This paper outlines the structure and operation of the system's loader.

The new system functions which affect the loader are related to the additional functions which the loader performs.

Descriptions of the algorithms employed by the loader for symbolic unit assignment and buffer allocation are included.

Design of an integrated programming and operating system

Part III: The expanded function of the loader

by R. Hedberg

additional functions of the loader

Historically, the principal functions of a loader have been to:

- Determine the operational locations of a program's parts (instructions and data).
- Translate all non-absolute references (relative location addresses of data and instructions, and inter-program part references) to their absolute forms.
- Make the initial placement of data and instructions in core so that the program may be properly executed.

However, the loader (IBLDR) within the IBSYS/IBJOB system performs a number of additional functions. The preceding paper (Part II) has explained how the system has been constructed to permit program segmentation with the use of control sections to accommodate modular program design. As a consequence of this feature of the system, the loader plays a larger role in assigning the location of program parts. Other features which affect the loader relate to channel and I/O unit assignment, and to buffering within the object programs. Thus, IBLDR must perform functions in addition to the historical ones. It must:

- Remove the control section structure and transform the instructions and data of the various program parts into a single operational entity.

- Make certain determinations relative to channel and I/O unit assignment.
- Make certain determinations relative to the number, core location and assignment of buffers.

In this paper we will discuss the loader relative to the above functions, outline the structure and operation of the loader, and describe the symbolic unit assignment and buffer allocation algorithms.

processing control sections

The program's control section structure is not disturbed by the assembly process. It is the loader that must eliminate the control section structure and transform the instructions and data of the various program parts into a single operational entity. This transformation requires the loader to determine the appropriate operational locations of the program's parts and to make the necessary memory assignments. The loader must interpret the control cards to determine, for example, whether:

- Only certain parts of the program are to be loaded to form the desired object program.
- Certain parts of the program are to be obtained from the library.
- Certain parts of the program are to be linked as program overlay[1] segments.

Relative to overlay linkage (which the programmer indicates by means of control cards inserted at appropriate points between subprogram decks), the loader assumes the responsibility of checking whether the resulting overlay operations are logically consistent (e.g., a call for a program part that would subsequently destroy the call would be improper). The loader automatically supplies the proper link reading instructions for the object program.

A large part of the loader's activity is concerned with control section processing. All intra-program references that have not been reduced to relative numeric addresses by IBMAP are in the form of inter-control-section references. As such, they are processed in a uniform manner. The techniques used for control section processing are of interest in that no table-searching mechanism is used. The process uses hashing,[2] chaining methods, and a special technique for detecting nested control sections. Control section processing is described in the literature.[3]

judgments by the loader

In the IBSYS/IBJOB system, the physical I/O units assigned to an object program are not identified until load time. A programmer specifies I/O units by means of unit requests. The symbolic unit assignment algorithm incorporated within the loader makes "judgments" as to which channels and units "best" satisfy the unit requests. The algorithm is described in some detail in a later section.

IBLDR also makes judgments as to the number and location in core of buffers associated with the input/output files. The I/O

buffer allocation algorithm is also described later in the paper and an illustrative example appears in the Appendix.

structure and operation

The IBJOB monitor supplies input to IBLDR in the form of a *load file.* Figure 1 displays the inputs and processes associated with the load file. A load file consists of a collection of assembled program decks together with appropriate control cards. In general, each program deck contains a control dictionary, a file dictionary, and a set of relocatable instructions known as *text.* The IBLDR program is divided into five sections. Section 1 stores the load file information internally in condensed tabular form:

- First, the control dictionaries are stored in ascending memory locations (the area labeled CDICT (Control Dictionary) in Figure 2).
- Second, control card information and file dictionary entries are stored in condensed form (the area labeled CISB (Control Information Storage Block) in Figure 2). This information is stored in descending core locations of CISB and a chaining mechanism is established so that each item of information is chained to all items of the same type in the order of their occurrence (e.g., the first occurrence of a particular type of control card is chained to the second occurrence of the same type, etc.).
- Third, relocatable instruction text is stored in the *internal file*

