement can be any statement except another `%IF`.

`%GO TO compile_time_label;`

This statement provides a jump to a `compile_time_label`. A `compile_time_label` can precede any statement but cannot be referenced in other than compiler-time statements. Both a forward or a backward jump is allowed.

A `compile_time_label` has the form:

`% identifier:`

`%TYPE identifier;`

`%TYPE` takes on a value indicating identifier type. The identifier type is determined in a declarative statement and is found in the leftmost four character positions of the instruction field in an assembly listing.

	<u>Flag bit indicator</u>	<u>Flag bit value</u>
	FIXED	8
	$\left. \begin{matrix} \text{see} \\ \text{left} \end{matrix} \right\}^*$	4 2
*If Arithmetic	multi-dimension	1
00 - default	-	8
01 - normalize/significant		
10 - upper	common (static)	4
11 - lower	DEC	2
	SPARSE	1
*If Character String	BIT	8
00 - count delim.	undefined	4
01 - mask	constant	2
10 - char	LABEL	1
11 - double	ARRAY	8
	CHAR	(right- most digit) 4
	REG	2
	half word	1

Example:

2 Full word register name
3 Half word register name
4 Character string
A Full word array with length/base register
B Half word array with length/base register
8 2 Bit string with length/base register
F F F F Undefined (first occurrence of identifier)

PROC { identifier † } { (parameter { , parameter † }) † } ;

This directive identifies the beginning of a procedure. PROC must be preceded by one or more labels limited to eight characters each. The last label becomes the module name. All of the labels are entry point names defined as an address symbol with values of the current address of the program location counter. If identifier is used, it is the name of the REGBLOCK loaded by the prolog generated. Also generated is the code necessary to stack the caller's registers.

Each of the parameters is assigned to an odd register beginning with register #FD, working toward register zero. This implies a default attribute of value. If it is desired that the parameter be a descriptor and therefore assigned to the appropriate odd register (even register minus one) the parameter must be given new attributes in a declaration statement.

```
A:PROC(A);  
    DCL A CHAR;
```

If an attribute of register appears in the redefinition of a parameter, a register transmit is generated to move the parameter from the parameter registers to the new register. The opposite transmit is generated at all subsequent RETURN macros.

If it is desired not to have the parameters returned at the time of RETURN, an asterisk (*) catenated to the front of the parameter in the PROC statement will suppress it.

```
A:PROC(A,*B);  
    DCL A CHAR;  
    DCL B A P;
```

label: ENTRY { (parameter { , parameter † ∫ }) † } ;

This directive identifies an entry point to the module. The necessary code for stacking the caller's registers and loading the initial register block is generated. ENTRY must be preceded by one or more labels limited to eight characters each. These labels become entry point names with a value of the current address of the program location counter.

The parameters are assigned in the same way as for PROC. If one of the parameters has the same name as one which appeared in the PROC, it must appear in the same position. For example,

```
A:PROC(A,B);  
.  
.  
.  
B:ENTRY(ALPHA,B);
```

If the redefinition of a parameter occurs which results in the generation of one or more register transmits, the same register transmits will occur at subsequent entry points if the parameter is listed.

```
T:PROC(A);  
    DCL A R ?  
  
S:ENTRY(A);  
  
U:ENTRY(B);
```

EXTERNAL identifier { , identifier † ∫ } ;

The EXTERNAL macro identifies linkage symbols that are used by a program but are defined in some other program. Identifier must appear as a label on a PROC or in an ENTRY. This macro allocates two registers per identifier and must therefore appear as the first statement in the initial register block.

CALL identifier $\{(\text{parameter } \{, \text{parameter} \int \}) \dagger\}$;

This macro creates a standard calling sequence for library routines and subprograms. Identifier is the entry name of a subprogram. Entry names are generated in a subprogram by PROC and ENTRY. The parameters can be either register names, array names, or constants, and it is assumed that each parameter represents two registers that correspond to the parameters in the appropriate PROC or ENTRY. The parameters are loaded sequentially starting with register #FD.

If the parameter is a descriptor it is loaded into the even register; if it is a value, it is loaded into the odd register, and if it is a register pair it is loaded into the even/odd pair respectively. If the parameter register to be loaded is the same as the register already containing the parameter, the load does not take place.

Upon return from the callee, the parameters are restored to where they were loaded from unless suppressed by a catenated asterisk (*) as in the PROC and ENTRY;

```
Example:          PROC (P1, P2, P3, P4);
                  DCL A REGP;
                  DCL B ARRAY PRESET;
                  DCL C R ? / ?;
                  DCL D ARRAY;

                  CALL SUBROUTINE (A, *B, C, P4, 3, D);
```

RETURN;

This macro generates a return to the program which called the subprogram in which the RETURN is found. First the status of the register file is restored, and then control is transferred to the return location.

If there were any parameter redefinitions resulting in the generation of register transmits, the parameters are restored just prior to the RETURN. For examples see PROC and ENTRY.

