CHAPTER 4

MASH SYSTEM STRUCTURE AND IMPLEMENTATION

This chapter describes the program structure of the MASH system and its related files and programs and describes the strategy and techniques of system implementation. The following description therefore is oriented to the reader who would like to understand how the system functions, rather than what functions it offers to its users.

Background

Systems for simulating microanalytic models of demographic and economic behavior have been constructed prior to MASH. To the best of the author's knowledge, the first such computer simulation system was constructed by Greenberger [G3] in support of research performed by Orcutt, Greenberger, Korbel, and Rivlin [O4]. The system for simulation constructed by Greenberger is of interest both because it logically precedes the present research effort and because its implementation reflects an earlier generation of computer hardware and software technology. Greenberger's system, named "Sussex," was written in SAP (*S*ymbolic *A*ssembly *P*rogram) for an IBM model 704 computer, having 8,192 words of immediate access memory, 10 magnetic tape drives with a recording density of 200 bits per inch, a magnetic drum having an additional 8,192 words of storage, and a card reader, card punch, and line printer. Although such a computer represented the state of the art in commercially available computing equipment in 1957, Sussex and the simulation experiments performed with it taxed the equipment's capacity severely.

Within SUSSEX, entities representing the micro population at any simulated time were stored sequentially on one reel of magnetic tape. The population was updated one time period by applying the operating characteristics sequentially to each population entity and recording the updated status of that entity on another reel of magnetic tape.[1] The records of persons to be married were stored temporarily on the magnetic drum. The drum was divided into 32 sectors, each of which was identified with homogeneous characteristics for the marriage process. Thus, the marriage market consisted of 32 marriage submarkets. Within each submarket, marriages were performed among those persons selected on a "first come, first married" basis, subject only to the constraint of marriages taking place between members of opposite sexes.

[1] Successive updates of the population were recorded in a cyclic pattern on three reels of magnetic tape, thereby providing at least one previous simulation state at any point in the computational sequence. In the event of machine failure, at most one simulated time period of computation was lost.

Assembly language programming was used exclusively primarily to minimize the operating costs of simulation. In 1957, higher level languages such as Fortran were just beginning to be available. Their syntax, efficiency, and flexibility were considerably less than their present day counterparts. Although the concept of subprogram had evolved at the assembly language level, it had not yet appeared in higher level languages. Object code produced by initial translators was relatively inefficient in execution and occupied more space than comparable code written by an average assembly language programmer. Translation, or compilation, time was very large compared with that for assembly language programs. Most important, the ratio of the costs of the factors of production incurred in program development -- computer time and programming labor -- was substantially higher than it is at present.

Various techniques were used by Greenberger to construct an efficient simulation system. Most functions were approximated by piecewise linear functions (linear splines) with up to 8 linear segments, and a general interpolation subroutine was used to evaluate the piecewise linear approximation. This technique allowed most functions to be stored as a sequence of $2(n+1)$ values for n line segments. Special attention was given to the efficiency of the pseudo random number generator, since the implementation of the microanalytic model relied heavily upon Monte Carlo techniques. The system was controlled by a sequence of control cards, each of which initiated the execution of a different function. This allowed maximum use to be made of the intervals of computer time available to users of the system and facilitated an easy manual means of specifying the simulation agenda.

In the intervening 15 years since Greenberger's Sussex system was constructed, advances in computer hardware technology and software tools have been substantial. The cost of automatic computing, measured in terms of computations per unit of money, has decreased several orders of magnitude. Advances in electronic technology have greatly increased the reliability of current hardware, although the increased complexity of current software executive systems have detracted somewhat from this gain. The capacities of components and their speed have also grown quite substantially.

Software advances have also been substantial. High level languages have been constructed for many different purposes, and most applications programming work now relies upon these languages and their translators. The languages are more powerful and specialized, and the techniques of writing efficient translators that produce efficient object code are widely known. The costs of programming -- resulting partially from an increasing ambitiousness of tasks undertaken and partially from increasing wage rates -- have risen substantially, even though programmer efficiency has also increased substantially due to capital investment in high level programming languages. Combined with the decrease in computer hardware costs, this has led to the ratio of factor costs -- of computing time to programming labor -- decreasing considerably. This implies that efficient utilization patterns of these resources would tend to exhibit an increasingly computer intensive production process, i.e. continuing substitution of machine resources for programming resources, and this phenomenon is commonly observed. In addition, there has always been a high level of capital investment in software that is hoped will increase programming labor productivity.

The environment in which experimental simulation systems such as Sussex and MASH should properly be viewed is considerably larger than simply its hardware and software components. Such systems are part of a larger production process, which is the production of research and policy tools and results. In addition to the inputs to a simulation system, inputs to this larger process include the existing stock of knowledge in the social sciences, additional research performed directly in support of model construction as described in [O6], interaction with individuals in policy making positions, and the substantial resources involved in integrating these outputs. The cost of constructing, implementing, and using a complex microanalytic model is substantially larger than the actual hardware and software costs incurred. Thus, although the costs of program development and use are by no means insignificant, they do not dominate the cost of such model construction. Emphasis upon the efficiency of the computational process is no longer as important a consideration as it once was due to the shift in the factor costs of production.

Implementation Goals and Strategy

A primary goal in the design and implementation of MASH has been to produce an on-line, interactive system that is used through a high level, user-oriented command language. Each of these attributes of the system was considered to be important in maximizing the contribution of the simulation system to the entire research task. Choosing such a goal, however, implies rejecting others. While command languages are more familiar to a user than strings of numeric parameter values, commands must be decoded by the system -- a process that requires time and program memory space. Interactive systems may more properly support the iterative nature of social research, but they may require more complex operating systems than those that support only batch processing. On-line storage is very useful for a variety of purposes -- especially when coupled with user interaction -- but off-line storage such as magnetic tape is less expensive. Programming the system in Fortran IV allows many programmers to construct object model operating characteristics and understand the structure of the system, but at some cost in memory used and system execution time. In addition, use of Fortran IV allows substantial transferability of both existing programs and programming skills. In designing MASH, the balance in each of these tradeoffs was evaluated and a mixed strategy chosen that was judged to represent a good balance to support the research task. Both the reader and the user of the system may judge the appropriateness of the overall design and implementation strategy and the particular choices made.

MASH is implemented using a combination of Fortran IV subprograms and PDP-10 assembly language (Macro-10) program. The great majority of the system is coded in Fortran IV. Among the reasons for selecting Fortran as the primary implementation language were: (1) widespread knowledge and readability among both professional programmers and economists with moderate computer exposure; (2) efficiency of the programming process using a high level language; (3) ease of interfacing Fortran programs with assembly language subprograms; (4) the relative ease of exporting and importing modules of code to and from other computer centers and research groups; and (5) the existence of efficient Fortran compilers that support interaction at execution time. Assembly language was used for functions where Fortran compiled code would have been grossly inefficient either in terms of execution time or memory required and where the inefficiency would result in substantially increased costs incurred during a typical micro simulation experiment.

Because of the relative age of the Fortran language and the availability of a number of languages oriented more directly to simulation techniques, the Fortran/Macro-10 implementation strategy requires some defense. Assembly language coding was considered attractive from the point of view of execution efficiency, but development time, debugging, and implementing operating characteristics would have been considerably costlier, and maintaining and exporting the system would have required specialized and scarce skills. COBOL was considered because of the substantial data manipulation and input-output activity inherent in processing large files, but it was rejected because of difficulty in interfacing with an algebraically oriented language processor, its inefficient and awkward algebraic capabilities, and its lack of overlay capabilities. PL/I, which syntactically contains both algebraic and file processing statement types, was considered a very attractive candidate, but the lack of an efficient PL/I processor residing on an available computer that supported efficient execution time interaction was considered to be a very significant drawback.

Of the computer simulation languages available, SIMSCRIPT II and its variants came closest to meeting the requirements of an adequate implementation language. In fact, SIMSCRIPT II offered many capabilities that were not required. For example, it supports quite complex event driven simulations involving a moderate number of entities. Furthermore, SIMSCRIPT lI contains comprehensive facilities for describing entity types, entities, their attributes and their values. Operations that perform state changes upon diverse entities are nicely and fairly naturally expressed in SIMSCRIPT II. The possibility of using the language was explored thoroughly.

SIMSCRIPT II was rejected for five reasons. First, power of the language is maximized when the entire model and simulation state reside in immediate access memory simultaneously, a situation that is not possible when large populations are used. Although program overlays are supported, it is difficult to communicate between overlay segments. Second, input and output of data are provided but only at a primitive level; and the structure of entities, attributes, and values is not maintained on external storage media. Third, when the implementation decision was made it was not possible to run SIMSCRIPT II in a virtual environment which would have provided sufficient program address space to accommodate all of the required program and data. Fourth, SIMSCRIPT II was only implemented on IBM/360 and IBM/370 series computers. These computers did not support well general interaction between executing programs and users. Fifth, the translator for the SIMSCRIPT II to object code translation process was then considered to be almost an order of magnitude more costly than that for PL/I, which was considerably costlier than the same process for Fortran. Although many of the syntax features of SIMSCRIPT II were very appealing, the language was rejected because of other of its limitations.