Figure 1

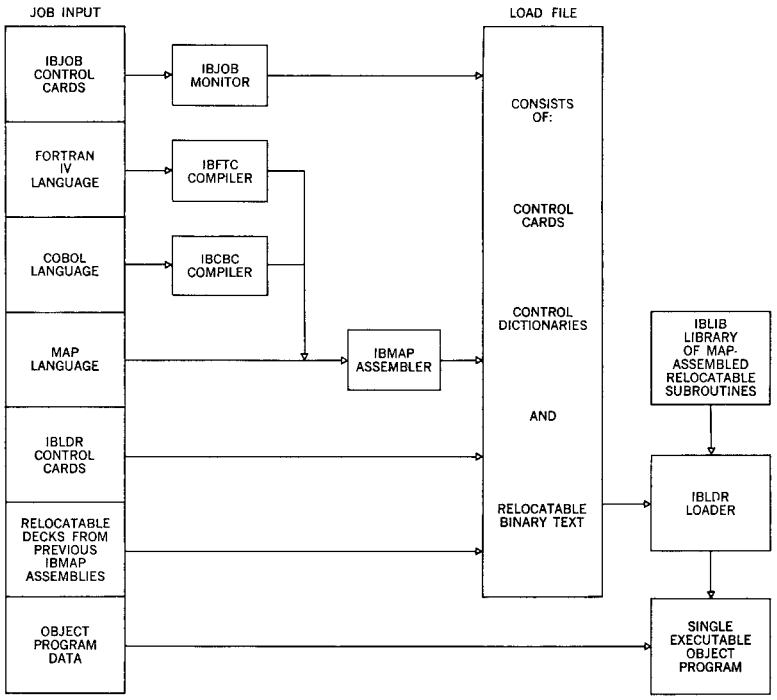
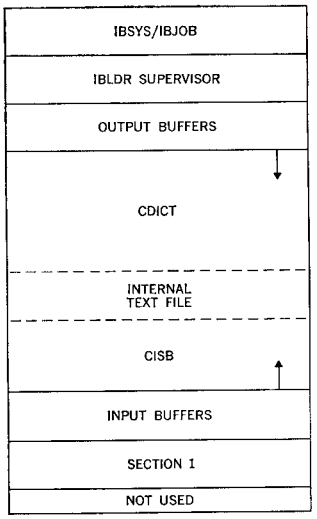


Figure 2



area. This area is a set of IOCS buffers, so that the text is stored as an IOCS internal file. The internal file area retains the properties of any IOCS file; it can, for example, be written and read again later. Further, it may be partially or totally transferred to any appropriate I/O device such as disk or tape.

The working storage requirements of Section 1 are small except for CDICT, the internal file, and CISB. The tables and file are assigned storage, as needed, by a single storage handling routine. A possible overlap of a table with the file is prevented by spilling the internal file to tape or disk. Generally, programs with less than 6000 source instructions may enter core and remain in memory until loading has been completed.

Section 2 of IBLDR is concerned mainly with determining which set of subroutines will be required from the subroutine library. The Librarian (IBLDR operating in a special mode) will have prepared a number of entries for each subroutine it has placed in the library. The first part of the library contains control section information for all the subroutines in the library. A second part lists all the control dictionaries and file dictionaries. A third part lists the actual text, subroutine by subroutine. Section 2 needs only to read the first part of the library to determine the identity of the complete set of called subroutines. It then obtains the corresponding control and file dictionaries, and calls on Section 1 (which has been left in core, see Figure 3) to complete the CDICT and CISB tables.

The information in CISB is then processed by type in order of occurrence (i.e., particular type control cards all at once, etc.). The ability to process these by type, regardless of order of occurrence, permits control cards to occur in arbitrary order in the load file. A set of tables is produced from CISB and stored just after CDICT. The tables contain the information required by Sections 3 and 4 to complete the loading procedure.

The main responsibility of Section 3 is to allocate storage for the object program. It assigns absolute core locations to the decks and sections as they are represented in CDICT. It also performs symbolic unit assignment and I/O buffer allocation. Before Section 3 is read into core, the maximum amount of working storage required by Sections 3 and 4 is determined from the number of control sections and the number of files in the object program. If the amount available is less than required, the internal text file is spilled before Section 3 is loaded.

Section 3 initiates the *absolute text internal file* and generates certain instruction sequences which augment the object program. Section 4 is loaded at the same time as Section 3 because the Section 4 routines which control formation of the absolute text internal file are also used by Section 3. The absolute text internal file is constructed in the IOCS internal file format and is located above Section 3 in core (see Figure 4).

