MAN-MACHINE INTERACTION FOR INDUSTRIAL PROCESS ANALYSIS

by

GALEN D. STACY

Bachelor of Science

Kansas State College of Pittsburg

Pittsburg, Kansas

1956

Submitted to the Faculty of the Graduate College
of the Oklahoma State University
in partial fulfillment of the requirements
for the Degree of
MASTER OF SCIENCE
May, 1971

MAN-MACHINE INTERACTION FOR INDUSTRIAL PROCESS ANALYSIS

Thesis Approved:

Charles M. Bacon
Thesis Adviser

Paul A. McCollum

J. L. Norton

D. D. Durham
Dean of the Graduate College

PREFACE

Over the past several years there has been an ever-increasing emphasis on the development and use of man-machine interactive terminals in computer systems which has brought the power and flexibility of the computer to more effective use in many applications areas. One of the most attractive, if not yet the most economical, of these terminals has been the various combinations of alphanumeric and graphic input devices using a cathode ray tube (CRT) display as the primary output means. Such terminals, together with appropriate software and communications networks, have been used or proposed for computer-aided design, time-sharing, information retrieval, computer assisted instruction, and a variety of other man-machine tasks.

The literature indicates that industrial experience with such devices in process systems under computer control is limited. Application of these systems has been generally limited to implementation of more or less conventional process operator functions. The writer has been convinced for several years that CRT terminals hold great promise for improving efficiency in the experimentation with and study of industrial processes, as well as in operator control. "Process study" includes all means by which improvement in knowledge of an industrial process is attained; i.e., a total systems approach to process analysis observation, calculation, analysis, control, and optimization. It is to the "analysis" function that this work is directed.

This study is concerned only with the software for carrying out interactive analysis of the process and, in particular, the basic operating system functions, data acquisition, data handling, and data analysis. Control and optimization techniques, while dependent on the analysis, are not treated in the present work. Although this study is based on a software (functional) approach, the results can aid in establishing hardware requirements as well as in specifying an efficient set of primitive operations for the implementation.

The central theme taken is that man must play an important part at many stages in the data collection, handling, and analysis to interject selection, direction, and <u>ideas</u> to the repetitive calculations and data handling which is best done by computer. The faster graphic input-output terminal (with appropriate software) can obviously provide quicker and more effective communication than can a typewriter terminal. However, there are many more subtle questions involved. For example, can process computer systems be programmed to adequately support such a terminal in addition to the data collection and control functions, what are the requirements for such analysis, and what, if any, advantages are there in doing on-line analysis with the process computer? As these questions imply, it is not clear whether on-line data analysis, even interactive data analysis, is practical using the process computer. There is little doubt that the efficiency of analysis work can be improved significantly; what is in question is whether a complete job can be done on-line, and how effectively the results of such analysis can be applied to the process in real or near-real time. This investigation was undertaken to provide the necessary framework for answering these questions.

I would like to take this opportunity to express my appreciation to Dr. Charles Bacon and Professor Paul McCollum, my thesis advisers, for invaluable criticisms, counsel, and interest in spite of my logistics problems; to Dr. Joseph Norton, my degree adviser, who helped greatly in planning, administering, and carrying out my study program; to Dean Norman Durham, who understood the problems of candidates remotely located; to Velda D. Davis, for final draft typing and co-ordinating; and to the management of Applied Automation, Inc., for financial assistance.

I would also like to acknowledge the contribution of my wife, Nancy, and sons, Charles, David, and Douglas, whose typing, patience, understanding and other support while I was deeply involved in this work, job, and travel really made this effort possible.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The general problem area to which this work is directed is shown in Figure 1. Process, Data Acquisition, and Models for Control (Blocks 1, 2, and 5) form the "inner" loop which implements computer control of the process. An "outer" loop contains the Process, Data Acquisition, Data Analysis and Reduction, and Model Identification (Blocks 1, 2, 3, and 4). This loop implements a path for systems information analysis.

Data analysis and model updating are generally carried out in a remote off-line environment. For example, in a typical case, a fairly complex test design may be carried out in a few days, while thorough analysis of the data at a remote site may require several weeks or even months, depending upon the sophistication of the remote analysis facilities. Because of time delays and associated environmental changes in industrial plants, problems inevitably arise in application of the analysis results. One such problem occurs because delayed analysis results from plant testing have limited applicability; i.e., the process, controls, or product specifications may have been modified in the interim. Another problem arises when several series of tests are planned to study different aspects of the process. With the analysis results from each series lagging behind, subsequent series must be carried out with limited benefit from previous work. This often leads to repetition of test series or failures in the analysis due to

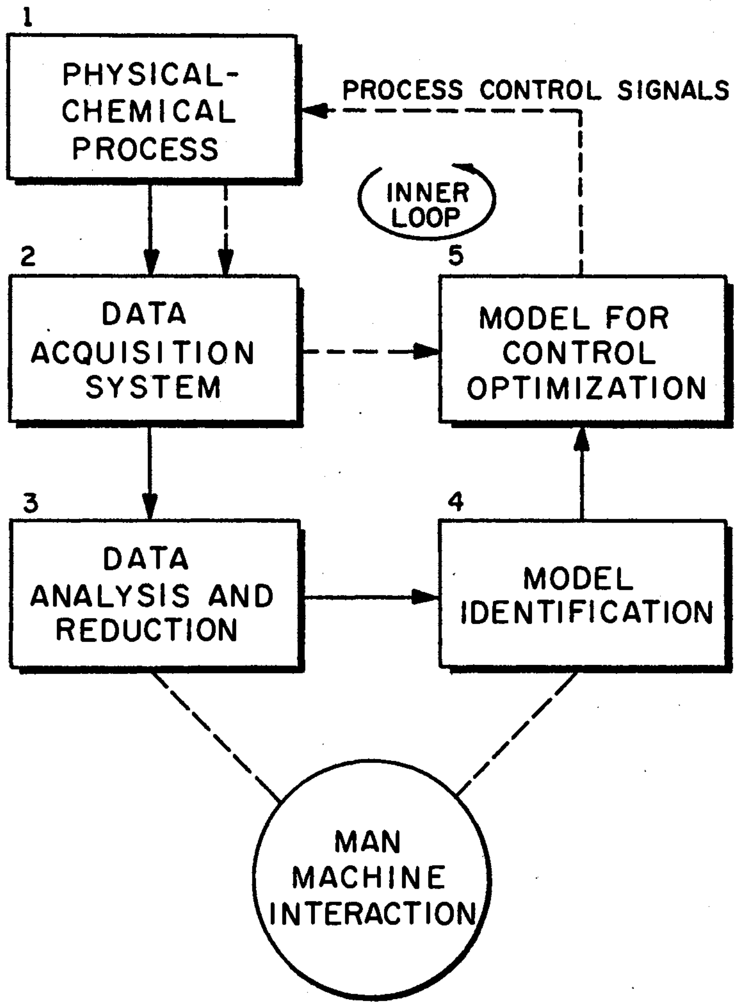


Figure 1. General Diagram, Process Control and Information Systems

unforeseen experimental control problems which must be solved before valid tests can be run. With the rate of generation of test data much greater than the rate of analysis, this problem of incomplete or late analysis reduces the value of the extensive and costly testing. Productivity of testing is also adversely affected by the anticipation of results of past work.

Such problems could be avoided, and at the same time significant analysis productivity benefits realized, if data analysis functions could be put on-line. Since analysis requires thinking as well as computer calculations, a high degree of man-computer interaction is indicated. This investigation is an attempt to outline and evaluate necessary functional specifications of such an on-line information analysis system, as applied to the study of industrial processes.

A graphic display terminal has been chosen for the interactive device because, in many cases, graphics are more efficient for man-machine communication. Many commercial models with more than adequate speed are now available and, when desired, hard copy can be obtained through alternative means from the same source data.

Coons (1) has outlined the general requirements for computer-aided design, many of which apply here, and has summarized very well, in general terms, the reasoning behind joining man and the computer (2). Van Dam (3) has written a summary which traces the history of display hardware technology and man-machine interaction. Extensive references to pertinent literature and bibliography are included. Requirements for "technically and economically feasible" man-machine interaction were outlined, the classic works with very large systems summarized, and several other applications mentioned. Essentially all references were directed to multiterminal design systems or complex picture generation and manipulation techniques, such as in (1).

The literature on the application of interactive devices in industrial process control is limited. Aronson (4) surveyed current installations in industrial process control. He pointed out that the power industry is leading the way in CRT display uses with the obvious applications of graphics for power transfer and distribution diagrams. Pipeline and process flow displays are similar applications in the petroleum and petrochemical industries. Typical uses of displays include setting valves in distribution systems, for large graphic panels, for computer control consoles, instrument displays, etc. Some emphasis is given to available hardware, but virtually nothing is said about the magnitude of the software effort needed to implement such applications, or the required sophistication of the computer hardware to drive the displays.

Currently, most of the interactive data analysis functions indicated in Figure 1 are carried out using off-line techniques. However, "on-line analysis" is being done where extensive hardware/software facilities and manpower exist. For example, Abraham, Betyar, and Johnston (5) describe a specialized system for collection and analysis of neurophysiological data using a 32K SDS 9300 computer augmented by an eight million character disc, CRT display, seven magnetic tape units, two plotters, and other assorted data processing peripheral devices. Lockemann and Knutsen (6) outlined a multiprogramming environment for on-line data acquisition and analysis implemented on IBM System/360 Models 44 and 50 with CRT display, disc, and presumably a normal complement of peripherals. A good summary of the characteristics of data analysis is given, but the description betrays a fundamental inconsistency common to both of these systems: The interactive console language is simple, yet "cascaded" to extremely complex large-computer operating systems. The former paper admitted to a factor of up to 30 slowdown over FORTRAN programs for some console interpreter routines. And FORTRAN code is considered inefficient as used on process control computers.

Moreover, the computer mainframe cost is a relatively small part of the total on-line system. In this view, one may question the economy of many users sharing a single complex large scale processing system, compared to what might be done in data reduction with relatively small dedicated parallel processors and a simple, direct interaction language allowing efficient programming. Ball et al.(7) described a small-computer display system, but the application work was done on a larger system.

The objective of the present work is to investigate the organizational and functional requirements of a software system to implement the data analysis loop in Figure 1 (Blocks 1, 2, 3, and 4). The proposed system would use the capabilities of an on-line industrial process control system dedicated to one or more units in a petroleum, chemical, or other manufacturing plant. Such systems are "on-line" to the industrial process in that they are directly connected through appropriate data acquisition hardware to various digital or analog instruments measuring such variables as flow rates, temperatures, pressures, product quality, etc.

There are several factors which discourage the use of present process computer software systems for combining data acquisition, control, and process analysis (8) (9) (10).

First, the objectives of these already very complex systems are different from the objectives here. For example, manufacturers are committed to provide systems which allow sales to users often not familiar with the detailed characteristics of computers. Moreover, of great importance to manufacturers is the simplicity of their own implementation across many installations with various hardware configurations. These objectives have resulted in unnecessarily large, complex, over-generalized operating systems which require an inordinately large proportion of the process system resources. There are usually many ways for users to accomplish the same results with such systems. Languages (such as FORTRAN) provide user programming convenience at the expense of system efficiency and simplicity. Because such systems have evolved over a number of years through efforts of many people, the investment is too great to correct fundamental errors made early in the development. Thus, unnecessary programming complexity is added to minimize the effect of these errors.

Second, because the analysis system objectives require interaction, duplication of many existing batch translation language functions with similar interpretive functions is indicated if these existing systems are used without modification.

Third, interactive analysis system hardware requirements are different from the configurations assumed in the design of existing operating systems.

And fourth, there is a law of diminishing return in efficiency as separate, large, modules of complex software are added to other such modules, as would be the case here using the extant systems.

