MAC-TR-8

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

THE OPS-1 MANUAL

by

Martin Greenberger
and the M.I.T. 15.599 Seminar

May, 1964

ABSTRACT:

The recent attainment and continuing development of personally accessible computer facilities have opened another chapter in the use of machines by man. A number of current research efforts, including Project MAC at M.I.T., are designing new conceptual systems to adapt the emerging technology to a wide range of human activity. Activities relating to management are the concern of a trial system at Project MAC called OPS-1. The OPS-1 system and the experiment that launched it are described in this manual.

"Work reported herein was supported (in part) by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr- 4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government."

This empty page was substituted for a blank page in the original document.

PREFACE

OPS-1 is a trial on-line system for general-purpose use of a compatible time-sharing operation, such as M.I.T.'s CTSS [8]. It is the product of an experiment which I conducted during the Spring of 1964 with a graduate research seminar (15.599) at the M.I.T. Sloan School of Management.

There were eleven students in the seminar.* Several of the students had no prior experience with CTSS, yet they were using it successfully within the first week of the term.

At the initial meeting of the seminar, we reviewed recent progress in the development of personally accessible computer systems, of which CTSS is an early form. We discussed the outlook for management, including applications to simulation methodology, data analysis, decision making, and the design of on-line systems [1, 3-7, 9-12]. Our theme was: much can be done; present technology is already more than adequate; there is a common thread running through many of the possibilities; the greatest need is for basic thinking and a simple, flexible approach; let's get started on a framework that will adapt and take form as it is applied.

During the second meeting of the seminar, I sketched in broad outline what later developed into the OPS-1 system. Previous use of a very preliminary implementation of the system led me to believe that the OPS concept had potential application to a variety of management functions. The OPS approach appeared to provide great power and versatility, while requiring minimal systems programming effort and almost no arbitrary conventions. From the pedagogical point of view, it offered the student a ready opportunity to be original and self-expressive, and gave him a little guidance to help him on his way.

The experimental plan was to have the seminar group program a generally acceptable version of the OPS system as a common project during

---

*John Brach, Gordon Everest, Malcolm Jones, George Klein, James Linderman, Richard Mezger, David Ness, Daniel Thornhill, Mayer Wantman, Robert Welsh, and Stephen Whitelaw.

the first part of the semester, and then spend the remainder of the term applying it to individual projects. We did not expect by this plan to produce any revolutions in management systems or concepts; but we did hope to uncover some central issues and achieve some basic understanding: a hope that was not in vain.

The experiment proceeded as follows. In order to bring up backgrounds to a common level, students with little CTSS experience were paired with students more familiar with the time-sharing system. Each team was assigned to program one or two general-purpose operators or OPS subroutines (described in Chapter III), using either the FAP or FORTRAN language. A standard procedure was established for collecting and consolidating information on the system as it grew. This communication procedure, based originally on the MEMO command of CTSS, led subsequently to the present manual.

Some students came to the seminar meeting the week after the assignments were made with completed programs. Most came, instead, with questions and suggestions: "How would the system do such and such?", "Would it not be better to do it this other way?", and so on.

One possibility would have been to encourage questioning and dissent at this stage, with the risk of the seminar's spending the better part of the semester entangled in endless considerations of system design. Another alternative, the one we chose, was to have a subgroup of the seminar* go into intensive deliberations, taking full account of points made in the seminar meetings, and emerge with final specifications for a first-phase system.

What steered us to the second alternative was the belief that many questions of systems design, such as the need for list structuring, would be better posed and answered a posteriori, or after having had some operating experience, than a priori and in vacuo. We felt that adoption of a straightforward OPS-1 would be in the spirit of the seminar, and could, as a by-product, provide guidelines for subsequent construction of a more

*Greenberger, Jones, Klein, and Ness.

sophisticated system. Simplicity and expediency were our watchwords for the time being.

The specifications that resulted from the meetings of the subgroup found surprising acceptance from the rest of the seminar the following week. Miraculously, the main objections that had been raised were all resolved by the modified system, and certain new features appeared as unexpected bonuses.

The OPS-1 system was programmed within the next two weeks to a state which allowed the individual projects to get underway. Except for some minor improvements, made later, the system was then (March 18) as it is now and as it is described in Chapters I, II, and III of this manual.

Chapter 1 defines OPS-1 and specifies its central components. Chapter II explains how to use and extend the system. Chapter III sets down the standard operators that were programmed by the students during the first weeks of the semester.

Later chapters present some of the individual projects. In Chapter IV, David Ness gives an extension of OPS-1 that incorporates several CTSS supervisory commands, and thus provides the facility for the OPS system to modify itself and its storage map.

In Chapter V, James Linderman extends OPS-1 in the opposite direction to obtain a first approximation to a "live" on-line programming facility. He calls his system OPTRAN.

In Chapter VI, Robert Welsh details an adaptation of OPS-1 to provide an on-line simulation capability which he refers to as OPSIM.

Additional chapters report on applications to management areas, including an automated stock exchange, an on-line project scheduler, a national credit exchange, a mechanized system for accounting and budgeting, an on-line statistical processor, and a simulation of activities on the floor of a savings bank.

The manual concludes with some discussion of how the OPS system might be improved, together with a description of some specific modifications that have already been programmed and tested.

Martin Greenberger

This empty page was substituted for a blank page in the original document.

TABLE OF CONTENTS 5/15/64

This empty page was substituted for a blank page in the original document.

This empty page was substituted for a blank page in the original document.

This empty page was substituted for a blank page in the original document.

Note

In order to keep this manual down to manageable size, reports on the following student projects have not been included.

1. A National Credit Exchange, by Malcolm Jones
2. Model Building, by George Klein
3. TELSIM--The Simulation of a Banking Floor Using OPS, by Richard Mezger
4. Modifications to the OPS System, by Daniel Thornhill

The omission of these chapters does not reflect on their quality in any way.

Also omitted from all chapters are program listings and detailed operational descriptions.

This empty page was substituted for a blank page in the original document.

INTRODUCTION

System Concepts

In setting out to design the on-line system, we were influenced by certain basic concepts that seemed important to us at the time. Specifically, OPS-1 has the following features. Our experience with OPS-1 has reaffirmed for us the importance of these features, and we regard most of them as essential to the success of any similar on-line system.

Simplicity

The system is simple to use and simple to understand, even by the relatively uninitiated. Complications can extract a high price from system resiliency, and they were avoided wherever possible. The user may add his own complexity as the needs of his application require, but normally this will not complicate the internal structure of the system itself, and it will not reduce overall flexibility.

Versatility

Unnecessary rules, conventions, and formats also were avoided. Thus the user is not forced into thinking about his problem in artificial or unnatural terms. He has ample room to express his problem and himself, and he has the freedom to be creative.

Generality

A wide range of applications is possible, from the structuring of simulation models, decision procedures, problem solvers, and information processes, to the design of programming systems and on-line operations, including CTSS itself. This generality is particularly significant in that it facilitates the construction of hybrid systems which combine two or more kinds of elements; for example, a real-time traffic controller with simulation elements and data analyzers; an on-line programming system with debugging aides and an instant-execution feature; a job-shop decision system with simulation, information processing, and problem

solving elements; or an information utility with all of these elements. It is a safe wager that man-machine organizations of the future will be large hybrid systems of this general character.

Individuality

In customary use, the system is expanded by the user to a form that matches the specific requirements of his application. The basic system performs like a catalyst in this process.

Expandability and Open-Endedness

Typically, the expansion of the system is by small increments which extend its usefulness, but not its structural complexity. There is no theoretical limit to how far the system can be expanded without its getting bogged down. Expansion may be gradual and continual over an indefinite period of time.

Modularity

The elements of the system are like pluggable parts. Except for the few main components, each part can be unplugged without disrupting or affecting any other part. Most of the elements are called operators. They do not call on one another directly, thus ensuring separation of function.

Immediate Execution

An operator can be executed as it is referenced. This feature facilitates the editing, probing, and testing of simulation models and information procedures while they are under construction. With the help of system operators, a model or procedure can be working almost as soon as it has a meaningful structure, and well before it is in final form.

Conditional Recalling

As elements of the system are referenced and examined in compound patterns, the patterns are automatically "remembered" for possible addition to the system.

Graduated Guidance

Helpful handholding is offered the user by the computer at first, but this guidance can be abbreviated as the user becomes more adept and sure-footed.

LIST OF REFERENCES

1. Anshen, M. and G. L. Bach (ed.), Management and Corporations 1985 (McGraw-Hill, 1960).

2. Corbató, F. J. et. al., "An Experimental Time-Sharing System," Proceedings of the Spring Joint Computer Conference (National Press, Palo Alto, California, 1962).

3. Greenberger, M., "The Computers of Tomorrow", The Atlantic Monthly, May, 1964.

4. Greenberger, M., (ed.), Computers and the World of the Future, (M.I.T. Press, 1962).

5. Greenberger, M., and H. W. Johnson, "Automation, Management, and the Future", The Technology Review, (M.I.T., May 1963).

6. Leavitt, H. J. and T. L. Whisler, "Management in the 1980's," Harvard Business Review, 36:6, Nov.-Dec., 1938.

7. Licklider, J. C. R., and W. E. Clark, "On-Line, Man-Computer Communications," AFIPS Proceedings, 1962.

8. The M.I.T. Computation Center, The Compatible Time-Sharing System: A Programmer's Guide, (M.I.T. Press, 1963).

9. Philipson (ed.), Automation--Implications for the Future (Vintage Books, 1962).

10. Shultz, G. P. and T. L Whisler (ed.), Management Organization and the Computer (Free Press of Glencoe, 1960).

11. Simon, H. A., The New Science of Management Decision (Harper, 1960).

12. Sprague, Richard E., Electronic Business Systems (Ronald Press, 1962).

This empty page was substituted for a blank page in the original document.

Chapter I

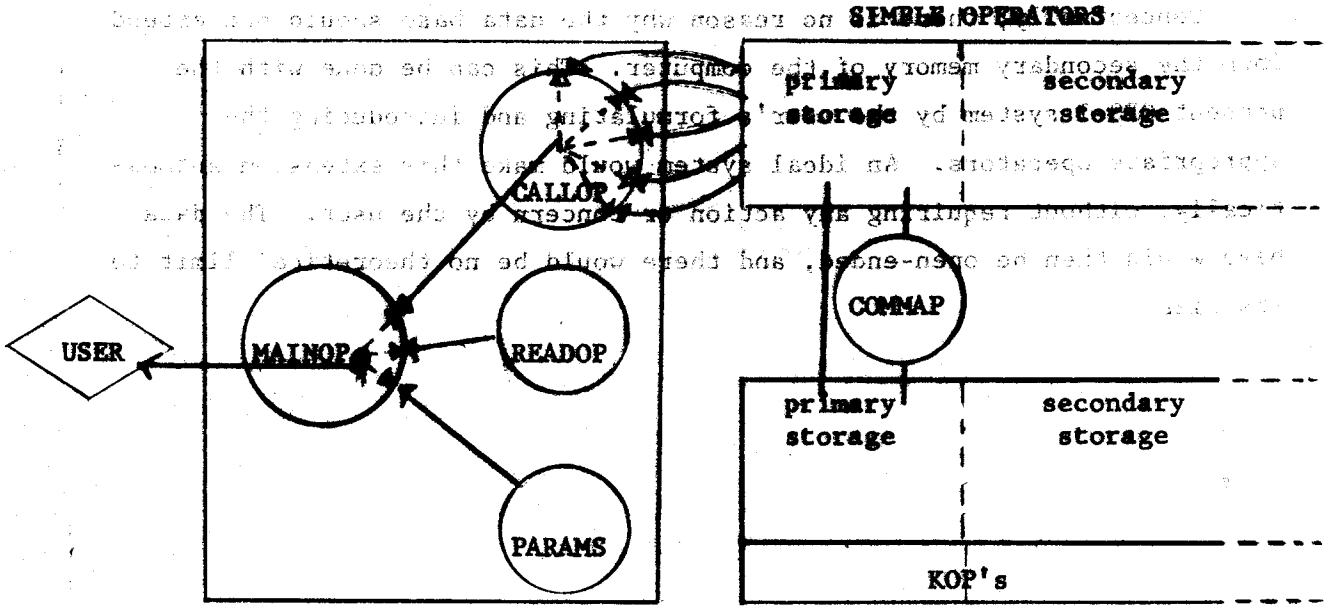
This empty page was substituted for a blank page in the original document.

SYSTEM DEFINITION

The Structure of OPS-1

The system has a basic structure that is easy to visualize. First, there is a common body of information, called the data base, that occupies one section of computer memory; then there is a set of simple operators that range over the data base (they are subroutines that were written originally either in FAP or FORTRAN before being compiled into machine language); and finally, there is a central mechanism that allows the user to execute and compound the simple operators in flexible combinations.

The structure of the system is shown schematically in the diagram below. The principal components of the system will be described briefly now, and in some detail in later pages.



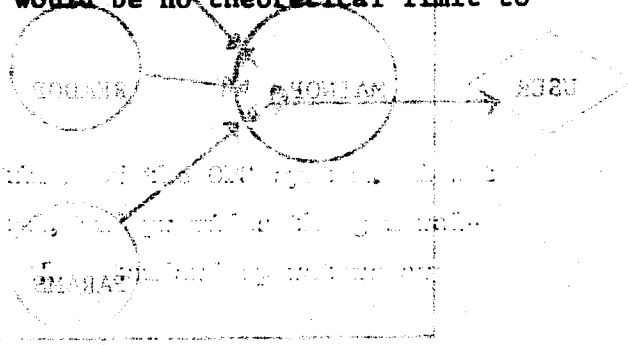
Data Base

The data base is the information center of the OPS system. It contains information of joint relevance to any two or more separate elements of the system. Its data may be arranged as single numbers, multi-dimensional arrays, or list structures.

The first part of the data base is associated with the basic system itself. The remainder is added by the user according to the needs of his application. It becomes a representation of the informational content or environment of his problem. For a simulation model, it is a portrayal of the process being modeled; for a real-time operation, it is a record of the present state of the system; and for an information processor, it is the bank of data that is being analyzed and manipulated.

At the present time, the entire data base is located within common storage in the primary memory of the computer. A map of the data base is associated with each operator, as well as with COMMAP, a service routine for operators. This provides the operators with the means for finding, moving, and modifying values within the data base.

Conceptually, there is no reason why the data base should not extend into the secondary memory of the computer. This can be done with the present OPS-1 system by the user's formulating and introducing the appropriate operators. An ideal system would make this extension automatically, without requiring any action or concern by the user. The data base would then be open-ended, and there would be no theoretical limit to its size.



Operators

Operators are the functional subroutines of the OPS system. A set of standard operators comes with the system, and provides the general-purpose services required by most users. The standard operators are described in Chapter III.

A user builds up the OPS system to fit his particular needs by enlarging the set of standard operators. That is, he programs and compiles his own operators, using a language such as FORTRAN or FAP, introduces them to the system, and makes corresponding additions to the data base. The precise procedure he follows in taking these steps is spelled out in Chapter II.

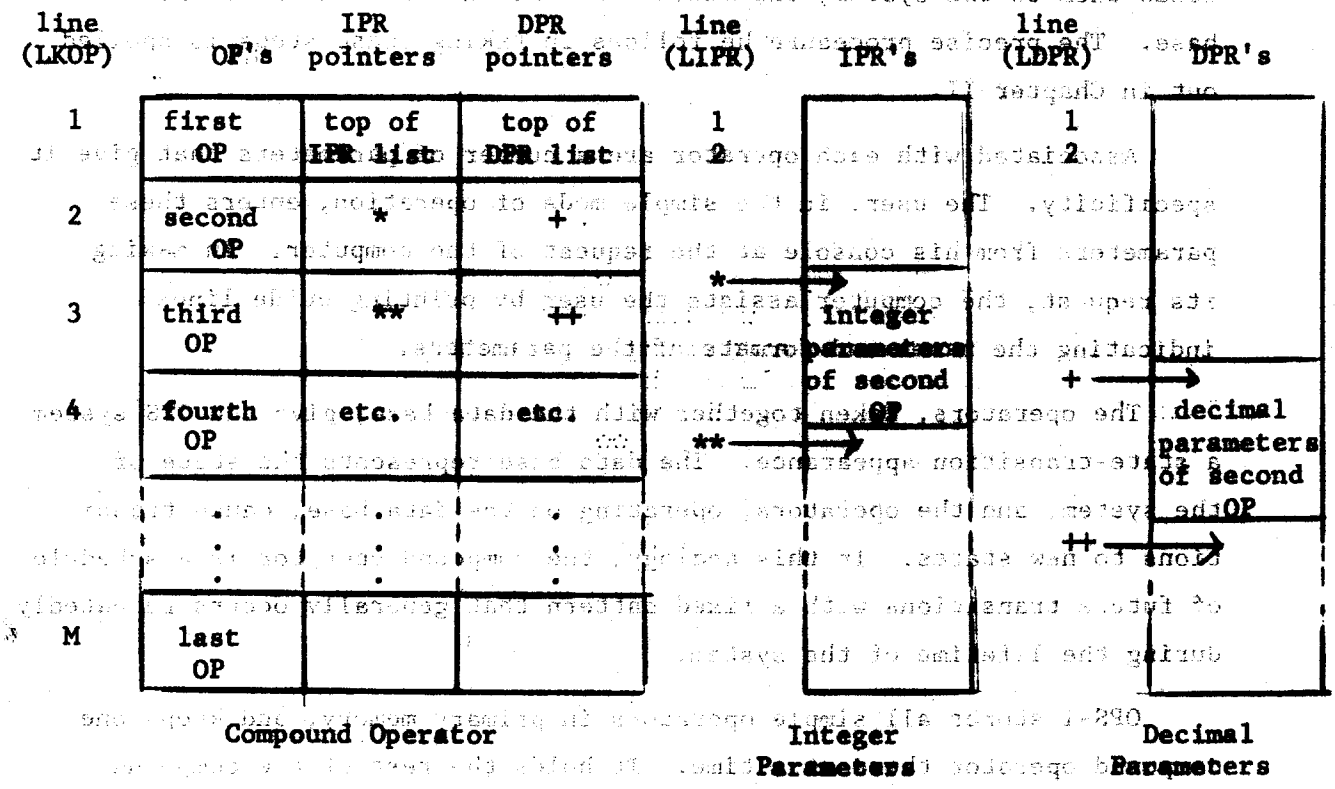
Associated with each operator are a number of parameters that give it specificity. The user, in the simple mode of operation, enters these parameters from his console at the request of the computer. In making its request, the computer assists the user by printing guide lines indicating the names and formats of the parameters.

The operators, taken together with the data base, give the OPS system a state-transition appearance. The data base represents the state of the system, and the operators, operating on the data base, cause transitions to new states. In this analogy, the compound operator is a schedule of future transitions with a fixed pattern that generally occurs repeatedly during the lifetime of the system.

OPS-1 stores all simple operators in primary memory, and keeps one compound operator there at a time. It holds the rest of the compound operators in secondary memory, and brings them in as needed. A preliminary improvement of OPS-1 also brings in the simple operators as needed. Ideally, compound operators could be used to schedule the dynamic allocation of simple operators by helping to anticipate when they will be required.

Compounding Operators

Simple operators may be strung together to form a compound operator, or KOP, and the KOP may be assigned an identification number and retained in list form as a permanent addition to the system. The KOP is represented by an Mx3 array (with structure as shown in the diagram below) plus two linear arrays for the associated parameters. These arrays are stored as part of the data base, and may therefore be modified by operators. That is, the operators can modify themselves and the KOP's.