Section 4 begins a second pass over the input and processes the

Figure 3

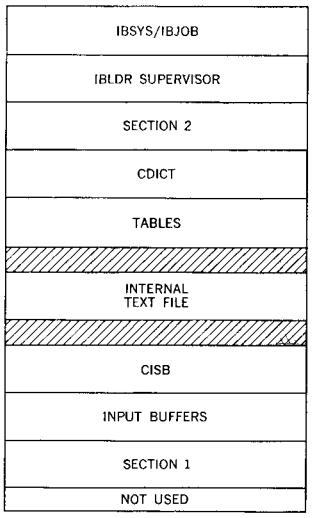


Figure 4

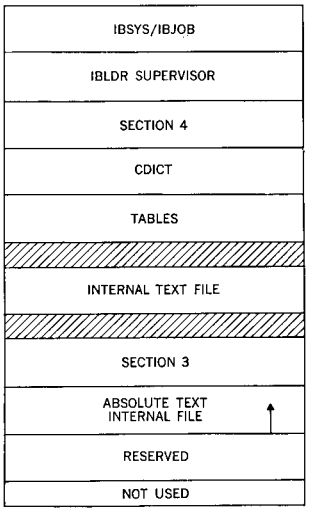


Figure 5

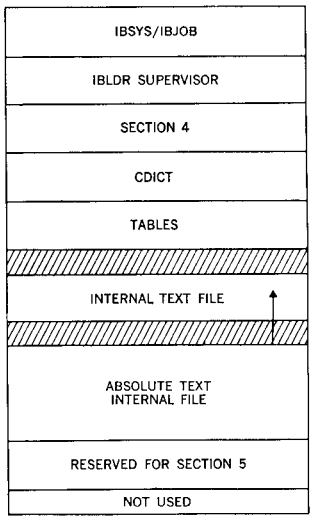
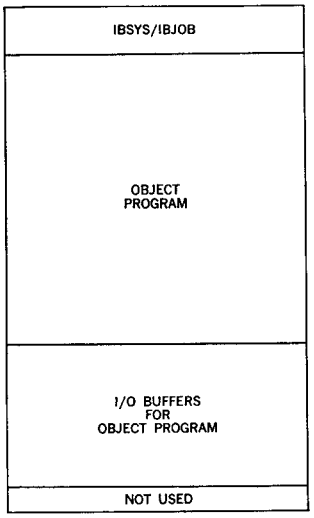


Figure 6



text stored by earlier sections. During operation of the first three sections, the tables and CDICT were stored in lower core. Since Section 4 is also loaded into lower core, the upper portion of core is available for storing the program text (see Figure 5).

Processing begins with the unrelocated text in the internal file found just above the midpoint of core. If that area is needed for working storage, text is spilled to an I/O device and processing is begun by reading it. As text is read and relocated, it is recorded in the absolute text internal file. The storage originally assigned to the internal text file is restricted so that the internal text could be completely relocated and stored in the absolute text internal file without conflict in storage. As soon as the unrelocated text file is empty, the area occupied by it becomes available for further growth of the absolute text internal file. The text of called subroutines is treated similarly. The absolute text internal file can hold approximately 8,000 object program instructions. For the larger programs which would require loading past the midpoint of core, an overflow absolute text file is used, but only for the amount that would fall above the lower boundary of the internal file of absolute text.

If the object program is to be executed, Section 5 performs the final movement of the absolute object program from the absolute text file (in core and/or on tape/disk) to its position of execution in core (see Figure 6).

The text of programs in the load file is generally not in contiguous loading order. The reordering method employed is to treat the text in its given order until the final movement prior to execution, at which time it is scatter-written into its proper order. This is accomplished by inserting control words into appropriate places in the text. The control words which contain the destination information then direct a simple IOCS read operation to reorder the text in transit.

IBLDR's internal structure allows it to adapt efficiently to different size loading jobs. The small program loads in system tape time plus input tape time. The large program (say greater than 8,000 total instructions) may require the use of two utility tapes during Pass 1 and three utility tapes during Pass 2.

Figure 7 indicates the paths that the text of a job-segment[4] may follow when it is processed by IBLDR. The text starts out on the load file and arrives in core as an absolute program load. All other indicated I/O activity is dependent upon the size and complexity of the object program. In Section 1, if the internal text file grows too large (in the direction of CISB), the internal text file "overflows" to UT4; if CDICT grows too large (in the direction of the internal text file), then the internal text file "spills" to UT3.