STORE identifier;

This macro stores the initial regblock, generated immediately following the PROC directive, into static space where it was loaded from by the prolog. Identifier is a register name containing the contents of the LINK register immediately after the prolog was executed.

```
A:PROC;
    DCL SIN LABEL EXT;
    DCL A R ? INIT 3.1417;
    DCL B R ?;
    $B= LINK;
    .
    .
    .
    CALL SIN(A),
    .
    .
    .
    SAVE B;
    .
    .
    .
```

If the link register is not destroyed by a CALL macro, it can appear as the identifier.

```
B:PROC;
    .
    .
    .
    DCL A A P;
    DCL B A P;
    .
    .
    .
    SAVE LINK;
    .
    .
    .
```

label: FORMAT (option {, option† ∫ });

This macro is comparable to the FORTRAN FORMAT. Options implemented are F, I, E, A, G, R, H, X, /, *Hollerith value*, and # (that is, hexadecimal values). Parentheses can be used for repeat groups. The first print position is not used for carriage control.

IOLIST list_name identifier {, identifier† ∫ };

List_name is used to name the collection of identifiers which follows. Identifier is a symbol or array to be printed. An array must be named with subscripts and is therefore printed in its entirety.

IOLIST and FORMAT are used in conjunction with a CALL to INPUTC or OUTPUTC.

CALL INPUTC (format_address, register_pair, list_name);

This macro is comparable to the FORTRAN READ (ENCODE) and is used to read coded input. The format_address is the label on a FORMAT macro; register_pair is the name of two registers holding the base address and index of the input area.

CALL OUTPUTC (format_address, register_pair, list_name);

This macro is comparable to the FORTRAN WRITE (DECODE) and is used to write coded output. The format_address is the label on a FORMAT macro, register_pair is the name of two registers holding the base address and index of the output area, and list_name is the first operand of the IOLIST macro. After execution the index is updated and the length set to #1F1C, but the base address remains the same. The list_name can be zero if the associated FORMAT has only Hollerith information. Compressed output is generated except when the output area address is below #8000. Also, for output area addresses below #8000, only one line as terminated by / or) is produced, and the final #1F (end-of-record) does not appear.

```

DCL H A(3) PRESET;DCL G A(5) P;
DCL I R ?, J R ?, K R ?;
DCL STORE REG ?/? P (L=0, B=20000000, X=0);
IOLIST ZARATHUSTRA G, H, I, J, K;
FORMX: FORMAT (3F12.0, /, 5E20.8, 11X 6HREGS =, 3I4);
/*NOTE THAT BOTH IOLIST AND FORMAT MUST APPEAR BEFORE
THE CALL TO OUTPUTC (OR INPUTC)*/
CALL OUTPUTC (FORMX, STORE, ZARATHUSTRA);

```

```

MESSAGE { [register-name] } ;
         [register-name]

```

This macro is used to communicate with the monitor with the exit force instruction. If the form MESSAGE [register-name] is used, the following code is generated at the program location counter:

```

09000000
00FF0000
0000XXXX

```

where XXXX is the bit address of the register-name. At this bit address in the register file is found a message to the monitor. If the form MESSAGE register-name is used, the following code is generated:

```

09000000
00EE00RR

```

where RR is the register number associated with register-name, the contents of register RR point to a message located in virtual memory.

```

RELOCOFF;
RELOCON;
PROLOGOFF;
PROLOGON;

```

By default the code generated by PL/STAR is relocatable. The exact format of an assembled module can be found in section 7. If in the calling sequence non-relocatable code is requested, the code, data and reg blocks are positioned by the user with the compile time code.

The use of RELOCOFF causes the symbols to follow which would normally be flagged as relocatable to be ignored in the relocation table. RELOCON reinitiates the process. For example, if a program were to be assembled in relocatable form the following

```
      .  
      .  
      .  
      DCL A_VECTOR ARRAY PRESET;  
      RELOCOFF;  
      DCL B_VECTOR ARRAY PRESET;  
      RELOCON;  
  
      $A_VECTOR(B)=X [ B_VECTOR];  
      RELOCOFF;  
      $A_VECTOR(B)=X [ B_VECTOR];  
      RELOCON;  
      .  
      .  
      .
```

would cause the contents of the register containing the present values for B_VECTOR not to be included in the relocation table. Likewise the second enter immediate instruction would not be relocatable. RELOCON cannot be used if absolute code is being generated.

The use of PROLOGOFF before a PROC or ENTRY directive inhibits the generation of the prolog necessary for stacking the register file and loading of a reg block. PROLOGON reinitiates the process. The use of PROLOGOFF might be used if a program runs strictly out of the temporary registers and does not need to stack the registers.