Modes of Operation

The OPS-1 system is operated by the user from his console. At any given time, he may, by his own choosing, operate in one of six possible modes. The modes are as follows:

1. Simple execution: execution of a simple operator (i.e., of an OP).

2. Create with execute: creation of a compound operator by adding an OP to the KOP list. The OP is executed and guide lines are printed.

3. Create without execute: same as 2, but without execution of the OP.

4. Compound execution with parameter entry: execution of one or more OP's of a KOP list. Each OP requests entry of its parameters from the console by printing guide lines, and stores these parameters in the appropriate locations of the parameter lists. A carriage return at the console (without the entry of any parameters) causes the parameters previously stored in the lists to be used by the OP.

5. Compound execution without parameter entry: same as 4, but with each OP automatically using parameters already stored in the parameter lists.

9. Blind create: same as 3, but without guide lines. The user is responsible for knowing the number and format of the parameters of all OP's in the KOP he is creating.

An Example

To illustrate how the OPS system might be used in a management context, let us consider an oversimplified example. Weekly sales data for the last twelve months have been collected and turned over to Mal, a new employee in the marketing department. Mal has been requested to develop a technique for forecasting future sales. He decides to try the exponential smoothing rule, $F_{t+1} = aS_t + (1-a)F_t$, where F_t is the forecast of sales in week t, S_t is the actual reported sales in week t, F_{t+1} is the forecast to be generated for the next week, and a is the smoothing constant. Mal wants to see if there is a smoothing constant that generates forecasts close to the actual weekly sales. As he thinks about this problem in the framework of the OPS system, he recognizes the need for three basic operators:

1. An input operator that reads data into a specific area of the data base.
2. A general arithmetic operator that can add, subtract, or multiply any of the variables in the data base.
3. A print operator that will write out the value of any variable in the data base.

Mal decides to determine whether the forecasts are good or bad as he sits at the console, and not to program this determination into his procedure.

Mal could now write these three special operators, but he first glances at the list of basic OP's in Chapter III of the OPS-1 manual and notes that the MODIFY OP (OP15) provides both the input and print functions that he desires. Glancing at Chapter V, he also notices that OPTRAN could be used to compute the forecasts.

However, he wants to get some experience writing his own operator, so he decides to write a general arithmetic OP. This operator could be programmed in a number of ways, but he decides on the following approach. He defines several data registers X_1, X_2, X_3, - - - X_9 that can contain input variables, constants, intermediate results, or final results. The

OP will have four parameters, all integers. The first parameter tells where to put the result, the second where to get the first operand, the third what the arithmetic operator is, and the fourth where to get the second operand. This might be stated as $X_i = X_j \theta_n X_k$ where X_i, X_j and X_k are data registers, and θ_n is either addition, subtraction, multiplication, or division. The corresponding parameter values are i, j, n, and k.

Mal programs this OP in FORTRAN, compiles it, adds it to the system as OP21, and checks it out using mode 1. Now he starts to write a compound OP. He can do this in either mode 2 or 3. He decides to use mode 2 so as to get an immediate check on the correctness of his process. First, he writes four OP15's to read in the constant 1, the constant a, the old forecast, F_t, and the last week's sales, S_t. Then, he writes four OP21's to calculate the forecast, F_{t+1}. Finally, he adds an OP15 to print out the result.

Satisfied with his 9-line KOP (see page 1.17), Mal decides to save it for safe keeping on a simulated tape. To do this he executes OP17 in mode 1. Now he is ready to return to the first line of the KOP and execute it repeatedly in mode 4, using the new values of F_t and S_t.

After repeating this KOP a few times, Mal realizes that some improvements are possible. First, he uses a MOVE OP (OP16) to replace the old forecast, F_t, with the new forecast, F_{t+1}, at the bottom of the KOP. He then adds a branch OP to loop back to the beginning of the KOP. To make these changes he reverts to mode 3. After a few more cycles through this new KOP, he realizes that there is only one OP (the OP15 that reads in the actual sales S_t) that requires a new parameter each time. Only this OP needs mode 4. All the other OP's have fixed parameters and can be executed in mode 5. Mal therefore decides to run in mode 5 and simply use two MODIFY OP's (OP15) to switch between modes 4 and 5 at the appropriate times (see page 1.17). Now he can automatically cycle through the KOP several times for each trial value of the smoothing constant. New smoothing constants may be inserted using mode 1.

This example could be extended to show how Mal might build up a more complex forecasting procedure with the help of the computer, but that would bring us beyond the purpose of the simple illustration that was intended.

Original KOP

OP15 ---------- read constant 1
OP15 ---------- read constant a
OP15 ---------- read forecast F_t
OP15 ---------- read last sales S_t
OP21 ---------- compute (1-a)
OP21 ---------- compute (1-a) x F_t
OP21 ---------- compute a x S_t
OP21 ---------- compute F_{t+1}
OP15 ---------- type out F_{t+1}

Revised KOP

OP15 ---------- read constant 1
OP15 ---------- read constant a
OP15 ---------- read forecast F_t
OP14 ---------- end of branch
OP16 ---------- change to mode 4
OP15 ---------- read last sales S_t
OP16 ---------- change to mode 5
OP21 ---------- compute (1-a)
OP21 ---------- compute (1-a) x F_t
OP21 ---------- compute a x S_t
OP21 ---------- compute F_{t+1}
OP15 ---------- type out F_{t+1}
OP16 ---------- move F_{t+1} to F_t
OP13 ---------- branch back to OP14

Central Mechanism

The diagram of the OPS-1 system on page 1.10 indicates the role that the central mechanism plays in OPS-1. It is the control center that switches the user to different parts of the system. The MAINOP routine actually does the switching.

At the beginning of each cycle, the user is connected to READOP and asked to issue instructions. If he specifies mode 9, he is connected to PARAMS with a direct line for entering parameters into the IPR and DPR lists. No guide lines are printed.

In all modes, the user is connected via CALLOP to the OP specified in his instructions. Except for modes 5 and 9, he can then communicate with that OP, or through that OP, with the data base. When the OP has completed its function, it returns control via CALLOP to MAINOP.

In addition to connecting the user to CALLOP, READOP and PARAMS, the MAINOP routine creates or executes compound operators (KOP's) and checks for errors.

COMMAP is a special table-look-up routine that permits an OP to communicate with any specified part of the data base.

The above routines are explained in detail in the following pages.

MAINOP

Purpose

The program MAINOP MADTRN is the executive program for the OPS system. It has the following functions:

1. Asks what operator you wish to execute next.
2. Executes the operator if in mode 1.
3. Creates a compound operator if you are in modes 2, 3, or 9.
4. Executes a compound operator if you are in modes 4 or 5.

Operation

To do these functions, MAINOP makes use of the three system subroutines CALLOP, READOP, and PARAMS. It uses READOP to find out what operator you wish to create and/or execute next. It uses CALLOP to transfer control to the specified operator. The operator is specified by the system variable MOP. It uses PARAMS to read in parameters for an operator if the mode is set to 9.

MAINOP creates compound operators (KOPs) by creating a table. There is one line in the table for each simple operator. Each line contains three pieces of information: (see page 1.13)

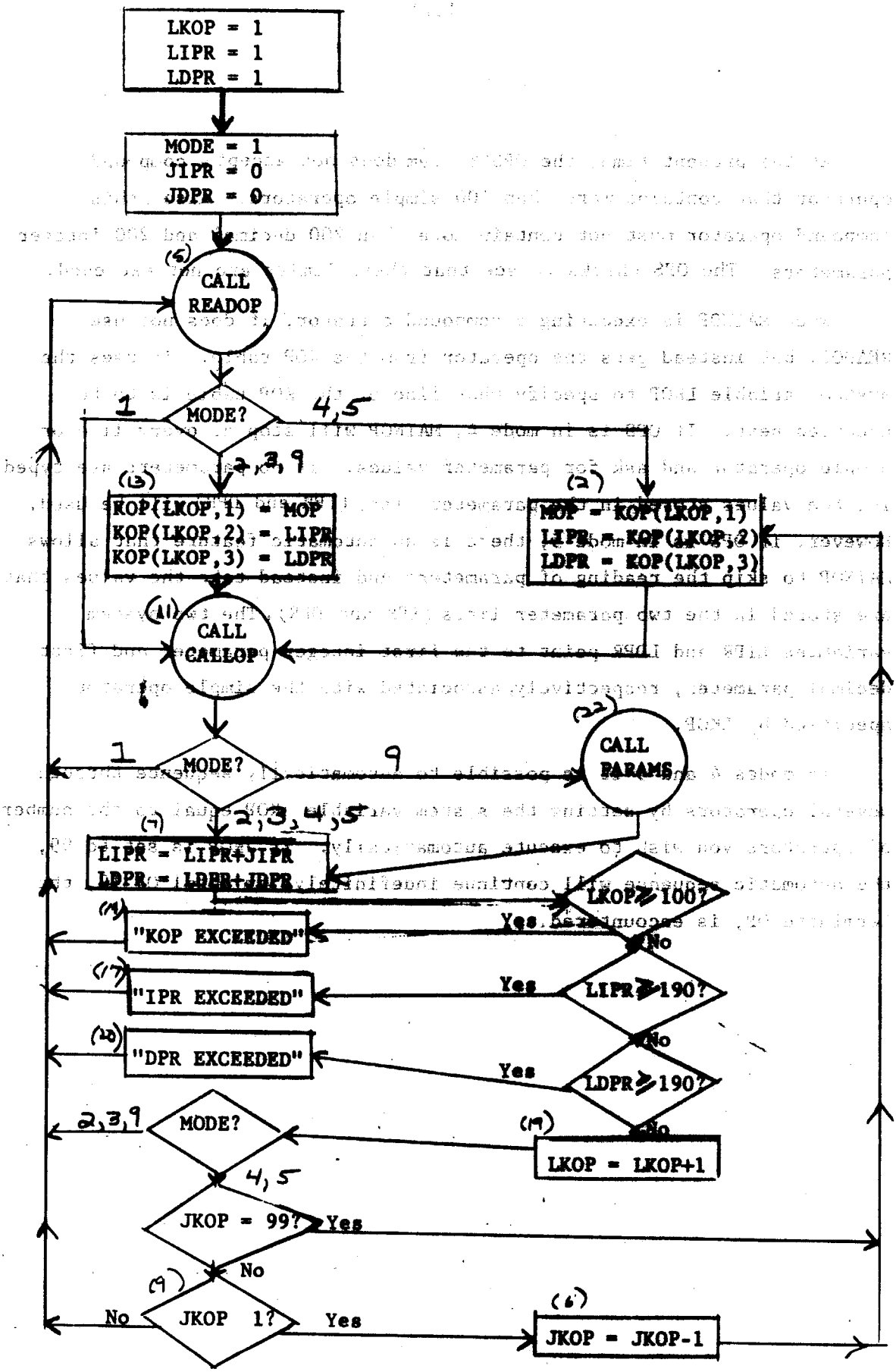
1. The name of the simple operator--a two-digit integer.
2. A pointer to the beginning of a list of integer parameters associated with the simple operator.
3. A pointer to the beginning of a list of decimal parameters associated with the simple operator.

To update the pointers, it makes use of two system variables, JIPR and JDPR. These are the number of integer parameters and the number of decimal parameters used by the simple operator.

At the present time, the OPS system does not accept a compound operator that contains more than 100 simple operators. Also, this compound operator must not contain more than 200 decimal and 200 integer parameters. The OPS checks to see that these limits are not exceeded.

When MAINOP is executing a compound operator, it does not use READOP, but instead gets the operator from the KOP table. It uses the system variable LKOP to specify what line of the KOP table is to be executed next. If OPS is in mode 4, MAINOP will stop at every line or simple operator and ask for parameter values. If no parameters are typed in, the values stored in the parameter lists (IPR and DPR) will be used. However, if OPS is in mode 5, there is an automatic feature that allows MAINOP to skip the reading of parameters and instead take the values that are stored in the two parameter lists (IPR and DPR). The two system variables LIPR and LDPR point to the first integer parameter and first decimal parameter, respectively, associated with the simple operator specified by LKOP.

In modes 4 and 5 it is possible to automatically sequence through several operators by setting the system variable JKOP equal to the number of operators you wish to execute automatically. If JKOP is set to 99, the automatic sequence will continue indefinitely, or until OP 19, the terminate OP, is encountered.


LKOP = 1
LIPR = 1
LDPR = 1
MODE = 1
JIPR = 0
JDPR = 0
(6)
CALL READOP
MODE?
1
4,5
2,3,9
(13)
KOP(LKOP,1) = MOP
KOP(LKOP,2) = LIPR
KOP(LKOP,3) = LDPR
(2)
MOP = KOP(LKOP,1)
LIPR = KOP(LKOP,2)
LDPR = KOP(LKOP,3)
(11)
CALL CALLOP
(22)
CALL PARAMS
MODE?
1
9
2,3,4,5
(7)
LIPR = LIPR+JIPR
LDPR = LDPR+JDPR
LKOP≥100?
Yes
(14)
"KOP EXCEEDED"
No
LIPR≥190?
Yes
(17)
"IPR EXCEEDED"
No
LDPR≥190?
Yes
(20)
"DPR EXCEEDED"
No
(19)
LKOP = LKOP+1
MODE?
2,3,9
4,5
JKOP = 99?
Yes
No
(9)
JKOP 1?
No
Yes
(6)
JKOP = JKOP-1

READOP

Purpose

The function of READOP is to allow the user to specify what action he wishes to have occur next.

Operation

If MODE is 1, 2, or 3 READOP does the following:

1. Prints current mode (MODE) and line (LKOP).
2. Prints a line in the form XX X XX, where

   XX = the next operator you wish to enter if you are going to be in modes 1, 2, 3 (MOP); or= the number of operators you wish to automatically execute in modes 4 or 5 before you again come back to manual operation (JKOP)*,

   X = the mode; be sure it corresponds to the way you intend the above to be used,

   XX = the line number at which you desire to

   a. start execution in modes 4 or 5,

   b. insert an operator into a KOP in modes 2 or 3.**
3. Reads in the operator number (or JKOP), mode and line number.
4. Echoes back numbers just read as a transmission check.
5. If MODE $\neq$ 0, sets mode to new number.
6. If LKOP $\neq$ 0, sets new line number and also sets the new LIPR and LDPR pointers.

---

*This jump (JKOP) can be used in both mode 4 and 5. In mode 4, you will be asked for parameters by each operator with no opportunity to change modes until all operators within this jump have been furnished parameters and executed. In mode 5, the stored parameters are used and again you can not stop the sequence until jump has been completed.

**If you set the line equal to something less than where you are presently, you will be writing over the old KOP, and hence, lose the relation to anything below the number you set.

7. Sets MOP or JKOP depending on whether new mode is 1, 2, 3, 9, or 4 and 5, respectively.

8. Returns control to MAINOP.

If MODE is 4 or 5 READOP does the following:

1. Prints next operator in the KOP table, the mode, and line number.

The subsequent actions are the same as steps 2-8, previously described.

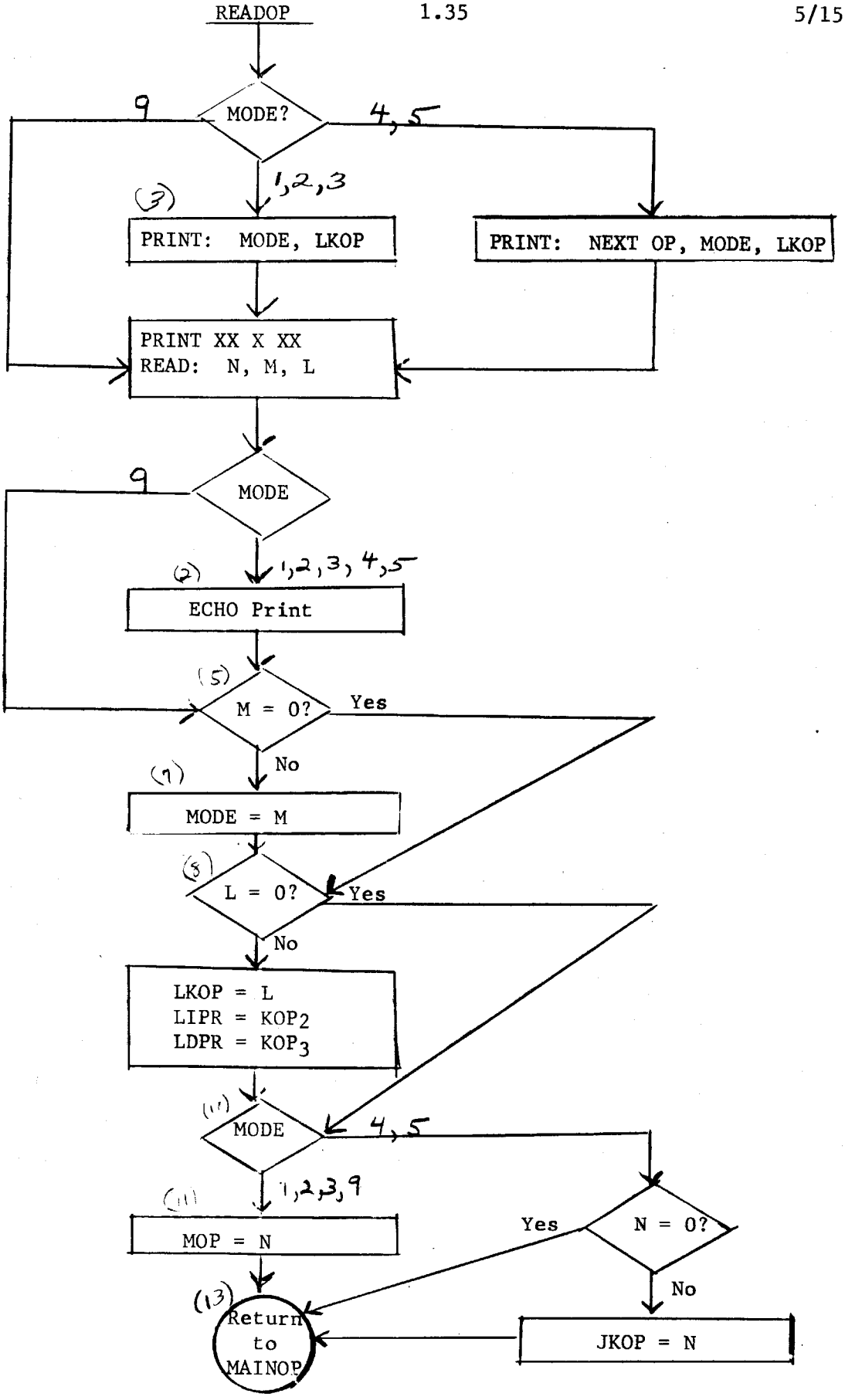
If mode is 9, READOP does the following:

1. Prints a line of the form XX X XX.

2. Reads in next operator, mode and line numbers.

The subsequent actions are the same as steps 5-8 previously described.

If any of the three fields represented by XX X XX are left blank, or if a 0 is typed, no change is made to the system variables associated with those fields. Thus, their previous values are used. This feature is particularly useful in modes 4 and 5 when JKOP = 14; then, typing a single carriage return provides a single step operation.