The symbolic unit assignment algorithm is designed to:

symbolic unit assignment algorithm

- Minimize the number of machine halts required to mount tapes.
- Balance channel usage for efficient overlapped I/O operations.

The physical I/O units are assigned to the object program in

response to the *unit requests* which have been specified by the programmer. There are several types of unit requests:

- *System unit.* The I/O unit being used for the named IBSYS/IBJOB system function is assigned.
- *True channel and unit.* The channel that is specified is the actual one assigned. The unit specification is interpreted relatively—unit 1 is the first unit on the channel not in system use.
- *True channel, no unit.* Any one of the available units on the specified channel may be assigned.
- *Reserve unit.* This type of request is used to obtain a unit whereby one job-segment may communicate with another. The unit request, when it appears in the first job-segment, is treated as a symbolic request (see below). The request, when it appears in the second job-segment, is assigned the identical unit that was assigned in the first job-segment.
- *Symbolic channel and unit.* A channel which has relatively little work already assigned and which has an appropriate unit available is selected for assignment. All channel requests using the same letter are, if possible, assigned to the same channel. A unit specification, *n*, is assigned, if possible, the *n*th unit in what is called an *availability* chain.[5] Identical symbolic requests are always assigned the same I/O unit.
- *Symbolic channel, no unit.* The channel is assigned as described in the preceding entry. Any one of the available units on the specified channel may be assigned. Two identical symbolic requests are assigned distinct units.

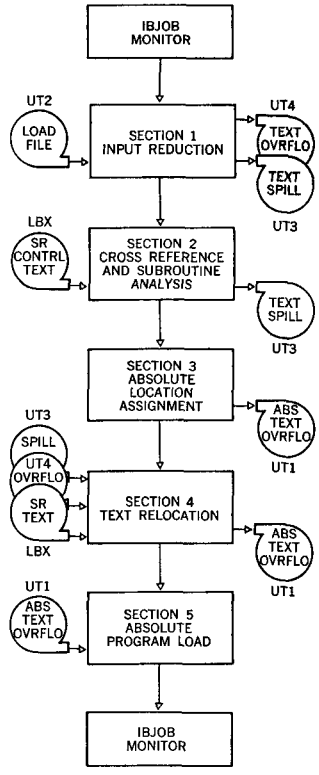
Table 1 illustrates the order in which units are assigned for the various types of unit assignment requests. A unit request with no channel or unit specified is also treated as a symbolic request (see below).

Table 1 IBLDR unit assignment order

Unit request type	*Sample requests*	*Order of assignment* 1	2	3	4
Card equipment	RDA, PUB	x			
True channel and unit	B(3), C(1)	x			
True channel, no unit	C, E		x		
System unit	OU, UT3	x			
Reserve unit (previously assigned)	J(1), K(2)	x			
Reserve unit (to be assigned)	K(2), L(1)				x
Symbolic channel and unit	S(2), T(3)				x
Symbolic channel no unit	T, U, V				x
No special request					x
Symbolic secondary unit	S(1), *				x
True secondary unit	A(1), *			x	
System secondary unit	UT(3), *			x	
Internal file	INT	x			
1301 disk	CD00/0	x			
Hypertape	EH03/1	x			

x denotes the relative time at which the specified unit is assigned.

Figure 7



The symbolic unit assignment algorithm has two parts. The first part constructs two tables: a channel characteristics table (CHACT) and a channel requirements table (CHART). For each true channel CHART lists:

- The number of units available.
- The number of units "ready" (units that are in a loaded ready-to-go position).
- The negative of the sum of activities of files already assigned (file activity numbers are normally supplied by the user, otherwise a value of 1 is chosen).
- The availability chain address (chain entries identify the available units).

For each symbolic channel CHART lists:

- The number of units required.
- The number of "secondary" units required (a secondary unit is the second unit required by a multi-reel file).
- The number of ready units required.
- The sum of activities of the associated files.
- A chain address (chain entries identify the symbolic unit request records).