Any code or data which does not appear within the bounds of the beginning and ending ORG and ORGD counter is not flagged as being relocatable. For example, if the beginning and ending values were #8000 and #100000, and #10000, #200000 and the following code appeared in the source

```
      %SAVE_ORGD=% ORGD;  
      %ORGD=#37000000;  
      DCL VECTOR A P;  
      %ORGD=% SAVE_ORGD;
```

the preset base address for VECTOR, #37000000, would not be relocatable.

CALLING SEQUENCE

PLSTAR (source_code, binary_output, listing, errors, abs_code, x_ref);

where source_code is a virtual address or file name,

binary_output is a virtual address or file name,

listing is a virtual address or file name,

errors is a virtual address or file name, and

abs_code is nonzero or zero denoting a request for nonrelocatable or
relocatable code respectively.

x_ref is a nonzero or zero denoting a request for a cross reference listing
of all symbols or not

The default conditions are: #10000000
Dynamic space
#20000000
#8000000
0
0

GENERAL DEFINITION

Some rules that must be considered when using this instruction set:

1. The two columns of numbers preceding the instruction model are the op-code and format type, respectively. Some instructions have an alternate form.
2. All upper case letters and delimiters except those described in paragraphs 3-6 following are part of the instruction and must appear as such.
3. All lower case terms are used to define a general class and are defined in 6. The upper case letter(s) following indicate the field in the instruction format to which the term applies.
4. The letter Q is not part of the syntax but denotes options in the syntax. The options are listed just before the sequence of instructions where the Q appears.
5. The user of brackets { } is not part of the syntax but denotes an option in the syntax.
6. Definition of general terms:
 - array: A symbol or combination of symbols describing a full or half word vector. The arrayC term can also include control vector notation. The qualifier (A) forces a full word vector.
 - bit: A symbol or register pair describing a bit string. The qualifier (Y) forces a bit string.
 - byn: A symbol or register pair describing a byte-aligned binary string. The qualifier (Y) forces a byte-aligned binary string.
 - char: A symbol or register pair describing a character string. A qualifier (C) forces a character string.
 - con: A constant or symbolic constant or expression, optionally preceded by a sign.
 - dec: A symbol or register pair describing a packed decimal string. The qualifier (D) forces a decimal string.

- full: A full word register name
- half: A half word register name
- label: A full word register name which contains a jump address
- label_pair: A full word register name or a full word register pair which contain a branch or base address and an index, respectively.
- sparse: A symbol or register pair describing a full word or half word, sparse vector. The qualifier (S) forces a sparse array. One of the sparse operands must either be defined as sparse or have a (S) qualifier or the vector form of the instruction is assembled.
- double: A symbol or register pair describing a full word or half word even-odd register pair. (Some may have a control vector format { 'NOT† } control_vector 'ON double.)

INSTRUCTION FORMATS

00	4	\$'IDLE;	Idle
04	4	\$'BKPT fullR;	Set breakpoint from R
05	4	\$'ALGO;	Execute algorithm (EM-1 only)
08	4	\$'CHAN conR;	Set channel flag from R
09	4	\$'EXIT;	Exit force (user mode to monitor)
09	4	\$'EXIT fullT, fullS;	Exit force (monitor to user mode)
0A	4	\$'MCLOCK= fullR;	Transmit R to monitor clock
0C	4	\$'STAR;	Store associative registers
0D	4	\$'LDAR;	Load associative registers
0E	4	\$fullT= 'CHAN:fullR;	Translate external interrupt
0F	4	\$fullT= fullS 'KEYS=fullR;	Load keys from R, translate S to T
10	A	\$fullT=B fullR;	Convert BCD to binary, fixed length
11	A	\$fullT=D fullR;	Convert binary to BCD, fixed length
12	7	\$fullT=G charR;	Load byte T per S, R
13	7	\$charR=P fullT;	Store byte T per S, R
14	7	\$bitT= bitS! bitR;	Bit compress
15	7	\$bitT= \ bitR. bitS\;	Bit merge
16	7	\$bitT= ! bitR. bitS!;	Bit mask