MODE?
9
4,5
1,2,3
(3)
PRINT: MODE, LKOP
PRINT: NEXT OP, MODE, LKOP
PRINT XX X XX
READ: N, M, L
MODE
9
1,2,3, 4,5
(2)
ECHO Print
(5)
M = 0?
Yes
No
(7)
MODE = M
(8)
L = 0?
Yes
No
LKOP = L
LIPR = KOP2
LDPR = KOP3
(10)
MODE
4,5
1,2,3,9
(11)
MOP = N
Yes
N = 0?
No
(13)
Return to MAINOP
JKOP = N

CALLOP

Purpose

The purpose of the CALLOP routine is to transfer control to the OP that the user has specified in READOP, or to the next OP in the KOP list. CALLOP also handles error returns from OP's.

Operation

CALLOP operates as follows: when called the first time, the routine immediately branches to TIME1. First the transfer to TIME1 is replaced so that this part of the process will be skipped in subsequent calls. Then we save the return to MAINOP and depress the console input level. Then the normal CALLOP is executed.

The normal CALLOP saves index register 1, then sets up the sense indicator register so that FAP subroutines may detect MODE (if they want to) using the SI. Then we create a transfer to OP+C(MOP) which will contain a transfer to OP(MOP). Then this transfer is executed.

At BRRET (return from a break signal) we re-depress the console input level and go to RETRN.

At RETRN we store zero in JIPR, JDPR and JKOP. Then we test to see if we are in mode 1. If we are we return to MAINOP. If we are not in mode 1 we first decrement LKOP in anticipation of MAINOP incrementing it. Then we return to MAINOP. RETRN may thus be used to return to MAINOP without changing pointers (for example when an error occurs, see 2.60).

CALLOP demands that all subroutines from OP00 to OP99 be present. Of course we rarely will have all 100 of the subroutines in memory. In order to circumvent these difficulties we make use of the fact that the BSS loader will, if presented with several definitions of one program, simply accept the last.

SUBCAL FAP defines all subroutines from OP00 to OP99. If there is no other version of a given subroutine loaded, then CALLOP will actually call SUBCAL. If a subroutine has actually been loaded then this definition will supersede the definition in SUBCAL and that routine will be called instead.

Thus if we call an operator which has not been defined we enter at OPXX. We begin a sequence of operations which leads us to the message OP. AB IS NOT YET DEFINED and we return to MAINOP through RETRN (thus not incrementing pointers).

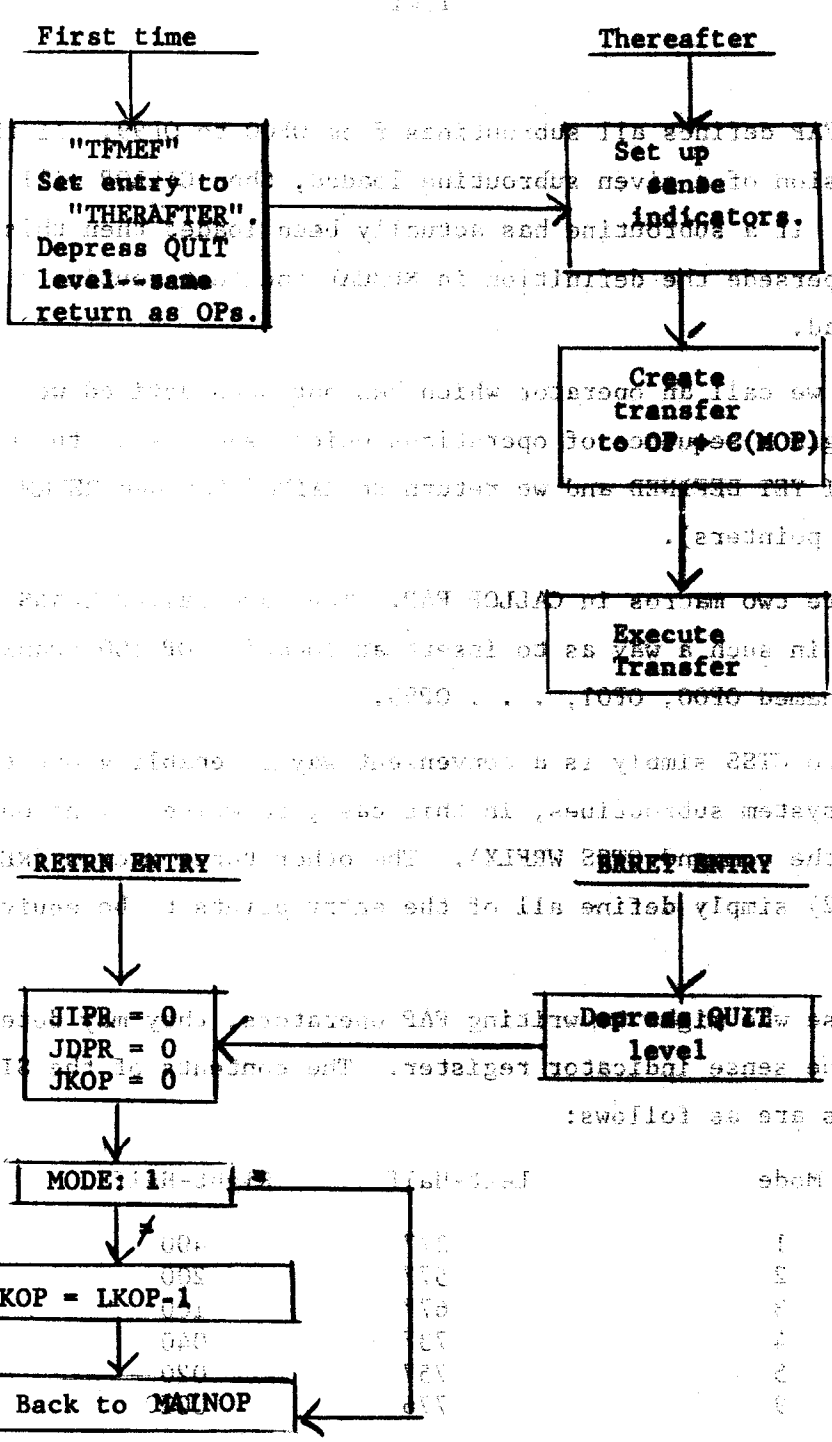
There are two macros in CALLOP FAP. They are called TRANS and FIX. They operate in such a way as to insert at location OP 100 transfers to subroutines named OP00, OP01, . . . OP99.

The macro CTSS simply is a convenient way of enabling one to get at the CTSS system subroutines, in this case, to write a line out on the typewriter (the command CTSS WRFLX). The other three macros (REQU, TAB1 and TAB2) simply define all of the entry points to be equivalent to OPXX.

For those who might be writing FAP operators, they may detect mode by testing the sense indicator register. The contents of the SI for the various modes are as follows:

Mode	Left-Half	Right-Half
1	377	400
2	577	200
3	677	100
4	737	040
5	757	020
9	776	001

Thus the command RNT 40 will execute the next instruction in sequence if we are not in mode 4 and will skip the next instruction if we are.


First time
Thereafter
"TFMEF"
Set entry to
"THERAFTER".
Depress QUIT
level--same
return as OPs.
Set up
sense
indicators.
Create
transfer
to OP+C(MOP)
Execute
Transfer
RETRN ENTRY
ERRET ENTRY
JIPR = 0
JDPR = 0
JKOP = 0
Depress QUIT
level
MODE: 1
LKOP = LKOP-1
Back to MAINOP

COMMAP

Purpose

The COMMAP routine is necessary for defining the data base, or data elements that are in COMMON storage. It is used by OP's 15 and 16 to locate individual data elements anywhere in the COMMON storage area.

Operation

COMMAP is written in FAP and makes use of a macro instruction called COMSET. The COMSET macro is a rather complex one. It performs the following functions:

1. Each time it is expanded it increments N by one.
2. Each time it is expanded, the variable named is placed in common.
3. The name, address of the variable (minus 1), and I,J,K dimension is placed in a table.

The entry points TABLE and MAX are used by GET and PUT* to obtain access to the given line of common. Essentially the process is this. Given a common line number, we test against MAX to see if we are beyond the range of common defined. If we pass this test, TABLE gives us the base address and dimension of the array referenced. From this information we can compute the exact address of the cell referenced. We use MAD (and thus MADTRN) type indexing, and thus this information is available for compatibility with MAD (and MADTRN) programs.

---

*See discussion on Service Routines, pg. 2,40.

We refer to common storage through "line numbers". The line numbers for system common are as follows:

Name of Common Array	Line No.
KOP	1
IPR	2
DPR	3
JIPR	4
JDPR	5
LKOP	6
LIPR	7
LDPR	8
MODE	9
NKOP	10
JKOP	11
MOP	12
INDEX	13
ISAT	14

When the user defines some common storage for himself (using the procedure discussed under "Updating of Common Map"), this list is simply extended. For example, if the user adds to the common above:

```
DIMENSION A(39),X(27,5,2)
COMMON    P, X, A
```

and modifies COMMAP by adding:

```
COMSET  P,0,0,0
COMSET  X, 27,5,2
COMSET  A,39,0,0
```

then P is line 15, X is line 16 and A is line 17. To refer to a given cell within an array we simply add the indices. For example to move X(22,3,1) to P using OP 16 we would enter as arguments to the first printout

```
("FROM LINE XX(XX,XX,XX) CAR RET. FOR PARAM")
          16 22 03 01      (Line 16, Cell 22,3,1)
```

and to the second printout

```
("TO LINE XX(XX,XX,XX)"etc.)
```

15 (or 15 00 00 00).

PARAMS

Purpose

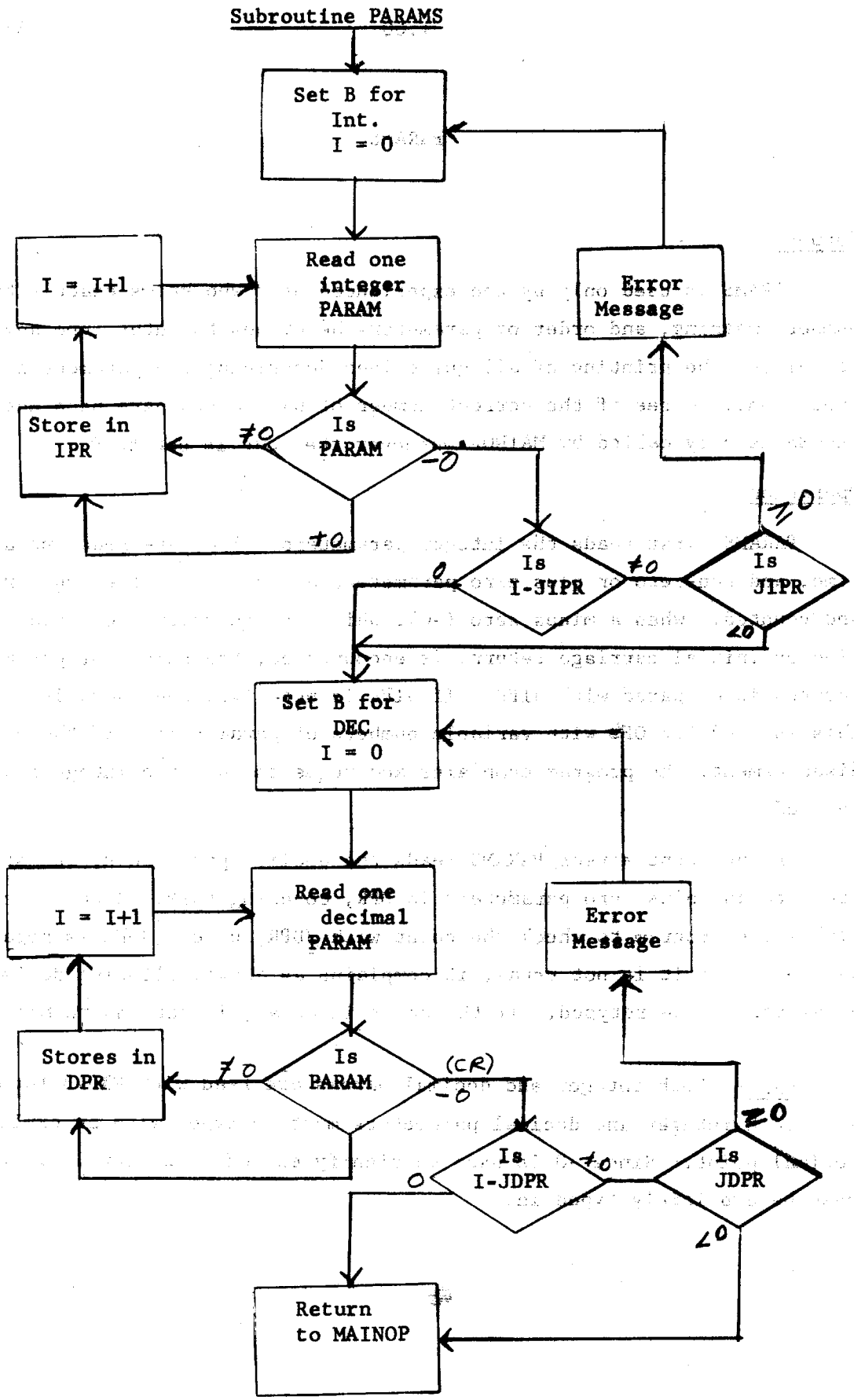
PARAMS is used only by the experienced user who knows exactly the number, meaning, and order of parameters he wishes to enter into any OP. It deletes the printing of all guidelines describing the parameters, and checks only to see if the correct number of parameters have been entered. PARAMS is only called by MAINOP whenever the mode is set to 9.

Operation

PARAMS first reads the integer parameters. They are read one at a time, and non-zero or plus zero parameters are stored in the IPR array and counted. When a minus zero (-0), which is equivalent to a blank line or initial carriage return, is encountered, the number of parameters counted is compared with JIPR. If JIPR is negative, the check is omitted. This is used for OPs with variable numbers of parameters. If there is a disagreement, the program complains and requests that the integers be retyped.

If the count agrees, PARAMS reads the decimal parameters, and stores non-zero and plus zero parameters in DPR, counting them. Minus zero causes the program to check the count with JDPR, unless JDPR is negative, as above. If it is not equal, it complains as above, allowing decimal parameters to be retyped. If the count is equal, it returns to MAINOP.

Note: Both integer and decimal params are read with F10.3 format. Thus both integer and decimal parameters must be typed with an explicit decimal point. Since -0 is used to signify end of parameter list, zero must be explicitly typed in.


Subroutine PARAMS
Set B for Int. I = 0
Read one integer PARAM
I = I+1
Error Message
Store in IPR
Is PARAM
≠0
=0
+0
Is I-JIPR
Is JIPR
0
≠0
≠0
<0
Set B for DEC I = 0
Read one decimal PARAM
I = I+1
Error Message
Stores in DPR
Is PARAM
≠0
(CR)
=0
Is I-JDPR
Is JDPR
0
+0
≠0
<0
Return to MAINOP

Chapter II

This empty page was substituted for a blank page in the original document.

The Structure of an OP

The following diagram shows the major steps of an OP. The numbers to the right indicate the modes that must and must not go through each step. Going through a step is optional for mode numbers not shown.

	Steps	Must be used by Modes	Must Not Be Used by Modes
	A. Subroutine OP XX	1,2,3,4,5,9	
	B. Storage Map Defines Common Storage	1,2,3,4,5	
	C. If MODE = 9 GO TO 9		
	D. Local Variables = IPR, DPR (from storage)	5	9
	E. IF MODE = 5 GO TO 5		
	F. Print Guide Lines Read from console into Local variables (carriage return in mode 4 causes stored values to be used)	1,2,3,4	5,9
	G. IF MODE = 3, GO TO 3		
5	H. Execution	1,2,4,5	3,9
	I. IF MODE = 1 GO TO 1		
3	J. Set IPR, DPR = local variables	2,3,4,5	1,9
9	K. Information Set JIPR, JDPR, (fixed)	2,3,4,5,9	
1	L. Return	1,2,3,4,5,9	

The OP must be written to read in and execute with variables local to this program so as not to clobber common storage when a simple execute (mode 1) is used temporarily in the middle of the execution of a KOP. Normally, do not change mode in an OP. Leave mode changes to READOP.

Depending on the nature of your OP, step 3 may be placed before or after step 5, the execute step.

Writing and Incorporating a new OP

All of the foregoing structure except the execution step has been preprogrammed in MADTRN. IP( ) and DP( ) are used as local variables for each OP. The execution step has been extracted and placed at the end for easy construction. Control is transferred to it (statement 5 CONTINUE) at the proper time. You complete the patch by transferring control to 7 (GO TO 7) at the end of the execution step which you write.

A. Construct common forms to suit your needs.

1. Start by EDITing one of the following forms from COMFIL 1 (or build your own in MADTRN, MAD or FAP if desired*):

   a. BFORM MADTRN for Both integer and decimal variables.

   b. IFORM MADTRN for Integers only.

   c. DFORM MADTRN for Decimals only.

   d. NFORM MADTRN for No variables at all,

2. Insert additional DIMENSION statements in lines 41 to 59 with the common elements first,

3. a. Insert additional COMMON statements in lines 91 to 99.

   b. Modify and recompile COMMAP to agree. See "Updating of COMMAP".

4. File as your own FORM for subsequent use.

B. Build the OP

1. Edit a suitable common form, constructed as above,

2. Change line 10 to read SUBROUTINE OPXX where the XX is your OP number. Normally, use numbers between 20 and 90.

3. Change line 20 to read your name and the date (of most recent change).

---

*Alternatively, the shorter FORM MADTRN or FORM FAP (also in COMFIL 1) may be used as a basic beginning in which case the programmer would build his own structure and use only part C of these instructions.

4. Update dimensions of IP and DP on line 50 if desired--these were set at 10 each initially. The dimensions must be changed if JIPR or JDPR exceed the values in the common form.

5. Set JIPR and JDPR on lines 100 and 110.

6. Add formats for printing guide lines* on line 560 and for reading IPRs and/or DPRs on lines 570/580 if needed. Standard formats will compile if not changed but will not read inputs.

7. Construct the execution part of the OP starting at location 600 in either manual or automatic mode.

8. Finish with statements GO TO 7 and END.

9. File OPXX MADTRN (or MAD or FAP) where the XX is the number of the OP.

C. Compile and Incorporate **

1. MADTRN (or MAD or FAP) OPXX.

2. Delete OPXX MAD (unless writing in MAD), OPXX MADTAB and OPXX SYMTB (if using FAP).

3. Load the new COMMAP first, followed by the OPS package (containing SYS and ALLOP), and then your personal OP's. The reasons for this sequence are as follows:

   a. All the COMMON storage required in any of the OP's must appear in the program loaded first.

   b. CALLOP in the OPS package defines all OP numbers from 0 to 99; if you load the OPS package after your personal OP's then your OP's will be replaced by the dummy definitions of CALLOP.

4. BSS files may be combined for easier loading by using the COMBIN* command. Be sure to use the * since resequencing clobbers the BSS instructions. (See pg. 1.70)

5. Do all this on your own number; not in COMFIL!

---

*Guide lines cue the human as to the contents and positioning of the next console input.

**See Chapter IV for an automatic way to perform these steps using OPS-1.

D. In writing an operator, it is essential to understand clearly the different sources of the parameters, called LOCAL VARIABLES, that are used by an operator during execution. LOCAL VARIABLES must be buffered from the parameters typed in at the console and the parameters that are (possibly) located in the IPR and/or DPR lists so that all of the options associated with the six modes can be performed. It is convenient to use the following definitions:

1. LOCAL VARIABLES: the values of an OP's parameters actually used during execution.

2. TYPED PARAMETERS: the values entered at the console.

3. STORAGE PARAMETERS: the values already in IPR and DPR storage.

Using these definitions, here is what happens in each mode:

Mode 1 - The TYPED PARAMETERS are placed only in the LOCAL VARIABLES and the OP is executed.