Other simulation languages that were examined did not seem to have any advantage over Fortran as an implementation vehicle. The specific features of computer simulation languages support functions such as queueing, event scheduling, process flow modelling, and other features that are inapplicable to the structure of the microanalytic model. Although the models implemented within MASH are certainly dynamic economic models, the supporting computing functions required may be better characterized as scientific computing techniques applied to large data files. Thus, much of the power of most simulation languages is of very limited application for implementing microanalytic models of the type described in [O6].

Program Structure

The acronym MASH (*M*icro*A*nalytic *S*imulation of *H*ouseholds) refers to the PDP-10 simulation system program for executing the object model forms of mixed micro and macro economic models. This program contains the great majority of programming that supports the implementation and simulation of the microanalytic model described in [O6]. In addition, there exist several other programs and a number of on-line data files whose use is integrated with the use of MASH for simulation experiments.

The MASH system, a mixture of Fortran and Macro-10 programs, executes as a standard user job under the PDP-10 (TOPS-10) time sharing monitor system. In the absence of overlays, MASH currently requires approximately 135K (K=1024) 36-bit words of main memory for program and variable storage. As the current model grows and as new functions are added to the system, this memory requirement will expand.

In order to provide for substantial growth of the system, the program has been organized in an overlay structure, which currently consists of one resident portion of approximately 14K words and 14 non-resident "links" of up to 12K in size. Approximately 2K additional memory is required for input and output buffers, so that the system currently requires 28K memory. An additional 5K is used by the Fortran support package, FORSE, which must be in memory simultaneously; however, this module is reentrant and therefore one copy is shared among all concurrent jobs requiring it.

A non-standard PDP-10 overlay program has been modified for MASH to provide "writeable overlays." When a link is to be loaded into the non-resident area of the MASH core image,[2] the overlay supervisor determines by program control whether the current contents of the non-resident area -- containing a copy of the last link loaded -- are to be saved. If the link is to be saved, the non-resident area is written into a temporary on-line file. The user may specify either that a fresh copy of a link be loaded, or that the previously used copy in the temporary file should be used instead. Saved copies of links contain values of all variables local to that link, so that the writeable overlay feature allows program variables to be saved over link changes. This feature therefore provides an effective program name space that can be substantially greater than the actual main memory available, albeit at the cost of the program logic required for manually programmed link replacement and with the restriction that variables local to a non-resident link are only available to the same link or to the resident link.[3]

The use of machine readable codebooks to describe micro data files is explained in the previous chapter and is a central concept within MASH. The description of MASH machine readable codebooks is specified in Appendix 3. A machine readable codebook consists of a sequence of lines of characters, separated by a carriage return-line feed character combination. Each line consists of 70 characters organized in fixed field format. The short line length was chosen so that codebooks might be prepared using punch cards, with 10 columns provided for identification and sequence numbers, if desired.

Several programs are available for processing machine readable codebook files. Two of these programs, BOOKB and BOOKP, are used to build and print codebook files. These programs are maintained on-line with others in a special project library area.

---

[2] The term *core image* refers to the copy of a program as it appears within the immediate access memory of a computer. It was coined when most immediate access memories were based upon magnetized ferrite cores. In recent years other technologies, most often large scale integrated semiconductor circuitry, have been used increasingly in place of magnetic core technology.

[3] Three newer versions of the PDP-10 central processing unit currently offer a hardware virtual memory capability, implemented with hardware paging. Bolt, Beranek and Newman's TENEX system offers each user job 256K words of memory for program name space and offers substantial compatibility with software supported by the manufacturer, Digital Equipment Corporation (DEC). The K10 and KL10 processors introduced by DEC contain paging hardware, and recent operating systems offer up to 256K word virtual space for each job. The current modular structure of MASH makes it likely that if it were implemented on either of these systems, the program would exhibit reasonable locality of program reference.

BOOKB provides an interactive method of creating the contents of a codebook file. BOOKB is an interrogative program; it asks its user the type of line to be specified and then requests all of the information required to complete the line. The output from the use of BOOKB is a machine readable codebook file which can be used to describe a survey data file to be read by MASH.

BOOKP transforms the compact fixed field format of the codebook file into a readable edited printout of the codebook file's contents. A variety of indices to its contents are printed following the description of its attributes. The input required by BOOKP consists of input and output units and file names.

An on-line attribute library is provided for the storage of attribute descriptions that are not part of any machine readable codebook, but which may be created and extracted to describe attributes that are to be imputed in an initial population for simulation or are to be defined during the course of a simulation. The library information is kept in two on-line files, ALCAT and ALLIB. The first file contains an index to the library's contents, called its "catalog," while the second file contains the lines describing the attributes themselves.

Both attribute library files are accessed in random access mode, and are structured as a set of line or card images. Each line of the catalog file contains identifying information for one attribute in the library. Attributes are identified by name alone, but may also be referenced indirectly by author (owner), date of creation, or date of last modification. The library is list structured. Each line of the library contains backward and forward line pointers, followed by the 70 character image of a codebook line. Space in the library file is expanded as additional lines are required. As deletions of lines or entire attributes occurs, the lines released are added to a "free space" list. As additional lines are inserted in current attribute definitions or as entire attributes are entered and catalogued, lines are reclaimed from the free space list. Additional file space is requested from the monitor system when the free space list is empty. Lines that constitute a description of an attribute are chained together in both directions and are linked to the catalog entry for the attribute.

An attribute library manager program, ALMAN, is provided for attribute library manipulations. Users may add, modify, erase, and selectively display attributes in the library. The MASH system extracts attribute definitions from this library in response to the command "INCLUDE LIBRARY ATTRIBUTE[S]...". The attribute library files and their manager program are stored in a separate user area within the PDP-10 on-line file system.

Data Structures

The MASH system creates and interacts with a variety of data files. Some of these files, such as survey data files and machine readable codebooks, are sequential in nature and are processed in that manner. However, most of the on-line files used by MASH require non-sequential access and are handled by a software virtual memory simulator which provides almost unlimited data address space for micro populations and dictionary entities. [4,5] The characteristics of the simulator and its uses are discussed below.

Virtual Memory Simulation

The concept of virtual memory is a product of the *storage allocation problem* in computing and the economics of computer design and production. Denning describes the reasons leading to this problem:

> "From the earliest days of electronic computing it has been recognized that fast access storage is so expensive, computer memories of overall capacity must be organized hierarchically, comprising at least two levels, "main memory" and "auxiliary memory." A program's information (i.e. instruction code and data) can be referenced only when it resides in main memory; thus, information having immediate likelihood of being referenced must reside in main memory, and all other information in auxiliary memory. The storage allocation problem is that of determining, at each moment of time, how information shall be distributed among levels of memory."
>
> "During the early years of computing, each programmer had to incorporate storage allocation procedures into his program whenever the totality of its information was expected to exceed the size of main memory. These procedures were relatively straightforward, amounting to dividing the program into a sequence of "segments" which would "overlay" (i.e. replace) one another in main memory. Since the programmer was intimately familiar with the details of both the machine and his algorithm, it was possible for him to devise efficient "overlay sequences" with relative ease.[6]

In summary, the storage allocation problem arises because the demand for memory resources at any time is quite likely to exceed the main memory available. Since main memory is almost always immediately addressable whereas there are delay costs associated with secondary memory, it was generally necessary in the past to plan an efficient scheme for transporting information among levels of the computer storage hierarchy.

Overlay, or chaining, mechanisms quickly evolved in monitor systems to support moving program text in and out of main memory. Generally such systems allowed program text to be organized in a tree

---

[4] Virtual memory implemented in hardware is not available on the PDP-10 KA10 processor; hence the necessity of simulating it in software. The term "virtual memory" referred to in PDP-10 operating system documentation in [D4] is space allocated on rotational swapping memory for user jobs not currently in main memory. It is therefore virtual to the operating system and non-existent to user jobs.

[5] More recent versions of the PDP-10 system have processors (KI10, KL10, TENEX) with hardware virtual memory. However the address space offered in both systems is still far short of what is required for simulation with populations of meaningful size.

[6] Peter J. Denning, "Virtual Memory." Computing Surveys, Vol. 2, September 1970, p. 153.

structure of some depth, and then provided a mechanism to replace various branches of the tree. In some instances, overlay control was handled manually by requiring explicit overlay replacement in the program text; in others, overlays were handled automatically through subprogram module calls, often at the expense of repetitive program relocation during the loading process. MASH program text is chained by a relatively primitive overlay procedure that only allows trees with two levels of code, resident and non-resident. With the addition of the "writeable overlay" feature, the procedure provides an elementary manually programmed solution to the storage allocation problem for MASH program text and program variables.