17	7	\$charT=\ charR.charS\ ;	Character string merge
18	7	\$>>charT,fullR.fullS;	Move bytes right (R)+(T) to (R)+(S)+(T)
19	7	\$'REV charT? 'NE conR;	Scan right for not equal byte
1A	7	\$charT=!conR;	Fill field with byte R
1B	7	\$charT=!fullR;	Fill field with byte from reg R
1C	7	\$fullT=Z fullR.fullS;	Form suffix vector, leading zeros
1D	7	\$fullT=O fullR.fullS;	Form prefix vector, leading ones
1E	7	\$fullT=Z bitR;	Maximum prefix function, count leading equals
1F	7	\$fullT=O bitR;	Bit string sum reduction, count ones
20	8	\$halfR'EQ halfS:fullT;	Branch if R equals S
21	8	\$halfR'NE halfS:fullT;	Branch if R not equal S
22	8	\$halfR'GE halfS:fullT;	Branch if R greater or equal S
23	8	\$halfR'LT halfS:fullT;	Branch if R less than S
22	8	\$halfS'LE halfR:fullT;	Branch if R greater or equal S
23	8	\$halfS'GT halfR:fullT;	Branch if R less than S
24	8	\$fullR'EQ fullS:fullT;	Branch if R equal S
25	8	\$fullR'NE fullS:fullT;	Branch if R not equal S
26	8	\$fullR'GE fullS:fullT;	Branch if R greater or equal to S
27	8	\$fullR'LT fullS:fullT;	Branch if R less than S
26	8	\$fullS'LE fullR:fullT;	Branch if R greater or equal to S
27	8	\$fullS'GT fullR:fullT;	Branch if R less than S
28	7	\$charT? 'EQ conR;	Scan equal to byte R
29	7	\$charT? 'NE conR;	Scan unequal to byte R
2A	6	\$fullR=E conT;	Enter length of R with I (16)
2B	4	\$fullT=E fullR+fullS;	Add to length field
31	7	\$fullR+1:label_pair;	Increase R and branch if R not equal 0

Both Bit Branch and Alter and Data Flag Branch instructions have relative branch capability, limited to plus or minus 256 half words. In place of the fullT field one can put an address constant or a relative count. The form for the relative count is n/F for forward jump and n/B for backward jump, where n is the number of half words.

3200	9	#{fullS/N }:{fullT/N };	Bit branch and alter(no JMP, no ALT)
3210	9	#{fullS/T }:{fullT/N };	Bit branch and alter(no JMP, toggle)
3220	9	#{fullS/S }:{fullT/N };	Bit branch and alter(no jump, set)

3230	9	$\$(fullS/R):\{fullT/N\};$	Bit branch and alter(no jump, reset)
3240	9	$\$(fullS/N):\{fullT\};$	Bit branch and alter (jump, no alter)
3250	9	$\$(fullS/T):\{fullT\};$	Bit branch and alter (jump, toggle)
3260	9	$\$(fullS/S):\{fullT\};$	Bit branch and alter(jump, set)
3270	9	$\$(fullS/R):\{fullT\};$	Bit branch and alter(jump, reset)
3230	9	$\$(fullS/N):\{fullT/O\};$	Bit branch and alter(JMP one, no alt)
3290	9	$\$(fullS/T):\{fullT/O\};$	Bit branch and alter(JMP one, toggle)
32A0	9	$\$(fullS/S):\{fullT/O\};$	Bit branch and alter(JMP one, set)
32B0	9	$\$(fullS/R):\{fullT/O\};$	Bit branch and alter(JMP one, reset)
32C0	9	$\$(fullS/N):\{fullT/Z\};$	Bit branch and alter(JMP zero, no alt)
32D0	9	$\$(fullS/T):\{fullT/Z\};$	Bit branch and alter(JMP zero, toggle)
32E0	9	$\$(fullS/S):\{fullT/Z\};$	Bit branch and alter(JMP zero, set)
32F0	9	$\$(fullS/R):\{fullT/Z\};$	Bit branch and alter(JMP zero, reset)
3300	B	$\$(conS/N):\{fullT/N\};$	DF branch and alter(no JMP, no alter)
3310	B	$\$(conS/T):\{fullT/N\};$	DF branch and alter(no JMP, toggle)
3320	B	$\$(conS/S):\{fullT/N\};$	DF branch and alter(no JMP, set)
3330	B	$\$(conS/R):\{fullT/N\};$	DF branch and alter(no JMP, reset)
3340	B	$\$(conS/N):\{fullT\};$	DF branch and alter(jump, no alter)
3350	B	$\$(conS/T):\{fullT\};$	DF branch and alter(jump, toggle)
3360	B	$\$(conS/S):\{fullT\};$	DF branch and alter(jump, set)
3370	B	$\$(conS/R):\{fullT\};$	DF branch and alter(jump, reset)
3380	B	$\$(conS/N):\{fullT/O\};$	DF branch and alter(jump one, no ALT)
3390	B	$\$(conS/T):\{fullT/O\};$	DF branch and alter(jump one, toggle)
33A0	B	$\$(conS/S):\{fullT/O\};$	DF branch and alter(jump one, set)
33B0	B	$\$(conS/R):\{fullT/O\};$	DF branch and alter(jump one, reset)
33C0	B	$\$(conS/N):\{fullT/Z\};$	DF branch and alter(jump zero, no ALT)
33D0	B	$\$(conS/T):\{fullT/Z\};$	DF branch and alter(jump zero, toggle)
33E0	B	$\$(conS/S):\{fullT/Z\};$	DF branch and alter(jump zero, set)
33F0	B	$\$(conS/R):\{fullT/Z\};$	DF branch and alter(jump zero, reset)
35	7	$\$fullR-1:label_pair;$	Decrease R and branch if R not equal 0
36	7	$\$fullR=:label_pair;$	Branch and set R to next instruction
38	A	$\$fullT=E fullR;$	Transmit R(0-15) to R(0-15)
39	A	$\$fullT= 'RCLOCK;$	Transmit real-time clock to T
3A	A	$\$fullT= 'JCLOCK=fullR;$	Transmit R to job clock
3B	A	$\$fullT= 'DFR = fullR;$	Data flag register load and store
3C	4	$\$halfT=X halfR*halfS;$	Half word index multiply R*S to T