Part 1 of the assignment algorithm sorts CHACT and CHART into descending order and then matches them entry for entry. Both CHACT and CHART use entire entries as sorting keys. For CHACT, the effecting sorting key fields are the number of available units, the number of ready units, and the negative of the sum of activities. CHART's effective sorting key fields are the number of units required, the number of secondary units required, the number of ready units required, and the sum of activities. The keys reflect the objectives of matching channels with many units available to channel requirements having large unit needs and, secondarily, assigning channels with low activities to high activity channel requirements.

Once the channels have been matched, symbolic requests are assigned units from the availability chain. The first unit in the chain which has the desired characteristics (ready, model type, etc.) is chosen for the first symbolic request.

In this manner, all assignment requests may be satisfied only if each symbolic channel can be matched with a channel having sufficient units. Requests not assignable and requests without channel specifications are assigned units by the second part of the algorithm.

Part 2 makes use of CHACT entries and another tabulation labeled TOUT. Each unit request not assigned by the CHACT-CHART sort and match procedure is placed in the TOUT table. For each symbolic unit request TOUT lists:

- The number of units required (one or two, two when a request also calls for a secondary unit).
- A "1" if the request is for a ready unit, "0" otherwise.
- The address of the symbolic unit request record.

The CHACT table, updated according to the Part 1 assignments, and the TOUT table are sorted in descending order. The sorting keys are again entire table entries. For TOUT, this yields two effective sorting keys: the number of units required and a ready request indication.

The first TOUT entry is assigned one unit (or two if requested and still possible) from the top CHACT entry, the second TOUT entry is assigned a unit from the second CHACT channel and so on until the end of TOUT or CHACT is reached. TOUT is then regenerated from the remaining symbolic requests, CHACT and TOUT are again sorted, and the one for one match repeated. This process is continued until all requests are assigned or until no appropriate units remain. The process tends to rotate the real channels as candidates for assignment to files without channel specifications and has the tendency to balance channel usage when there is no exact fit of a symbolic channel to a real channel.

In summary, the algorithm minimizes the number of tape mounting stops by selecting ready units whenever possible. Efficient I/O operation is achieved by balancing channel usage according to activity. A final point not previously mentioned—the algorithm also provides complete directions as to the actions required by the machine operator.

I/O buffer allocation algorithm

It is often difficult to decide how a fixed piece of memory should be allocated for buffering a set of files. Two questions arise:

- After all the procedural parts of a program are combined, how much memory is left for use in buffering files?
- If available memory is limited so that the optimum number of buffers for each file is not possible, which files should be given preference?

In this system it is not necessary for the programmer to determine the buffer allocation specifications. If buffering is not explicitly given by the programmer, the loader calls on its buffer allocation algorithm. The algorithm will determine buffer specifications and, in doing so, will make an effort to allocate the buffering on an "optimal" basis.

Some terminology will be needed to describe the algorithm:

- *Block size.* The length of a physical I/O record where "physical" is used to suggest a record as it appears on an I/O device. (A procedural record description is usually not descriptive of the physical record.)
- *Buffer.* A core area used for communication between an I/O device and the CPU. Its length is equal to or greater than the block size.
- *Pool.* A group of buffers of the same size.
- *Activity number.* A number assigned to a file which reflects its expected usage relative to the expected usage of the set of files of the program. Activity specifications are normally supplied by a user. If not, a value of 1 is chosen.

The algorithm does not allocate buffers directly to individual files but rather to pools. This is done to allow flexibility in IOCS operation (when called on by the object program, IOCS assigns buffers to files as needed).

There are three distinct parts to the allocation algorithm. First, the "minimum" number of buffers per pool is allocated (in the absence of explicit user specification, one buffer per file is the minimum). If this minimum need cannot be met, an appropriate error message is prepared and object program execution is inhibited.

During the second part of the algorithm, a "preferred" number of buffers per pool is allocated if possible (the preferred number being twice the number of files in the pool since two buffers per file allow the file to be processed simultaneously by computer and data channel). If available storage is not sufficient to satisfy all preferred needs, then a buffer by buffer assignment is made to those pools with the greatest individual "need". A pool's need is measured by the product of the sum of activities and the "deficit" number (the preferred number minus the number assigned) of buffers.

After the above rules have been applied, the third part of the algorithm distributes any remaining storage (in buffer size units) in proportion to each pool's output activity number (the sum of activities of its output files). Only output file activities are considered since IOCS does not normally use more than two buffers per input file, whereas IOCS may (according to demand) use any number of buffers for an output file. Note that the only candidates for "extra" buffers under Part 3 of the buffer allocation algorithm are those pools with preferred numbers already assigned.