In the early days of electronic computing, large data files were generally processed sequentially, and software that supported such processing, such as file sorts and merges, underwent rapid development. As random access storage devices of significant capacity became available, techniques for processing random access were developed, and random access functions were introduced into higher level languages. Since then, indexing functions for random processing have also been institutionalized in forms such as ISAM (*I*ndexed *S*equential *A*ccess *M*ethod) and various data management systems, allowing the applications programmer to concentrate upon program logic rather than management of large data files.

The ability to address large data files as if they were in main (immediate access) memory contributes substantially in certain cases to simplification within MASH of both object module program structure and content. For example, the construction and implementation of inter-unit operating characteristics becomes considerably simplified. To implement all inter-unit characteristics planned for the present microanalytic model, including those that modify population structure such as marriage and divorce, using a sequential population file structure becomes quite complicated; using a random access file structure, such an implementation is possible but requires substantial file bookkeeping by each of the operating characteristic program modules. However, if the entire population can be assumed to reside in main memory, then the implementation of inter-unit operating characteristics differs from that of intra-unit operating characteristics by only one level of indirection -- which is provided by pointers to the other micro unit or units involved.

The concept of virtual memory provides a mechanism for allowing the programmer to treat all program text as well as data as if it were resident in main memory. The term *virtual* is used because this condition is an *illusion;* in fact, an automatic memory management mechanism is provided so that those subsets of information that are to be retrieved or modified are transported into main memory from secondary memory whenever necessary. As described by Denning:

> "... *virtual memory* gives the programmer the *illusion* that he has a very large main memory at his disposal, even though the computer actually has a very small main memory, At the heart ... is the notion that "address" is a concept distinct from "physical location." It becomes the responsibility of the computer hardware and software automatically and propitiously to move information into main memory when and only when it is required for

> processing, and to arrange that program generated addresses be directed to the memory locations that happen to contain the information addressed..."
>
> "As mentioned earlier, virtual memory may be used to give the programmer the illusion that memory is much larger than in reality. To do this, it is necessary to allow the programmer to use a set of addresses different from that provided by the memory and to provide a mechanism for translating program-generated addresses into the correct memory location addresses. An address used by the programmer is called a "name" or a "virtual address," and the set of such names is called the *address space,* or *name space.* An address used by the memory is called a "location" or "memory address," and the set of such locations is called the *memory space.*"
>
> "The price to be paid for there being no *a priori* correspondence between virtual address and memory locations is increased complexity in the addressing mechanism. We must incorporate a way of associating names with locations during execution. To this end we define a function *f*"... (from the address space to the memory space) ... "This function *f* is known as the *address map* or the *address-translation function.*" [7]

Virtual memory is most efficiently implemented if the address translation function is implemented at least in part in hardware. Often this implementation takes the form of a set of *associative registers* which are used to determine whether a given virtual address is currently mapped into real memory and if so, what its real memory address is. If the virtual address is not in memory and is required to be transported, then software is used to schedule and execute this function. Hardware implementation may include structuring the instruction word in such a way that the entire virtual memory may be directly addressed by the address syllable of an instruction, or it may require that (large) virtual addresses be computed indirectly through indexing and/or indirect addressing. A thorough discussion of one virtual memory design, that for the IBM 360/67, appears in [Al]. A more advanced hardware virtual memory system called MULTICS, in which the hierarchy of all on-line storage devices is included in the user's virtual memory space, is described in [C2]. Virtual memory capability is now available on models 135, 145, 158 and 168 of IBM's System 370 computer line, as well as being offered by an increasing number of suppliers.

While the PDP-10 on which MASH is implemented does not support virtual address spaces for user jobs, a restricted virtual space can be simulated through a software module. Such a simulator can provide only virtual space for data, since without an interpreter it is not possible to provide virtual space for program text. Simulation of data arrays is possible, since the simulator can be interposed at the program level between the calling program and the programs that store and retrieve the data.

The MASH virtual memory simulator is programmed in Macro-10 assembly language, and is structured as a subset of common code, plus an assembler macro that expands to provide the code to handle one two dimensional virtual array. The size of the virtual array is limited only by the amount of on-line storage available to the user. The number of virtual arrays that may be active at any one time is limited by

[7] Denning, *ibid,* pp. 156-158.

the number of PDP-10 software input-output channels [8] available which is 16 at present. Virtual arrays are, in fact, PDP-10 on-line files that are kept "open" while in use in order to minimize the transfer time of information between secondary and main storage.

The assembler macro that expands to define code for a virtual array has five arguments: (1) the "name" of the array; (2) a sequential sequencing index used to index external global symbols associated with the array and to distinguish labels in different macro expansions; (3) the page size; (4) the number of page frames to be allocated in main memory; and (5) a bit indicating whether the array is to be referenced directly by word or indirectly by byte.

A virtual array name may be any legal Fortran variable name that is 1 to 5 characters in length. This name is used to generate two entry points to the simulator and to name the on-line PDP-10 file that contains the array. For example, the name FAM is chosen to contain data at the family level. The entry points FAM and SFAM are generated for retrieving and storing data in this array. To the programmer, entry points in this simulator appear as functions and subroutine calls, with some necessary concession to Fortran syntax rules. Instead of writing:

```
DIMENSION    FAM(20,30000)
INCOME = FAM (I,N)
FAM(J,N) = RENTS
```

to retrieve and store data values, the MASH applications programmer writes:

```
CALL INITVA ('FAM', 20, 30000)
INCOME = FAM,(I,N)
CALL SFAM (J, N, RENTS)
```

Thus data retrieval is syntactically identical, but variable assignment requires a subroutine call.

Virtual memory is organized in subdivisions of fixed length blocks, called *pages*. Each page consists of N PDP-10 words, of which the first 2 are for identification and control purposes, and the remaining N-2 contain cells of the virtual array. The first words contains the array's sequence number in the left half of the word and the sequential page number within the array in the right half of the word; this word is checked when the page is read to insure against input addressing errors. The second word contains a sequence

[8] The PDP-10 contains both hardware channels and "software channels." Hardware channels are bidirectional data paths between peripheral devices (auxiliary memory) and main memory that transfer data independent of the central processor once a transfer command or sequence of commands has been initiated. "Software channels" consist of logical 4-bit designators that are used to reference input-output operations referring to a specific device. Such a shorthand reference device economizes on instruction size at the expense of flexibility in input-output operations.

number denoting the order in which the page was last written, which is expected to be useful in analyzing the efficiency of page replacement algorithms for simulation runs.[9]

The choice of page size depends upon the characteristics of secondary memory, the charging algorithms used, the transport time for a single page, the number of page frames available in main memory, and the expected pattern of access of the array. This access pattern is the subset of the *program address trace* that falls within this array. The basic unit of secondary storage on the PDP-10 is the *block* which contains 128 36-bit words. In the absence of tight main memory constraints, a choice of N as a multiple of 128 makes efficient use of secondary storage. Depending upon the dimensions of the array and the relevant subset of the program address trace, a different value of N might produce greater execution efficiency at a cost of increased secondary storage resources.

Page frames are contiguous blocks of storage in main memory into which pages from secondary memory are transported. In MASH, page frames are contained within the virtual memory simulator; each page frame is N-2 words long and is assigned at assembly time to one virtual array. The number of page frames to be generated for each array is an assembly parameter. The simulator maintains statistics that allow the efficiency of the paging activity to be studied. The assembly parameters may then be altered to provide more efficient operation.[10]

Two different forms of virtual arrays are provided by the simulator as the last macro parameter option. The first form provides word structured arrays. Once the dimensions of an array of this form have been declared, then the unit of storage and retrieval is an entire 36-bit PDP-10 word. This corresponds to the main memory organization of the PDP-10.

The second form of virtual array is byte structured. The motivation for this form of memory organization arises from the limited range that many attributes of micro population entities have, and the implicit partial word storage required by them. For example, the attribute SEX has two values, 1 and 2. Using the existing code, only 2 bits are required to contain any value of this attribute. If linear transformations of the attribute are allowed, only 1 bit is required, rather than the 36 bits contained in each PDP-10 word. Similarly, the AGE attribute has a range of 0-99, requiring only 7 bits of storage, and the relationship to family head attribute generally contains not more than 9 distinct values, which translates

[9] Hardware paging machines have no such control words, since they would complicate the address translation function unnecessarily.