3D	4	$\$fullT=X fullR *fullS;$	Index multiply R*S to T
3E	6	$\$fullR=X conI;$	Enter R with I (16)
3F	6	$\$fullR=+ conI;$	Increase R by I (16)
40	4	$\$halfT=U halfR +halfS;$	Add upper R+S to T
41	4	$\$halfT=L halfR +halfS;$	Add lower R+S to T
42	4	$\$halfT=N halfR +halfS;$	Add normalized R+S to T
44	4	$\$halfT=U halfR - halfS;$	Subtract upper R-S to T
45	4	$\$halfT=L halfR -halfS;$	Subtract lower R-S to T
46	4	$\$halfT=N halfR -halfS;$	Subtract normalized R-S to T
48	4	$\$halfT=U halfR *halfS;$	Multiply upper R*S to T
49	4	$\$halfT=L halfR *halfS;$	Multiply lower R*S to T
4B	4	$\$halfT=S half R *halfS;$	Multiply significance R*S to T
4C	4	$\$halfT=U halfR /halfS;$	Divide upper R/S to T
4D	6	$\$halfR=X conI;$	Half word enter R with I (16)
4E	6	$\$halfR=+ conI;$	Half word increase R with I (16)
4F	4	$\$halfT=S half R /halfS;$	Divide significance R/S to T
50	A	$\$halfT=T half R;$	Truncate R to T
51	A	$\$halfT=F halfR;$	Floor R to T
52	A	$\$halfT=C halfR;$	Ceiling R to T
53	A	$\$halfT=S halfR;$	Significance square root of R to T
54	4	$\$halfT=S halfS .halfR;$	Adjust significance of R per S to T
55	4	$\$halfT=E halfS .halfR;$	Adjust exponent of R per S to T
58	A	$\$halfT=halfR;$	Transmit R to T
59	A	$\$halfT=A halfR;$	Absolute R to T
5A	A	$\$halfT=U .halfR;$	Exponent of R to T
5B	4	$\$halfT=U halfS .halfR;$	Pack R, S to T
5C	A	$\$fullT=> halfR;$	Extend R(32) to T(64)
5D	A	$\$fullT=X halfR;$	Index extend R(32) to T(64)
5E	7	$\$halfT=G label_pair;$	Load T per S, R
5F	7	$\$label_pair=P halfT;$	Store T per S, R
60	4	$\$fullT=U fullR +fullS;$	Add upper R+S to T
61	4	$\$fullT=L fullR +fullS;$	Add lower R+S to T
62	4	$\$fullT=N fullR +fullS;$	Add normalized R+S to T
63	4	$\$fullT=X fullR +fullS;$	Add address R+S to T

64	4	\$fullT=U fullR -fullS;	Subtract Upper R-S to T
65	4	\$fullT=L fullR -fullS;	Subtract lower R-S to T
66	4	\$fullT=N fullR -fullS;	Subtract normalized R-S to T
67	4	\$fullT=X fullR -fullS;	Subtract address R-S to T
68	4	\$fullT=U fullR*fullS;	Multiply upper R*S to T
69	4	\$fullT=L fullR*fullS;	Multiply lower R*S to T
6B	4	\$fullT=S fullR*fullS;	Multiply significance R*S to T
6C	4	\$fullT=U fullR/fullS;	Divide R/S to T
6F	4	\$fullT=S fullR/fullS;	Divide significance R/S to T
70	A	\$fullT=T fullR;	Truncate R to T
71	A	\$fullT=F fullR;	Floor R to T
72	A	\$fullT=C fullR;	Ceiling R to T
73	A	\$fullT=S fullR;	Significance square root of R to T
74	4	\$fullT=S fullS. fullR;	Adjust significance or R per S to T
75	4	\$fullT=E fullS. fullR;	Adjust exponent of R per S to T
76	A	\$halfT=< fullR;	Contract R(64) to T(32)
77	A	\$halfT=R fullR;	Rounded contract R(64) to T(32)
78	A	\$fullT= fullR;	Transmit R to T
79	A	\$fullT=A fullR;	Absolute R to T
7A	A	\$fullT=U .fullR;	Exponent of R to T
7B	4	\$fullT=U fullS. fullR;	Pack R,S to T
7C	A	\$fullT=E .fullR;	Length of R to T
7E	7	\$fullT=G label_pair;	Load T per S,R
7F	7	\$label_pair=P fullT;	Store T per S,R