Consequently, these operating systems are considered of limited suitability as a starting point for this work. Development of a simpler operating system, with emphasis upon the functional process analysis requirements rather than nonfunctional system requirements, is needed. This, of course, does not preclude use of input-output or other appropriate basic functions directly from such systems.

Therefore, the major objective of this thesis is to outline operating system and functional requirements for applying a CRT display to interactive process analysis using an on-line process computer. This general objective is organized into the following problem areas:

1. Overview and Notation. To accomplish the thesis objective, it is necessary first to carefully outline the general functions to be included which requires development of appropriate definitions and a notational basis for use in the description. This problem is dealt with in Chapter II.
2. Data Basis. One of the most difficult to define problems in software is to determine the organization for system data and parameter structures to best serve the various routines operating with these structures and still retain as much flexibility, efficiency, and simplicity in documentation as possible. This problem, as it relates to the process computer system with interactive data analysis, is approached in Chapter III. With consideration for machine storage efficiency, subroutine structuring, and process flexibility, the best structure for storing and documenting the many parameters characteristic of this system is the simplest, i.e., the matrix. Variable length parameter elements and

system identification of these elements are found to be necessary for the application. A special case of linked allocation list structuring is used to define the matrix identifying structure.

3. Priority Structure and Scheduling. Using the data basis and notation of Chapters II and III, Chapter IV outlines in some detail a formalization of the types of priority structures used in real-time systems, and a rationale for selection of the formal lattice structure. A data basis for the parameters of this lattice structure is developed, and the logic of the program scheduler is discussed. A working core organization for efficient but simplified multiprogramming using this structure is proposed. The concept of the re-entrant level executive to further simplify application program scheduling is introduced.

4. Other Process System Functions. Within the framework of the priority structure of Chapter IV, a general discussion of how the data acquisition, process calculations, data handling, input-output, and error-alarm control may be organized to facilitate interactive manipulations is given in Chapter V.

5. The Interactive Analysis Subsystem. In Chapter VI, the functions of the interactive analysis subsystem are described and related to the previously outlined operating system. These functions generally consist of procedure definition, data definition, procedure execution, procedure or data modification, and verification of results.

6. Example and Conclusions. A comprehensive example of the application of the interactive analysis system is given in Chapter VII. The concepts given in Chapter VI are used assuming the system described in Chapters II through V to tie together and illustrate the power of interactive analysis functions. The conclusions and significance of this work are given in Chapter VIII.

CHAPTER II

OVERVIEW OF SYSTEM FUNCTIONS

Formalization

In the development of a science, one of the most significant steps occurs when large volumes of specific observations, analyses, and conclusions are classified into laws, species, or theories, which organize, simplify, and generalize the technology into a more compact and manageable form. This is referred to as the formalization of the science. Of significance in formalization is the adoption of notations for the structural units such that the science may be applied by use of the notation to observe and describe relationships between the basic structural units. Thus new knowledge "fits" into the old formalization. Formalization also makes available an efficient organization from the fundamental to the more complex, which facilitates teaching as well as new discovery in the science.

In a similar manner, there is recognized a need for some formalization of the languages, procedures, data, and structures used in the collection, manipulation, and output of data in industrial process computing systems. From such formalizations, one hopes to simplify existing functions and, perhaps, provide a more reliable means for evaluating system resource requirements for new functions. The extent to which new functions can be described by the basic formalization would be a measure of success of the formalization.

Language Basis

The first step in formalization of experimental on-line analysis is the selection of a suitable notation or language. Knuth (11) outlined in the Preface page x, the general reasons for adopting a machine-oriented language rather than a more sophisticated compiler language for his classic work. At least two of those principles apply extremely well to the present problem.

First, the programmer is greatly influenced by the language in which he writes his programs, and will select constructions best in the language rather than those which might be best for the machine. The truth in this principle has been demonstrated to the writer many times in his own experience with various versions of FORTRAN and machine languages. The FORTRAN language introduces numerous restrictions to data and program structural efficiency, primarily in the logic and input-output statements. Moreover, selecting constructions best for the machine obviously simplifies the translation function. The opposite view, of course, is that machine resources are less expensive than programming resources, compilers conserve programming resources, thus the compilers are justified. The complex tradeoffs are unique to each situation, and the correct solution actually depends as much upon the skills of the people involved as on the application. Since this work is concerned with a system which allows conservation of machine resources, a language directly relatable to machine language is considered highly desirable.

Second, today's high-level languages, particularly in process control, are not suitable for input-output buffering, problems involving packed data, searching, recursion, and multiple-precision arithmetic, all of which are needed both to write and use the interactive analysis system.

A third significant factor is that compact notations, particularly in an engineer-computer interactive environment, are actually an advantage in use, compared to the bulky English language words of most compilers. An excellent example of a compact notational language is Iverson's APL language (12), which is built from extremely basic (one character stroke) primitives, but accepts and operates upon complete vectors. From such primitives, one familiar with the notation can build and execute complex functions in a short period of time. Unfortunately, it is more difficult to learn such extremely compact notation, and documentation is similarly difficult to read. A compromise between compactness and readability is indicated.

The compromise preferred here is to apply the time-tested macrocoding principle at the assembly language level. Use of mnemonic abbreviations simplifies learning. As Kent (13) points out in his survey of assembly level macro techniques, the use of macros written in assembler language provides a tool powerful enough so that programs for a given application area can be written using only macros. Benefits such as reduced coding effort, flexibility, fewer bugs, and standardized coding conventions are realized; specific machine characteristics need not be considered.

This last point is an important point to consider in the over-all compiler versus assembler question. For, once an (assembler) language level is reached which sufficiently isolates the programmer from hardware or undue format restrictions peculiar to the machine or assembler, then the compiler level has, in a very real sense, been reached. That is, there is an appropriate assembler language macro to implement any desired function of the compiler language; only the questions of arrangement of symbols (format) and definition of macros from predefined macros down to the primitive or machine language level (translation) remain. This approach to system programming has the added advantage of limiting the "compiler language" to only those functions desired for the current application.

The memory efficiency of the macro system will depend, in large measure, upon the number and extent of reuse of each macro in building new macros. There is as yet no scientific approach which will insure optimum reuse; therefore, it is considered desirable that redefinition and other updating of macros be made as easy as possible, so that improved definitions can be introduced as they are discovered.

There is much said in support of high level languages for the purpose of allowing process and control engineers without a detailed knowledge of computers to implement computer control schemes (see, for example, reference (14)). Yet, machine language principles are fundamental to all computing if not engineering. If one who does not know fundamental principles uses a high level language, this has the effect of encouraging the inefficient use of the system. Hence, if machine resources are a consideration (as well as programming resources), users should be familiar with fundamental principles. The language need is for relief from tedious, non-functional hardware peculiarities and unnecessary format detail.

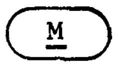
In describing the system and the operational notation for macro-code, terms are used which have various meanings in the literature.

Therefore, the particular usage applicable here is defined in the next section. The reader may find it preferable to skip the following section until a need for the specific meaning of terms or notation is apparent.

Definitions and Notation

The *process* is a real, operating chemical-physical complex of materials and equipment, characteristics of which can be *measured* instrumentally and sensed by the process computing system. *Parameters* are needed by system programs to identify and format data, direct programs, or perform needed modifications upon data. Measurements and parameters are stored as *data units*, each consisting of (a) a single real number, or (b) an element of an ordered sequence of packed data (n-tuple), often called a control word, data parameter, or code word. *A data unit may require less than or more than one hardware word or byte length.* Where appropriate, each data unit can be referred to by its *name*. By establishing a systematic *mnemonic* naming scheme, the name can be used associatively to identify the meaning, class, or use of the data unit to man. By accepting these names, the interactive operating system can assume at execution time, a simple but important function of the compiler or assembler.

Memory data units are classified and referred to in groups (*sets*), or *collections*, of any manageable number of *elements*. Names of data units or collections are underlined in the text to distinguish them from names of programs; collections are capitalized while data units are usually lower case (x from *X*). Elements of collections are not necessarily physically related in memory. A *block* of data units is a collection whose elements are *physically related*, i.e., contained in a single physical array in memory. A *vector* is a block with a particular sequence, mathematically a column vector. Blocks may be considered vectors which may have null elements in order to apply *operations* (below) to them. Physically, a matrix is a vector of vectors. In schematic drawings data group *M* is shown



.

An *operation macro*, or *operation*, is a basic unit of a computer program, or *procedure*, and is a closed subroutine-algorithm. When man is on-line to the computer, he is concerned with: procedures, which are stand-alone sequences of procedural elements (operations or sub-procedures), and *data* input-output (I/O) of procedures. Operations/procedures are analogous to closed subroutines/computer programs without the latter's ambiguity. For instance, an operation differs from a subroutine in the FORTRAN sense in that it must always be available by name without special provision in the interactive environment of the user, regardless of time or priority.

An operation or procedure may be parameterized by *internal* or *external* data parameters. Internal parameters are associated with named system data previously defined from the console; external parameters are entered from the console at execution or procedure build time as part of the definition of the operation. External parameters may become internal parameters as an operation or procedure is defined. In this manner, relatively inconvenient parameter lists arising from generalization may be "buried" for frequently used specific instances of the operation. This tends to reduce programming redundancy, a primary objective here.

Since consideration is being given in this study to "on-line" environment, procedures and operations may be scheduled, or executed at certain times, on different *levels of priority*. Such scheduled procedures are called *system programs* or real-time programs. Higher levels interrupt (and delay) lower levels; if the same data (same name) are generated on different levels, the user is responsible to see that procedures do not logically conflict. The system must be responsible to see that procedures do not cause system break-down. An operation or procedure which may be used correctly on any priority level is *re-entrant*. Classify the I/O of operations and procedures into real numbers, names (alphanumeric) and n-tuples according to the internal format of control words. Note an integer may be a "1-tuple." *Functions*, or functional procedures, are process-dependent procedures (e.g., a process model) and may be restricted to some priority level. *Data treatments* are process-independent procedures for general data analysis, display, or optimization, and may be similarly restricted. Let *all* operations be re-entrant.

Define mathematical operations:

(a) explicitly, e.g.,

$$\text{OPN}\ (\underline{A}, \underline{B}):\ \underline{A} \leftarrow \underline{B}\ \underline{N}$$

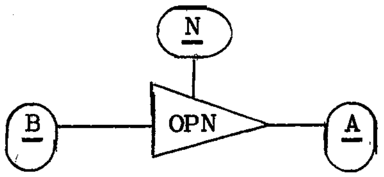
meaning: "$\underline{B}$ postmultiplied by $\underline{N}$ is moved to (replaces) $\underline{A}$" by the operation OPN", or

(b) as operators, e.g.,

$$\text{OPN}\ (\underline{A}, \underline{B}),$$

meaning: "OPN operates with $\underline{A}$ and $\underline{B}$ as external, $\underline{N}$ as internal, parameters."

A schematic symbol for this operation would be



.

The triangle shape of the symbol for operation (or procedure) is selected in preference to other shapes primarily because it allows attaching internal parameters into the sides of the block without confusion of block diagram flow, and implies direction of flow without arrows. Other blocks are used more conventionally.

A vector sample of data taken at points in time from a process status vector $\underline{X}$ in order to relate a subset of valid, controlled, independent variables to a subset of response (dependent) variables in the face of a subset of measured uncontrollable variables is called a <u>data point</u>.

One final term, used later, is introduced here. In the practical environment of industrial process computer data sampling and interpretation, accuracy and validity are of fundamental importance to reliable results, yet plant measurements are often of low quality, especially in absolute accuracy. An error analysis, verified by experience, will demonstrate the dangers of relying upon data as obtained. A basic function of process computers has traditionally been to check limits of observed input process data. Here, the requirement will be added that the system provide feedback to determine or help determine that <u>response</u> variables, functions of many measured variables, are within prescribed error tolerances and if possible what may or may not be correct about the measurements. A process so checked and found to be within tolerances is said to be in a state of process calibration.