[10] Within MASH, the value of N is set to 128 for all arrays to match the size of the fundamental unit of disk storage, the *block*, which is 128 words long. The number of page frames allocated to each array is determined partially by the thrashing analysis performed below, and varies among the arrays.

into a 4-bit storage requirement. These bit strings are called bytes; these bytes may be of variable length.[11] In general, if the range of an attribute consists of V distinct values, then values of the attribute may be stored in B bytes, where B is the smallest power of 2 such that $V<2^B+1$. If the distinct values are not sequential or do not fall into a regular pattern, then the efficiency of this form of representation depends upon the complexity of the mapping and its inverse that maps the distinct values into the sequential, non-negative integers. Since most attributes arising in survey data collection are either coded or have some natural form of ordinal progression, e.g. years in school, in general the above mapping consists of at most an addition or subtraction. Somewhat more efficient data compaction schemes are available through the use of other techniques, but at the expense of greater complexity in the storage and retrieval functions. Meyers [M5] discusses one of these methods, mixed radix arithmetic compaction, and develops the implications of byte structured data files for other forms of file organization such as transposed files. MASH, however, stores attribute values either as they are defined in the original data file or as they are computed by the operating characteristics.

Byte structured virtual memory is addressed indirectly through an array of PDP-10 byte pointers[12] that are stored in main memory. In this organization, each column of a virtual array consists of N PDP-10 words which are further arbitrarily subdivided into a total of M (>=N) variable length byte fields. In order that PDP-10 hardware instructions may be used efficiently for byte storage and retrieval, bytes may not overlap word boundaries. Each word in the column may be divided independently of the manner of division of any other word in the column. A pointer array of length M in main storage is associated with the virtual array; the I'th location of this array points to the byte field in a column corresponding to the I'th byte. Within the implementation of MASH, the I'th byte contains the value of the I'th attribute according to some ordering of attributes for each entity type.

Variable length byte structured virtual data arrays and machine readable codebook information are used by MASH to restructure micro population data into a compact form for storage. As attributes are specified either for extraction from the survey data file or from the attribute library, the minimum byte size required by the values of that attribute -- assuming no compression mapping -- is computed from its codebook value specification. The byte fields required by the M attributes for an entity type are then allocated in a

[11] The terminology used by IBM defines a byte as a fixed length string of 8 bits. This equivalence is a misuse of a generic term. However, its use has become so pervasive that to avoid misunderstanding we will use the expression variable length byte. This does not imply that the bytes are dynamically variable in length, but that byte fields of size from 1 to 36 bits (on a PDP-10) may be defined and intermixed in a virtual array.

[12] A PDP-10 byte pointer is a byte definition word containing a main memory address, indexable and perhaps specifying indirect addressing, and a bit field within the effective address. Byte pointer words are used efficiently by the *load byte* and *deposit byte* hardware instructions to retrieve and store an arbitrary bit field within a word.

compact manner to a block of N PDP-10 words, where N is chosen to be as small as possible. Then M PDP-10 byte pointers are created for the M attributes.[13]

Byte structured virtual memory arrays are accessed indirectly through an array of byte pointers. For example, person data is stored in array PER which is the third array defined within the simulator, and the array PNTR03(128) contains up to 128 byte pointers for person attributes. To retrieve and store the I'th attribute in this byte structured array, the following statements may be used:

```
L = PER (PNTR03(I), J)
CALL SPER (PNTR03(I), J, L)
```

The byte is accessed indirectly through the attribute's byte pointer. Since this mechanism is clumsy and depends upon knowledge of the number and order of attributes in a population prior to coding operating characteristics or performing expensive table lookups, alternate entry points have been defined that allow addressing attributes by name. These entry points are illustrated in the following statements, which retrieve the value of AGE for the person whose data are stored in column PA of the person virtual array and add one to it, storing the result back in the array:

```
A = PERS ('AGE', PA)
CALL STPER ('AGE', PA, A+l)
```

The first argument, 'AGE', is a Fortran character string constant corresponding to the attribute whose codebook name is AGE, and PERS is the alternate entry point that allows addressing by name.

The implementation of this "call by name" feature is accomplished by altering the object code of the calling sequence the first time it is executed to revert to the primary entry point with the correct first argument for the current micropopulation. Attribute names and byte pointers for each population are kept in another virtual array containing codebook summary information for that population. Functions that employ a call by attribute name actually replace the call to themselves by a call to the alternate entry point and replace the address of the character string literal with the address in main memory of the attribute's byte pointer. This replacement need only occur once per function call, since the writeable overlay mechanism preserves the altered object code. Object code modification destroys potential reentrancy for the system, however, and if that were undesirable, an alternate implementation strategy would be to maintain a hash table of attribute names and byte pointer addresses in main memory, and retrieve the byte pointer each time

[13] The structure of the byte pointers created is as follows. The address field contains the offset within the N word record of the word containing the byte, and indexing by one of the PDP-10 general registers is specified. During execution, this register is loaded with the actual main memory address of the page frame which contains the first word of data for this entity. It is therefore required that all N words containing values for an entity reside in the same virtual memory page. This requirement is imposed when a population is created, and the process is terminated if it is violated. The restriction has not affected any work to date, although it introduces an additional minor consideration into any optimization of page sizes for three micro population data arrays.

an attribute reference was made. This alternative would require somewhat more main memory and additional time for each retrieval.[14]

It is helpful that software virtual memory arrays are also addressable as PDP-10 files. Virtual arrays may be "closed" by MASH, retained between jobs as on-line files, "opened" during subsequent runs, and may be processed by other system software. Two virtual arrays discussed below, the user dictionary and its index, are maintained on-line at all times for all MASH users.

The primary cost of using software implemented virtual memory is increased execution time. On a PDP-10 computer with a KA-10 processor, a typical retrieval of a word from real memory using Fortran compiled object code requires execution of 4 instructions and takes approximately 17 microseconds. The same retrieval using the software virtual memory executes 31 instructions in approximately 103 microseconds. The difference in costs due to page transport time are small.

In addition to processing time differences, however, there may also be a substantial secondary real cost to the user of the virtual memory simulator in terms of throughput, or turnaround time. When the number of page transports required becomes a non-negligible fraction of the number of virtual memory accesses performed, then there may be substantial delays inherent in the interruptions in processing caused by the input and output operations of the simulator. These delays are especially likely in a swapping system, in which real memory is often oversubscribed in order to accommodate a large number of users whose computing patterns are expected to have a strong interactive component. Since these costs may be quite significant in terms of turnaround time, it is generally prudent not to treat virtual memory as if it were an extension of immediate access memory, but rather to be aware of the pattern of accesses to it and its implication for the rate of paging. This topic is discussed in more detailed below.

---

[14] An initial implementation strategy for providing attribute referencing by name attempted to bind names to locations at compilation time, but had to be abandoned for reasons of efficiency. It consisted of creating a source file of equivalence statements such as:

```
EQUIVALENCE (PNTR03(9), AGE)
```

containing the equivalence between byte pointers and Fortran variable names that matched attribute codebook names, and then inserting this file into each operating characteristic program module and compiling a version of that module for that population. This procedure is similar in character to Cobol's use of a Data Division at compile time to bind input fields to computational code. This procedure was implemented but became economically infeasible almost immediately for two reasons: (1) the composition of the micro population changed rapidly, requiring frequent recompilation and a proliferation of program modules for different populations; and (2) the number of program modules that required recompilation grew above even the most extreme estimate. In addition, the above strategy required the use of reserved Fortran variable names that could not be determined in advance. While this was temporarily solved by adopting non-overlapping name length conventions for Fortran variables and attribute names, such restrictions were burdensome while constructing the initial operating characteristic program modules and could have led to errors that might not be easily detected.

The software virtual memory implementation described here allows the integration of a number of convenient functions. For example, the PDP-10 monitor system provides *examine* and *deposit* commands that allow a user to display and modify any word of main memory in his job's core image; display is in octal, and alterations are also specified in octal. Two MASH commands provide these same functions in somewhat more convenient form for virtual memory arrays; they are the EXAMINE and DEPOSIT commands. Words may be displayed and altered in any of our modes, alphabetic, integer, octal, and real. For example:

```
EXAMINE PERAD (2,73)  INTEGER;
DEPOSIT ALPHABETIC ABCDE IN CBOOK (7,5);
```

For byte structured arrays, the first subscript is taken to mean the number of the attribute for this entity, and the byte will be retrieved or stored. Full words will be retrieved and stored for word structured arrays.

Another advantage to a software implementation is that array subscript checking may be performed. Subscript checking code is conditionally assembled within the simulator; this code serves as a useful error detection device, and may be removed by reassembly to produce a more efficient module.[15] The simulator maintains the following statistics for each virtual array that is opened:

1. Number of array accesses that refer to the same page as the previous access.
2. Number of array accesses that refer to a page that is resident in a main memory page frame, but not to the page previously accessed.
3. Number of array accesses that refer to a page that is not resident in main memory and must therefore be fetched from secondary storage.
4. Number of write accesses into this array.
5. Number of read accesses from this array.
6. Number of pages transported from secondary storage into main memory.
7. Number of pages transported from main memory back to secondary storage.
8. Maximum page number used.

These statistics give some indication of the relative efficiency of the simulator, and provide data for decisions regarding the size and number of page frames and their allocation among virtual arrays.