Mode 2 - The TYPED PARAMETERS are placed both in the LOCAL VARIABLES and the OP's STORAGE PARAMETERS and the OP is executed.

Mode 3 - The TYPED PARAMETERS are placed in the STORAGE PARAMETERS but the OP is not executed.

Mode 4 - The user is given the choice of using the STORAGE PARAMETERS as the LOCAL VARIABLES or typing in TYPED PARAMETERS to be used as LOCAL VARIABLES. The user must take some action for the run to continue; if he wishes to use the STORAGE PARAMETERS, he hits c.r. The TYPED PARAMETERS must not be placed in the STORAGE PARAMETERS.

Mode 5 - Parameters cannot be entered from the console.

These arrangements are mandatory for several reasons--perhaps the most important is permitting the use of Mode 1 at any time without destroying any parts of the OPS system. Normally, mode changes should not be made in an operation. Leave mode changes to READOP.

Updating of Common Map (COMMAP)

To keep the common map up to date, it is only necessary to insert lines of coding for each array or variable added to the common list. These lines take the form:

(tab) COMSET (tab) NAME,I,J,K

where NAME is the name of the common area, and I,J,K are its dimensions, for example:

```
COMSET    KOP,100,3,0,
COMSET    MODE,0,0,0
etc.
```

These lines should be inserted at the end of the TABLE list, and just before the line

```
MAX    PZE    N
```

Service Subroutines

Several subroutines are used by OP 16 to retrieve and store information in COMMON. These subroutines can be used by any operator. We will describe the effect and the calling sequence for each of these routines below:

CALL GET(L,I,J,K) retrieves the (I,J,K)th element of the Lth line of COMMON. It places the contents of the cell referenced in a register called EXCH.

CALL PUT(L,I,J,K) places the contents of the EXCH register into the (I,J,K)th element of the Lth line of COMMON.

CALL LE(A) places the contents of A in the EXCH register.

CALL EE(A) places the contents of the EXCH register in A.

CALL GETLST(L,I,J) places the contents of the Jth parameter (integer parameter if I=0, decimal parameter if I=1) of the Lth line of the KOP list in EXCH.

CALL PUTLST(L,I,J) places the contents of the EXCH register into the position of the Jth integer (I=0) or decimal (I=1) parameter of the Lth line of the KOP list.

The function OCTALF(I) has a value which is the octal equivalent of the BCD number in I.

These subroutines are available for use by any operator.

Note: If one gets a parameter from the DPR (for example) and empties it into I, there will be no conversion to the integer mode.

```
CALL LE(X)
CALL EE(I)
```

puts the contents of X into I, but it does not convert X to an integer. Thus I looks like a very strange integer.

Creation and Contents of Combined BSS Files

The CTSS operator COMBIN *X BSS Y Z A . . . has the effect of combining the files Y BSS, Z BSS, A BSS, . . ., under the name X BSS. This feature is used to make it easier to load programs (only a few combined files, replacing many individual files).

The file SYS BSS, for example, contains MAINOP BSS, CALLOP BSS, READOP BSS, PARAMS BSS, SERVIC BSS. The CALLOP BSS and SERVIC BSS files are themselves combinations of further individual files.

We have established the procedure of dating each FAP and MADTRN file. It is impossible, however, to date the BSS files for any such information would cause them to operate incorrectly. For this reason, we have established the procedure of creating files with such names as SYS FACT. SYS FACT simply contains the date on which SYS BSS was put together, along with a description of the contents of SYS. FACT files will usually appear only for files for which no MADTRN or FAP file exists. In a few cases, however, it is necessary to point out some peripheral facts, and one will appear in these cases. For example CALLOP FAP is the routine which does the calling of operators. CALLOP BSS, however, is a combination of this file and one called SUBCAL FAP, and thus the file CALLOP FACT appears, and makes note of this fact. Should the user wish a part, but not all, of some combined file, he must copy each part from the COMFIL and combine them under his own number.

Error Conditions in Subroutines

This note has two purposes: 1) to explain the behavior of OPS when certain error conditions arise; and 2) to give the user a method for handling error conditions within his own subroutines.

If we have a subroutine which is common to several operators (such as GET, etc., which is common to OP 16 and OP 15 and others), and an error condition arises which is uncorrectable within the subroutine, then we have a problem because if we return to the calling operator it will behave as though the subroutine has operated properly. To get around this problem we have established a return to MAINOP without incrementing any pointers regardless of mode.

To use this error routine simply CALL RETRN. Control will pass to RETRN (which is actually part of CALLOP) where we will test mode. If in mode 1, control simply passes back to MAINOP. If in some other mode, we first decrement LKOP in anticipation of MAINOP's incrementing it. In all cases we set JIPR, JDPR and JKOP to zero. Thus, whatever the mode, we return to MAINOP without changing any pointers.

Variable Numbers of Parameters

For certain operators it is necessary that the number of parameters be variable (i.e., OP's 91 and 99 described in Chapter IV). Certain problems, however, are created by this and this note will attempt to explain, mode by mode, how they may be resolved.

In mode 1, a variable number of parameters clearly creates no problem, since the parameters are only used locally.

In modes 2, 3 and 4, we are reading the parameters from the console, and thus we accept however many are entered. The one restriction on these modes is that if we are writing in the middle of an OP we must be careful not to enter more parameters than there is space. Thus if we erroneously enter 7 parameters for an operator on line 5, we may go back (say when we are now on line 10) and enter six parameters on line 5, but we may not correct it to eight parameters. If we need eight parameters, we must return to line 5 and continue from there.

In mode 5, the routine must know how many parameters are used by the OP. It can find this out by differencing KOP(LKOP+1, 2 or 3) and KOP(LKOP, 2 or 3).

In mode 9 we must notify the PARAMS routine that any number of parameters is acceptable. This is done by setting JIPR or JDPR negative. PARAMS interprets this as a signal that whatever number of parameters are entered is all right.

Chapter III

This empty page was substituted for a blank page in the original document.

NO OP (OP 00)

Purpose

The main use of this OP is to replace OP's that you wish to delete from an already constructed KOP.

Operation

1. No parameters.
2. If mode is 1, 2, 3, 9, set
   JIPR = 0, JDPR = 0.
3. If mode is 4 or 5, set
   JIPR = KOP(LKOP+1,2) - KOP(LKOP,2)
   JDPR = KOP(LKOP+1,3) - KOP(LKOP,3)

PRINT KOP (OP 10)

Purpose

Prints KOP's and associated IPR and DPR lists.

Operation

Prints out:

FROM XXX TO XXX (XXX referring to line numbers).

Will print out from N_1 to N_2-1 in KOP list.

LOOP (OP's 11 and 12)

Purpose

These two OP's provide a means for controlled repetition of any sequence of other OP's. Their existence allows iteration of strings of OP's and further provides an indexing parameter (INDEX) which may be referred to or altered by the OP's of the sequence. By virtue of the nesting feature, the LOOP OP themselves may be elements of the sequence of other LOOP OP's. OP 11 is used to initiate a loop, and OP 12 to terminate the range of the loop's sequence.

Parameters used by OP's 11 and 12

1. A one-digit integer to designate which of nine 1 x 4 arrays in common is to be used in the loop defined by OP 11 at the beginning and OP 12 at the end. The nine arrays allow loops to be nested nine deep.

   INDEX(i,1) is the actual value of the index i.

   INDEX(i,2) is the highest value index i shall be permitted to take.

   INDEX(i,3) is the incremental step by which index i is altered during each cycle.

   INDEX(i,4) is reserved for internal bookkeeping, and refers to the range of the loop. INDEX(i,4) should never be used by a programmer, nor need it be.

2. The starting value of the loop index.

3. The highest value of the loop index to be permitted.

   Note: This value may never be attained if the increment is greater than 1, but in no case will it be exceeded.

4. The amount the index is to be incremented during each loop. (This is set equal to one if read in as zero.)

These parameters may be read in by either OP 11, OP 12, or both. Any parameter read in by OP 12 will replace what was read in by OP 11.

Operation

1. Specify OP's and read in parameters while in mode 3.
2. During execution, the OP's will test to see if the loop is satisfied before executing the first pass. When the loop is satisfied, the KOP will continue with the OP following OP 12.
3. It is possible for any OP to modify or refer to the value of the index, the increment, or the final value, during its execution.

CONDITIONAL BRANCH (OP's 13 and 14)

Objectives

OP 13 and 14 are used to control the sequence in which operators are executed in a KOP. OP 13 causes control to conditionally branch to the OP 14 which has the same first parameter. Any number of OP 13's may branch to the same OP 14.

Parameters used by OP 13

1. A two-digit integer between 01 and 30 which gives the location in a single dimension array in common, ISAT, which contains the LKOP of an OP 14 so that OP 13 knows where to branch.
2. Parameters required by GET routine to retrieve the contents of a location in common. The contents of this location will be referred to below as Var 1.
3. Another set of parameters required by the GET routine. The contents of this location will be referred to below as Var 2.
4. The type of conditional branch.

Parameters used by OP 14

1. A two-digit integer between 1 and 30 which specifies the location in a single dimension array in common, ISAT, in which the LKOP of this OP 14 will be placed.

Operation of OPS 13 and 14

1. If all parameters are zero except the location in ISAT, OP 13 executes an unconditional branch to the OP 14 which has the same first parameters as this OP 13.
2. Otherwise, OP 13 executes a branch to the OP 14 with the same first parameter when the type of conditional branch is satisfied.

Type of Conditional Branch	Relationship of Var 1 to Var 2
E- - -0	Var1 = Var 2
NE - -1	Var1 $\neq$ Var 2
L - - 2	Var 1 $<$ Var 2
LE - -3	Var 1 $\leq$ Var 2
G - - 4	Var 1 $>$ Var 2
GE - -5	Var 1 $\geq$ Var 2

If the parameters pointing to Var 2 are all zero, Var 2 will be set equal to zero.

The only parameter called for by OP 14 is the location in ISAT.

Modify Operation (OP 15)

Purpose

OP 15 is a utility operation of the OPS System, which allows the user to look at the contents of any single cell in COMMON storage and to change the contents, if desired. The cell in COMMON storage can be in the data base, including the KOP and parameter lists. All cell locations can also be specified by indirectly addressing the particular OP of interest.

As presently written, OP 15 is able to deal with only one cell or word of COMMON storage at a time. In the future it may be desirable to provide for printing and possible (selective) modification of blocks of COMMON storage at one time. It would also be advantageous to address a line of the KOP relative to this OP 15 when it is being used.

Description of Parameters

Guide lines will be printed as shown below when OP 15 is called in modes 1 to 4 inclusive. This is a request for the five integer parameters which OP 15 requires: CODE, LINE, SUBSCRIPT I, SUBSCRIPT J, and SUBSCRIPT K.

```
                    1-FIX DIR, 2-FLT DIR, 3-FIX IND, 4-FLT IND
                    C LN SSI SSJ SSK
User types:         X XX XXX XXX XXX
```

Five Integer Parameters

I* CODE indicates both the type of data being addressed and the method of addressing. As listed in the first line of the guide lines, the alternative values of CODE are:

1--direct addressing; integer data

2--direct addressing; decimal data

3--indirect addressing; integer data

4--indirect addressing; decimal data

	Direct Addressing	Indirect Addressing
2*	LINE of COMMON storage being addressed (see MAP MADTRN or MAP FAP)	LINE of KOP which contains the OP whose IPR or DPR parameter is being studied. This is the absolute line of KOP, not relative.
3	SSI is the value of I in ARRAY(I,J,K). If I is 1 or the cell being addressed is not an array, zero or one can be used.	Relative location in the selected OP's parameter list. (A 0 or carriage return will be set to 1 by the OP 15.)
4	SSJ is the value of J in ARRAY(I,J,K)	(not used)
5	SSK is the value of K in ARRAY(I,J,K)	(not used)

*Only the first two entries are mandatory. If the addressed variable is not dimensional then these two parameters can be entered followed by a carriage return.

One Decimal Parameter

OP 15 can be used in modes 4 and 5 to automatically initialize, re-initialize, or change a data word in COMMON storage by putting the new value of the specified word in the DPR list as OP 15's one DPR parameter. This parameter, be it integer or decimal when used, is always stored in decimal form and must be entered on the console in decimal form.

Operation of OP 15

To clarify terms in the description by mode, the following definitions are given:

VARIABLE: the cell or word in COMMON storage being referenced by this particular OP 15 for inspection and/or possible modification.

VALUE OF THE VARIABLE: (PARAM VALUE, STORAGE VALUE, NEW VALUE)
When speaking of the value of the variable we must specify where -- in the parameter set, in COMMON storage, or being read from the console.

PARAMETER SET: (REFERENCE, PARAM VALUE)
The six parameters associated with this particular OP 15.

REFERENCE: the five integer parameters which specify the location of the VARIABLE in COMMON storage.

PARAM VALUE: the one decimal parameter which is needed to facilitate the automatic execution of OP 15 in modes 4 and 5. It is used to reset the STORAGE VALUE of the variable.

STORAGE VALUE: the value of the referenced VARIABLE in COMMON storage. This is always used to represent the current value at the time this particular OP 15 is encountered or when the inquiry is made.

NEW VALUE: that value of the VARIABLE which is read from the console (as typed by the user).

All modes prior to exit will set JIPR = 5, and JDPR = 1.

MODE 1: OP 15 will print out the guide lines as shown above and read from the console the REFERENCE parameters. If a carriage return is given control will be released from OP 15 (i.e., RETURN is given).

The STORAGE VALUE of the variable so referenced will be printed on the console.

OP 15 now asks for a NEW VALUE to be typed on the console. It must be typed with decimal point regardless of the specification in the first parameter. If it is to be an integer, the OP 15 will make the necessary conversion. This conversion removes the necessity of typing in redundant leading zeros.

The value so typed will replace the STORAGE VALUE of the variable.

If a carriage return is given no change will be made to the STORAGE VALUE of the variable.

MODE 2: As mode 1 except that the REFERENCE parameters will also be entered into the KOP list of this particular OP 15 and the PARAM VALUE of the variable will be set equal to the NEW VALUE if one is typed on the console, or it will be set equal to the STORAGE VALUE if a carriage return is given.

MODE 3: As in mode 2 except that the STORAGE VALUE of the variable is left unchanged (this is the no execution feature of this mode). As before, the PARAM VALUE of the variable will be set equal to the NEW VALUE if one is typed on the console, or it will be set equal to the STORAGE VALUE if a carriage return is given.

MODE 4: The REFERENCE parameters, having previously been set, are printed on the console immediately following the guide lines. If a new REFERENCE is typed in, it will replace the old one in the KOP parameter list for this particular OP 15. If a carriage return is given, no change will be made in the REFERENCE.
Note: If a change is to be made, all five of the parameters must be retyped on the console.

The STORAGE VALUE and PARAM VALUE of the variable so referenced will be typed on the console and OP 15 will then ask for a NEW VALUE of the variable to be typed on the console.

The value so typed (with decimal point--see Mode 1) will replace both the STORAGE VALUE and the PARAM VALUE of the variable.

If a carriage return is given, the PARAM VALUE will replace the STORAGE VALUE of the VARIABLE.

MODE 5: In this automatic KOP execution mode the only action taken when OP 15 is encountered is that the STORAGE VALUE is replaced by the PARAM VALUE of the variable referenced by the REFERENCE parameters. Nothing is printed on the console. This action might be useful in some forms of initialization.

MOVE (OP 16)

Purpose

The purpose of MOVE is to allow one to move any cell in common storage to any other common location.

Use

Each variable and array in common is assigned a line number for external notation. These are assigned in order, i.e., 1=KOP, 2=IPR, 3=DPR, . . . etc. We would refer to KOP(62,2) as 01,62,02,00 and MODE as 9,0,0,0.

This operator has eight integer parameters.

References may be made directly to a common cell, or indirectly through the pointer list to the IPR or DPR. Such a reference is made relative to the current LKOP and takes the following form:

RELAT, XPR, INCR

which references the INCRth parameter of the IPR(XPR=0) or DPR(XPR=1) of the (LKOP+RELAT)th line of the operator list.*

Entering the operator causes the message FROM LINE (I,J,K) CAR. RET. FOR PARAM to be typed. If a direct reference is desired we type the line number and as many indicies as necessary. If we wish to make a relative reference we carriage return and the message

RELAT, 0=IPR 1=DPR, INCR. is typed. Then we type in the parameters indicated above.

After completing either of these procedures, the process is repeated for the receiving location. The message TO LINE (I,J,K) CAR. RET. FOR PARAM is typed and similar entries made.

*Relative references forward (RELAT > 0) will not be executed properly in the CREATE AND EXECUTE mode.

Writing KOP onto Tape (OP 17)

Purpose

OP 17 writes a KOP, with associated integer and decimal parameters, onto a simulated tape.

Operation

The user is asked to specify the number of the KOP, the line number of the first OP in the new KOP, and the line number of the last OP which is to be included.

If the first line number is not specified, the number 01 is automatically assigned; if the ending line number is not specified, it is assigned at execution time and will be the line number of the OP immediately preceding the OP 17 being executed.

Once the beginning and ending lines of the KOP list, as well as its number N, have been determined, OP 17 deletes from the disk file any file .TAPE. N. It then writes 5 quantities onto .TAPE. N:

1. The number of OP's which are to be included in the new KOP.
2. The number of integer parameters associated with the OP's being included in the KOP.
3. The number of floating point parameters associated with the OP's being included in the KOP.
4. The integer pointer of the first OP of the new KOP.
5. The floating point pointer of the first OP of the new KOP.

These five quantities are used when the KOP is read back into the computer by OP 18.

After these five quantities, the array of OP's and pointers in the KOP, the integer parameters, and the floating point parameters are written.

Use of OP 17

When OP 17 is entered, it types out the request: WRITE KOP XXX, LINE XX TO LINE XX. It expects input in the form XXX XX XX. The KOP number must be specified; if the two line numbers are omitted they will be assigned as outlined above. If you are in the create mode, the reassigned numbers will be stored in the IPR list. The KOP and its associated parameters are then written on a simulated tape XXX, with the creation of a file .TAPE. XXX.

Reading KOP from Tape (OP 18)

Purpose

OP 18 reads the KOP back into the computer. It asks for the number of the KOP, for the line on which the KOP is to be read in, and for the first line of execution. If no line number is given, the program assumes the KOP is to be read in starting with line 1.

During the reading-in operation, OP 18 re-assigns the pointers of the KOP being read in, and adds the numbers of the IPR and DPR lists in their proper places.

If an attempt is made to read a KOP in from tape and the first line number is not 1, there must be an OP, with pointers, in the preceding line. If there is not, OP 18 will not know where to put the new members of the IPR and DPR lists.

Use of OP 18

OP 18 types out

READ KOP XXX, STARTING WITH LINE XX, EXECUTION STARTS ON LINE XX

It expects input in the form XXX XX XX. The starting line indicates the line on which the first OP of the KOP is to be placed, with the others following sequentially. If no starting line is specified, line 1 is assumed; if no execution-start line is specified, line 2 is assumed.

The OP computes, with the first five numbers on tape, the new values of the pointers of the KOP being read. It then assigns those new pointers and updates LKOP, LIPR and LDPR.