Overview

Using the above schematic notation, the over-all data handing functions of the interactive analysis system may be outlined as an informative example of the use of this scheme. Refer to Figure 2. Inputs to the process system in the form of analog (N) and digital (D) signals are converted to engineering units values Y by data acquisition procedures DAQ under control of a system data parameter group YDP. All procedures are scheduled from a program scheduler parameter set P which controls the times of entry into each program. A series of process calculation or model functions f operate on the system current status block X, which may include some manually entered data M, to extend the system data base with Z. Z is a set of calculated outputs from f. X would be identified and controlled by system data parameters SDP. Sampling from X takes place using data sample and data point functions DSF and/or DPF, built to produce a matrix of data points MDP consisting of subsets of X defined by vector definitions V.

From MDP, build matrix functions BMD would be interactively defined to obtain sample matrices S, the input to (interactively) selected interfaces to provide necessary preprocessing and final entry to data treatments (XIA) which produce results observable on the display console. Modifications of this procedure would be reiterated until functions g are obtained from the data which, when executed on current data, predict values similar to functions f (or direct measurements Y)

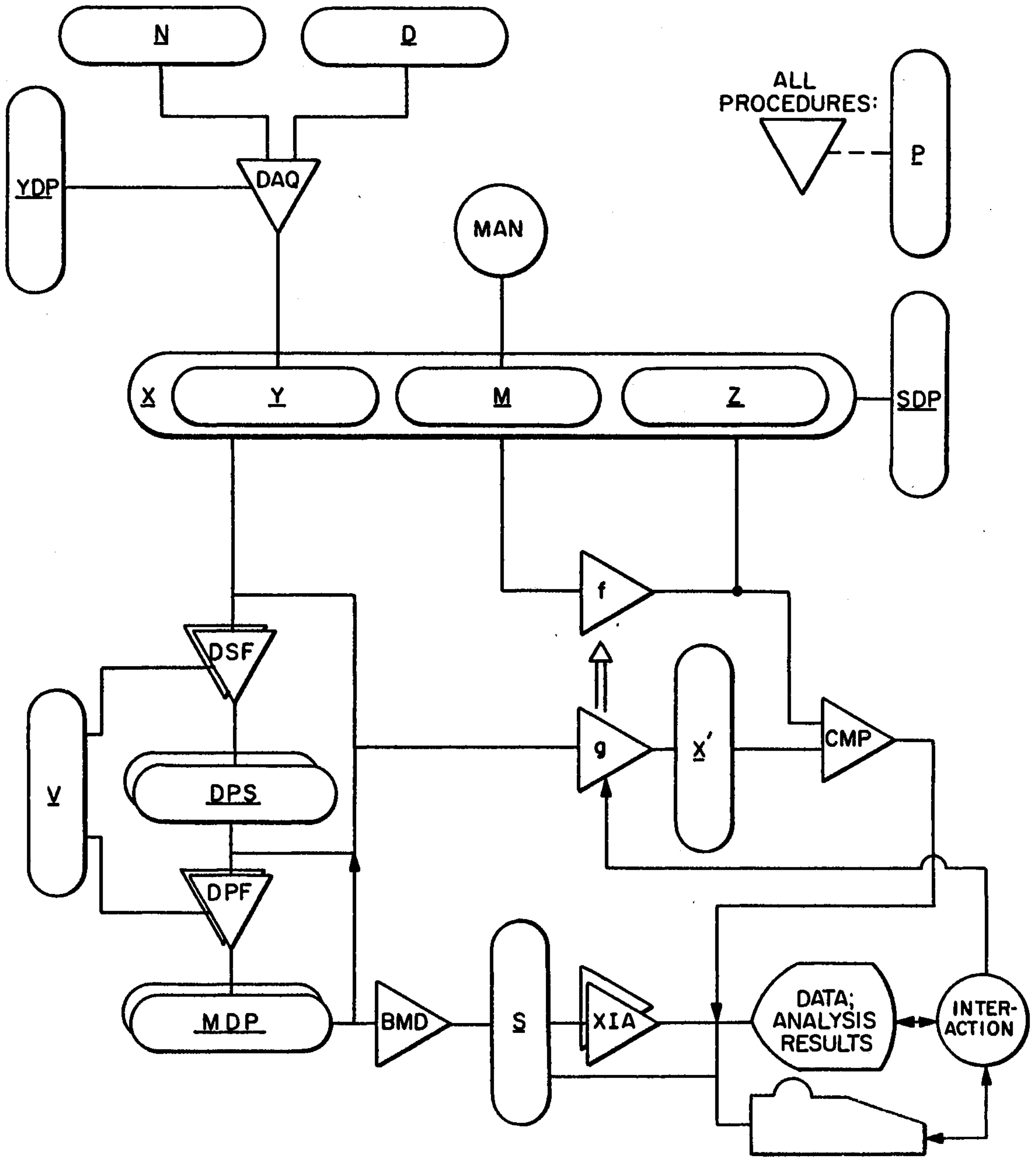


Figure 2. Data System for On-Line Industrial Process Analysis

which may be compared through CMP functions to obtain verification of the analysis results.

The non-computer oriented reader may observe at this point the difficult problem of description in depth of this system. Natural questions are, why all this abstraction, and what is the necessity for and content of all these data parameters? The former question can only be answered by further study of this thesis or similar descriptions of computer logic and software in the literature. Being electronic implementations of mathematical and logical mental processes, computers are abstract by nature and require precise and highly detailed rules by which to operate. Each computer application, thus, leads to its own complex set of abstractions. To describe such a system, precise abstractions are infinitely more suitable than imprecise abstractions, and these are, unfortunately, the only choices available. As for the necessity of the application data parameters, each is ultimately justified (or not justified) by the reward of its availability and the flexibility it offers in eventual use of the system. At this design stage, therefore, each parameter is included on the basis of designer experience, judgment, and reason as to the desired flexibility, simplicity, and expected need. While prepared to justify the parameters included, the writer's view is that the important contribution of the present work is not so much which application parameters are included as it is the fundamental data structure for their specification and organization into similar matrix groups which lead to common data manipulation operations and a compatible, unified, whole. This specification results in both a simpler over-all programming task and a structure which is easier to improve upon in the early design stages.

However, if only to avoid intolerable abstraction, one must list the parameters and their use in this application. Therefore, a data parameter basis for the interactive analysis system is described in Chapter III, immediately following. Unfortunately its structural value may not be fully apparent until study of Chapter VII, the example, is completed. Chapters IV through VI, (primarily Chapter VI), provide the necessary background for understanding the example.

CHAPTER III

DATA BASIS FOR THE OPERATING SYSTEM

Since the computer is on line to the process, the data and language structures begin with the "conventional" process computer functions at the process interface. A formal data base for the operating system is developed in this chapter; this formal organization must meet all basic requirements for the interactive analysis subsystem as well. Some of these conventional process computer functions are also outlined.

Let $\underline{N}$ be a vector containing conventional analog input signals. Then a linear input conversion function would conveniently be written

$$\underline{Y} \leftarrow \underline{A}\,\underline{N} + \underline{B} \quad ,$$

where $\underline{A}$ is a diagonal scaling matrix, $\underline{B}$ is an offset vector, and $\underline{Y}$ is the storage vector for the results in engineering units. This function would normally be scheduled periodically by a scheduler (Chapter IV), such that $\underline{Y}$ always contains the *latest* values of process inputs in engineering units. There are, of course, several other sources of data: pulse frequency and other digital inputs, manual inputs, and inputs which must undergo nonlinear conversion (flows) to engineering units. Moreover, process calculations may generate many additional results. To include capacity for these, let $\underline{X}$ be the *current status block*, which contains the subsets $\underline{Y}$ (converted analog and digital inputs), $\underline{M}$ (manual inputs), and $\underline{Z}$ (other calculated values). $\underline{X}$ then becomes the only source of current (real time) data input to system programs, and represents the basic data characterization of the process. Because of its importance and wide use, $\underline{X}$ should remain in permanent core (primary) storage and is the most extensively "parameterized," or coupled, data in the system. All functional operations, procedures, and data treatments will operate from or to $\underline{X}$, or recorded instances of subsets of $\underline{X}$.

The simple organization of all current data into one permanent block has several important advantages over alternate organizations:

1. Because it is in fixed primary memory it insures fast and direct availability to all functions of current data, regardless of time or priority. System resources are not required for non-functional fetches of needed data.
2. Within $\underline{X}$, elements have a fixed position, allowing simple one-for-one coupling of system parameters for identification, scaling, limiting, lagging, etc., to $\underline{X}$.
3. It enforces the scheduling of data sampling functions on an efficient, coordinated basis, and consideration of the effects of different instrument cycles or process lags on system functions f.
4. It is directly translatable into machine resource (memory) requirements.

If $\underline{X}$ contains a subset $\underline{Z}$ of significant size, it may not be desirable to execute a process-calculations function(s)

$$\underline{Z} \leftarrow f\ (\underline{X})$$

on as high a frequency as $\underline{X}$ is generated. This may relax the necessity for having $\underline{Z}$ in permanent core. Therefore, a subset $\underline{Z}$ of $\underline{X}$ may reside on a bulk or secondary, memory (disc, drum, tape, etc.) and be generated only when f is called. In this case nonfunctional system resources (bulk transfer) must be used to supply $\underline{Z}$ to all functional procedures requiring elements from $\underline{Z}$.

Consider the various <u>system data parameters</u> which may be needed for each element of $\underline{X}$. For interactive references and identification of data, a data name is required. Lower and upper limits, lag parameters, scaling constants, alarm instructions, and output formats, are other parameters which may be needed in various programs for each element of $\underline{X}$. None of the current process operating systems provide a direct capability for mnemonically referring to process data at execution or interactive procedure-build time; and, therefore, compilation or assembly is required. Interpretative translation of names is a key factor in adapting the process system to interactive analysis. Significant memory may be required to store so many parameters one-for-one with $\underline{X}$. The following paragraph will illustrate how the notation can be used to evaluate data storage requirements for such parameters.

Consider a typical linear operation

$$\text{CLS:}\quad \underline{P} \leftarrow \underline{\underline{C}}\,\underline{X} + \underline{D}\ ,$$

where $\underline{C}$ is a diagonal parameter scaling matrix and $\underline{D}$ an offset parameter vector to project $\underline{X}$ onto $\underline{P}$. Depending upon the nature of $\underline{X}$ and CLS, there may be enough redundancy in $\underline{C}$ and $\underline{D}$ to evaluate an alternative approach. For example, packed indirect references i_c and $\underline{i_d}$ (requiring fewer bits than $\underline{C}$ and $\underline{D}$) might be used for access to smaller $\underline{C}$ and $\underline{D}$ blocks. The CLS operation would then be

$$\text{CLS:}\quad \underline{P} \leftarrow \underline{\underline{C}}(\underline{i_c})\,\underline{X} + \underline{D}(\underline{i_d})$$

where the diagonal scaling matrix $\underline{C}(i_c)$ and $\underline{D}(i_d)$ denote vectors generated by the relative indirect addresses i_c and i_d. In some cases auxiliary memory for $\underline{C}$ and $\underline{D}$ might be eliminated altogether by making $\underline{C}$ and $\underline{D}$ direct functions of i_c and i_d (where $f(i_c, i_d)$ does not require real parameters unique to each c and d), or the packed integers i_c and i_d might be used directly for some operation. The reader may imagine other alternatives. The point is that generally real number storage versus n-tuple packed storage and some speed can be traded for memory conservation where the notation relates directly to hardware structure. This would be impractical for a process system using a compiler language such as FORTRAN, because it is not possible to efficiently handle packed data and indirect addressing in such languages.