[15] Another function which was considered for the simulator was a virtual memory address stop feature, since such a feature could be implemented in software. Such a stop could be made conditional on "fetch" or "store" conditions, and therefore could be used as an additional debugging tool. However such an address stop mechanism could not be expected to be foolproof, since there is no way to protect the integrity of pages in page frames in main memory. The feature was therefore not implemented.

The simulator implements *demand paging*. A page is not transported into main memory until a retrieve or store operation takes place that references the page. *Anticipatory paging,* which is potentially more efficient if the characteristics of the array address trace are known, is attractive for this application, since reference to certain arrays is highly likely to be sequential.[16] However its effectiveness has not been explored.[17]

The simulator currently uses a "least recently used" page replacement policy. Under this policy, when a non-resident page is demanded and all page frames allocated to that array are full, the frame containing that page which was accessed farthest in the past is released. If the write count for that page is zero, the new page overlays the old page; if the write count is positive, the old page must be first transported to secondary storage before the page demanded can be fetched. Several page replacement policies are contained within the simulator; a conditional assembly parameter is used to select a specific policy.

User Dictionary Structure

The MASH user dictionary is contained in two PDP-10 files in on-line secondary storage. Each of these files is created using the virtual memory software described above, and all dictionary operations use this software to process the contents of the files. The files are named "DICT.SIM" and "INDEX.SIM"; the first contains the dictionary entities themselves, while the second contains index information for the dictionary contents. A single pair of dictionary files is used for all MASH users; user dictionaries are logical subsets of these files.[18]

[16] The value of anticipatory paging controlled by user software depends upon the characteristics of the monitor under which it executes. If input-output operations initiated by an anticipatory paging algorithm generally result in a monitor interrupt and transfer of control to another runnable user job, then the gain accruing from the algorithm might be small. However, if the scheduler allows simultaneous compute and input-output operations or if there are few or no other runnable jobs, then the gain in throughput from anticipatory paging could be considerable.

[17] Since this virtual memory is implemented in software, it is relatively easy to study the effects of various paging schemes in detail. The sequence of references to an array during a simulation run may be recorded in a file by adding some optional recording code to the simulator. Then a program that simulates the simulator may be constructed to read this file and determine the effect of modifying the number and size of the page frames and the page replacement policy for that sequence of references. Since this program need simulate only array references, it provides a relatively inexpensive method of tuning the virtual memory simulation process for a specific class of array access patterns.

[18] Although this implementation using two on-line files was initially convenient, it has become increasingly inefficient due in large part to the success of the on-line user dictionary as a concept. The dictionary was initially conceived of as a repository for standardized lists of attributes and macro model equations, and it was not thought that it would be used frequently. In practice, it has been heavily used and users have brought substantial pressure to bear for more powerful extensions and commands than were originally designed. The author believes that this increased use reflects the attractiveness of features similar to those offered by the host time sharing system. However, such increased use has led to the requirement that a very large dictionary file be maintained in on-line storage whenever MASH is used. In addition to this storage requirement, each entity fetch and store is more costly in a large dictionary than in a smaller one, even though the effects of size are somewhat mitigated by chaining entities by owner internally. Concurrent access to the dictionary by concurrent MASH jobs is possible and is anticipated and handled correctly by the dictionary access software. A design that segments user dictionary and index files into individual paired files now appears to be a more efficient long run choice.

Access to the dictionary by multiple MASH jobs can be concurrent but not simultaneous. The virtual memory simulator opens files in "random access" mode, which allows both reading and writing. In order to prevent simultaneous alterations to the files, parallel modification is inhibited, and the files must be closed by one user before they can be opened by another. If the dictionary files are not available to a job, that job is "put to sleep" (using a PDP-10 monitor function) for a few seconds, and then availability is checked again.

The index to the dictionary contains 10 words of information for each entity in the dictionary. It identifies each entity by the name of its owner, the name of its entity type, and its name. An entity is uniquely identified by these three names. The index also contains entries for each active owner, i.e. each owner who owns at least one entity. The index has an initial prefix of 20 words for housekeeping data.

Owner entries in the index are doubly chained together. Each owner entry also contains pointers to the first and last entities that it owns, and all of the entities belonging to that owner are doubly chained together, as well as pointing to their owner entry in the index file. Each entity's entry in the index contains a pointer to the actual entity in the dictionary file. When an owner or entity is added to the dictionary by a series of MASH commands, a new entry is made in the index and is inserted in the appropriate chain. When an entity is erased from the dictionary, it is only logically removed. Its initial word is changed to indicate logical erasure, and its index entry is unchained and also modified to indicate logical erasure. Owners with no entities remain in the owner chain. Erased entities are actually removed from the dictionary when the CONDENSE command is given:

```
CONDENSE DICTIONARY;
```

This command locks all other users out of the dictionary and index files, and then performs a compaction and reorganization of the dictionary file and a corresponding reconstruction of the index.

Entities are retrieved by opening the dictionary files, searching the owner chain for a match and then searching the entity chain for that owner for a match of type and name. The entity is then referenced through the pointer from the index entry. Entities are stored by first performing the same search to determine whether an entity of that description already exists. If so, then unless the entity is protected -- which is recorded in its index entry -- it will be superseded.

An initial dictionary entry length is associated with each entity type. If the initial entity definition fits, this initial length is allocated. If the initial entity does not fit in the default length, it is expanded in fixed increments until sufficient space is available. If an entity is being superseded and will fit in the dictionary space initially allocated, the new definition overlays the old one; if the superseding entity is too long to fit in the previously allocated space, that space is flagged as being erased and new space is allocated for the revised entity. Automatic expansion of dictionary entities is important, since some entity types will grow as research continues, e.g. attribute lists, while others such as time series may be expanded substantially under program control during a simulation exercise.

Although the array dimensions declared for the user dictionary are (1,1000000) providing effectively no capacity limitation, all entities now stored within it are organized into triplets of words, and the dictionary entity buffer within MASH has shape (3,150). Most entity types entered by a user at command level are checked for proper syntax upon entry. If no syntactic errors are detected, the entity is stored in the dictionary in somewhat modified form.[19]

Lists and other entities that are similarly structured are divided into symbols of up to 10 MASH alphabet characters and up to 5 contiguous MASH separator characters. The symbol occupies the first two words of a dictionary triplet, and the separator character or characters occupy the third word. Parameter lists are syntactically no different from lists; for correct interpretation, however, each odd numbered symbol must be a scalar name, while each even numbered symbol must be a real or integer numeric value. The character "-" is interpreted as part of the symbol alphabet for non-algebraic entity types rather than as an operator so that it is recognized as a legitimate part of negative numeric strings, such as <u>-34.2</u>. Time series entities are stored somewhat like lists; the first column of the entity contains the starting and ending years of the series as binary integers in words 1 and 2. All succeeding columns of the entry contain values in symbolic form in words 1 and 2.

Code entities are used to map the real line into one of a series of discrete values in the range of the code. Code definitions entered by a user are converted into a sequence of triplets of real values (v1, v2, v3) where vl<=v2. Each triplet defines the following partial function: for all x such that vl<x<=v2, assign v3 to x. These triplets of real numbers are stored in successive columns of their dictionary entry.

[19] The merging of input and parsing functions was predicated upon a relatively static set of dictionary entries that would be used for production simulations. This expectation has not been met, and the dictionary is now being used more and in different ways than was originally anticipated. A better long run strategy would be to retain the original input string in the dictionary and parse it for syntactic errors but not alter its form. While the cost of such a strategy would consist of having to reinterpret the entity each time it was fetched for use, the benefit of being able to perform text string operations directly upon the original input string is now more important. The entity could also be displayed in exactly its original or modified form, which is not possible starting with the current canonical forms due to the loss of semantically redundant but possibly helpful space and separator characters. Alternatively, both the original string and a parsed form of the entity could be maintained in the dictionary at a cost of substantially increasing the size of the dictionary.

The entity types "attribute", "sample", and "equation" contain algebraic and boolean expressions as part of their definition strings. These strings are parsed at input time into Polish, or parenthesis free, notation and are stored in this form. Each element of the Polish string is tagged with an operator or operand code. The operand codes distinguish between variables, constants, codes, and functions. The operator codes are dispatch indices that are used to branch to the correct interpretive code when the string is evaluated. Time series and references to lagged micro population variables are syntactically indistinguishable from function references at input time and are therefore identified as functions. The interpretive routines distinguish time series variables and micro time series from standard function references at evaluation time.

MASH commands may be stored in the dictionary either by using the DEFINE COMMAND command or by preceding a command with a label followed by a colon. The command entered is parsed like a list with punctuation contained and stored in the dictionary. The command is retrieved for execution via an EXECUTE command, and it is transferred into the MASH dictionary buffer in the same form as originally parsed.[20]

The DEFINE and DISPLAY commands are used to enter entities into the MASH dictionary and retrieve them in a readable form. At the programming level, two subprograms, ENTER and FETCH are available for storing and retrieving dictionary entities. Both have arguments for specifying the entity owner, type, and name, and both return information regarding whether the operation was successfully executed, and if not, why not.