An example illustrating the operation of the buffer allocation algorithm is given in the Appendix.

Appendix: An example to illustrate operation of the buffer allocation algorithm

The buffer allocation procedure consists of six phases. The functions of the phases are:

Phase 1. Determines the number of buffer pools to be generated.
Phase 2. Assigns files to pools.
Phase 3. Computes storage requirements for each pool.
Phase 4. Assigns extra storage according to the activity specification and buffer size requirements.
Phase 5. Generates the IOCS file list including the location of any non-standard label processing routine.
Phase 6. Generates calling sequences for IOCS initialization at object time.

To illustrate the phase-by-phase application of the algorithm, consider the example given in Table A1. Phase 1 groups the files by block size and develops the information shown in Table A2. Phase 2 summarizes the pool requirements as shown in Table A3 where:

Table A1 Sample set of file requirements

File	Block size	Activity	Type
F1	198	2	Output
F2	198	1	Input
F3	98	0	Output
F4	98	0	Output
F5	98	1	Output
F6	48	0	Input
F7	48	3	Input
F8	48	1	Input
F9	98	2	Output

Assume 1670 words of storage are available for buffers.

Table A2 Pool requirements

Pool	Buffer size	Number of files
1	200	2 (F1, F2)
2	100	4 (F3, F4, F5, F9)
3	50	3 (F6, F7, F8)

Note: IOCS requires two control words for each buffer.

Table A3 Summarized pool requirements, first form

Pool	No. of files	Buffer size	Minimum buffers	Deficit buffers	Sum of activity	Sum of output activity
1	2	200	2	2	3	2
2	4	100	4	4	3	3
3	3	50	3	3	4	0

Table A4 Summarized pool requirements, second form

Pool	Buffer size	Minimum buffers	Deficit buffers	Minimum storage	Preferred storage	Sum of activity
1	200	2	2	402	802	3
2	100	4	4	402	802	3
3	50	3	3	152	302	4
			Totals	956	1906	10

- The minimum number of buffers per pool is taking as being the number of files per pool (unless otherwise specified by the user). Rule—one buffer per file is the minimum requirement.
- The deficit number of buffers per pool is set equal to the minimum number of buffers per pool. Rule—the deficit buffer count measures the difference between the preferred number of buffers and the minimum number of buffers (the preferred number is twice the number of files).
- The summation of activity per pool is the sum of the activities of the members of that pool.
- The summation of output activity per pool is the sum of the activities of the output members of that pool.

Phase 3 further processes the information to obtain Table A4 where:

Table A5 Weighing factors

Pool	*Deficit*	*Activity*	*D*A*
1	2	3	6
2	4	3	12
3	3	4	12

- The minimum storage per pool equals the buffer size times the minimum number of buffers plus two cells for control words.
- The preferred storage per pool equals the buffer size times the optimum number of buffers plus two cells for control words.

As originally stated in the example, 1670 words are available for i/o buffer assignment. In this example it is shown that 1906 words are required for preferred storage and 956 for minimum storage. The minimum number of 956 is assigned first. Phase 3 then computes a table of weighting factors by forming the products of the deficit and activity numbers. The weighting factors are shown in the "*D*A*" column of Table A5.

The weighting factors table is sorted in descending *D*A* order and, within that order, in descending buffer size. The pool with the largest *D*A* value is assigned one buffer if storage permits. In this case, Pool 2 is assigned an additional buffer. Deficits are adjusted, weighting factors are recomputed, and the pool with the largest remaining *D*A* is assigned a buffer if possible. This process is continued until a pool cannot be assigned a buffer because of insufficient storage (its deficit is considered unreconcilable and set equal to zero) and until all deficits equal zero. Table A6 summarizes the deficit buffer assignment for the example and Table A7 shows the final pool assignments.

In this example, if there had been at least 1906 words (the preferred storage) available, then all buffers would have been assigned automatically and the last mentioned procedure would have been by-passed.