If in mode 4 or 5, execution will begin at line XX of the KOP being read in. For example, if XXX XX XX were specified as 100 20 05, the KOP 100 would be read in. The first OP would be placed in line 20, the second in line 21, etc., of the KOP list. If in mode 4 or 5, execution would continue with line 5 (line 24 of the KOP list) of the KOP which had just been read in.

END OP (OP 19)

Purpose

The purpose of the operator is to provide an automatic termination for KOP's. It also provides a simple way of exiting from the OPS system back to CTSS.

Operation

1. No parameters.
2. If mode = 1, EXIT will be called.
3. If mode is 2, 4 or 5, prints out "END OF KOP" and sets JKOP=0.
4. If mode is 3 or 9 OP is not executed.

This empty page was substituted for a blank page in the original document.

Chapter IV

This empty page was substituted for a blank page in the original document.

CTSS OPERATORS

by

David Ness

In building up a system from the standard OPS base, the user programs new operators and modifies old ones. He may also want to alter his map of common storage, and perform other activities normally within the jurisdiction of the time-sharing supervisor.

For some purposes, it is convenient to be able to do these things as though by operators of the user's system; that is, without returning control to CTSS. The following set of operators gives the user this ability for any CTSS command. These operators were programmed before the RUNCOM command of CTSS was available.

OP96: Input and Compile

Purpose

Inputs a file and compiles it into machine code. The sequence is automatic.

Parameters

Two integer (actually BCD) parameters: the class of the input file and the name of the input file (in that order).

Operation

OP96 obtains its parameters either from the console or from the IPR list. It then executes the CTSS command, INPUT. When the FILE command is executed at the end of the input sequence, the operator automatically calls for the compilation of the file. If the compilation is successful, extra files created by the compilation are automatically deleted.

Guide Lines

INPUT: CLASS NAME. When this information is received, INPUT YOUR FILE is typed.

Note

If we supply the operator with parameters MADTRN X, for example, and then file our input under the name Y MADTRN, the routine will compile the X MADTRN file.

OP97: Edit and Compile

Purpose

Edits a file and compiles it into machine code.

Parameters and Operation

Exactly as in OP96 except that an existing file is edited, rather than a new file created.

Guide Lines

EDIT: CLASS NAME

Note

See the note to OP96.

OP99: System Load

Purpose

Loads new operators, etc., without leaving OPS.

Parameters

Variable number of integer (actually BCD) parameters. These parameters are the BCD names of the files to be loaded.

Operation

Loads COMMAP, SYS, ALLOP and any other files indicated as parameters. The sequence is as follows:

1. Get names of files other than COMMAP, SYS, and ALLOP from the console or from the IPR
2. CTEST4 COMMAP SYS ALLOP and the indicated files
3. START (i.e. execute a START command).

Guide Lines

TYPE NAMES OF OP'S. User types a line of BSS file names. For example typing the line "OP92 ABC OP91 XYS" would cause COMMAP, SYS, ALLOP, OP92, ABC, OP91 and XYS to be loaded.

Note

This operator, when executed, destroys the common area. We will later describe a pair of operators which can be used to save and restore common.

OP93: Define Own Common

Purpose

Eliminates the necessity of the user's constantly modifying his COMMAP.

Parameters and Mode

OP93 is executed regardless of mode and thus it does not change the IPR in any way.

Operation

Allows the user to extend his common map. It does this by creating a file called OWNCOM FAP which is the map of the user's own common area (i.e. his extension to system common).

Guide Lines

CREATE COMMON: NAME,I,J,K CR / CR. User types an arbitrary number of input lines of the form: ABC,I,J,K where ABC is the name to be assigned to the common array and I,J,K are the dimension (explicity zeros for unused indicies). When all of user common has been entered an extra carriage return terminates the operation of this operator.

Example

To add an array A(27,3,4) and two single cells X and Y to common (in that order) we would type:

```
A,26,3,4
X,0,0,0
Y,0,0,0
(carriage return)
```

Note

When OP93 is executed a new definition of user common is made. Nothing of previous definitions (except system common) remains. OP89 can be used to extend common by adding to the current OWNCOM file.

OP89: Extend Own Common

Purpose

Extends (not recreates) OWNCOM FAP.

Parameters and Mode

OP89 has no parameters and it operates regardless of mode.

Operation

Equivalent to OP93 except that current definitions of common are extended, not begun again.

Guide Lines

EXTEND COMMON: NAME,I,J,K CR / CR. User input is same as in OP93.

OP92: Incorporate Own Common

Purpose

Incorporates a common map written by OP93 (i.e. to create the BSS version of COMMAP with the user's own common).

Parameters and Mode

OP92 has no parameters and operates (like OP93) regardless of mode.

Operation

Assembles the file COMMAP FAP with OWNCOM FAP incorporated, deletes the SYMTB file created and returns control to OPS. It does not automatically load the new version.

Guide Lines

FAP ENTERED TO DEFINE COMMON. There is no user input.

Note

If OP92 is executed with no OWNCOM FAP present, it first creates a blank OWNCOM.

OP98: Combine BSS Files

Purpose

Combines any BSS files (used to create ALLOP).

Parameters

Variable number of integer (actually BCD) parameters. These parameters are the BCD names of the files to be combined.

Operation

Requests the name of the combined file and names of the files to be combined into this file. It automatically sets up and executes a CTSS COMBIN operation.

Guide Lines

ENTER NEWNAME THEN ALL TO BE COMBINED. User types the new name and the names of all files to be combined under that name.

Example

To add the files OP27 BSS and SUBR BSS to our present ALLOP we would type "ALLOP ALLOP OP27 SUBR" which would combine under the name ALLOP the files ALLOP (the old one) OP27 and SUBR.

Note

This OP can only be used with BSS files.

OP91: CTSS Command

Purpose

Allows access within OPS to any CTSS command.

Parameters

Variable number of integer (actually BCD) parameters. Each parameter is a single word of a CTSS command (in sequence).

Operation

Reads in one line of information which it expects to be a CTSS operator. It then executes this operator.

Guide Lines

CTSS LINE. User types one line of input in the normal CTSS fashion.

Example

If we were to type :RENAME OP27 MADTRN OP28 MADTRN" we would execute the rename command without leaving OPS.

OP95: Write Common Area on Disk

Purpose

Allows the user to write any part of the common area into a file.

Parameters

Nine integer (one of which is actually BCD) parameters. They are (in order) one BCD name, four integers denoting "from" cell and four integers denoting "to" cell (explained below).

Operation

Reads a name (e.g. MAY20) and creates a file (named MAY20 COMMON) which contains all of the cells from the cell denoted by the first set of parameters to the cell denoted by the second set of parameters. Cells are referenced as discussed in "Addressing Common Storage", (see p.) above. This file will be available for reading by OP94.

Guide Lines

FILE NAME OF COMMON. User types one word which will become the name of the common area file being created. It then types TYPE IN BEG L,I,J,K, OR ALL and waits for the line number and (i,j,k) of any array to be entered. This is a reference to the cell from which it is desired to begin writing. One may also type the message ALL at this point and the operator will automatically write out all of present common. If one does not select the ALL option the routine types out TYPE IN END L,I,J,K and waits for line number and i,j,k of the ending cell to be entered. It then determines the location of this section of common and writes it into a file.

Note

See note following description of OP94 for further comment on OP's 94 and 95.

OP94: Read Common Area from Disk

Purpose

Reads a file written by an OP95.

Parameters

One integer (actually BCD) parameter. This parameter is the name of the common area file desired.

Operation

Reads indication from the file referenced as to where to begin writing. It then reads the rest of the file into this segment.

Guide Lines

NAME OF COMMON FILE DESIRED and user types the name of a COMMON (class name) file. It then types RETRIEVING THE COMMON AREA and proceeds to obtain the file.

Notes on OP94 and OP95

1. OP95 writes indication (beginning location and number of words) of the source of the information at the beginning of the file and then writes the information. Thus OP94 can retrieve the information without knowing where to put it, because this information is at the beginning of the file.

2. If common is changed (by extending it, for example) the file will be read into the correct positions so long as no change has occurred within the part which was written into the file. This should be made clear by the examples which end this chapter.

3. In writing out an array, one cannot write out just one index (for two or three dimensions). OP95 writes out a sequence of registers. Writing from KOP(1,1) to KOP(5,3) writes out (1,1), (1,2), (1,3) ... (5,2), (5,3).

4. Parameters are input as in the following example. To write out all of system common (lines 1,1,1 to line 14,30) we would type as follows:

```
TYPE IN BEG L,I,J,K OR ALL          (machine)

1 1 1                               (user)

TYPE IN END L,I,J,K                 (machine)

14 30                               (user)
```

Any i,j, or k not entered is regarded as zero.

OP90: Print Common File

Purpose

Prints a file written by OP95.

Parameters

One integer (actually BCD) parameter. It is the name of the file to be printed.

Operation

Prints the common file indicated by the name mentioned. The file is printed in reverse order (i.e. in the actual order of instructions in the machine, since the common area is stored in reverse). The first five words of the file are the L,I,J,K of the ending location and the number of words in the file.

Guide Lines

FILE TO BE PRINTED. User types the name of the common area file which he desires. It then types out "BCD OCTAL DECIMAL FLOATING POINT" and proceeds to print out the bed, octal, decimal and floating point equivalents of the word. (Under BCD any illegal character is printed as *). The routine automatically suppresses the printing of zeros, but it counts them and prints out the message "ZEROS: XX" (where XX is the number of zeros suppressed).

An Example of a KOP

To add A(5) to our common area without changing anything else the following sequence may be executed:

```
OP89 (Extend Own Common)
A,5,0,0
OP92 (Incorporate Own Common)
OP95 (Write Common Area File)
TEMP (Name of file)
ALL  (Parameters of Common)
```

OP99 (System Load)
OP94 (Read Common Area File)
OP91 (CTSS Command)
DELETE TEMP COMMON

This first extends common, then incorporates this extension. Then we save the common area, reload the system, reload the common area and then finally delete the temporary file.

5.00

Chapter V

This empty page was substituted for a blank page in the original document.

OPTRAN

by

James Linderman

The OPS system simultaneously provides a need and a facility for an on-line algebraic programming language. It provides a need for algebraic manipulation of storage with a requirement that execution be concurrent with programming. It provides a facility for recording a program as it is written and for re-executing the program, or any part of it, at will.

OPTRAN is an algebraic language similar to FORTRAN, which gives the user the option of executing as he is programming. OPTRAN is embedded in the OPS system and was produced in less than a term of intermittent work. For this reason many desirable features had to be left for later addition, and emphasis was placed on a working, easy-to-use version.

Description of the Language

Although OPTRAN might easily include statements of every type provided by FORTRAN (substitution, conditional transfer, and iteration) the initial version has only substitution or algebraic replacement statements. The decision to postpone coding of the other types was based partially on the existence of workable loop and transfer OP's in the OPS system itself. A limited vocabulary of dimensioning, clearing, and printing statements also are provided but mode declarations and EQUIVALENCE and COMMON statements are omitted.

The replacement statement is of the form a = b with the following meaning:

1. a is the variable to be replaced,
2. b is an algebraic expression, consisting of one or more variables and/or constants connected in a meaningful algebraic sequence by standard arithmetic operators:

   \+ unary or binary (addition)

   \- unary (negation) or binary (subtraction)

   * multiplication

   ** exponentiation

   / division

   ' (apostrophe) a convention for taking absolute value of the expression following

3. The variables in the replacement statement may be of either integer or floating point mode, and may be subscripted by using the usual parenthesis convention. ANY arithmetic expression may be used as a subscript expression. Parentheses also may be used in the same way as in ordinary algebra to specify the order of the computation, and standard FORTRAN hierarchy is observed for operators in an unparenthesized expression.

4. For the time being, variables may have only one subscript. To accomodate double or multiple subscripts the programmer must reduce the multiple subscripts to a single subscript by computing the appropriate algebraic function of the multiple subscripts. For example in the two dimensional case, ARRAY (I,J), is replaced by ARRAY((I-1)*JMAX + J) where JMAX is maximum value of the J subscript.

To provide compatibility with the OPS system, all of the COMMON variables used in programming a standard OP have been predefined and appropriately located. Additional common storage is allocated as additional variables are introduced by the OPTRAN user. The FORTRAN convention of identifying fixed point variables by initial character (i,j, ... ,n) has been retained.

To dimension arrays, the statement

DIMENSION ARRAY1(ISIZE1),ARRAY2(ISIZE2),...,ARRAYn(ISIZEn)

may be used. Both fixed and floating point arrays may be dimensioned in this way, but the symbols ARRAYi must not have occurred previously, and the variables ISIZEi must be integers. These integers may be constants OR, unlike FORTRAN, previously defined integer variables in which case the current variables are used.

The command CLEAR clears out all symbols defined in OPTRAN, and thereby makes their values inaccessible. It does NOT clear the COMMON variables of the OPS system. The statement

PRINT: VAR1,VAR2, ..., VARn

causes the printing, one to a line, of the n variables VARi, in fixed or floating mode. If a symbol is undefined, a line to this effect is printed. NO SUBSCRIPTS ARE ALLOWED as yet. This is a temporary restriction, and may require you to set an unsubscripted variable to the value of a subscripted variable, using the replacement statement before printing.

Symbols may be no more than 6 characters in length, and are right adjusted with preceding blanks as in CTSS, but not as in FORTRAN. Symbols with more than 6 characters will be truncated leaving only the terminal 6 characters. Blanks are everywhere ignored in the system.

Limited diagnostics are given. In most cases of error, control is returned to the OPS system through the RETRN entry point. In a few recoverable cases control is returned to OPTRAN itself for corrective action.

Use

OPTRAN is initiated within OPS by a call to OP20. OPTRAN then continues to call OP20 itself, until a carriage return restores control to OPS. All relevant arrays, notably KOP and IPR, are continually updated so that a single call to OP20 in mode 2 or 3 may result in many occurences of OP20 in the KOP list.

OP20 invites a statement from the user by printing TYPE. The user then has the option of typing an OPTRAN statement, or giving a carriage return to restore control to the OPS system.

Any OPTRAN statement referring to KOP, IPR, DPR, JIPR, JDPR, LKOP, LIPR, LDPR, MODE, NKOP, JKOP, MOP, INDEX or ISAT will actually address these common locations, subject to the restriction of one subscript only. Common storage has been extended to 5000 locations, and symbols other than the standard COMMON symbols given above are added into common storage as they are specified.

"Compilation" consists of construction of a Polish list, usually about 3 to 15 elements long. This list is stored in modes 2 and 3 in the IPR list, with the addition of either

1. a blank word (OCT 606060606060) to indicate end of list, or
2. an integer count of the elements in the list in IPR(LIPR), the actual elements following in IPR(LIPR+1).

OPTRAN makes no distinction between modes 4 or 5. In either of these modes, the IPR array is used for the Polish list and recompilation is unnecessary.

In execution modes, the Polish list is processed and printing or substitution/modification of storage is effected. Since DIMENSIONING IS NOT A REPEATABLE OPERATION, repetitions of a dimension statement during a compound execution result in no operations after the first occurrence.

Diagnostics and Messages

input (always printed when ready to accept input line)

TYPE:

dimensioning

ILLEGAL DIMENSION FOR 'XXXXXX'

'XXXXXX' DIMENSIONED AT nnnnnn

replacement statement compilation

MISPLACED LEFT PARENTHESIS

MISPLACED RIGHT PARENTHESIS

THE SYMBOL x IS MISPLACED

FOR, CALLED TOO MANY TIMES

PUSH, CALLED TOO MANY TIMES

printing

XXXXXX = 0.nnnnnnE+nn (if XXXXXX is floating point mode)

XXXXXX = nnnnnnnn (if XXXXXX is integer mode)

'XXXXXX' not in symbol table

Possible Extensions and Revisions of the OPTRAN Language

Easy alterations to OP20 as it now stands include:

1. Different combinations of predefined symbols to accomodate different common variables.

2. Alterations to the size of symbol tables, and to the maximal allowable statement length.

3. Additional diagnostics or message formats.

4. Different integer-mode characters (other than i,j,k,l,m,n).

Somewhat more difficult alterations include:

1. Additional operations, i.e., extending the operator set beyond (*,+,/, etc.).

2. Multiple subscripting.

Difficult but still desirable alterations include:

1. Incorporating loop and transfer operations into OPTRAN.

2. Simultaneous maintenance of a parallel FORTRAN file for creation of a standard FORTRAN program.

Conclusion

OPTRAN provides the OPS system with an on-line facility for programming with simultaneous execution and immediate diagnostics. One interesting application is the creation of new operators. After being debugged using OPTRAN, a new operator can be added to the OPS system as a KOP. The introduction of a mechanism to produce a parallel FORTRAN file on the disc would also allow the operator to be added to the system as a standard OP, if this were more desirable.

This empty page was substituted for a blank page in the original document.

Chapter VI

This empty page was substituted for a blank page in the original document.

OPSIM

by

Robert Welsh

Introduction

Current general purpose simulation languages such as SIMSCRIPT, GPSS, and DYNAMO have made it markedly easier to build complex simulation models. Although these languages were not designed for use in a time-shared computer, suitable modifications can make them compatible with a time-sharing system. DYNAMO and GPSS have already been modified in such a manner, but as yet these systems do not take full advantage of time-sharing possibilities. Specifically, programmer interaction with the model during execution is very limited.

The advantages of programmer interaction are several. First such a capability permits the user to introduce new decision rules or input data while the model is running. Of even greater significance is the fact that the programmer can build his model on-line and test the various parts as they are introduced. This manner of testing a model permits the programmer to consider detailed aspects of his model at a time when he is most aware of the problems involved, rather than forcing him to deal with the model in toto when many of the same details are rather obscured by the complexities of the model.

Although it is possible to modify current simulation languages to make them interactive, it appears more promising to incorporate the functions provided by these languages into an existing on-line control system such as OPS-1. The following presents a method for accomplishing this end, the OPSIM system.

---

[1] Welsh, R. L., The Development of an On-Line Computer Simulation System, unpublished M.S. thesis, M.I.T., June 6, 1964.

The OPSIM System

As an example of how one builds a simulation model with OPSIM, consider the model of an order-up inventory problem described by Galliher.[2] Here the object is to compare the effect of different "order to" levels on inventory level, given that the ordering is done once a week and that the lead time is constant at one week.

A way of thinking about this problem is the following: Four distinct events can be recognized:

1. place an order
2. draw a weekly demand
3. report the status of the system
4. record the arrival of ordered goods.

The simulation consists of iterating through this series of events, advancing time one week on each cycle.

The segregation of these events in the above manner suggests the use of an OP to describe each event and the subsequent combination of these OP's into a KOP, the execution of which would be one cycle of the simulation. Those variables used by more than one OP (such as inventory level, quantity on order, etc.) are maintained in common storage.

Although the user could program all of the OPs mentioned above, it is easier to utilize certain special purpose OP's of OPSIM. For example the demand event could be described by the following OPSIM OP's:

OP47 - to obtain random demand from a distribution of demands
OP16 - to move the value of the demand to OP51
OP51 - to perform the remaining functions necessary to complete the description of the demand event.

OPSIM special purpose OP's can also be used to simplify the accumulation of the desired outputs of the simulation.

---

[2] Galliher, H. P., "Simulation of Random Processes", Notes on Operations Research 1959, assembled by the O. R. Center, M.I.T., p. 231-250.