After having introduced these parameters into the system definition, a fundamental software design consideration is the location of these system data parameters in the core/bulk memory system. As pointed out above, these parameters involve considerable memory for large $\underline{X}$, indicating they should reside on bulk memory and be called when needed. On the other hand, parameters from this group are essential or highly desirable to several system and functional programs, e.g., man-machine interaction (names, formats), process model and control programs (limits, lags, etc.), alarm programs (direction of action on alarm), and data handling routines (names, formats, scaling constants). Furthermore, as procedures are added, the inefficiencies multiply in the storage of system programs without a common organization. The tradeoff, therefore, is in the relative size and speed of the several functional programs for the two basic storage alternatives (in core, or on bulk with and without internal storage of parameters) and the various priority structures. An efficient compromise would seem to be to organize the functional programs around desired subgroups of the needed data parameters and, using system efficiency as a criterion, link these subgroups to their parent programs as internal parameters. That is, where parameters are needed for a given routine, queue the singly stored subgroups whenever the routine is queued.

The above discussion leads into the specification of a standard data and parameter base for the process operating system. This operating system, in contrast to most, if not all, process computer operating systems provided by manufacturers, revolves not about the non-functional hardware options, compilers, assemblers, and peripherals, but about the functional process data. A simplified, yet more functional, system results.

Let the sex-tuple

$$\text{sdp:}\quad \{s, \ell, r, i(xv), b_0, b_1\} \tag{1}$$

represent the following system data parameters needed for appropriate disposition of $\underline{X}$:

s = the name of x

ℓ, r = the number of characters to left (ℓ) and right (r) of decimal when formating real numbers for manual <u>input</u> or output. Let r also contain codes for defining variables in an integer or alphanumeric format.

$i(xv)$ = addresses of the triplet $\{g, h, n_c\}$,

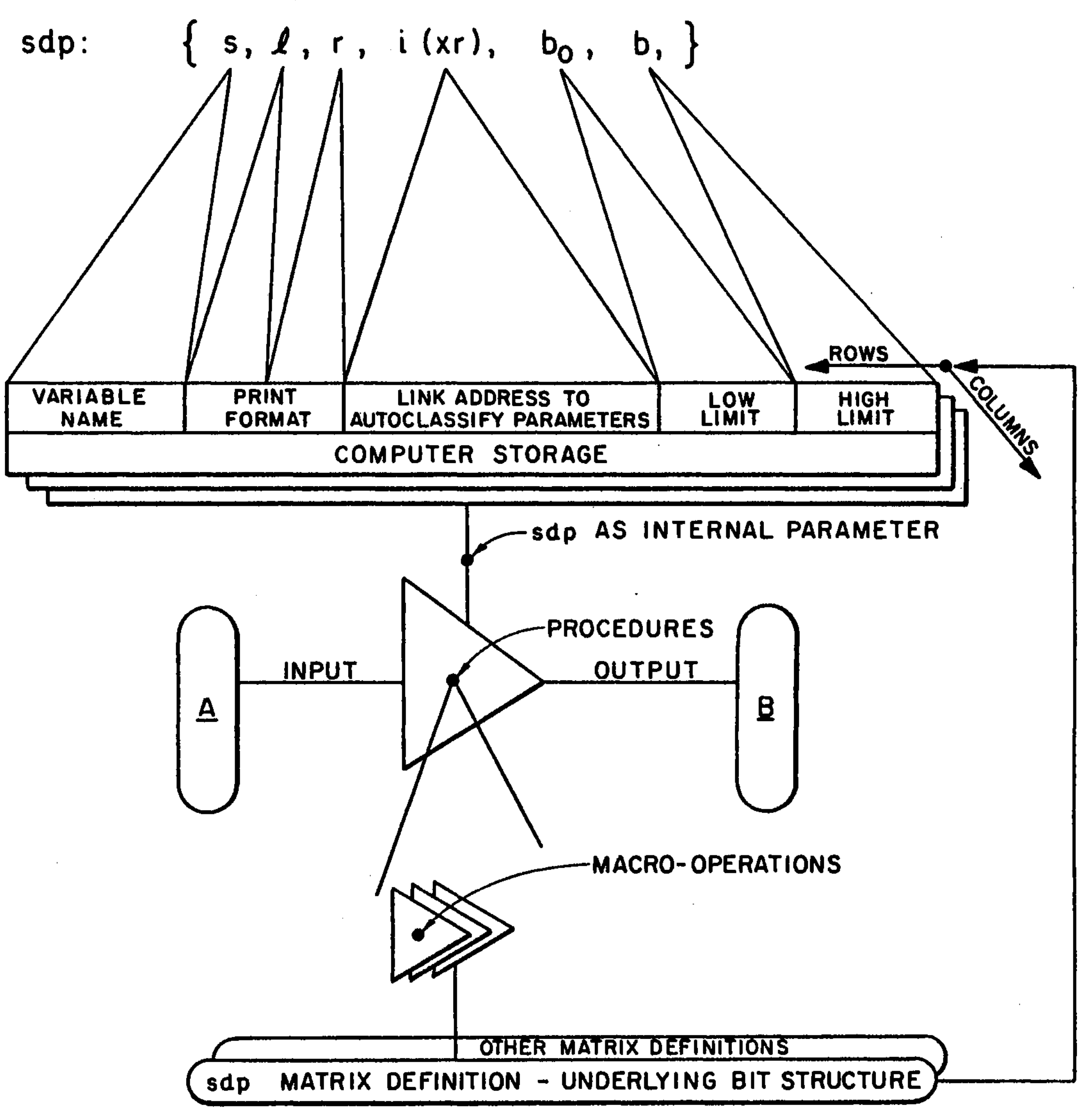
where

g, h = lower and upper boundary, respectively, for projection of x into <u>class</u> intervals and n_c is the number of classification intervals between g and h for x. See Appendix A.

b_0, b_1 = lower (b_0) and upper (b_1) operating limit for x; used on $\underline{X}$ to allow use of constraints where f may not be applicable outside certain ranges.

The set theory notation used in (1) above shall be used throughout this thesis to describe data bases having a common physical structure. How this structure relates to operations and procedures is shown in Figure 3. The names (e.g., s, ℓ, etc.) in a data base definition refer to data units of any length making up the row definition of the matrix containing the actual parameters. Some parameters may be link addresses to other system matrices with a similar structure. Link addresses are used primarily to keep the matrices down to core-manageable sizes by separating logically-related and perhaps less frequently used subsets of parameters. Any system matrix may be either an input ($\underline{A}$), an internal parameter (as shown), or output ($\underline{B}$) to interactively defined procedures, which use closed, re-entrant, macro-operations. It is in these primitive macro-operations that the underlying bit structure of the packed matrix is found by reference to the appropriate (sdp) matrix definition. The matrix defining system matrices is itself a system matrix.

In this manner, only the name of large groups of parameters need be used at the man-machine interface, while the <u>same</u> <u>set</u> of primitive, most efficient, macro-operations may be used for manipulating either data or system parameter matrices. Moreover, maximum storage efficiency is assured by allowing packed data structures. At the same time the user building procedures is concerned with data and parameters only



sdp: { s, ℓ, r, i (xr), b0, b, }
VARIABLE NAME
PRINT FORMAT
LINK ADDRESS TO AUTOCLASSIFY PARAMETERS
LOW LIMIT
HIGH LIMIT
ROWS
COLUMNS
COMPUTER STORAGE
sdp AS INTERNAL PARAMETER
PROCEDURES
A
INPUT
OUTPUT
B
MACRO-OPERATIONS
OTHER MATRIX DEFINITIONS
sdp MATRIX DEFINITION - UNDERLYING BIT STRUCTURE

Figure 3. Relation of Data Basis Notation to System Storage and Use

at the highest functional level.

For $\underline{Y}$, the process signal input subset of $\underline{X}$, additional input and scaling parameters would be defined, e.g., as the 12-tuple

$$\text{ydp:}\quad \{s,\ n,\ i(\pi),\ i(a_0),\ i(a_1),\ b_2,\ b_3,\ i(k),\ z_a,\ w,\ d_1,\ d_2\}, \qquad (2)$$

where

s = the symbolic name of y in $\underline{X}$ (and in the sub block $\underline{Y}$ of $\underline{X}$) for the result of the process input sampling and scaling,

n = the value of the input signal before scaling,

$i(\pi)$ = the relative (to π) indirect address of suboperations for frequently encountered linear, quadratic, flow, etc., scaling of process input signals,

$i(a_0)$, $i(a_1)$ = the relative indirect addresses of offset (a_0) and scaling (a_1) constants for linear input conversion.

For $i(\pi)$ not a linear conversion suboperation, a_0 and a_1 may be unique constants or references defined by the appropriate suboperation.

b_2, b_3 = low and high signal (instruments) limits, respectively,

$i(k)$ = indirect address to lag constant k for RC filtering of noise components of x ($i(k) = 0$; none),

z_a = a code for disposition of alarm events (e.g., ignore, normal typeout, out and in limits, with or without $\underline{X}$ storage of the alarmed value, set buzzer or bell, etc.),

w = input source of n.

Actually w may be an n-tuple defining various hardware parameters necessary for obtaining n: e.g., analog multiplexer address, gain selection, pulse accumulation, etc. Since w is unique for a particular hardware design and input signal, it will be sufficient for the purposes of this writer to define w as an address of the raw signal. $d_1 (d_2) =$ low (high) limit alarm switches.

With $\underline{X}$ and the associated $\underline{SDP}$ defined as the data base of the process system, all process data groups handled by the system may be considered to be derivatives of $\underline{X}$. In this manner, the writer obtains a formalization which has significance to efficiency in both set-up and operation of the system: the data matrix (and vector) definitions. A physical data matrix or vector is identified by the matrix-name sep-tuple

$$sv_v : \{\underline{V}, \underline{m}_v, \underline{n}_v, c_v, d_v, e_v, \theta_v, \xi_v, \Phi_v\}, \; v = 1, 2, \ldots, \text{no. of vectors} \quad (3)$$

where $\underline{V}$ is the vector name (v is the index, or position, of $\underline{V}$ in this system vector definition matrix), m_v and n_v are the row and column dimensions, respectively, c_v is a link address to a core or bulk memory area where the <u>matrix</u> (or vector) <u>row definition</u> (below) is stored, d_v is a link address to the area where the matrix column definition (below) is stored, e_v is a link address to the first data unit, θ_v is the system assigned currently active column number for updating matrices, ξ_v is a retrieval code calculated from the vector definition, and Φ_v refers to the general data type: system data (floating point, integer, or alpha according to $\{\ell, r\} \subseteq \underline{SDP}$), or system data parameters (e.g., $\underline{SDP}$). If system data parameters, the binary packing parameters are stored in the c_v area. A data matrix or parameter definition (c_v) contains a vector of $\underline{x}$-indices defining the rows of the matrix or vector in terms of <u>variables</u> (or packing parameters). A column definition is an arbitrary list of sequence (observation) numbers assigned by the operating system and used to identify and retrieve data in the interactive environment. A vector definition ($n_v = 1$) contains no column definition ($d_v = \emptyset$). With this form of definition each physical data collection may be uniquely identified, while links from different data matrices to their <u>common</u> vector definitions preclude redundant storage. ξ_v serves as an abbreviation of the variable definition for efficient matrix updating. This is a special combination of linear and linked-allocation list structuring, the generalization for which would be too inefficient for practical use here. See Chapter II of (11) for a sound general discussion of list processing approaches, and particularly the Introduction to that Chapter, for when to intelligently use them.