SIGN CONTROL capability for vector instructions, 80 through 8F (except 83 and 87), and for all sparse vector instructions, A0 through AF:

Q₁ - 'NOT, 'ABS, -, omitted

Q₂ - 'ABS, omitted

The above are sign control options for that vector:

omitted use operands in normal manner
 'ABS use magnitude of operands
 'NOT use complement of operands
 - (minus) make all operands negative before use

80	1	\$arrayC=U Q ₁ arrayA+ Q ₂ arrayB;	Vector add upper A+B to C
81	1	\$arrayC=L Q ₁ arrayA+ Q ₂ arrayB;	Vector add lower A+B to C
82	1	\$arrayC=N Q ₁ arrayA+ Q ₂ arrayB;	Vector add normalized A+B to C
83	1	\$arrayC=X arrayA+ arrayB;	Vector add address A+B to C
84	1	\$arrayC=U Q ₁ arrayA- Q ₂ arrayB;	Vector subtract upper A-B to C
85	1	\$arrayC=L Q ₁ arrayA- Q ₂ arrayB;	Vector subtract lower A-B to C
86	1	\$arrayC=N Q ₁ arrayA- Q ₂ arrayB;	Vector subtract normalized A-B to C
87	1	\$arrayC=X arrayA- arrayB;	Vector subtract address A-B to C
88	1	\$arrayC=U Q ₁ arrayA* Q ₂ arrayB;	Vector multiply upper A*B to C
89	1	\$arrayC=L Q ₁ arrayA* Q ₂ arrayB;	Vector multiply lower A*B to C
8B	1	\$arrayC=S Q ₁ arrayA* Q ₂ arrayB;	Vector multiply significant A*B to C
8C	1	\$arrayC=U Q ₁ arrayA/ Q ₂ arrayB;	Vector divide upper A/B to C
8F	1	\$arrayC=S Q ₁ arrayA/ Q ₂ arrayB;	Vector divide significant A/B to C
90	1	\$arrayC=T arrayA;	Vector truncate A to C
91	1	\$arrayC=F arrayA;	Vector floor A to C
92	1	\$arrayC=C arrayA;	Vector ceiling A to C
93	1	\$arrayC=S QarrayA;	Vector significant square root of A to C
94	1	\$arrayC=S arrayS. arrayR;	Vector adjust significance of A per B to C
95	1	\$arrayC=E arrayS. arrayR;	Vector adjust exponent of A per B to C
96	1	\$half_arrayC=< full_arrayA;	Vector contract A(64) to C(32)
97	1	\$half_arrayC=R full_arrayA;	Vector rounded contract A(64) to C(32)
98	1	\$arrayC= arrayA;	Vector transmit A to C
99	1	\$arrayC=A arrayA;	Vector absolute A to C
9A	1	\$arrayC=U . arrayA;	Vector exponent of A to C
9B	1	\$arrayC=U arrayB. arrayA;	Vector pack A, B to C
9C	1	\$full_arrayC⇒half_arrayA;	Vector extend A(32) to C(64)
A0	2	\$sparseC=U Q ₁ sparseA+Q ₂ sparseB;	Sparse add upper A+B to C

A1	2	$\$sparseC=L Q_1 sparseA+ Q_2 sparseB;$	Sparse add lower A+B to C
A2	2	$\$sparseC=N Q_1 sparseA+ Q_2 sparseB;$	Sparse and normalized A+B to C
A4	2	$\$sparseC=U Q_1 sparseA- Q_2 sparseB;$	Sparse subtract upper A-B to C
A5	2	$\$sparseC=L Q_1 sparseA- Q_2 sparseB;$	Sparse subtract lower A-B to C
A6	2	$\$sparseC=N Q_1 sparseA- Q_2 sparseB;$	Sparse subtract normalized A-B to C
A8	2	$\$sparseC=U Q_1 sparseA* Q_2 sparseB;$	Sparse multiply upper A*B to C
A9	2	$\$sparseC=L Q_1 sparseA* Q_2 sparseB;$	Sparse multiply lower A*B to C
AB	2	$\$sparseC=S Q_1 sparseA* Q_2 sparseB;$	Sparse multiply significant A*B to C
AC	2	$\$sparseC=U Q_1 sparseA/ Q_2 sparseB;$	Sparse divide upper A/B to C
AF	2	$\$sparseC=S Q_1 sparseA/ Q_2 sparseB;$	Sparse divide significant A/B to C

In the B0 through B5 instructions branch can be one of the following:

1. label pair: branch address is contained in register B with a halfword index in Y
2. address constant: the B field contains a halfword index to the address given
3. n/F or n/B where n is a constant indicating a jump n halfwords forward or backward, respectively