Thus, when the user has the appropriate OP's for his simulation, he begins building his model by creating the OPSIM control system, a KOP which coordinates the execution of the event OP's. The OPSIM control is generally composed of OP's 14, 40, 17, 18. OP14 permits any KOP to terminate by branching to the control system. OP40 performs the major functions of the control system. When executed in the create mode, it requests that the programmer input the numbers and times of the first execution of all event KOP's, and uses this information to initialize a list of all scheduled future occurrences of these event KOP's. It also creates the OP's 17 and 18, OP's for reading and writing KOP's, automatically.

After completing the OP's control system and executing OP40 in the create mode, the user creates the first event KOP. Upon completion, he stores it on a simulated tape file. He then creates and stores the other event KOP's. To begin a simulation, the user executes OP40 in mode 5. OP40 will read in and begin the execution of the earliest scheduled event KOP, for only one KOP is in core at any time, the remainder residing on the disk.

Because of the nature of the model under consideration, the events could have been listed on the same event KOP. Since in general the order of the occurrence of the events will not be fixed, such a simplification will not be possible.

To build a model in which the lead time is probablistic, the programmer would remove the arrival OP from the event KOP and file it as a separate KOP. OPSIM OP's 47 and 41 are added to the original event KOP to draw a lead time from a distribution and to cause an occurrence of the arrival event at the current time plus lead time.

Similarly, if the ordering rule were now changed from ordering once a week to ordering at an order point, the order OP would also be removed from the event KOP and would be executed only if the ordering condition were satisfied. OPSIM OP's 13 and 14 would be used in the event KOP to test for this condition, and to effect the execution of the order OP when required. In the same manner, the programmer utilizes the modularity of the QPS-1 system to describe other variations of the model, controlling the event sequencing with the OPSIM OP's.

Conclusion

The results of this research show that OPSIM is successful in building and testing simple models like the one discussed here. It now needs to be used on a complex problem. It is this type of problem that is hardest to test using the current simulation language, and it is here where the on-line approach to building simulation models is expected to show its greatest advantage.

OPSIM will be improved in several ways. First, it will automatically incorporate improvements in the power and convenience with which OPS-1 allows the programmer to control his model. Secondly, it will improve from the addition of new OP's which add more specialized simulation functions. Finally, it will improve most from use. As programmers make use of the characteristics that it now has, they will discover things they would prefer to have done differently. It is impossible to foresee the best way to organize such a system as this, and in many parts, the best way is dependent on the problem being solved. However, OPSIM is a start. It provides a general on-line system that can be used, experimented with, and extended.

OP's in the OPSIM System

OP40 the main control OP for the system

OP50 the OP which determines the size of an order

OP51 the OP which executes the operation caused by a demand

OP52 the OP which executes an arrival

OP53 the OP which prints the statistics of a simulation

OP4143 the OP which causes an event KOP

OP47 the OP which makes a draw from a random distribution

OP46 the OP to plot frequency distributions

Other OP's to control data location and movement comprise the remainder of the OPSIM system. All the OP's of OPSIM are discussed in detail in the thesis by Welsh.

Chapter VII

This empty page was substituted for a blank page in the original document.

APPLICATION OF ON-LINE COMPUTATIONAL TECHNIQUES TO MANAGEMENT OBJECTIVES IN THE AREA OF PROJECT SCHEDULING AND CONTROL

by

John Brach

Introduction

This chapter describes an application of the OPS system to scheduling and control of a project made up of individual activities. The objective was to schedule activities in a way that minimizes over-all project cost and thereafter to provide a facility to monitor project progress and whenever necessary, to reschedule the remainder of the project.

Six OP's form the basis of the system. These six represent a compromise between flexibility for the user and efficiency in computation.

Background

The scheduling algorithm used is based on the critical path method. A project is broken down into its individual activities and a network is developed using these activities in such a way that the actual relationship of one activity to another is depicted. Each activity is supplied with a normal time and a crash time and a cost associated with each. The critical path analysis generates a series of completion times, each with an associated direct project cost, which is a minimum for the duration. There will be one sequence of activities which determines the completion time. This sequence or path is called the critical path. The activities on paths other than a critical path will have float.

Although the first project schedule, obtained with each activity at normal time, will have the lowest direct cost, there are two situations which render importance to the remaining schedules. First, a deadline may exist which is earlier than the first or normal schedule. In this case, we would have knowledge of the activities to speed up or "crash" to minimize additional direct cost. Second, a consideration of indirect cost on a project usually indicates a <u>total</u> project cost minimum at a project duration somewhat less than indicated by the normal schedule.

All the terms mentioned have strict definitions too lengthy to include here. A good reference is "Lecture Notes on Critical Path Scheduling", by J. Lloyd Cutcliffe, a publication of the Department of Civil Engineering, Massachusetts Institute of Technology.

OP35

This OP is a network input program. It [illegible] a file number from which to read an activity array. This array [illegible] of [illegible] quantities of information about each activity in the network.

TCR = the crash time

This OP [illegible]

A [illegible]

set [illegible]

stored in [illegible]. The last function of [illegible]

of activities, in the network.

OP36

OP36 is a routine to compress the activities as represented in the data base. This is necessary to insure that they conform to the requirements of the scheduling algorithm. The activities must be ordered by I and J and the modes must be numbered in a strictly sequential manner. This means every number must be present from one to the maximum node number and every activity must have a J value greater than its I value. OP36 takes a network numbered loosely and compresses the node numbers as required.

This OP will be required under two circumstances: first, original network input may be loosely numbered to allow for manual revision of the network without renumbering the entire thing. For instance, activities may be added. The second case wherein OP36 will be used is following an automatic updating of a network by OP40.

The original node numbers are saved for output to identify the activities to the user.

OP37

OP37 is the scheduling algorithm. It operates on the network stored in the common data base. A schedule of project durations and associated costs for crash programs are summarized and printed. A tape file is created corresponding to each schedule number and contains the earliest event time of each node for the current iteration.

The scheduling algorithm is based on Fulkerson's solution to the critical path problem. It consists of four sections: the flow computation, backflow computation, activity flow updating, and flow summary/output. The logical sequence of program execution proceeds as follows:

1. The flow computation is entered and a forward pass over the network is performed labeling each node.

2. The backflow computation is entered and a reverse pass over the network is performed. When a backflow condition is detected, control transfers to (1) to attempt to revise forward flow. The algorithm iterates between (1) and (2) until no more revision of node labeling is obtained.

3. Activity flow is updated according to current node labeling.

4. A summary of flow reaching the end node on each iteration is maintained and printed together with the event time of the last node.

OP38

For each schedule generated by OP37, there corresponds an array of individual activities with their starts, finishes, durations, and floats. OP38 accepts a schedule number and outputs the activity array corresponding to that schedule. To do this, a tape of earliest event times produced by OP37 is read and the computation is performed. The quantities calculated by this OP are not saved. The original activity node numbers are present in the output so that these activities may be related directly to the network diagram.

OP39

OP39 is called the interface generator for lack of a better name. It requests a schedule number (previously generated by OP37) and a day. The day specified is the day on which you desire knowledge of activities in progress. For the schedule specified, OP39 supplies a list of all activities scheduled to be in progress or completed as of that day, with their durations and percent complete. Because many activities will possess float and therefore leeway exists in the time when an activity can be performed, the cross section viewed has each activity scheduled at its latest possible starting time. This means that if any activity is behind in relation to the interface generated, the project will be delayed and not finished on schedule. The interface can only be defined, in certain cases, by the completed activities to the left of the interface as well as those scheduled at that time. This is because the position where the interface passes a chain of the network may be a float of an activity rather than the activity scheduled itself and therefore, no activity on that chain would otherwise appear.

OP40

This operator performs the updating of a project network at any time during the actual project. It, in effect, creates a new project network representing the remainder of the project from that point.

The first information OP40 requests is a tape number on which to place the updated network. This should be greater than twenty so as not to correspond to a schedule output tape. Next, a description of the interface currently existing between completed activities and non-completed activities. Five pieces of information about each activity are required: 1) the tail node I; 2) the head node J; 3) the new estimated normal time to complete the activity; 4) the new crash time estimated; 5) the slope, i.e., cost/day to shorten from (3) to (4).

With this information, a new network is developed by bringing all the specified activities together to a common starting node. A tape file is then written.

Sample Problem

A sample problem is included to illustrate the input and output of the system as well as the logical sequence of steps to analyze a physical problem. [illegible]

The project selected will be called "Project X". It was chosen not so much for its physical resemblance to [illegible] but because the [illegible] the flow algorithm. A network diagram of Project X is included herein.

The analysis starts with OP36. Project X is stored on tape 1003. Next OP37 is executed to schedule the project. Seven schedules are produced for Project X. This information allows us to plot directly the project direct cost curve.

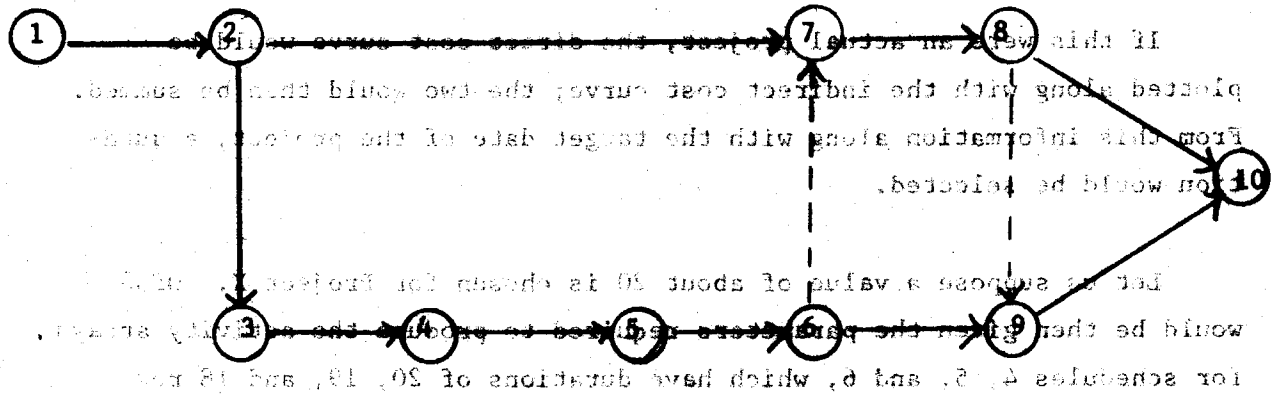
If this were an actual project, the direct cost curve would be plotted along with the indirect cost curve; the two would then be summed. From this information along with the target date of the project, a duration would be selected.

Let us suppose a value of about 20 is chosen for Project X. OP38 would be then [illegible] the activity arrays for schedules 4, 5, and 6, which have durations of 20, 19, and 18 respectively.

With the preceding information at our disposal, we could now schedule Project X. All this would be done prior to its actual start.

The remainder of the sample problem has to do with control of Project X during its execution. At any time, OP39 may be executed to get the

interface between activities scheduled to be completed and those not scheduled to start as yet. OP39 was executed on day 13 and reported that activities 6-9 and 7-8 may still be in progress but all those previous should have been completed. Assuming the project to be behind on day 13, and in need of rescheduling for possible reduced extra cost, OP40 was executed to reschedule the remainder of the project. New data for activities 6-9 and 7-8 were supplied. OP36 was executed to renumber the new project X after OP35 read it in. Then OP37 produced a new set of schedules



PROJECT X

Chapter VIII

This empty page was substituted for a blank page in the original document.

AN AUTOMATED STOCK EXCHANGE USING OPS

by

Stephen Whitelaw

Introduction

The following is a preliminary report of a project to automate large financial markets, particularly ones such as the New York Stock Exchange. The program as it now stands enables the computer to function as both a broker's office and a crude market. The user may directly enter transactions through the computer and also have his account kept up to date. In effect, the console appears to the user as his broker's office. The user has an account, can buy or sell stocks, obtain quotations on any and all stocks, and also deposit money in his account. The various operators to perform these functions are called using the standard OPS system.

Automating the Exchange

The most difficult task in automating the exchange is the programming of the specialist function. The bookkeeping part of the job is conceptually comparatively easy. The specialist, however, not only has to bring matching orders together, but he also must set a price and decide how much stock to offer. His decisions are based on the following variables:

1. The Book of outstanding orders
2. The particular stock's trend
3. The specialist's current position.

It is also conceivable that the specialist may want to consider

4. The entire market trend
5. The general position of the economy
6. The particular company status
7. External events.

These last four variables are conceptually different from the first three in that they are external to the market. It may be that, except in exceptional cases, the specialist should let these influences work themselves out in the daily trading rather than let them influence the market directly by changing the decision rules. The market should reflect the general condition of the economy and the particular stocks, rather than having the market affect the value of the stock by a change in the market mechanism. The programmed specialist would consider the three variables in making his decisions on pricing and amount of stock offered. The resulting prices would then be tested to see if they fall within a certain tolerance limit. If they are outside the limits, the operator will have a message typed back and he can either interfere by setting the price himself or allow the price to stand. If the market is affected by an external event that would cause a run on the stock that the normal decision rules could not handle

the operator could intervene by changing the parameters of the decision rules themselves in order to insure an orderly drop or rise in price. This gives the operator the ability to affect individual transaction prices and price decisions in general.

The computer also must consider the position it has taken in the market. As more and more is invested in a particular stock the point spread will probably grow wider as the specialist is forced to withdraw his support. The emphasis should be on an orderly decline and the capital behind the specialist should be enough to insure this. If the amount invested is greater than a limit either the market will have to be temporarily closed or some form of human intervention will be necessary.

By insuring that all decisions are made according to standardized procedures any conflicts of interest will be minimized. Of course, every time the specialist enters into trading some conflict of interest will ensue but it will be equally likely to happen to any customer. This is the price that must be paid for a continuous stable market. The computer will not try to make money other than the difference in the spread. This should eliminate the prime conflict of interest since the computer will not be prevented from taking a position when necessary due to self interest. The amount earned on the point spread will probably be enough to cover losses and costs. If it is not, a commission can be charged on all transactions over and above broker's fees. This system will not allow the use of not-held or other discretionary orders, but all other including market, stop-loss, limit, and possibly stop-limit orders could be handled.

There are two primary objections to this system voiced by the New York Stock Exchange:

1. Who will bank the specialist?
2. How can a computer be programmed to meet all contingencies?

In answer to the first question, there is no doubt that this scheme will entail a thorough reorganization of the specialist function. The exchange will probably have to bank each separate specialist for each separate stock. The operation will be, like the exchange itself, a non-profit operation. There is no doubt that short run profits and losses will be made, but these will be programmed so they even out in the long run. Funds for the original capital investment need not be raised all at once, since the exchange could gradually convert the individual markets as specialists retire, with a deadline on conversion of all markets. Certainly the amount needed to back the market need not be too large, since the requirements for specialist investment are, at present, very low. Eventually these requirements would have to be raised in order to provide adequate stability.

The second question is partially answered by providing human interaction with the computer. However, as I said before, it is not absolutely clear that this feature is always desirable since the market should reflect values, not affect them. This is shown by the intense dislike of anyone connected with the exchange for any external control over their market. What they don't realize is that the specialist, by employing his knowledge of the outside world, is affecting the market externally himself. What I propose to do is to affect the market by considering outside influence, but do it systematically and with a minimum of external influence and personal bias.

Note: The following operators were the first ones programmed, and form only part of the operator set as of August, 1964. The set is being expanded to include the specialist's function.

Description of Operators

OP41

This operators is always the first one called in the system. It is necessary to set up a random number generator to enable my "specialist" to operate in a random manner. If the user wishes to have the list of operators available, he can hit carriage return after the console types back SUPPRESS 1.

OP42

This operator sets up an account for the user. Each separate user only uses this operator once unless he wishes to have several accounts. Account numbers are assigned, and if the user already is in possession of some stocks, he may give them to his broker who will put them in his account. He does this by specifying the number of different types of stocks he has, then the particular stock number and the corresponding number of shares. The stocks are entered at the current market price and from then on until they are sold the computer keeps track of their net capital gain (or loss). The user can inquire at any time what his gain (or loss) is by using OP44. Only six accounts can be kept at present, but this will be extended.

OP43

This operator will give the current market price of any of the ten stocks listed. The user merely enters the stock number and the price is returned to him.

OP44

This operator gives the user the current status of his portfolio including an inventory of the number of shares of each stock, the price at which the stock was bought or the market price at the time he entered it at the broker's office, the present market value of the stock, the net gain (or loss), and the number of shares held. Then his total original investment in stocks, his total present worth, and his total net gain is printed.

OP45

This operator enters in market orders to buy certain stocks. In a real market, these orders are either filled by the specialist or by his book of outstanding sales. When the market order is received, the program transfers to an update routine. This routine determines the market price of the stock and executes the order. The market price is determined by a random number generator which is modified by the book of outstanding limit orders (see OP49). The price of market orders is determined as if a specialist picked a random price between the highest price of an order and the lowest price of a sale on his order book and executed the transaction at that price. If the price he picks corresponds to a limit order, that limit order is also executed. The operator prints a confirmation of the order, the purchase price and total amount of the order. It also prints the user's old account balance, amount spent, commission, and new balance.

OP46

This operator is much the same as OP45 except it is used to execute sales at the market price. The price is determined the same way as in OP46. The operator prints the sale price, the proceeds of the sale and the user's new balance.

OP47

This operator is used to obtain prices on all ten stocks in the market.

OP48

This operator is used to deposit or withdraw money in the user's account. If the user attempts to purchase a stock and his balance is insufficient, he will get a message back telling him to deposit more money. His transaction will not be consummated if his balance is too low.

OP49

This operator is used to enter limit and stop-loss orders and sales. The user enters his account number, the number of the stock to be bought or sold, the number of shares to be bought or sold and the price above which or below which the transaction should take place. These orders are entered into the specialist's book. As soon as the market price goes above or below the triggering prices the orders are activated and executed at the triggering price. The book is updated each time the market price changes. The market price will change every time one of the nine operators are used, so the specialist always keeps his book updated. As soon as a book transaction is completed the console will type a message announcing the sale or order and the price of the transaction, plus other information on number of shares, new balance, etc.

Other Subroutines

UP

This subroutine is called by all OP's 42-49 and is used to update all the prices of the stocks. A random number generator is used.

CL

This is called by update and it drives the clock by an increment of time each time it is called.

CH

This is called by update after the prices have been established. It checks to see if any transactions in the book should be activated.

OC and SC

These are called by CH if an order or a sale in the book is to be executed because the price has gone past the trigger price.

General Comments

This program is realistic in its method of handling limit orders, but not in its handling of market orders. When the market goes above or below the triggering price the book transactions are activated as it is done at the stock exchange. Market orders, however, are determined at a price somewhere in between, or at, the highest book order and the lowest book sale price as is conventionally done, but the actual price in between these limits is random with a factor added in proportional to the number of shares bought or sold in the last period. This of course is purely arbitrary but it does give fairly reasonable looking prices. Market orders can be considered as being bought by a specialist at a price that differs by a random amount from the last price. If this price happens to be above or below a triggering price the book sale that triggers the price is consummated at this price, however there is no reason to assume that these two transactions were an order and a sale with each other. All market orders are actually made with our omnipotent specialist, and when he decides the time is ripe he will trigger his book orders.

Presently, only six accounts are kept and only ten stocks handled. Obviously this can be extended. For each user, both his stock inventory and his money balance accounts are continuously kept.

This empty page was substituted for a blank page in the original document.

Chapter IX

This empty page was substituted for a blank page in the original document.