CHAPTER IV

PRIORITY STRUCTURE AND SCHEDULING

Introduction

The priority structure of most process operating systems, while very "flexible," is vague, complex, and filled with exceptions and duplications (8) (9) (10). For example, complexity is added to one system to allow full multiprogramming and relocatability of all programs from discs. Yet, frequently, exceptions occur when one must run a program using unavailable or fixed core features, or with interrupts inhibited to avoid problems introduced by the complexity of the multiprogramming design. In another case, the user has to remember what kind (interrupt core load, main line, core resident, etc.) of interrupt program he is in in order to properly set up and exit his program. A common error is to describe the "batch" (variously called the "non-process," "background," or "free time") mode as a special feature worthy of extensive influence in the design of the systems. As a consequence, the complexity of system set-up and execution (but not necessarily operation) is increased significantly and, in this view, artificially. The driving forces behind this common approach seem to be: (1) the desire to put relatively large, resource-consuming compilers (FORTRAN) and assemblers, with various degrees of debug capability, on-line and (2) the influence of the data processing "job-shop" computing room carrying over to the large process systems. While this writer does not immediately propose complete elimination of these forces, the guiding principles here are more to the point: (1) devoting system resources primarily to the process data and operations on the data, (2) formalizing primitive and procedural data handling functions such that system set up and execution will require a minimum of system resources, and (3) providing a man-machine interactive mode through which most (eventually all) functional procedures leading to optimization of the process may be set up and executed without batch compilation and debug, i.e., in an interpretive mode, using fast alphanumeric and graphic capability.

While much attention has been paid to the process computer's need to respond quickly to external events presumably signaled through switches around the process tied to system hardware interrupts, there rarely has been, in the writer's experience with systems on several petrochemical processes, a clearly defined need to signal a system program directly through a process interrupt (alarms are usually connected directly to enunciators, and the obvious system interrupt needs for operating peripheral equipment, clock signals, etc., are excepted). Most functional procedures on-line can be handled by execution either on a demand or cyclic basis. One reason for this is the state of the art: "fast" response with respect to the process is often "slow" response with respect to the computer speed. It is remarkable that operating systems do not provide for efficient, direct set up of cyclic scheduling while going to great lengths to allow process-interrupt or programmed scheduling of functional programs.

Formal Priority Structures

Formal priority structures may be <u>vertical</u> (GE RTMOS), <u>horizontal</u> (IBM TSX), or have a combination, or <u>lattice</u> structure. Vertical structure refers to priority and implies that each functional program operates at a different (higher or lower) priority level ℓ. Note that high priority corresponds to a numerically low ℓ and vice-versa. Horizontal structure puts all "main-line" programs on essentially one level, with a minimum of vertical levels interrupting for scheduling, control, input-output, etc. Usually all interrupts are inhibited while a given interrupt is serviced. In each case peripheral equipment is driven from within the operating system without on-line control by the user, and nearly always the user can achieve vertical structure from horizontal, and vice-versa, by specification and/or programming. The lattice structure allows both vertical and horizontal structuring directly from the operating system.

Vertical structuring involves relatively simple priority decisions but requires extensive overhead in re-entrancy techniques and unnecessary shifting from one program to another. "Executives" often end up being written by the user to provide some horizontal structuring. Horizontal structuring simplifies priority decisions still further but imposes unnecessary complexity or restrictions in scheduling of higher level programs, and problems with response time may arise as the system becomes loaded. This results in system resources being spent on various ways for scheduling core-resident programs and achieving vertical structuring for bulk-resident programs. The lattice structure offers a single solution for all alternatives, but introduces some complexity into priority decisions. The advantages of the lattice in system and programming resources saved will tend to grow as system functions are changed or added. Therefore, the lattice structure is selected. A simplification, or at least, early burial, of the priority decision problem, will be necessary.

Data Basis for Lattice Structure

Let the control word parameter matrix $\underline{PQ}$ for the lattice program scheduler be defined by the primary pentuple

$$p: \quad \{n_p, \Delta, \lambda_p, tx, x_p\} \quad p = 1, 2, \ldots \tag{4}$$

coupled to the pentuple

$$q: \quad \{\ell, i(b), \tau, sc, \Psi\} \quad , \tag{5}$$

where p contains scheduling parameters and q is the queued program parameters which are stored in the priority bulk transfer queue; n_p = the program name, Δ = the time interval between iterative (cyclic) executions of n_p (or a code for "no cyclic execution"), λ_p = reference to a hardware interrupt to activate this program or a digital switch to allow or inhibit cyclic execution from one console, tx = scheduler-updated time for next execution of n_p, and x_p = link to next program execution for chaining programs. In q, ℓ = the priority level of n_p, i(b) = indirect reference to the bulk storage address and length (or coreaddress if permanent core), τ = current relative entry location for n_p, sc = current starting core location for n_p, and Ψ = system-updated state code for current status of n_p:

State Ψ_0 = program locked out and not executable on schedule;

Ψ_1 = n_p has been scheduled or demanded, and queued, but is not in core;

Ψ_2 = n_p is queued and in core at sc (entry at τ);

Ψ_3 = n_p has been entered but not completed;

Ψ_4 = n_p has entered but not completed execution and has been transferred to disc for higher priority work;

Ψ_5 = n_p has been completed and is presently inactive.

As expected, the lattice structure requires some system attention to relative priority decisions in queueing bulk transfers. The data for these decisions may be provided by a currently active program list

$$\underline{\text{XPL}}: \quad \{p_o(\ell)\} \ , \quad \underline{\ell} = 1, 2, \ldots, \text{no. of levels} \ , \qquad (6)$$

and a currently active level cell $\underline{\ell_o}$. These data point to the currently active system program $\{\underline{p_o}, \underline{\ell_o}\}$ for use by the operating system. Now, every transition in level takes place through a hardware interrupt response (the hardware interrupt may be program generated) which consists of a conventional save registers routine (all interrupts inhibited; masking of lower level interrupts) during which

$$\tau(p_o, \ell_o) \leftarrow \tau(\lambda) - sc(\ell_o, p_o) \ , \qquad (7)$$

takes place, where $\tau(\lambda)$ is the location of the instruction to be executed on ℓ_o when interruption to the higher level occurred. Equation (7) insures the retention of the relative point of interruption of each program on each level so that $\{p_o, \ell_o\}$ may be transferred to disc in case the interruption queues a program with $\ell < \ell_o$.

Following execution of (7) the level transition normally takes place:

$$\ell_o \leftarrow \ell(\lambda) \ . \qquad (8)$$

The Scheduler

The scheduler, operating at a high priority from a clock interrupt λ_s:

1. Updates the system clock t_n (and date).
2. Scans $\underline{PQ}$ and executes Steps 3. - 5. for all $tx_p \leq t_n$ and $\Psi_5 = 1$ (1 means state is true, 0 means state is false); otherwise exits.
3. $tx_p \leftarrow tx_p + \Delta$, $\Delta \neq 0$ (reset for next cyclic execution).
4. Changes state $\Psi_{(1 \text{ or } 2)_p} \leftarrow \Psi_{5_p}$ and $\lambda_{\ell_p} \leftarrow \Psi_{2_p}$ where λ_{ℓ_p} is a single hardware priority interrupt for each priority level. This interrupt causes entry (following a typical system register save) into the level executive (below).
5. If n_p is bulk resident, a bulk priority queue $\underline{BQ}$ is linked to q_{n_p} ($\Psi_1 = 1$) and λ_b, the queue for the bulk transfer control routine, is set.

Also in the scheduler package are routines for user calls, such as "turn on p," "schedule p at ---," "cancel p," etc., which update the appropriate elements of $\underline{PQ}$.

Working Core Organization

Available process operating systems organize working core storage all the way from single programs sequentially loaded and executed into one, fixed address, binary "core load" area (9), to completely relocatable programs simultaneously loaded into various groups of fairly small memory blocks which are dynamically mapped to provide a flexible multiprogramming capability (8). The former seems to limit rather severely the number of effective priority levels (horizontal structure), especially for the interactive system, and the latter seems too complex for effective implementation where system resources must be conserved. Additionally, observe that the throughput advantages of multiprogramming are concentrated in software - overlapped I/Ø, including bulk transfer operations, so that other (even lower priority) programs may proceed when one program is waiting for completion of an I/Ø operation. A compromise approach is suggested which allows significant I/Ø overlap and limits the dynamic mapping problem. This will facilitate efficient integration of the on-line and the interactive subsystems.

Divide working core $\underline{wc}$ into $\underline{m}$ unit areas chosen for their favorable comparison in size to most functional programs. That is, most programs will fit into one area but some programs may require two or more (up to all) areas of working core. This is a "rough cut" between using all $\underline{wc}$ for one program, and the fine division of $\underline{wc}$ into areas too small for most programs. This division is defined by the triplet

$$wc: \quad \{\underline{ad}, \underline{ln}, p\}, \quad i = 1, 2, \ldots, m, \tag{9}$$

where $\underline{ad}$ and $\underline{ln}$ are the starting address and length, respectively, of each core area, and p is the number of the program currently occupying each area. By dividing working core into only two or three areas, I/Ø overlap (loading of one area while another is in execution) can be achieved, which is sufficient for attaining substantial advantages of multiprogramming without many of its complexities.

Bulk Transfer Control

The bulk transfer control routine may be a part of or queued by the scheduler, or may operate independently on its own cyclic schedule. It also contains response routines for "transfer complete" interrupts. This routine, operating at high priority (no level transition):

1. Checks busy status of the bulk transfer control channel(s), and, when a transfer is complete, sets:

   $\Psi_{2p} \leftarrow \Psi_{1p}$ (new program p into core) .

2. Whenever a bulk transfer channel is not busy and <u>BQ</u> (format of pentuple q, Equation (4)) is not empty, this routine:

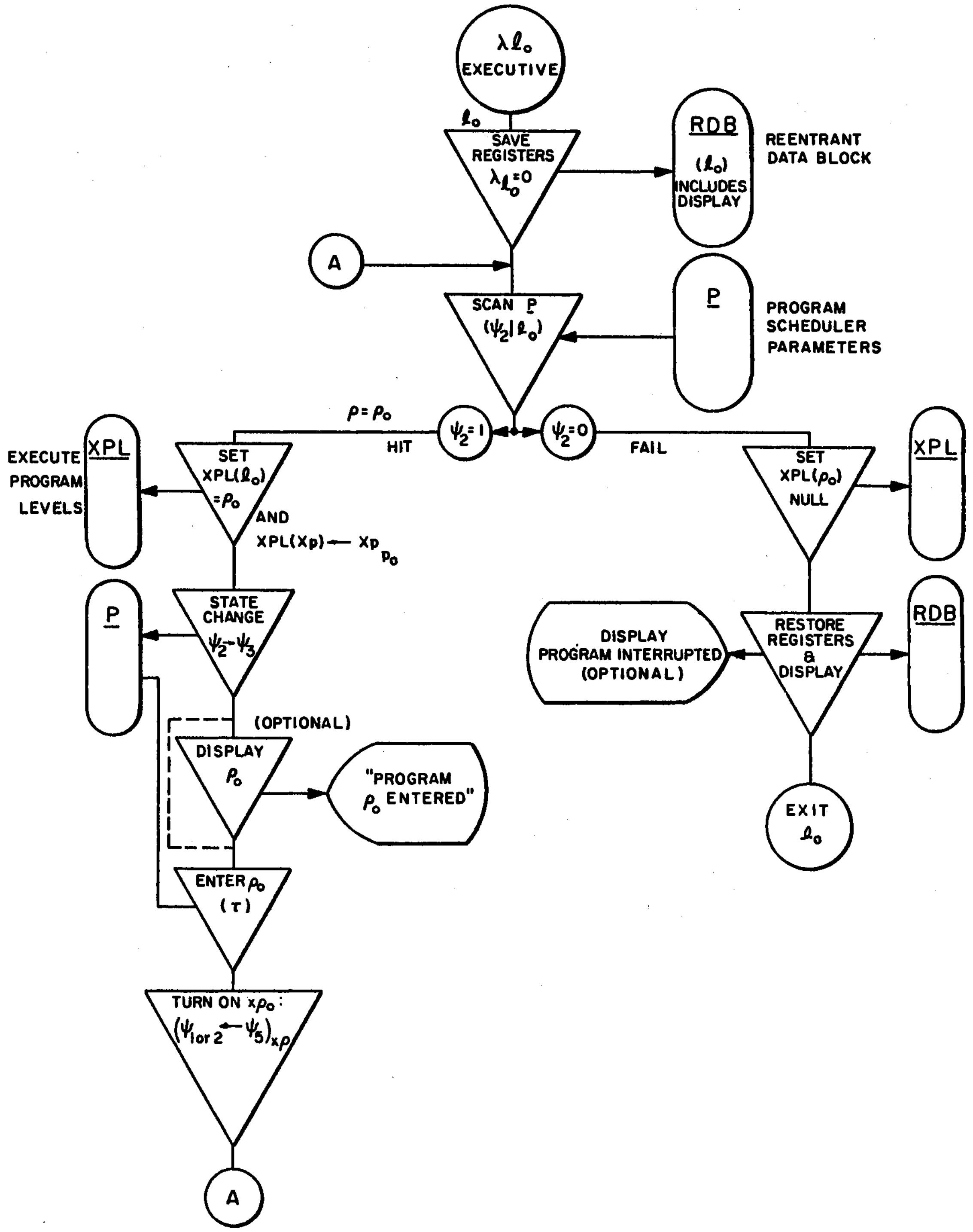
   a. Scans <u>BQ</u> for $(\ell \subseteq \underline{BQ}) < \ell_0$ from highest to $(\ell_0 - 1)$ priority. If found, initiates transfer (deletes q from <u>BQ</u>) of the associated p to core. If all core is busy, initiates transfer of any p' in core with $\ell_{p'} > (\ell \subseteq \underline{BQ})$ to make room for p_ℓ, $\ell < \ell_0$. At the same time that transfer of p' to bulk is initiated, changes state of p' $(\Psi_{4_{p'}} \leftarrow \Psi_{3_{p'}})$, updates XPL, and requeues p' into <u>BQ</u> $(\Psi_{4_{p'}} = 1)$ so that p' may be returned to core $(\Psi_{3_{p'}} \leftarrow \Psi_{4_{p'}})$ when priorities allow.