B0	C	$\{\$fullC\}=X\{\$fullA\}+\{\$fullX\}'EQ\{\$fullZ\};branch$	Index branch if (A)+(X) EQ (Z)
B1	C	$\{\$fullC\}=X\{\$fullA\}+\{\$fullX\}'NE\{\$fullZ\};branch$	Index branch if (A)+(X) NE (Z)
B2	C	$\{\$fullC\}=X\{\$fullA\}+\{\$fullX\}'GE\{\$fullZ\};branch$	Index branch if (A)+(X) GE (Z)
B3	C	$\{\$fullC\}=X\{\$fullA\}+\{\$fullX\}'LT\{\$fullZ\};branch$	Index branch if (A)+(X) LT (Z)
B4	C	$\{\$fullC\}=X\{\$fullA\}+\{\$fullX\}'LE\{\$fullZ\};branch$	Index branch if (A)+(X) LE (Z)
B5	C	$\{\$fullC\}=X\{\$fullA\}+\{\$fullX\}'GT\{\$fullZ\};branch$	Index branch if (A)+(X) GT (Z)
B6	5	$\$fullR+; conI;$	Branch to immediate address (R)+I(48)
B7	1	$\$arrayC[arrayA]=arrayB;$	Transmit list to indexed C
B8	1	$\$arrayC='REV arrayA;$	Transmit reverse A to C
B9	1	$\{\$fullZ'ON\}fullB=arrayA. arrayC=fullY;$	Transpose move

BA	1	\$arrayC= arrayB[arrayA];	Transmit indexed list to C
BB	2	\$arrayC= !arrayA. bitZ. arrayB!;	Mask A, B to C per Z
BC	2	\$arrayC= {'NOT'} bitZ! arrayA;	Compress A to C per Z
BD	2	\$arrayC= \arrayA. bitZ. arrayB\;	Merge A, B to C per Z
BE	5	\$fullR=X conI;	Enter R with I(48)
BF	5	\$fullR=+ conI;	Increase R by I(48)
C0	1	\$fullC= arrayA'EQ arrayB;	Select A EQ B, item count to C
C1	1	\$fullC= arrayA'NE arrayB;	Select A NE B, item count to C
C2	1	\$fullC= arrayA'GE arrayB;	Select A GE B, item count to C
C3	1	\$fullC= arrayA'LT arrayB;	Select A LT B, item count to C
C2	1	\$fullC= arrayB'LE arrayA;	Select B LE A, item count to C
C3	1	\$fullC= arrayB'GT arrayA;	Select B GT A, item count to C
C4	1	\$bitZ= arrayA'EQ arrayB;	Compare A EQ B, order vector to C
C5	1	\$bitZ= arrayA'NE arrayB;	Compare A NE B, order vector to C
C6	1	\$bitZ= arrayA'GE arrayB;	Compare A GE B, order vector to C
C7	1	\$bitZ= arrayA'LT arrayB;	Compare A LT B, order vector to C

In the C8 through CB instructions the optional item, /I, indicates that the start of the search is at the location of the last hit in arrayB (rather than the beginning) for each element of arrayA.

C8	1	\$arrayC= arrayA'EQ arrayB{/I };	Search EQ, index list to C
C9	1	\$arrayC= arrayA'NE arrayB{/I };	Search NE, index list to C
CA	1	\$arrayC= arrayA'GE arrayB{/I };	Search GE, index list to C
CB	1	\$arrayC= arrayA'LT arrayB{/I };	Search LT, index list to C
CD	5	\$halfR=X conI;	Halfword enter R with I(24)
CE	5	\$halfR=+ conI;	Halfword increase R with I(24)
CF	1	\$sparseC= 'GE Q2arrayB! Q1arrayA;	Compress A to C per B
D0	1	\$arrayC= 'AVG arrayA+arrayB;	Average ((A(N) + B(N))/2 to C(N)
D1	1	\$arrayC= 'AVG arrayA;	Adjacent mean ((A(N+1)+A(N))/2 to C(N)