AN ARRAY PROCESSOR

by

Mayer Wantman

Introduction

VECOPS was designed as a general purpose array manipulator which allows the user to inject himself into the analysis of his data. It is written within the framework of the OPS-1 system.

Data is represented within the computer as a two-dimensional array of 50 rows and 20 columns, or 20 vectors of 50 elements each. This particular configuration accomodates a broad class of problems, but was chosen arbitrarily and can be changed.

Description of Operators

OP26

This OP has two different modes of operation.

A. Simple polynomial regression.

The user specifies the column indices of the independent and dependent variables, and the degree of the polynomial to be used. Results are printed out as the equation representing the best fit. In addition the RMS deviation, as a measure of the goodness of fit, is printed on line.

A vector of weights may be specified if the user wants to be sure to fit certain points better than others. The final curve is displayed on a cathode-ray tube to help the user develop a "feel" for how the curve fits the data, where the large deviations occur, etc.

B. Multiple linear regression.

The user specifies that vector K is to be regressed on vectors I through J. The regression coefficients are printed in equation form. No curves are displayed.

OP28

OP28 allows the user to specify and apply a fairly broad class of transforms to any two vectors to produce a third vector. The transform is of the form $X_i = AF(X_j) + BX_k$, where i is the index of the new column. A and B are constants specified by the user, F is one of seven transformations, and j and k are the two transformed vectors. Transformations proceed element-by-element, and elements may be chosen selectively.

The seven transformations now available are:

1. Addition
2. Absolute value
3. Square root
4. Multiplication
5. Division
6. Exponentiation
7. Logarithm

OP28 also provides for the automatic entry and use of two standard vectors: the vector of all ones, and the vector of consecutive positive integers.

OP29, OP30

OP29 is the principal means of entering and changing information in the data array. The user can print the contents of the array (comprehensively or selectively), edit the array by rows, or input data directly (with or without echo check to verify correct transmission).

OP30 provides for storing and retrieving all or part of the vector array in a permanent disk file.

OP45

This operator can be used only with a particular display system at M.I.T. The basic philosophy is straightforward, however, and could be easily implemented on any digital display device.

OP45 displays as many points as there are rows in the data array. The columns to be treated as x and y are specified, and the OP calculates that scaling which will ensure full utilization of the display arrow. The points are then displayed.

Example

As an example of the application of VECOPS, suppose we are interested in simulating the operation of a metropolitan transportation system. To do this it is necessary to have some idea of the arrival rate of people at, say, bus stops. Assuming that the arrival process is Poisson, we want to know how the average arrival rate changes during a typical day.

Our experimental data consists of a series of observations of hourly arrival rates, so we have two vectors of information: vector 1 contains the hour of the day, and vector 2 contains the number of arrivals since the last hour.

We can now use the array processor to select a suitable approximation for use in the model.

The data are input to the computer in the following form:

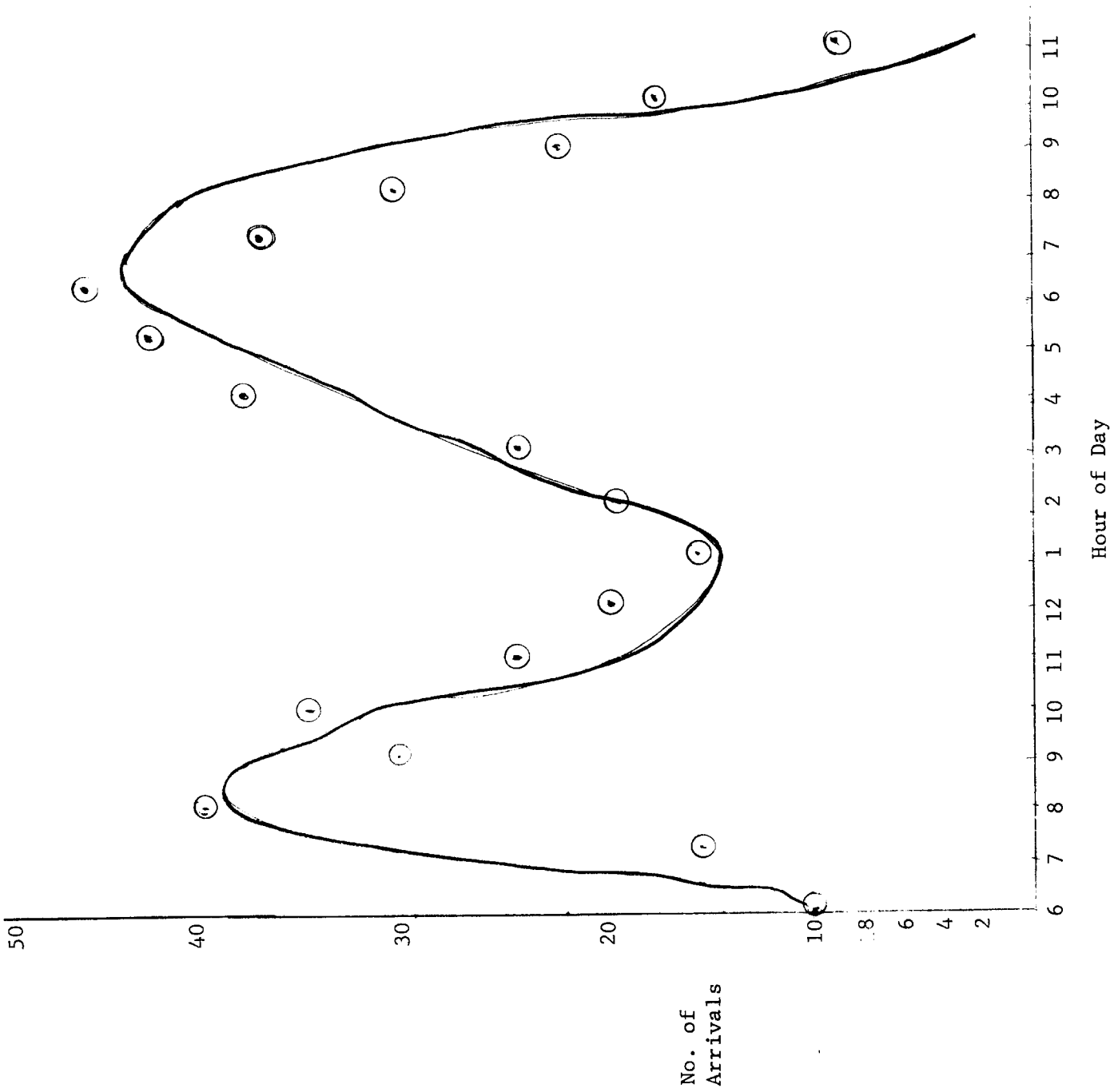
Time	No. Arrivals
6 a.m.	10
7	15
8	40
9	30
10	33
11	25
12	20
13	15
14	18
15	22
16	38
17	42
18	45
19	34
20	28
21	22
22	17
23	8

These data can then be plotted, using an OP45, to yield a display which looks like the accompanying graph. The appearance of the points suggests a 4th degree polynomial fit. This fit is then applied with OP26, yielding the equation

$$y = -.01175X^4 + .6527X^3 - 12.98X^2 + 109.5X - 304$$

as the best 4th degree fit. The RMS deviation is 7.1

Raising the degree of the fit to 5 affords little reduction in the RMS deviation, further indicating the appropriateness of the 4th degree fit.


No. of Arrivals
50
40
30
20
10
8
6
4
2
6 7 8 9 10 11 12 1 2 3 4 5 6 7 8 9 10 11
Hour of Day

Chapter X

This empty page was substituted for a blank page in the original document.

AN ACCOUNTING SYSTEM WITHIN ØPS

by

Gordon C. Everest

Introduction

Traditionally, accounting has been a process of successive aggregation of data to present more and more summarized reports to management, owners and the government. It should not be necessary to maintain a relatively frozen, hierarchical structure of reports. Ideally, we would like to maintain the identity of every single business and financial transaction of an organization. The information associated with each transaction would include such things as the date, with whom the transaction was performed, what tangible or intangible commodity was received and in what quantity, where the transaction was performed, for what reason the transaction was performed, and who, in the organization, is responsible for the transaction.

By performing the accounting function within a computer system it is possible to achieve more timely extraction of information. The input to such a system would consist of initial levels of resources (the Balance Sheet) and a record of the flows of resources through time. Interrogation of the data base could take place when information as to levels or flows is needed and not necessarily only on a periodic basis. Within the ØPS system it is possible to achieve great flexibility and open-endedness in the construction of operators to extract information from or manipulate data in the data base. With this facility there is no end to the way information can be derived as to the flows and levels of resources within an organization.

In searching for a concrete basis from which to build up an accounting system within ØPS I decided to use my own personal code of accounts. For three years I have been keeping a detailed record of my income, expenses, assets, and liabilities and in that period I have

developed a very useful system of accounts and reports for my purposes, which are essentially the same as for an organization. In this way I am in a better position to evaluate the uses of the resultant reports and to suggest possible improvements. I can vividly recognize the need for a change or further information, and I can experiment with new ways of presenting information.

Data Base Approach

[redacted]

accumulated depreciation, and the nature of the asset.

Since income and expense items represent resource flows it is appropriate to store these amounts in a multi-dimensional array with the dimensions consisting of object, purpose, cost center, and time.

The data base specifically consists of the following arrays:

DT(10) IT(10)	Words which may be used for the temporary storage of both fixed and floating data in COMMON Storage.
BS(3,10)	Balance Sheet accounts - up to 30 classes.
PL(3,10,10)	Profit and Loss accounts matrix with 30 object classes and 10 purpose classes.
LS(300,5)	An array within which a list structure is set up to store data which is subsidiary to, in support of, or to further break-down the amounts contained in the Balance Sheet items. It may also be used to store P and L data pertaining to past time periods.

Data Base

Since balance sheet items represent resource levels it is convenient to store the amounts in a one-dimensional array with an attached list structure for storing subsidiary information. By doing this we recognize that some accounts or resources do not lend themselves to being categorized according to the same dimensions as others. For example, accounts receivable should be broken down by the name of the debtor and, further, by the date of the transaction while a fixed asset should possess such information as date of purchase, estimated life, estimated scrap value, amount of capital repairs or additions, amount of appreciation, accumulated depreciation, and the nature of the asset.

Since income and expense items represent resource flows it is appropriate to store these amounts in a multi-dimensional array with the dimensions consisting of objects, purpose, cost center, and time.

The data base specifically consists of the following arrays:

DT(10) IT(10)	Words which may be used for the temporary storage of both fixed and floating data in COMMON Storage.
BS(3,10)	Balance Sheet accounts - up to 30 classes.
PL(3,10,10)	Profit and Loss accounts matrix with 30 object classes and 10 purpose classes.
LS(300,5)	An array within which a list structure is set up to store data which is subsidiary to, in support of, or to further break-down the amounts contained in the Balance Sheet items. It may also be used to store P and L data pertaining to past time periods.

Subsidiary List

Each cell within the list structure consists of five integer fields and, at present, there are 300 cells defined but this is purely arbitrary.

Cell:	ID	NAME1	NAME2	DATA	POINTER

ID — Contains the Balance Sheet account number which is negative for the first cell of a sublist.

NAME — 12 alphabetic characters made up of NAME1 and NAME2. This will be the name of a Balance Sheet account or of an account in its subsidiary list.

DATA — If the cell represents a Balance Sheet account, this field will contain the pointer to the first cell in the subsidiary list associated with this account. If the subsidiary list is empty, this field will be set to zero.

POINTER — will point to the next cell within the same level list. If this cell is the last one in the list, the pointer will be set to zero.

DATA — If the cell is in a subsidiary list, the data field will contain the amount (cents = dollars x 100) which is to be associated with the NAME. If this amount goes to zero after it has been updated with a transaction, this cell will be taken from the subsidiary list and returned to the List of Available Storage (LAVS).

When the system is initialized all cells should be attached to the List of Available Storage, through the use of ØP23. The first cell of the list array [LS(1,n)] will always contain pointers to the major list as follows:

LS(1,1) — pointer to the first cell of the ACCOUNTS LIST.

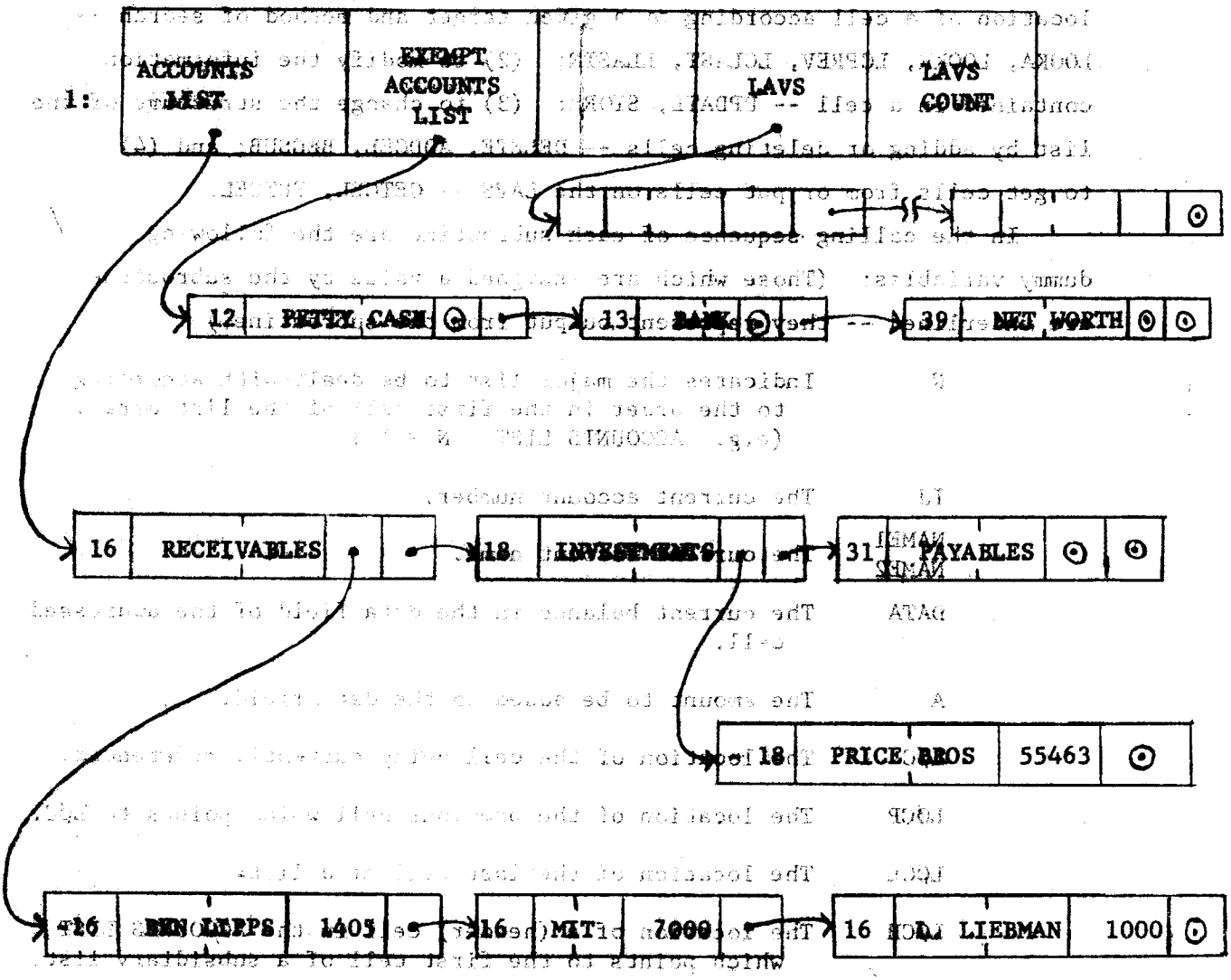
LS(1,2) — pointer to the first cell of the EXEMPT ACCOUNTS LIST. (those accounts which don't have a subsidiary list)

LS(1,3) — not used at present

LS(1,4) pointer to the first cell of the List of Available Storage.

LS(1,5) contains a current count of the number of cells on LAVS.

The following is a graphical presentation of an illustrative list structure.



Subroutines to Operate on the List

In order to facilitate the use of the list structure a set of 12 subroutines have been written: (1) to search the list to find the location of a cell according to a given target and method of search -- LOOKA, LOOKN, LCPREV, LCLAST, LLASTN; (2) to modify the information contained in a cell -- UPDATE, STORE; (3) to change the structure of the list by adding or deleting cells -- DELETE, ADDCEL, RECSUB; and (4) to get cells from or put cells on the LAVS -- GETCEL, PUTCEL.

In the calling sequence of each subroutine are the following dummy variables: (Those which are assigned a value by the subroutine are underlined -- they represent output from the subroutine.)

N	Indicates the major list to be dealt with according to the order in the first cell of the list array. (e.g. ACCOUNTS LIST: N = 1)
IJ	The current account number.
NAME1 NAME2	The current account name.
DATA	The current balance in the data field of the addressed cell.
A	The amount to be added to the data field.
LØC	The location of the cell being currently referenced.
LØCP	The location of the previous cell which points to LØC.
LØCL	The location of the last cell of a list.
LØCH	The location of a (header) cell in the ACCOUNTS LIST which points to the first cell of a subsidiary list.

The functioning of each of the list processing subroutines is described as follows. Except for DELETE which calls PUTCEL, and PUTCEL, which calls STORE, the subroutines are independent of each other.

LOOKA (N, IJ, LØC)

To search the major list designated by N for a cell which has an account number equal to IJ. If the search is successful, LØC will

be set equal to the location in the array of the corresponding cell. If the search is unsuccessful, in other words, account number IJ doesn't exist in the major list N, LØC will be set to zero.

LOOKN (LØCH, NAME1, NAME2, LØC)

To search the subsidiary list of LØCH for a cell which has a NAME equal to the name represented by NAME1 and NAME2. If the search is successful, LØC will be set equal to the location of the corresponding cell. If the search is unsuccessful, in other words, no cell within the subsidiary list contained the name NAME1+NAME2, LØC will be set to zero.

LCPREV (LØC, N, LØCP)

To find the location of the cell which points to LØC. Every cell has one and only one cell pointing to it. If the search is successful LØCP will contain the location of the previous cell. If the search is unsuccessful, LØCP will be set to zero.

LCLAST (N, LØCL)

LØCL will be set equal to the location of the last cell in the major list designated by N. (Note: this may be 1 if the specified major list is empty).

LLASTN (N, IJ, LØCL)

To find the location of the last cell of the subsidiary list of IJ, and assign it to LØCL. LØCL will be set to zero if the subsidiary list of account IJ is empty or if account IJ does not appear in the major list N.

UPDATE (LØC, A, DATA)

To add the amount A to the DATA field of the cell whose location is LØC. DATA will contain the new amount in the data field.

STØRE (LØC, ID, NAME1, NAME2, DATA)

To store infomation in the first 4 fields of the cell whose location is LØC. If ID = -1, or NAME1 = -1, or NAME 2 = -1, or DATA = -.001, then the corresponding field in the addressed cell will be left untouched.

DELETE (LØC, LØCP, ISW)

To delete the cell whose address is LØC from the list structure and update the appropriate pointer of cell LØCP. It is possible to attempt to delete a cell representing an account which has a subsidiary list. If ISW = 0 then no test for this condition will be made. If ISW = 1 and the condition exists then this subroutine will ask the user via the console whether or not he wants to carry out the deletion. If the deletion is to be carried out then all the cells in the subsidiary list will be returned to the LAVS. If not, no action will be taken. If the cell LØC is the first of a subsidiary list, then the minus sign will be moved to the first field of the next cell.