   b. Scans remainder of $(\ell \subseteq \underline{BQ})$ from ℓ_0 to lowest priority. When found, check core for a working area <u>wc</u> with $\ell(\underline{p}) = \emptyset$ (null). If found, initiate transfer to core. Otherwise, exits.

Level Executives

The level executive routine, diagrammed in Figure 4, is a ℓ-parameterized call to a single, common, re-entrant scan executive which simply finds each $\Psi_{2_p} = 1$ and enters $(\Psi_3 \leftarrow \Psi_2)_p$ program p. Upon completion of each p_0 execution, the chained program xp (if any) is queued. Control then returns to the ℓp scan loop so that all programs queued and in core on the same level at the same time are executed in turn. When all programs on a level are completed, the executive exits through a level transition scan $\ell_0 \leftarrow \ell + 1$, $\ell + 2$, ... until a level $\ell + i$ is reached for which $\Psi_2(p(\ell + i)) = 1$. The $\underline{Q}$ parameters sc and τ are then used to (re) enter $p(\ell + i)$.

The above scheme is believed to be unique; it offers a highly efficient means for the priority structure of a complete real time system to be organized and executed by formal specification of data parameters. The three most frequently encountered methods for queueing real time process programs are built directly into the operating system with no calls necessary in the programs themselves. These methods are cyclic execution, chaining, and interrupt-queueing. Modification is simple, I/Ø overlap can be used where needed without multiprogramming complexity, the display allows visual on-line monitoring for debug and operational "feel," and any desired combination of horizontal or vertical structuring may be specified, allowing maximum effectiveness of a given process application. With the "rough cut" of working core, it is suitable for interactive analysis during on-line execution.



$\lambda \ell_o$ EXECUTIVE
ℓ_o
SAVE REGISTERS $\lambda_{\ell_o} = 0$
RDB
(ℓ_o) INCLUDES DISPLAY
REENTRANT DATA BLOCK
A
SCAN $\underline{P}$ $(\psi_2 | \ell_o)$
$\underline{P}$
PROGRAM SCHEDULER PARAMETERS
$\rho = \rho_o$
$\psi_2 = 1$
HIT
$\psi_2 = 0$
FAIL
EXECUTE PROGRAM LEVELS
XPL
SET XPL(ℓ_o) $= \rho_o$ AND XPL(Xp) ← Xp_{ρ_o}
SET XPL(ρ_o) NULL
XPL
STATE CHANGE $\psi_2 \rightarrow \psi_3$
$\underline{P}$
(OPTIONAL)
DISPLAY ρ_o
"PROGRAM ρ_o ENTERED"
ENTER ρ_o (τ)
TURN ON $x\rho_o$: $(\psi_{1 or 2} \leftarrow \psi_5)_{x\rho}$
A
DISPLAY PROGRAM INTERRUPTED (OPTIONAL)
RESTORE REGISTERS & DISPLAY
RDB
EXIT ℓ_o

Figure 4. Re-Entrant Level Executive

CHAPTER V

ORGANIZATION OF OTHER PROCESS FUNCTIONS

With the data and scheduling basis outlined, the next step is to organize conventional system functions into formal program sections. This will provide some insight on good ways to define the interactive primitive operations.

Data Acquisition

Process input, checking, and conversion to engineering units may be categorized into one section, or procedure, data acquisition. The data basis for data acquisition was given in Chapter III, Equation (2):

$$\underline{ydp}\colon\ \{s,\ n,\ i(\pi),\ i(a_0),\ i(a_1),\ b_2,\ b_3,\ i(k),\ ƶ_a,\ w,\ d_1,\ d_2\}. \qquad (2)$$

The computational elements of this program are:

1. Given a <u>ydp</u>, set $n \leftarrow f(w)$, i.e., sample the signal (depending upon hardware, DAQ may exit at this point with recall through an analog signal ready interrupt $\underline{\lambda_a}$ when n ready).
2. If $b_2 \leq n \leq b_3$, go to Step 3, otherwise to Step 7.
3. Set $\underline{PAST} = y_{ydp}$.
4. Given n, $i(\pi)$, $i(a_0)$ and $i(a_1)$, execute π:
   $y = f(n, a_0, a_1)$ [conversion to engineering units].
5. Given i(k) and y, lag y according to
   $y = ky + (1 - k)\ (\underline{PAST})$.

6. Continue (usually loop to scan all inputs).

7. Set $d_1 = 1$ (alarm code) if $n < b_2$ or $d_2 = 1$ if $n > b_3$, and turn on input-output alarm program if the limit violation has occurred for the first time this scan. Go to Step 6.

Process Calculations, Models

There may be any number of process-oriented procedures for operation on the process inputs to produce response variables. Such procedures assume the implicit forms

$$\underline{X} \leftarrow f(\underline{X}, t) \text{ or} \qquad (10)$$

$$\underline{E} \leftarrow g(\underline{E}, t) \qquad (11)$$

where $\underline{E}$ may be any previously defined system data vector or matrix, such as $\underline{X}$ (including $\underline{Y}$ and $\underline{M}$), for on-line computations, or some data point matrix $\underline{MDP}$, of historic subset samples from $\underline{X}$. The parameter t implies f and g may be functions of time. $\underline{E}$ may be resident on bulk storage. If $\underline{MDP}$ is resident on an input device (paper tape, cards), a system data input function (see Input-Output) must first be executed. Note that f is limited to operations from $\underline{X}$ to $\underline{X}$; i.e., to real time, on-line process calculations, while g may be a function of any system vector. f is the class of real time process functions written by the user; g may be user written, or generated by interactive analysis, below.

Data Handling

The total process status block $\underline{X}$ contains independent, dependent (or response), and control variables representing the present state of the process and the quality of process calibration. Most data sampling will take place directly from $\underline{X}$, but a significant amount of memory of past process conditions will be required for interactive learning about the process to take place. The primary tool for sampling and storing history is the data point function DPF, in conjunction with a <u>system vector definition</u>. A data point was defined in Chapter II. The function would take the form

$$\underline{DPj}\ (\underline{V}) \leftarrow \text{DPF}\ (\underline{V}) \tag{12}$$

$$j = 1,\ 2,\ \ldots,\ n_v,\ 1,\ 2,\ \ldots,$$

where $\underline{DPj}$ is a column of $\underline{MDP}$, n_v is the maximum number of columns of $\underline{MDP}$, and $\underline{V}$ is the name of a system vector. A system vector was defined by Equation (3). The purpose of a data point is to <u>select</u> a reliable and relevant subset from $\underline{X}$ for later use. Of course, $\underline{V}$ may specify all of $\underline{X}$. However, one important function of the analysis system is to help reduce data insofar as possible. Moreover, the system may be connected to more than one process unit and only one is of interest at a particular time. Thus, several different such subsets may be desired; therefore, several data point vector definitions may be stored.

In order to increase the reliability of the data point selection, an average of several samples of $\underline{X}$ may be desired (when the data point is steady-state). Here the data point would be

1. $\underline{DPS}\ (j,\ \underline{V}) \leftarrow \text{DSF}\ (\underline{V})$ (cyclic) $j = 1,\ 2,\ \ldots,\ n,\ 1,\ 2,\ \ldots$ (13)
2. $\underline{DP}\ (j,\ \underline{V}) \leftarrow \text{DPF}\ (\underline{DPS})$ (14)

where the Data Sample Function DSF has $\underline{X}$ and $\underline{SDP}$ (s, l, r, g, h) as internal parameters and is simply a sampling of $\underline{X}$ with time into a circular by-column matrix. DPF is executed upon external demand to call for retention of the row-by-row averages of <u>DPS</u> in the current column of <u>DP</u>. Other possible modifications of the data point function might be implemented in more specific applications. For example, dynamic data sampling might transfer only <u>changes</u> of x-values with time (a common dynamic data reduction technique).

An important function of the data point is to provide a flexible interface for input of process data to various <u>data treatments</u>, i.e., process-independent display, analysis, or optimization techniques which may be applied through the interface to provide data for display at the man-machine console for the learning process, decisions, modification of data or analysis, and conversion of the results to permanent form usable by both computer and man.

Several of these data treatments may require significant machine resources. For example, steady-state treatments may include linear (or non-linear) regression and correlation, analysis of variance, parametric plotting, factor analysis, etc. Dynamic data techniques might include Laplace, Fourier, or other frequency response analysis, or adaptive controllor tuning techniques. It is very important, therefore, to arrive at this interface with significant machine resources remaining if such techniques are to be accomplished with the on-line system. The accomplishment of data treatment functions with the on-line computer will require very efficient man-machine interaction, since the effectiveness of the on-line system will be measured by the degree of useful feedback to the process accomplished through the data system. Figure 2 outlined this "outer loop" the writer is trying to "close." Both man and computer must work together in this outer loop. The writer knows of no instance where this interactive analysis loop has been effectively closed in a practical environment using a process computer.

Input-Output Control (IØC)

Conventional input-output has always required a substantial portion of system resources and, with this requirement for interactive IØC as well, there is substantial motivation for system economies here. Process input is handled by the DAQ procedures (above); it is to be expected that process output would be treated in a similar fashion; however, due to its complexity, the output function is outside the scope of the present work. Input-output from bulk storage is considered under the operating system techniques. Therefore, the main concern here is with system input-output from the primary man-machine devices, including the terminal. Such devices already have fairly standard (ASC II - code) interfaces for acceptance of alphanumeric data; moreover, magnetic tape units or cassettes, line printers, typewriters, paper tape punches, and (with some logic) cards, can accept similar alphanumeric records, the differences being in the hardware and timing. For alphanumeric output, then, one can gain considerable simplicity by demanding that all alphanumeric output devices be code-compatible such that a record may be sent from memory to any of these devices by simply addressing the desired device(s). It is not too much for a system to be compatible with itself. Where necessary, page or file control may be built into the actual hardware driver routine, and any codes not possible on one device may be formally interpreted in the software drivers.