Micropopulation Management

The complete description of a micro population is contained in ten PDP-10 on-line files. Of these, nine are random access files containing virtual memory arrays, and the tenth is a small sequential file containing current population status data. Of the nine virtual arrays, three are byte structured containing data for all interview unit, family, and person entities in the population; four arrays contain locational and structural data linking these entities, and two arrays contain the machine readable codebook and an attribute documentation array for the population.

As described in the previous chapter, each interview unit, family, and person in a population is assigned a *name*. These names are chosen to be consecutive positive integers for programming convenience. Each population entity is also assigned a logical *address*. Addresses are also consecutive positive integers. The address of an entity is the number of the column of the entity data array in which the

[20] Since DEFINE commands cannot be stored and executed indirectly, there is no conflict between non-list syntactic forms being included in a command to be stored in the dictionary.

data for the entity are stored. Thus for example, each column in the person data array contains the value of all attributes including micro time series attributes associated with the person who "lives at" that address, i.e. the column number of that person array. Each column consists of the variable byte length fields that have been created and initialized when the micro population was created. Thus, while the number of columns of each of the data arrays expands to contain the growing population, the number of rows of each array is minimized at population creation time to be the smallest number required to contain values for all attributes named in the INCLUDE and GENERATE statements used to create the population.

Current name-address correspondences and membership and containment information are maintained in four structural virtual arrays. FAMAD (*fam*ily *ad*dresses) and PERAD (*per*son *ad*dresses) each contain 2 rows and a large number of columns. The I'th column of PERAD contains: (1) in row 1, the address of the person whose name is I; and (2) in row 2, the name of the containing entity -- in this case a family -- for this person. The I'th column of FAMAD contains corresponding information for the family level. FMINU (*fam*ilies *in* interview *u*nit) and PRINF (*per*sons *in f*amily) contain the count of and names of the families and persons in interview units and families, respectively. These arrays currently have 20 rows each and as many columns as are required to hold data for the population. The names of families that have been members of interview units in the past but have left the unit are also kept in the FMINU array; names of departed persons are also kept in the PRINF array. Because these arrays have a fixed number of rows, however, only the most recent 19 members' names are retained. If there are more than 19 current members in an entity, the population must be recreated with a larger number of rows in these arrays. Departed members, however, may have their names dropped if capacity is exceeded.[21]

Entities in newly created populations initially have equal names and addresses, i.e. as they are created they are assigned consecutive names and the values of their attributes are stored in consecutive columns. However, any structural demographic change involving a family or person causes one or more family or person entities to be moved to a new address. For every such move, forward and backward address pointers are defined as entity attributes in both the old and the new addresses and in addition, the year of the move and the reason for it are also recorded as values of attributes at both addresses. Since addresses are never reused, it is possible to trace a person's or a family's history back through all genealogical changes that affected it. Since the containment arrays PRINF and FMINU maintain (to within the limits of capacity) the names of all members of the unit present or past, the entire genealogy of the person produced by the

[21] Had these membership lists been implemented in a list structured context, such capacity restrictions would not exist for any one entity, and the amount of virtual space required would have decreased. These features were sacrificed for the direct addressability feature that exists with regular arrays. Capacity problems are insured against by choosing a large maximum for the number of members allowed, and the waste of space caused by the raggedness of the array is not serious since the space is virtual, not real. Transfer time is of course increased for these arrays by choosing this strategy.

simulation may be reconstructed by program instructions within MASH. The browsing command "EXHIBIT HISTORY ..." may be used to trace the history of an entity at the command level.

Most demographic operating characteristics can produce one or more changes in the micro population structure. Persons born are assigned new person names and addresses, and are added to the mother's family. If the mother is unmarried, is contained within an existing family, and has no previous children, then a new family is created within the mother's interview unit, containing the mother and her new child or children. Deaths cause a person to be logically removed from his family. If a family becomes empty, it is logically removed from its containing interview unit. If no other families remain, the interview unit is logically removed from the population.

Marriages may cause a new family and interview unit to be created. If neither bride nor groom is already the head of a family, both move out of their current family and form a new family and interview unit. If either bride or groom is the head of a family already, the other partner moves into that already existing unit. Divorce results in the removal of the male participant from the family; he becomes the head of a newly created family in a new interview unit. Separation of a young person from home results in a new one person family in a new interview unit. Each such structural change in the population is recorded in the genealogical, membership, and containment reference information.

Several of the aspects of the structure of the first seven data arrays describing a population are illustrated in Figure 4-1. The figure depicts an interview unit named 5 containing 1 family named 6 living at family address 7, which in turn contains 4 persons named 19, 20, 21, and 38 living at person addresses 33, 34, 35, and 57 respectively. In addition, person 22, who used to live at person address 36, used to be a member of family 6 but "died" during the simulation. Person 22's death is denoted by a negative address for the person in the PERAD array.

The remaining two arrays that describe a micro population are ATABL(9,1000) and CBOOK(14,100000). ATABL contains contains one column for each attribute in the micro population; the column contains the attribute's name and level, its micro time series specification if any, and pointers to attribute information in CBOOK. CBOOK contains the entire codebook describing the micro population. It is used for supporting on-line browsing commands and for deriving codebooks for survey files. Unlike the previous files, these two files are static; they are reference files and do not change after the population has been created.

Figure 4-1. Micropopulation Structure Example

The tenth file that describes a micro population is the *population status file.* It is a small sequential file that contains the date of creation and last update of the population, the maximum names and addresses dispensed for each entity level, the dimensions of the virtual arrays, the maximum page number of each array, and other such housekeeping information. The population status file is read whenever a micro population is opened, and it is rewritten with updated information whenever a micro population is saved or closed.[22]

[22] The implementation of the virtual memory simulator would have been more self-contained had each array contained within it its own dimensions and length; however, completeness would have been gained at some expense in clean design. In fact, more indirection would have probably been useful. An expanded population status file could contain the file names, owners, and devices of each of the virtual array files comprising a micro population. Such an expansion would make possible a more flexible and automatic system for administering a base of many population files.

The power derived from using simulated virtual memory to store micro population data depends upon three considerations: (1) efficient use of virtual space due to *a priori* knowledge of memory access patterns during simulation; (2) transparency of memory management considerations in programming operating characteristics; and (3) flexibility of extension to multi-sector microanalytic models. These points are discussed more fully below.

Efficient use of virtual space. During one simulation pass through the micro population, interview units are processed in order of increasing interview unit name. Entities contained within interview units are processed with the interview unit; within the unit, processing is in "left list" order. The manner in which entity names and initial entity addresses are assigned ensures that with the exception of new births, the passes for the initial year of simulation will be in sequential order of address in each of the virtual data arrays containing data and structural information for the population. Thus, simulation processing in the initial year differs only marginally from sequential processing in terms of the number of input-output operations required.

As demographic events alter the structure of the population through new births, deaths, marriages, and divorces, the original name-address correspondence will diverge from equality, causing population units to be processed in non-sequential order. The effect of these changes upon processing efficiency depends upon both the structure of the simulator and the nature of the structural changes. In order to evaluate this effect, consider the set of references to any one virtual population array made during one pass of the simulation program. These references may be characterized by the row and column indices of each reference, and are called the subset of the *program address trace* that fall within this array:

$$(r_1, c_1), (r_2, c_2), (r_3, c_3), \ldots, (r_k, c_k), \ldots$$

where the r's refer to the row being referenced, and the c's refer to the columns. Since column numbers are identical with entity logical addresses, all references to the same column number reference the same entity.

Achieving input-output efficiency depends upon minimizing the number of input and output block transfers required to perform a certain processing task. Assuming that all information within the array that corresponds to each entity is contained on the same page, i.e. array columns do not span two array pages, then the address trace within this virtual array may be represented by the columns referenced:

$$c_1, c_2, c_3, \ldots, c_k, \ldots$$

The *program reference string* is derived from the above address trace by: (1) substituting for each column number the number of the virtual page containing it; and (2) eliminating consecutive references to the same page from the resulting string. The program reference string therefore contains the history of the demand for pages made by the simulation program.[23]

A primary factor in evaluating the efficiency of paged virtual memory for supporting population management for simulation depends upon the nature of the program reference string:

$$P_1, P_2, P_3, \ldots, P_k, \ldots$$

as the simulation progresses. Let us examine how this string is affected by each of the current demographic operating characteristics.