Table A6 The deficit buffer assignment procedure

		Pool 1		*Pool 2*		*Pool 3*		
	Avail-	*Buff size = 200*		*Buff size = 100*		*Buff size = 50*		*Pool*
Iteration	*able words*	*D*	*D*A*	*D*	*D*A*	*D*	*D*A*	*selected*
0	714	2	6	4	12	3	12	Pool 2
1	614	2	6	3	9	3	12	Pool 3
2	564	2	6	3	9	2	8	Pool 2
3	464	2	6	2	6	2	8	Pool 3
4	414	2	6	2	6	1	4	Pool 1
5	214	1	3	2	6	1	4	Pool 2
6	114	1	3	1	3	1	4	Pool 3
7	64	1	3	1	3	0	0	None
8	64	1	0	1	3	0	0	None
9	64	1	0	1	0	0	0	None

Table A7 Final pool assignments

Pool	*Minimum buffers*	*Newly assigned buffers*	*Total buffers*	*Buffer size*	*Total storage*
1	2	1	3	200	602
2	4	3	7	100	702
3	3	3	6	50	302

Total words used = 1606, words unused = 64.

The assignment of extra buffers to buffer pools (beyond those assigned in Phases 1 to 3) is performed by Phase 4 in accordance with the criteria:

- If the user specified a buffer count for some pool, then that pool is not eligible for extra storage assignment.
- The activity of input files is not counted in the determination of extra storage assignments.

The amount of extra storage assigned to each pool is determined as a certain fraction of the total storage available. The fraction is computed as the ratio of each pool's output activity to the sum of the output activity for pools which have not yet been considered for extra storage (zero divided by zero is taken to be equal to zero). The ratios are computed in one pass over the table by processing it backwards. The pool with the largest buffer size is assigned extra storage first. The ratio calculations for the example are shown in Table A8.

Table A8 Ratio calculations

Pool	*Buffer size*	*Output activity*	*Ratio*
1	200	2	0.4
2	100	3	1.0
3	50	0	0.0

Storage is then assigned to pools in forward order throughout the table. The example was constructed to illustrate the Phase 3 process and does not illustrate Phase 4.

Thus, we will change the example by assuming that there are 3000 words of storage available in addition to the preferred storage of 1906 words. Then the procedure of Phase 4 is illustrated by Table A9. It will be noted that in this example the ratio of extra storage assigned is 1800 to 1200 or 3 to 2, which reflects the 3 to 2 output activities of the two pools.

Table A9 The extra storage assignment procedure

Available storage	*Pool*	*Ratio*	*Storage times ratio*	*Buffer size*	*Buffers assigned*	*Storage used*	*Remaining storage*
3000	1	0.4	1200	200	6	1200	1800
1800	2	1.0	1800	100	18	1800	0
0	3	0.0	0	50	0	0	0

Phases 5 and 6 of the buffer allocation procedure generate the correct initialization instructions for IOCS and perform validity checks on the file dictionary information. The buffer counts and sizes in the generated instructions are taken from the entries in the pool table.

ACKNOWLEDGMENT

The IBLDR system design philosophy and programming techniques described in this paper represent the work of four additional members of Programming Systems, IBM Data Systems Division, Los Angeles, California. D. Stark, co-designer of the system; N. Gentry, unit assignment; M. Minami, text relocation; and R. Jacobson, first pass input and control card processing.

CITED REFERENCES AND FOOTNOTES

1. Overlay program techniques are discussed in Reference 3.
2. Hashing refers to the process of using an arithmetic manipulation on a name in such a way that the name determines a unique storage location in some prescribed table.
3. IBM 7090/7094 Programming Systems: IBJOB Processor, Systems Reference Library C28-6275, International Business Machines Corporation, 1963.
4. A job in IBSYS may be viewed as the deck of cards submitted to IBSYS by a single user (its beginning and end are indicated by control cards). It may require the functioning of a number of the IBSYS subsystems (IBJOB, 90SORT, etc.). A job-segment on the other hand is taken to mean a single functioning of one of the subsystems. It again may be viewed as a deck of cards, a subdeck of a job deck. One functioning of the IBJOB subsystem (a job-segment) corresponds to at most one functioning of IBLDR.
5. The availability chain was originally kept in reverse order of most recent usage. The intent was that successive object program loads would make use of different I/O units for their symbolic unit needs, thereby spreading the work load among the tape drives and giving the operator time to remove tapes. A recent IBSYS design change generalizing the notion of a job (see 4 above) has negated the above by effectively keeping the availability chain in one fixed order.