The most likely "hard copy" device(s) on this relatively small process system are (interactive) typewriters. A good tutorial discussion of human factors and functional specifications for such devices used in a similar (time-sharing) application is given in (15).

Input-output devices are generally single-level devices; i.e., interruption of an incomplete operation to a particular device to execute another operation to the same device is rarely necessary or desirable. The single exception might be if the same typewriter is used for alarms as well as routine output. This, of course, causes alarms to be intermixed with logs. In this system, the relatively fast display with appropriate buzzers or blinking functions may be preferred for alarms. Therefore, drivers for other input-output devices may be restricted to one level.

Input-output coding, formating and checkout is a major part of any complete programming job, even (particularly) when compilers such as FØRTRAN are used. From this formalization of data structures, a standardization of typing functions and formats is suggested.

Routine process tracking logs may be defined by the simplest possible specification: a list (vector definition of the names (internally, indices) of variables desired in each output vector. Thus, the tracking log function TLF (for subsets of $\underline{X}$) is

$$\text{LØG1} \leftarrow \text{TLF}\ (\underline{V}) \tag{15}$$

where LØG1 addresses the output device. $\underline{L}$, $\underline{R}$, and $\underline{S}$ (from $\underline{\text{SDP}}$) are internal parameters used to format the output. The actual shape of the log on the page may be a standard form with a row of variable headings followed by the row of data values in $\{\ell, r\}$ format. Note that alphanumeric variables may also be logged in the same manner ($\underline{r}$ may contain a code for "alpha," in which case $\underline{\ell}$ may be the number of characters). From this basic and simple format, any desired data log may be built. There is no need for the tedium of FORMAT statements, recompilation, and associated checkout. Additionally, formating and identifying information is stored permanently only once; there is no redundancy.

Other vector logs may take advantage of the same standard format and software, but the source of the data, as well as the vector definition, is specified, and it is called a vector log function VLF:

$$\text{L}\emptyset\text{G1} \leftarrow \text{VLF } (\underline{\text{V}}) \tag{16}$$

where $\underline{\text{V}}$ is the name of the data vector to be output. A column (e.g., Observation Number 2) of a matrix may be logged

$$\text{L}\emptyset\text{G1} \leftarrow \text{VLF } (\underline{\text{MDP}}(2)) \text{ .}$$

Matrix printout (Matrix Log Function MLF) would be similar, except here it is usually better to output columns across the page, and use the vertical dimension for variables:

$$\text{L}\emptyset\text{G2} \leftarrow \text{MLF } (\underline{\text{MDP}}) \text{ .} \tag{17}$$

Output of data vectors and matrices to other output devices would have similar forms; e.g., to a magnetic tape (WMT - Write Magnetic Tape)

$$\text{MT1} \leftarrow \text{WMT } (\underline{\text{V}}) \tag{18}$$

or

$$\text{MT2} \leftarrow \text{WMT } (\underline{\text{MDP}}). \tag{19}$$

Input of data vectors and matrices is defined as the reverse of output; e.g., from paper tape (RPT - Read Paper Tape):

$$\underline{V} \leftarrow \text{RPT} \quad . \tag{20}$$

The binary packing format for each system data parameter collection is identified in its vector definition, Equation (3). Thus, these packing parameters may become internal parameters in output of system data parameter blocks:

$$\text{LØG1} \leftarrow \text{MLF } (\underline{\text{SDP}}) \ , \tag{21}$$

or

$$\text{CRT} \leftarrow \text{MLF } (\underline{\text{SDP}}) \tag{22}$$

for display.

Error - Alarm Control

Consider here only the process input limit alarm, turned on by Step 7 of DAQ (above). However, the Error Alarm Control program in practice isolates the source of any alarm and takes appropriate action. The process input limit alarm examines $\{d_1 , d_2, s\}$ and takes the specified action. When display is required, the triplet $\{s, l, r\}$ is used to identify the y out of limits.

CHAPTER VI

FUNCTIONS OF THE INTERACTIVE ANALYSIS SUBSYSTEM

The operating environment described above provides a formal data, scheduling, and input-output framework for the interactive analysis subsystem. It can be shown that this framework is simple enough and, at the same time, sufficiently symbolic that the interactive analysis language may be expressed in terms directly related to this framework. In such a manner, a compact, mnemonic, notation may be used (desirable to man); yet much translation may be done using existing tools of the operating system (desirable for machine resource efficiency).

Several facts support the contentions of simplicity and high efficiency. In the first place, the exact, complete, priority structure of the system may be displayed or typed in a single matrix. For example, a core-bulk interactive system with several programs distributed over six priority levels is given in Figure 5. This system is used in the following chapter for illustration. System data structures may be similarly documented by one table per system data parameter matrix. Updating is a relatively simple task of loading data into appropriate tables (equivalent to complex "fill in the blanks" systems without the software overhead). Secondly, most functions may be built from more primitive, common, functions which have their basis in operating system code, which is the most efficient code. For example, the same table lookup functions are used in the scheduler, bulk-transfer

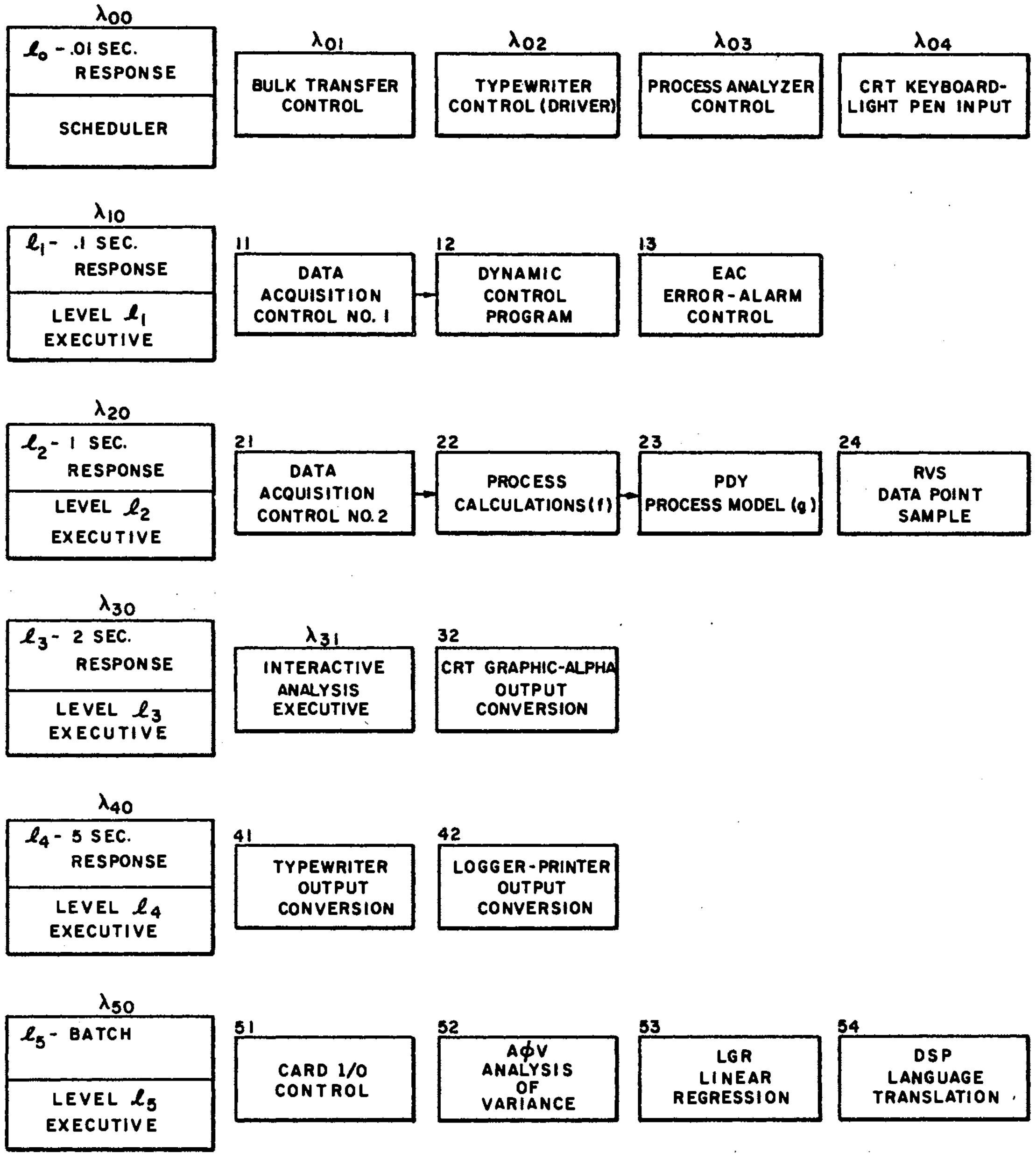


Figure 5. Example of System with Lattice Structure

control, and interactive translation. Thirdly, the system vector concept allows extensive internal parameter references, which preclude many inefficiencies generated in more conventional systems through redundant storage and handling of data in different programs, and passing data through external argument lists.

The general sequence for interactive analysis is given in Figure 6. Procedures and data are first defined in the language of the operating system (Blocks 1 and 2). Following execution of a procedure (Block 3), various results of this execution are displayed in a form which may suggest modification or redefinition of the data or procedure (Block 4). Re-execution and remodification follows until a result is obtained which must then be independently verified by further experimentation (Block 5). This "Procedure Execution Mode" is the primary operational mode for the interactive analysis. Following a brief discussion of this writer's interactive terminal operation and concepts, these functions will be discussed in more detail.

Interactive Terminal Operation

Interactive operation begins with an ENTER key depression to call and initialize the Interactive Analysis Executive. Whenever interactive terminal operation is indicated, error correction capability is implied: i.e., entry of characters from the CRT terminal keyboard appear immediately on the face of the display, but no translation of the information takes place until:

1. A complete interactive function (e.g., a procedure or data definition) has been defined and
2. An EXECUTE function key is punched from the keyboard.

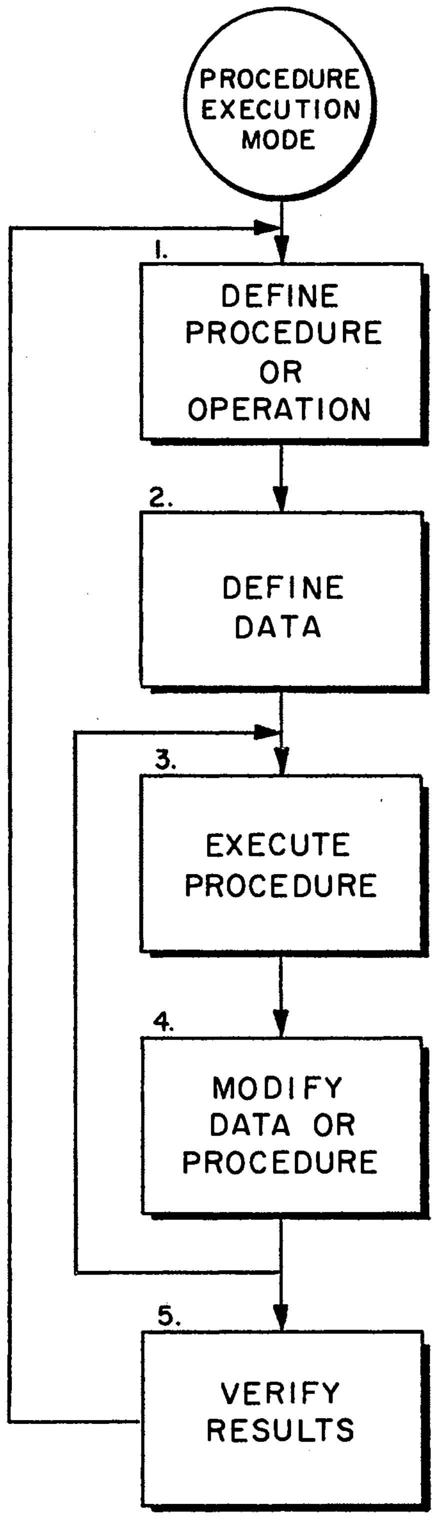


Figure 6. Interaction in Procedure Execution Mode

Thus, corrections of any part of the displayed information may take place at any time before EXECUTE by moving the CRT cursor to the erroneous data, and replacing it with correct information. In some operations, such as those when the light pen is used to successively point to displayed information, errors can be corrected by a DELETE/ RESTART function key whose effect is described below.