During each simulated year, new births are generated, and are given names and addresses one greater than the last name and address assigned. Thus, within a specific year new births have consecutive names and nearly consecutive addresses; the addresses would be consecutive except for intervening structural changes. Births therefore add in each year an independent "thread" of consecutive page references to each simulation pass. Divorces move an individual from an existing family to a new family and a new interview unit. The person removed is processed later in simulation execution time than he otherwise would be, but apart from transferring entity data from one set of columns to another set, no irregularity or discontinuity is introduced into the program reference string. Likewise, separation of a child from parents involves a transfer of person data and the creation of a new interview unit and family, but does not otherwise alter the regularity of the program reference string. Marriages differ in their effects upon the string. If neither of the persons marrying is the head of a family, they are both moved to adjacent new addresses and a new family and interview unit are created. Apart from the initial move, the string's regularity is not altered. However, when one of the partners is a head of family, then the other partner is moved to a new address which will introduce a discontinuity in the program reference string when the new, combined family unit is processed. This discontinuity will be permanent unless the person moves out due to death or divorce at a future time.

Thus, discontinuities in the program reference string are introduced both by births and by marriages between a non-head and the head of a family. However, the discontinuities created in any one simulated year have a regular pattern; they form a continuous sequential thread of page numbers that occasionally interact in processing with the sequence of page numbers in the reference string for the previous time period.

[23] The program reference string only contains a part of the total paging history for the array. It does not include either the multiplicity of reference count associated with each occurrence of a page in the string, nor does it contain information on page modification. This information would be required if an overall measure of computational efficiency were desired, rather than one limited to paging activity.

Furthermore, since new addresses are assigned sequentially without regard to the reason for filling them, the sequences produced by marriage, birth, and divorce are really *one* sequence since that sequence is generated as the simulation pass proceeds. There may be some second order effects, such as a woman with a child born in some prior simulated year both moving to a new address, but these events are infrequent in any one year compared with births and marriages.[24]

The set of active pages at any one time during processing may be termed the program's working set within that array. Minimization of input-output or paging activity during a simulation exercise is roughly equivalent to ordering the computation so that the composition of the working set changes slowly relative to the amount of computing being performed. If the working set changes quite rapidly so that the number of pages required for any significant amount of computation exceeds the number of available page frames, then we say that *thrashing* occurs. Thus whether a program thrashes or not depends to a considerable extent upon the number of virtual memory page frames available to it as well as its program reference string. [25] When the working set becomes greater than the number of page frames allocated for it, thrashing occurs. Until this occurs, the automatic transfer of data between main and secondary storage may be as efficient as sequential file processing.

Thus, ignoring secondary effects and assuming that the current computational sequence is retained, a necessary condition for insuring against thrashing during a simulation year Y (Y=1,2,...) is that the number of page frames for each population array is at least Y+1. These frames may be functionally allocated approximately as follows: (1) one frame will be used to process population entities sequentially and update them in place; (2) one frame will be used to generate the new thread of data derived from births and some marriages that occur during the year; and (3) Y-l frames will be used to process those independent threads in the program reference string that were created in the previous Y-l years of simulation. Of course, the allocation of page frames to specific subsets of pages or the program reference string is not static as suggested, but rather depends upon the *page replacement algorithm* used by the virtual memory routines. MASH currently employs a "least recently used" replacement algorithm, which is believed to give efficient

---

[24] This analysis of paging activity is not applicable to every set of possible operating characteristics. The assumptions under which it is accurate are that all new entities are created during one pass of simulation, that the new entities are stored in a continuous "thread" of entity addresses at each level, and that operating characteristics are almost totally intraunit in nature. The introduction of an operating characteristic that violates either of these assumptions, such as altering the marriage characteristic so that all persons of marriageable age search the entire population for the best possible mate and then a small number of actual matings are chosen, would lead to severe thrashing.

[25] The term "significant amount of computation" is of course relative to the characteristics of the program being executed and the computer environment in which it is executing. Other factors such as the speed of the central processor and the transport time per page and the resources required for transport are important in judging what is thrashing behavior. See Denning [D1] for an excellent discussion of those topics.

performance given prior knowledge about the sequential nature of independent sequential threads of the reference string. Confirmation of this choice, however, must be supplied through empirical studies.

The cost of including Y+1 page frames for population data arrays is not insignificant for moderately large Y. This is in part because page frame storage is allocated statically at assembly time; thus, Y+l page frames are allocated for all simulated years, even though they do not improve efficiency in the initial years. An alternative strategy would be to modify the simulator to allocate and deallocate page frames during execution of a simulation, either according to specific programmed instructions or according to its own analysis of the ratio of paging activity to array accesses.

Another technique, suggested by Orcutt, involves altering the population structure slightly so as to reduce both the number and length of new page threads substantially. Using this technique, empty columns are created in data arrays, where growth in a containing entity might occur. For example, in the person data array PER, at least N columns are allocated to each family when the population is created and are available to contain new persons who join the family. New page threads are therefore created only for additions to families of size N or larger. Currently used population management algorithms implicitly implement such a policy with N = 1. Larger values of N increase the size of the initial data array but reduce the number and length of threads generated by the creation of new entities. For moderately long simulations, this additional technique appears quite promising in terms of increasing program efficiency.

Transparency of Memory Management. The second reason for using simulated virtual memory for population data storage is that it removes file management considerations from the programmer who implements operating characteristics within a microanalytic model. This is of course one of the major advantages claimed for programming within existing virtual memory environments by proponents of automatic memory management through virtual space.

The development of a microanalytic model of the household sector is a complex process, and the implementation of the model for computer simulation can also be complex. As the number of operating characteristics grows, the interdependence among these processes is likely to increase; and as the implications of each operating characteristic become known, the characteristic is likely to undergo iterations of modification and expansion. Confusion and added complexity would be introduced if file management problems were to be introduced in addition to the primary concerns of microanalytic model specification and implementation. Using virtual memory, the management of the population files, or arrays, is transparent to the programmer. The costs incurred are: (1) somewhat greater fetch and store times for data; and (2) for inter-unit characteristics, some additional input and output operations. The first cost can be roughly characterized as exchanging a PDP-10 with 33K words with an average memory cycle time of approximately 2 microseconds for an IBM 709 computer with 5 million words of memory and an average cycle time of 12 microseconds; this is an exchange that is attractive for certain programs. The second cost must be balanced against its alternative, which is the manual management of the queuing, scheduling, and timing tasks that are inherent in any implementation of inter-unit operating characteristics. The author's preference is for using virtual memory in both cases.

Flexibility of Extension to Multisector Micro Models. The third major reason for using virtual memory arrays for micro population storage is the ease of both conceptual and actual extension to more comprehensive microanalytic models. While plans for implementing the current model include representing only the household sector in microanalytic form, for some purposes extensions to other sectors are desirable. For example, labor market demand is now represented primarily by equations in the macro submodel for determining the demand for labor in broad occupational categories; this demand depends upon aggregate variables such as the unemployment rate and is governed by the amount supplied by individuals from the micro model. A more realistic representation of the labor market would be provided by a microanalytic representation of the business sector, consisting of firms producing various product mixes and employing varying quantities of certain occupations as inputs. Operating characteristics in such a model would link employment of individuals by firms at the micro level.

Combining two or more microanalytic models in a computer simulation increases the degree of complexity of the data management problem substantially beyond what is experienced with a single micro model. While using virtual memory to represent other sectors and simplify the problems of intersector interaction may not be the most elegant long run solution for micro data management, it represents a conceptually simple way of extending a basic micro model. Moreover, it allows implementation of such models to proceed in a cumulative fashion rather than requiring more specialized data representations to be modified in order to gain more complete and efficient interaction. Thus, an implementation based upon virtual memory for storage of large microdata sets has the potential for offering a relatively easy transition to more complex microanalytic models that may be valuable even if some degradation of efficiency occurs.

Command Language Interpretation

The MASH command language is implemented as an interpretive system in which the commands are interpreted directly with no intermediate form.

Each command, whether entered directly from the user's terminal or fetched indirectly through an EXECUTE command, is first stored in reformatted form in the system command string buffer. This buffer has shape (3,150), where the first two rows are used to store the string symbols and the third row holds the inter-symbol separator characters (punctuation). A dispatch table in the command interpreting link contains a list of command key words and dispatch indices. The dispatch indices are assigned in groups, with each group assigned to a specific link for further interpretation. If the command is preceded by a label, it is stripped off and used as the name of the command, which is then stored. Unless the associated dispatch index indicates that the command is to be interpreted in the command interpretation link, control is returned to the resident link which calls in the link implied by the dispatch index. Control is then passed to that link, which branches to one of several commands.[26]

Procedures are executed sequentially in the order in which they appear in the procedure definition. The list of command names in a procedure is extracted before any commands are executed; hence, a procedure may modify itself, but the modification will only become effective upon subsequent executions of that procedure. Procedures may not contain the names of any other procedures, nor may they call themselves recursively.

Syntax errors are detected as the command is interpreted. Unless a message is typed to the contrary, no command execution takes place, and control is returned to the command interpreter. If the command is being executed as part of a procedure, the procedure is aborted at that command, and the user is notified of this outcome. Errors in non-algebraic definition commands are detected only after the entire command has been entered. Because of the combining of algebraic definition input and conversion to parenthesis free notation, syntax errors in algebraic expressions may be detected prior to completion of their entry.