D4	1	\$arrayC= 'AVG arrayA-arrayB;	Average difference ((A(N)-B(N))/2 to C(N)
D5	1	\$arrayC= 'DEL arrayA;	Delta (A(N+1)-A(N)) to C(N)
D6	3	\$(bitC'ON } bitA?bitB.{fullG);	Search for masked key, bit; A, B per C
D7	3	\$!arrayC== charA.charB/Q;	Translate and mark A per B to C
D8	1	\$fullC, fullB= 'MAX Q ₁ arrayA;	Maximum of A to C, item count to E
D9	1	\$fullC, fullB= 'MIN Q ₁ arrayA;	Minimum of A to C, item count to B
DA	1	\$doubleC= +!arrayA;	Vector sum (A0+A1+...AN) to C and C+1
DB	1	\$fullC= *!arrayA;	Vector product
DC	1	\$doubleC= arrayA**arrayB;	Vector dot product to C and C+1
DD	1	\$doubleC= sparseA**sparseB;	Sparse dot product to C and C+1
DE	1	\$arrayC= arrayA**arrayB;	Polynomial evaluation A(N) per B to C(N)
DF	1	\$arrayC= fullA++fullB;	Interval A per B to C
E0	3	\$bynC=B bynA+bynB;	Binary add A+B to C
E1	3	\$bynC=B bynA-bydB;	Binary subtract A-B to C
E2	3	\$bynC=B bynA*bynB;	Binary multiply A*B to C
E3	3	\$bynC=B bynA/bydB;	Binary divide A/B to C
E4	3	\$decC=D decA+decB;	Decimal add A+B to C
E5	3	\$decC=D decA-decB;	Decimal subtract A-B to C
E6	3	\$decC=D dec*decB;	Decimal multiply A*B to C
E7	3	\$decC=D decA/decB;	Decimal divide A/B to C
E8	3	\$bynA ?bynB;	Compare binary (less, equal, greater)
E9	3	\$decA ?decB;	Compare decimal (less, equal, greater)
EA	3	\$charC= \charA.conG.charB\;	Merge per byte mask A, B per G to C
EB	3	\$charC=% charB'ON decA.fullG;	Edit and mark A per B to C
EC	3	\$charC=# charA+charB.conG;	Modulo add A+B to C, modulo G
ED	3	\$charC=# charA-charB.conG;	Modulo subtract A+B to C, modulo G

In the character string operations (D7, EE, EF, F8, F9, FD) several delimiting and length options are denoted by following the word char with a slash and a Q, which stands for

/C string is count delimited, count in bytes
 /K byte delimiter, right-justified
 /D double byte (16-bit) delimiter
 /M 8-bit mark, with 8-bit delimiter
 omitted count delimited assumed unless otherwise specified in DCL

In addition N or I can be catenated to any of the above (except D7) to indicate the type of index update: N for do not update index, and I (default) for update index based on how the instruction terminates.

EE	3	\$charC==charA/Q.charB;	Translate A per B to C
EF	3	\$fullC.fullZ==charA/Q.charB;	Translate and test A per B to C
F0	3	\$bitC= bitA'XOR bitB;	Logical EXCLUSIVE OR A, B to C
F1	3	\$bitC= bitA'AND bitB;	Logical AND
F2	3	\$bitC= bitA'OR bitB;	Logical INCLUSIVE OR
F3	3	\$bitC= 'NOT bitA'OR 'NOT bitB;	Logical STROKE
F4	3	\$bitC= 'NOT bitA'AND 'NOT bitB;	Logical PIERCE
F5	3	\$bitC= bitA'OR 'NOT bitB;	Logical IMPLICATION(B implies A)
F6	3	\$bitC= bitA'AND 'NOT bitB;	Logical INHIBIT
F7	3	\$bitC= bitA'XOR 'NOT bitB;	Logical EQUIVALENCE
F8	3	\$charC/Q<<charA/Q{.conB };	Move bytes left A to C
F9	3	\$charC/Q<<'NOT charA/Q{.conB };	Move bytes left ones complement A to C
FA	3	\$decC<>fullB.decA;	Move and scale A to C

In the pack, unpack instructions the three sign control options are denoted by a trailing slash and a Q_S, which stands for

/T trailing sign, G subfunction = #80
 /P positive sign, G subfunction = #CO
 omitted G subfunction = #00

FB	3	\$decC<]charA/Q _S ;	Pack zoned to BCD A to C
FC	3	\$charC [<decA/Q _S ;	Unpack BCD to zoned A to C
FD	3	\$(charC'ON } charA/Q?charB;	Compare bytes A, B per maskfield C
FE	3	\$(charC'ON } charA?charB.{fullG };	Search for masked key, byte;A, B per C
FF	3	\$(charC'ON }arrayA?arrayB.{fullG };	Search for masked key, word;A B per C

COMMENT SHEET

MANUAL TITLE STAR Software System

Reference Manual

PUBLICATION NO. 59156400 REVISION B

FROM: NAME: _____

BUSINESS
ADDRESS: _____

COMMENTS:

This form is not intended to be used as an order blank. Your evaluation of this manual will be welcomed by Control Data Corporation. Any errors, suggested additions or deletions, or general comments may be made below. Please include page number references and fill in publication revision level as shown by the last entry on the Record of Revision page at the front of the manual. Customer engineers are urged to use the TAR.

CUT ALONG LINE

PRINTED IN U.S.A.

AA9419 REV. 11/69

NO POSTAGE STAMP NECESSARY IF MAILED IN U. S. A.

FOLD ON DOTTED LINES AND STAPLE

STAPLE

STAPLE

FOLD

FOLD

FIRST CLASS
PERMIT NO. 8241
MINNEAPOLIS, MINN.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY
CONTROL DATA CORPORATION
ADVANCED SYSTEMS LABORATORY
4201 NORTH LEXINGTON AVENUE
ST. PAUL, MINNESOTA 55112

ATTN: DOCUMENTATION GROUP



CUT ALONG LINE

FOLD

FOLD

Advanced Design Laboratory