ADDCEL (LØC, LØCL, N)

To add a cell whose location is LØC to the end of the major list designated by N. LØCL is the location of the last cell in the major list. If LØCL = 1, then this subroutine will serve to start a major list. Also, to add a cell to a non-empty subsidiary list.

BEGSUB (LØC, LØCH)

To begin a subsidiary list for the account represented by the cell LØCH and to set the first field of LØC equal to the negative of the corresponding account number.

GETCEL (LØC)

To get the location of the first cell in the List of Available Storage and update the appropriate pointers. The LAVS COUNT is

decremented by one and then tested for zero. If the COUNT is found to be zero then the message 'LIST OF AVAILABLE STORAGE IS EMPTY' is typed on the console. This message may occur if the LAVS is not initialized prior to its use.

PUTCEL (LØC)

To add the cell whose location is LØC to the LAVS and increment the LAVS COUNT by one. Then the first 4 fields of the cell are set to zero.

Income and Expense Account Classification

Income and expense accounts are normally classified only on one dimension and frequent attempts to incorporate further break-downs are evident in many existing accounting systems. In setting up a system here I have attempted to classify income and expense items on the basis of four dimensions -- object, purpose, cost center and time. The classification on any one dimension must consist of a set of mutually exclusive and collectively exhaustive elements (or as nearly so as possible).

Generally, we would like to get information as to what has happened so far this month, a comparison of data collected in past months, cumulative figures for the period from the first of the fiscal year to the present, and comparisons between budget and actual, and also a framework within which the budgets for future time periods can prepared.

There is one problem which sometimes arises -- how to handle transactions which affect the Income Statements of past periods. If we get one that affects future time periods then no problem arises because the item is merely held on the Balance Sheet as a prepaid expense or income received in advance until such time as the affected time period is arrived at. In the present system this entry affecting a past period becomes easy to deal with. Since all past records are retained in secondary storage (on simulated tape files) the affected file can be read into memory; updated and then saved again. This would also require that the Balance Sheet data for all periods from then until now be all correspondingly updated. Although this could be done automatically, it is not done within the present system. A similar problem to this one is what to do with items which are large and non-recurring? It is undesirable to put them into the regular classification since they would tend to distort the true picture. This problem has also not been dealt with in the present system.

On the Object dimension the main question to be asked in setting up the code of accounts is: What commodity or service is received or

given in the transaction? In traditional accounting codes this dimension probably has the greatest influence. However, we will often see a division made on the basis of purpose through the definition of two or more object classes. By clearly defining a system of classification on the basis of two dimensions instead of just one it is possible to arrive at a concise set of codes that can lead to more meaningful reports.

On the Purpose dimension we can ask such questions as: Why or for what purpose is the money spent? or What do we hope to accomplish? This, I suggest, is the most difficult of the dimensions to classify.

On the third dimension we divide up the income and expenses according to a Cost or Profit Center by asking Who is responsible? or who should be charged? or perhaps Where or what should be charged? This is probably the easiest dimension on which to break transactions down into classes which are mutually exclusive and collectively exhaustive. It can be done on the basis of geographical location, departments, products, persons in charge, or functions (the later may overlap with purpose)..

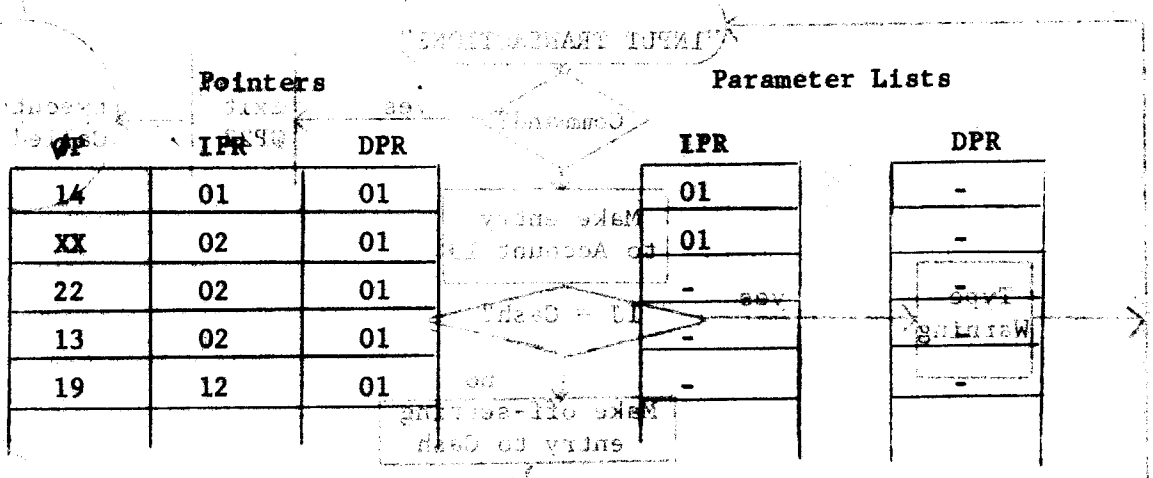
The last dimension is that of Time -- to include the transaction in the time period in which the benefit was derived. This is not always a clear cut decision but it is one that must be made. As suggested earlier, this is probably the first dimension that will break-down with the use of time-sharing systems. We would then ask: Is there another way to represent the dynamic, continuous nature of income and expenditures? On this dimension information would be stored in an exponentional manner. In other words, as we move back in time, the level of aggregation across this dimension would increase.

Within the present system I have eliminated the Cost Center dimension since it is the easiest to define and also, in the light of my personal income and expenses I am the only one responsible and, hence, the only cost center. The object and purpose dimensions are included in the array which is defined within the system. The time dimension is achieved by storing information relating to other time periods in

a file kept in secondary storage. This does not impose any great restrictions since generally, only one time period will be dealt with at any one time.

Operation of the System

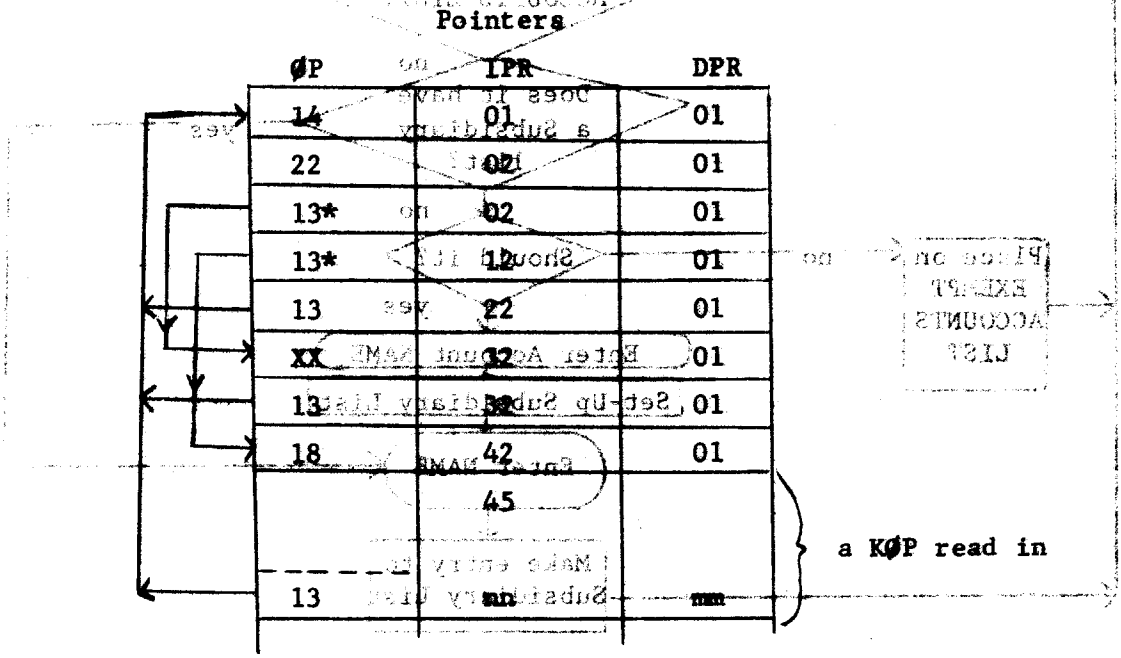
The KØP list is permanently set up as follows:



Pointers			Parameter Lists	
ØP	IPR	DPR	IPR	DPR
14	01	01	01	-
XX	02	01	01	-
22	02	01	-	-
13	02	01	-	
19	12	01	-	-

Note that execution of the KØP must be started on line 01 in order to set up the index in ISAT for the transfer operators (13 and 14).

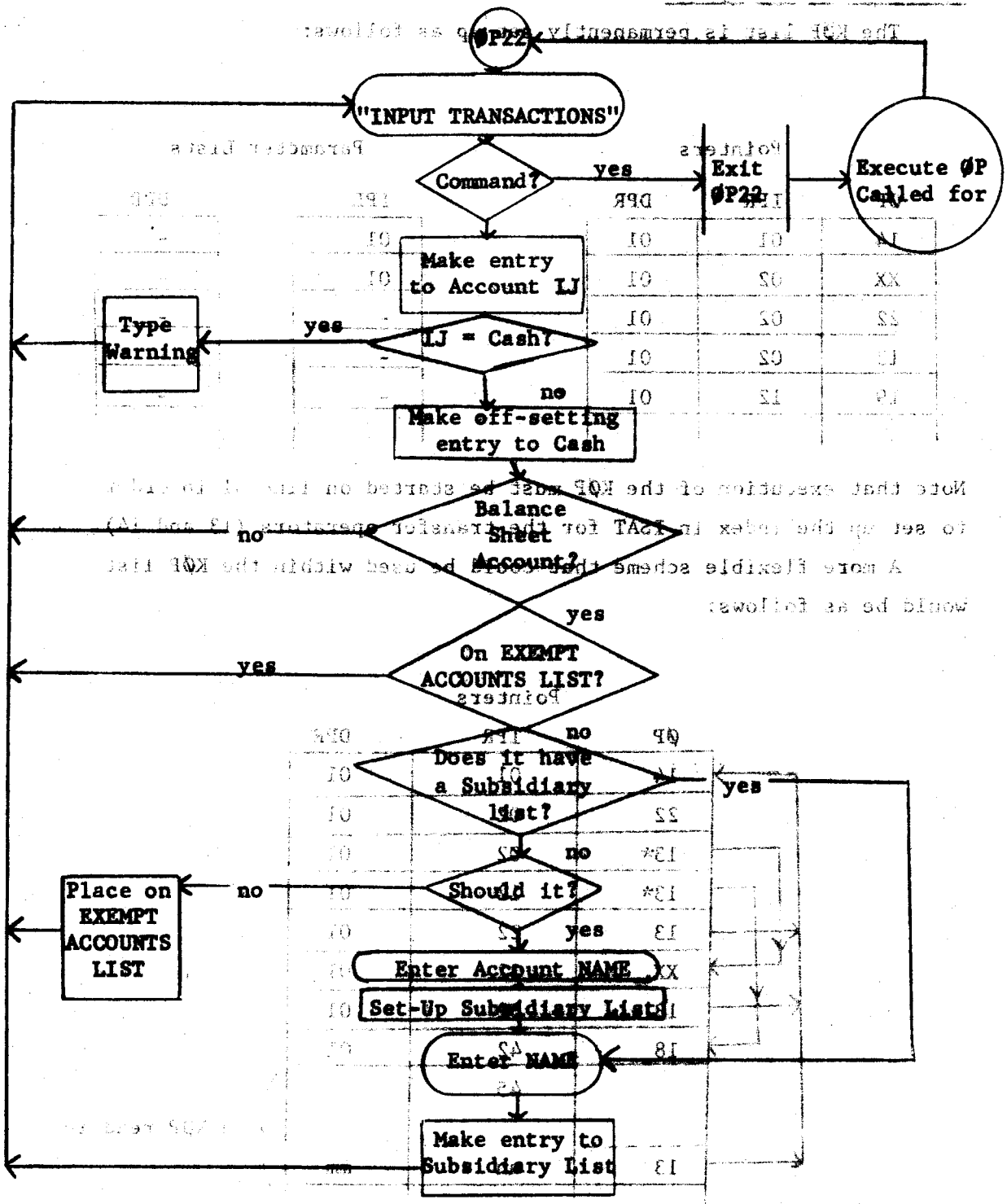
A more flexible scheme that could be used within the KØP list would be as follows:



Pointers		
ØP	IPR	DPR
14	01	01
22	02	01
13*	02	01
13*	12	01
13	22	01
XX	32	01
13	32	01
18	42	01
	45	
13	nn	nn

a KØP read in

FLOW CHART - the system centered around ØP22.



[illegible]

[illegible] a duplication [illegible]

[illegible]

where IJK is the account number [illegible]

[illegible]

he is asked to give the QP number.

After each entry is [illegible]

[illegible] the Balance Sheet (BS) Accounts, or the Income and Expense [illegible]

[illegible]

the entry is for account [illegible] user is warned of this since it will put the "books" out of balance if left. If the entry is [illegible] in the subsidiary [illegible] than the QP will not [illegible] up. [illegible]

[illegible]

This QP closes out all the [illegible] and Expense Accounts to Account 39[illegible]. This QP [illegible] sets the [illegible] to [illegible].

QP40

QP40 is used to print out the current balance of account IJK. For [illegible] of only two digits [illegible], the total balance over all purposes will be printed. This QP will

ØP25

ØP25 prints out NN lines of the KØP, IPR, and DPR lists starting at line SS. If SS is omitted then it is assumed to be line 01. (This is a duplication of ØP10 which I was not prone to use.)

ØP29

This ØP is used to delete account IJ from the list array -- both the cell representing the account and all the cells making up its subsidiary list. All deleted cells are returned to the LAVS. A warning is issued if an attempt is made to delete an account cell which has a sublist and the user is given the option of deleting it or not.

ØP31

This ØP is used to write a file consisting of the current status of the Balance Sheet (BS) Accounts, or the Income and Expense (P&L) Accounts, or the Subsidiary Lists (LS) or all three. The user is required to supply tape numbers for the files and is expected to keep his own record of all the numbers used. This bookkeeping function is one that should eventually be delegated to the system.

ØP32

ØP32 is used to read in a file previously created by ØP31. The user is asked to supply the tape number to be used.

ØP35

This ØP closes out all the Income and Expense Accounts to Account 39, Net Worth.

ØP40

ØP40 is used to print out the current balance of account IJK. For Balance Sheet Accounts the K may be omitted since they all consist of only two digits. For Income and Expense accounts, if K=0 (or blank), the total balance over all purposes will be printed. This ØP will

repeatedly ask for an account number until a carriage return is given.

ØP41

ØP41 will print out a Trial Balance of all the accounts. The user is asked to provide the date to be printed on the statement. This is another function that should eventually be delegated to the system so that the user need enter the date only once. It would then be stored in some common location so that it could be referenced by the other ØPs.

ØP42

This ØP will print out the Income and Expense Statement. The user is asked to supply which purpose will be used. If a 0 purpose is entered, a statement over all purpose will be printed.

ØP43

ØP43 will print out the subsidiary lists of a series of Balance Sheet Accounts, if they have subsidiary lists (up to ten accounts).

ØP44

This ØP will print the Balance Sheet. It will net out the balance of all Income and Expense accounts to arrive at a figure for Net Income. This amount is then included in the Balance Sheet if it is non-zero.

For the Future

A very useful addition to the accounting system would be the facility to set up, modify, and analyze budgets. Any of the operations defined so far which use actual data could be used equally as well with budget data. When a budget is prepared it would simply be filed in its current status, just as files of data are created now. One major change would have to be made to allow a comparison of budget data to actual data: two complete sets of account balances in the memory of the computer at the same time. This would require an expansion of the common data base. With such an extension to the present system it would be possible to compare the two sets of data, be it budget versus actual or one month versus another, and issue reports on the differences between accounts.

Another possibility for the future is an extension of receivables and payables to include sufficient information to allow us to age them, prepare billings, and check on the credit ratings or limits on certain accounts.

11.00

Chapter XI

This empty page was substituted for a blank page in the original document.

DYNAMIC STORAGE ALLOCATION

by

Francis Douglas Tuggle

"When they saw Alice coming, they cried out 'No room!' 'What do you mean,' said Alice, 'there's plenty of room. Move down.' 'More room!' ..."

--Charles Lutwidge Dodgson

Introduction

Dynamic storage allocation refers to the process by which parts of programs are put into memory during the execution of other parts of the program. It stands in contrast to the (now) standard method of having all of the necessary programs and subroutines in memory before starting execution. My work provides a method of carrying out dynamic storage allocation under the OPS-1 system.*

Dynamic storage allocation may be desirable for any of several reasons: to meet space restrictions, to increase running time, or to decrease costs. Two FAP programs, CALOP1 and CALOP2, were written to replace CALLOP and solve my dynamic storage allocation problem.

---

*For a fuller discussion of this work, see an unpublished bachelor's thesis by Francis D. Tuggle, "A Programmed Method of Dynamic Storage Allocation for a Time-Shared Computer," M.I.T., May 20, 1964, Sloan School of Management.

Use

To use CALOP1 in the OPS-1 system, one merely types in LOADGO MAINOP READOP CALOP1. (Any special subroutines used by the OP's should be loaded in between READOP and CALOP1.) From here on, one proceeds to use the OPS-1 system as described earlier. Unbeknownst to the user, here is what CALOP1 is doing: CALOP1 gets the number of the desired OP and loads it into memory; the computer types out EXECUTION, CALOP1 transfers to the OP it has just loaded in. After the OP is through executing, it is deleted from memory (althouth the user is given no indication of this), and control passes back to the program that originally called CALLOP. The user continues in this manner. Except for the computer typing out EXECUTION, he would have no a priori knowledge that all the OP subprograms had not been previously loaded into memory. Should he give either CALOP1 or CALOP2 an illegal OP number (of an OP that does not exist in BSS form on his files), an error message is printed out and control just passes back to the program that called CALLOP.

CALOP2 is similar in nature to CALOP1 but is just a bit more sophisticated. The loading of the main programs is similar. However, the computer then types EXECUTION three times, which indicates that two programs have been loaded, FIRST((FAP) and READR (FORTRAN), which are only used once and then deleted. FIRST(asks for two parameters (READR is used to read the parameters from the consoles) which should be of the form .XXXXX (each--trailing zeros can be deleted). When that fraction (indicated by the initial parameter) of memory has been used up by the user's programs, CALOP2 will announce that fact by printing CHECK TIME and the amount of memory the user has used (in octal) and asking him if he would like to delete all of his subprograms (note that I use .Q. in place of ?). If he does, he should type in at least three characters

Added Note (by Malcolm M. Jones)

It has not been possible to adapt CALOP2 to the OPS system because of some undetected errors, perhaps in the BSS loader.

The intensive studying of CALOP2, however, did yield some positive results. The BSS loader creates a table called MOVIE) which contains the names and entry points of all subroutines loaded into core memory. CALOP2 scans the MOVIE) table to see if a routine is in core before going to the disk to find it. This use of the MOVIE) table has been incorporated into a new, simpler, and smaller CALLOP, which has worked well. By further extending the use of the MOVIE) table, CALOP2 itself could be considerably simplified and a list which it keeps of current OP's in memory could be eliminated.