Simulation Execution

This section describes the implementation of the execution of the microanalytic object model and the aggregate submodel within MASH.

The implementation of a microanalytic object model within MASH consists of three parts: (1) the control program for simulation; (2) the agendas for invoking operating characteristics; and (3) the program modules that are the object form of the operating characteristics themselves. The control program for simulation, MICSIM, consists of five nested loops for processing micro population entities and conditional transfers to the macro model solution program for any aggregate models that have been prepared for simulation. The five nested loops span, from outside to inside, years of simulation, passes over the

[26] The necessity to pass control to a non-resident link to read and interpret successive commands indicates the shortage of main memory available to support MASH. If more main memory were available, this module would be one of the first to be shifted into the resident link. However, the current memory allocation is balanced in the following sense; during periods of substantial interaction the program must only react to the human user's sense of promptness and link swapping does not degrade this response time, whereas during simulation processing the interpreting link need not be swapped into main memory unless the user wishes to interrupt. Interactive support is therefore expensive relative to simulation activity, but interactive systems *should* provide a responsive user interface even at some larger cost per interaction.

simulation population within one year, interview units in order of increasing name, families within the interview unit, and persons within the family. The 11 control points at which pauses may occur are scheduled at the beginning and at the end of each of these nested loops and at the end of the simulation exercise. This structure is described in detail in the previous chapter. In addition, MICSIM checks if there is an input line from the user's terminal in the input buffer before processing each person; if a line has been entered, a pause is generated before processing the person.[27] Thus, if a user wishes to interrupt a micro simulation, it is sufficient to type "carriage return" while the simulation is proceeding. This action halts the simulation and displays the current simulation pointers.

The simulation control program MICSIM calls a master agenda program, AGENDA, at each of the control points. AGENDA in turn determines which of the non-resident links should be resident in main memory so that control can be passed to the proper link.

Each link containing microanalytic object model modules is organized by an agenda that specifies the actions to be taken during one pass through the micro population. The micro simulation agenda for pass 'k' is contained in the program MAGk. Each micro agenda contains a precisely ordered series of calls to operating characteristic modules at each of the 11 control points, as described more fully in the previous chapter. An example of a micro simulation agenda program is contained in Appendix 4.

The operating characteristic modules are written in Fortran IV according to specific interface standards described in the previous chapter. Interfaces are required with the micro agenda program and with the population data management facilities provided through the virtual memory simulator.

The interface between the operating characteristic modules and the population data arrays is well defined; it consists of a set of twelve entry points which may be divided into two sets of six each. The set which allows "call by name" storage and retrieval using attribute names consists of:

```
K = INTUN  ('attribute-name', intunit-address)
CALL STIU  ('attribute-name', intunit-address, K)
K = FAMLY  ('attribute-name', family-address)
CALL STFAM ('attribute name', family-address, K)
K = PERS   ('attribute-name', person-address)
CALL STPER ('attribute-name', person-address, K)
```

where K is the name of the integer Fortran variable to be retrieved or stored, and all attribute names correspond to the same hierarchical level as the function or subroutine in which they appear as arguments. The second set requires that the location of the attribute's byte pointer appear as the address of the first

[27] The cost to the MASH user of having this interruptibility feature is approximately 1 percent of the processor time required to execute the simulation procedure.

argument, and the entry point names are similar to the above set. As described previously in this chapter, the call by name facility is implemented at first execution time by table lookup followed by entry point and first argument address substitution. Thus, if an attribute is never referenced during a simulation exercise, it may appear as part of non-executed code within an object model program module but it need not be included in the micro population.

All program modules corresponding to microanalytic object model operating characteristics are kept in an on-line library of operating characteristic modules. After the micro model has been specified by the precise formulation of the micro agenda programs, the operating characteristic library is searched and those modules called by any micro agenda are linked to that agenda in the same non-resident link.[28] This operation provides for flexibility in building versions of MASH that contain different microanalytic models. It also provides substantial efficiency during simulation exercises, since all of the object model program modules for one pass over the micro population are brought into main memory together and retained there until the pass has been completed.

The simulation control program, MICSIM, contains two additional features. At the end of the first micro simulation pass, that subset of the micro marriage operating characteristic that performs mate matching and subsequent family restructuring procedures is invoked unless the marriage operating characteristic has been suppressed by the PREVENT command. The names of the participants in the marriage market for the current year have already been written in a file named ALTAR during the first pass; that file will be read and exactly the statistically expected number of marriages will be produced by the second part of the operating characteristic.

At the end of each micro model pass, the aggregate model solution program will be invoked if an aggregate model was specified for this pass by the PREPARE command. The arguments for this transfer of control are: (1) the calendar year for which aggregate results are to be first produced; (2) the pass number after which the transfer of control is occurring; and (3) the number of years which are to be simulated, currently set to 1. The first argument initializes the special aggregate model variable 'T'; the second argument specifies the choice of model; and the third argument specifies the simulation length. Since model solutions are not simultaneous and depend only upon exogenous and lagged endogenous variables, it is possible in one year to extend an aggregate model's solution several years in the future, use the future

[28] Linking operating characteristic modules with micro agendas is accomplished by using standard features of the PDP-10 LOADER program, which both links independent subprograms together and creates an absolute "core image" resident file and chain file. The operating characteristic object modules are kept in a multi-program file which is loaded in library search mode immediately after each micro agenda has been loaded. Only those operating characteristics invoked in the agenda for the pass are loaded with it. The library is ordered so that one sequential search identifies and loads all necessary modules from that library.

projections in micro operating characteristics, e.g. consumer or producer expectations, and then in the next year restart the model solution for several years beginning with the current simulation year. The duration of aggregate simulations has not been incorporated into the MASH command language, but a minor amount of effort would allow the model builder to specify such variable length aggregate projections as part of the model.

While the specification of an aggregate model within MASH is considerably more flexible at the command level than the specification of a microanalytic model, its solution is correspondingly less efficient. Aggregate models are not compiled; the Polish strings corresponding to their equations are evaluated by an interpreter emulating a stack processor such as the Burroughs B5500. [29] For flexibility, the names corresponding to scalars, time-dependent variables, and functions are retained in the strings.

The preparation of one or more aggregate or macro models results in the formation of a model table, an equation table, and the extraction of all Polish strings corresponding to the equations. Constants are then extracted into a constant pool and scalars into a scalar pool. Function names are matched with a table of built-in functions; all names that do not match are assumed to be references to time dependent series in the time series data bank. The scalar table consists of scalar names, current values, and a defined/undefined status variable. If parameter lists are searched, a name match results in the scalar being defined with its corresponding value. The variable 'T' is reserved and denotes calendar time in years, e.g. T=1973.

The association and tracking tables are used to link aggregate model variable names and time series in the aggregate data bank. Entries to these tables are made in the order in which they are entered, and both tables are searched in reverse order during interpretive execution of an equation. No modifications are made automatically to either of these tables during execution, although the search order allows an initial specification to be effectively cancelled by a subsequent specification. [30]

Conclusion

The computer industry has been characterized throughout its history by rapidly decreasing costs per unit of computation and by an increasing diversity of available hardware and software. The computer systems available for implementing microanalytic models have become substantially more powerful since

[29] The algorithm for building the emulator was suggested by the Compilogram board game [B8]. The algorithm was modified slightly and extended to include consistent treatment of function references and unary operators.

[30] Only the relative unimportance of the aggregate model for the end result of our research coupled with the expectation that aggregate models will be small in size justifies this implementation strategy. It is simple and inefficient but is easily implemented with little investment. If either of the above assumptions became invalid, other techniques could be used. For example, the tables could be restructured each time an insertion was made so that the search process could be replaced by direct links between variables and relevant table entries and time series. The simple strategy used was chosen for its ease of implementation and its flexibility for the user; tracking and association specifications may be altered at any time during a simulation, a characteristic thought to be highly desirable in a research oriented computing program.

such models were first attempted, and today the range of possible environments is even greater than in 1970 when the design of MASH occurred.

The design of MASH capitalized upon two then recent and very important developments in computer systems; interactive computing environments based upon resource sharing operating systems, and virtual address spaces for transparent memory management. Other important implementation goals were the construction of a user-oriented command language for microanalytic model construction and execution and data definition independence. Within the framework of a medium scale "time sharing" system, the PDP-10, and based upon Fortran IV and assembly language programming, MASH was created. Its structure is largely faithful to the design criteria specified, with some changes dictated by limitations in the environment and efficiency considerations. And, while MASH represents a considerable evolutionary step in computer systems for microanalytic modelling, there is every reason to expect that the technical and intellectual advances that made it possible will continue to occur and will eventually produce a considerably more powerful successor to it.