Once a graph has been obtained from the interactive data analysis which illustrates a verification of experimental work, or optimization of some process objective, a hard copy of the display would undoubtedly be desired. Several hardware devices are available which perform the translation from the display buffer to hard copy simply by pressing a PRINT key on the keyboard. For this logically simple function, such hardware would probably be preferred over software techniques.

Interactive Programming Concept

For understanding the following discussion, it is well to carefully point out the distinction between <u>object</u> procedures and <u>source</u> procedures. (The terms object and source are used because of the similarity to conventional batch computing functions of assembly or compilation from a source language to an object language.) Execution of object procedures generates an object output from a source input; the object output may be <u>data or a procedure</u>, depending upon the function of the executed object procedure. All executable procedures must be object; i.e., they must first have been defined by a procedure definition.

Several procedures and operations involve basic modifications to system data parameters. These require careful checkout. Such procedures are denoted by an asterisk (*) below to indicate a status which may demand a special "System Generation Mode" to enable them.

Procedure Definition

An operation or procedure is defined by identifying a name and, optionally, a parameter (argument) - list with a group of previously defined (object) operations or procedures. Such a definition may take the form

Define System Operation DSØ (ØPR (list)), (23)

or

Define System Procedure DSP (PRC (list)), (24)

ØP1 (list),

ØP2 (list),

⋮

This procedure when executed, accepts the sequence of operation (or procedure) calls with optional parameters and associates the new operation ØPR or procedure PRC with this sequence of calls. The principal differences between operations and procedures are (a) an operation is re-entrant and (b) the (macro) operation is organizationally a more permanent and widely used fixture, while a procedure may be a task set up to be executed only once. Thus, operation definitions require the system to provide volatile data usage information so that re-entrancy and register use requirements may be checked. An operation may not call a procedure.

An objective beyond the present work is to fit the basic programming languages into the body of these definitions to allow insertion of assembly or other language code where necessary. In this case, an option would allow insertion of prepared paper tape or cards in the appropriate reader and specifying this in the body of the DSP or DSØ.

Procedures may be purged from system files after a predetermined period of time, or by a Purge System Procedures function

$$\text{PSP (list)}, \tag{25}$$

where the list is of those procedures being deleted. Any procedure in on-line use would not be deleted by this function.

Each procedure defined by interaction is assigned a $\{p, q\}$ record (Chapter IV) on the background (lowest) level with a code for "no cyclic execution." The general operation to do this is the LEM given below in Equation (34). A more frequent special case of LEM for assigning procedures to the real time lattice structure is

$$\text{Assign to Real Time ART } (n_p, \ell, \Delta, xp) , \tag{26}$$

where xp is optional. A procedure to be assigned to real time must use internal parameters only.

Data Definition

The matrices of data points (<u>MDP</u>, etc.) become the primary source of input to data treatments, the objective being to reduce, from collections of experimental observations, data to a form which can be used to improve performance of the process. However, a data point will often contain more elements than needed for a given treatment. For example, only independent and dependent variables are necessary for input to a linear regression analysis, but the data point includes other experimentally controlled variables as well. Moreover, the necessity for randomizing the sequence or obtaining some points outside of an experimental design, the need to size $\underline{\text{MDP}}$ for many experimental series, and the vagaries of the complex plant environment combine to establish the necessity for easy selection and manipulation of data points. Therefore, the first step in selecting input to a treatment is usually to sample an input submatrix from one or more data point matrices. A completely flexible scheme for combining various $\underline{\text{MDP}}$ subsets can be a very large retrieval problem. At its simplest, such a scheme would require listing all rows and columns, together with any matrix concatenating information desired. This is a tedious chore except for very small matrices. With a light pen and the CRT display, this task can be reduced to a relatively quick "pointing" operation, which leads to the interactive functions

$$\text{Display System Matrix CRTM } (\underline{\text{MDP}}), \qquad (27)$$

and

$$\text{Build Sample Matrix from Display BMD } (\underline{\text{S}}), \qquad (28)$$

which would be followed by a light pen function pointing sequentially to each row and column to define variables and observations desired for the submatrix sample. A matrix definition for $\underline{\text{S}}$ results from this function, but is purged when a new matrix is built in $\underline{\text{S}}$. BMD is "executed" twice; once after all rows and columns have been selected. At this time, $\underline{\text{S}}$ replaces $\underline{\text{MDP}}$ on the display, allowing visual verification of a correct sample matrix.

Note that keying in the BMD (which requires the display) with $\underline{\text{MDP}}$ already on the display poses a housekeeping problem. One immediate solution would be to avoid the problem by combining CRTM and BMD into one function. However, operating on displayed data from the keyboard/ light pen combination is a basic and very applicable interactive function to be encountered repeatedly. Thus, one might as well face the problem here. Another solution would be to divide the display into two sections, but this would further limit the capacity of the display, which already limits the size of data matrix handled at any given time. Hardware "window" techniques (horizontal and vertical step) for stepping through matrices too large for display all at once would be preferred in this limited-resource system; purely software techniques would probably require too much housekeeping in themselves. Moreover, the interactive mode and nature of the application limits the size of most matrices to those which can be easily handled all at once.

A remaining solution is to build the necessary logic for two separate interactive display buffers into the interactive control program: an "Executed Display Output" XDØ buffer and "Display Keyboard Input" DKI buffer. Interactive display procedures are executed through XDØ while keyboard entry of a new procedure uses DKI. Edit-display functions change or update the displayed data in DKI, and other interactive functions referring to displayed data cause the DKI buffer to be loaded from XDØ as they are executed. The above sequence in more detail would therefore be:

1. The CRTM (MDP) is entered into DKI (causing display of DKI),
2. DKI is corrected if necessary,
3. CRTM (MDP) is executed, causing storage of MDP into XDØ and display of XDØ,
4. BMD (S) is entered into DKI, causing display of DKI,

5. BMD ($\underline{S}$) is corrected if necessary in $\underline{\text{DKI}}$, and
6. BMD ($\underline{S}$) is executed, causing $\underline{\text{DKI}} \leftarrow \underline{\text{XDØ}}$ (changing display back to $\underline{\text{MDP}}$), and a change of state of the display input routine to identify, in $\underline{\text{DKI}}$, the light pen inputs as rows and columns of the displayed data.
7. The light pen is used to point to the desired rows and columns. If an error is made during input from light pen-keyboard, the DELETE/RESTART key causes reinitialization ($\underline{\text{DKI}} \leftarrow \underline{\text{XDØ}}$) and, thus, the matrix definition may start again.
8. A second EXECUTE for $\underline{\text{BMD}}$ causes termination of the light pen inputs and the matrix definition, followed by selection and display of $\underline{S}$ through $\underline{\text{XDØ}}$.

Analysis of data from various system matrices often leads to a need for improved definition of system data matrices or parameters. In conventional systems recompilation would be required. Here the simple translation needed is built in. System macro-operations to accomplish data vector definition and generation are

$$\text{Define Vector in } \underline{X}\text{:}\quad \text{DVX } (\underline{V}, \text{ s-list}), \tag{29}$$

or

$$\text{Define System Vector:}\quad \text{DSV } (\underline{V}, \text{ s-list}), \tag{30}$$

or

$$\text{Define Parameter Vector DPV } (\underline{V}, \text{ spf-list}). \tag{31}$$

$\underline{V}$ is a new vector name and s-list is a list of variable names from $\underline{\text{SDP}}$ to define $\underline{V}$. Both DVX and DSV generate a vector name septuple $\underline{sv}$ (Chapter III) and vector definition list. However, for DVX, $\underline{V}$ refers to data in $\underline{X}$, while for DSV space on bulk is reserved for $\underline{V}$, and data must be transferred there by GCM, below. DPV refers to definition of data parameter lists (such as sdp, sv, pq, etc.), and spf-list, therefore, includes a parameter name, number of bits, and format code (alphanumeric, integer, binary, etc.) for each data unit. This structure, again, allows most system built functions to be accomplished interactively without recompilation.

For matrix definition, a previously defined vector name $\underline{V}$ is used:

$$\text{Define System Matrix DSM } (\underline{\text{MDP}}\ (\underline{V},\ \underline{n})), \tag{32}$$

which loads a matrix name septuple under $\underline{\text{MDP}}$ using vector definition $\underline{V}$ and column dimension $\underline{n}$ to assign storage. Normal sampling from $\underline{X}$ is done with

$$\text{Generate Column of Matrix GCM } (\underline{\text{MDP}}), \tag{33}$$

which uses the $\underline{\text{MDP}}$ matrix definition to transfer a data vector to $\underline{\text{MDP}}$ from $\underline{X}$, and

$$\text{Load Element of Matrix LEM } (\underline{\text{MDP}}\ (\underline{r},\ \underline{k}),\ \underline{u}), \tag{34}$$

where $\underline{r}$ and $\underline{k}$ are the row name and column number of the system data or parameter matrix $\underline{\text{MDP}}$, and $\underline{u}$ is the data unit value in the defined format. A special $\underline{r}$ or $\underline{k}$ code may refer to an entire row or column, in which case $\underline{u}$ is a list. Deletion of definitions may be done by

$$\text{Purge System Data Definition PSD } (\underline{\text{MDP}}), \tag{35}$$

which causes release of storage for $\underline{\text{MDP}}$ and elimination of the sv septuple. The associated vector definition is also deleted if $\xi_{\underline{\text{MDP}}} \neq \xi_v (v = 1, 2, \ldots, n_v)$ after $\underline{\text{MDP}}$ deletion.

Procedure Execution

Once a name has been assigned to a specific procedure using DSP, and data are defined, this new object procedure may be executed in the same manner as are the interactive executive functions; i.e., by entering the name of the procedure with arguments, if any, and pressing EXECUTE:

ENTER

PRC (list) (36)

EXECUTE

If the procedure obtains all its data from the argument list and/or internal definition, this is all that is required to obtain the PRC output. However, if PRC requires that, for example, external options be supplied, the interactive analysis executive may contain query-response subroutines which can be called upon to generate multiple-choice or direct response questions on the display of the form

$$\{mch,\ a,\ q_t,\ n_a,\ a_{1t},\ a_{2t},\ \ldots,\ a_{n_a t}\} \qquad (37)$$

or

$$\{md,\ a,\ q_t,\ l_a,\ r_a\}\ , \qquad (38)$$

where

<u>mch</u> and <u>md</u> are unique names of the respective query parameters,

a	is a (dummy) data-unit field for returning the answer,
q_t	is a one-line alphanumeric text for the query,
n_a	is the number of choices for multiple-choice queries,
$a_{1t}, a_{2t}, \ldots, a_{n_a t}$	are the alphanumeric lines of text for identification of multiple choice responses, and

(ℓ_a, r_a) defines the left and right (or alpha code) format for a.

Thus, the main system overhead for interactive queries (in addition to these query-response subroutines) is the storage for the query itself and the call from the procedure requesting an interactive response.

In execution, the query is displayed through the DKI buffer and the interactive analysis executive is switched to a wait state. The response is made either by light-pen pointing to a multiple choice answer displayed on the console, or by keying direct answers, followed by an EXECUTE, which transfers the DKI - stored answer back to the calling procedure. As above, any mistake made before EXECUTE may be corrected using DELETE/RESTART.

The query-response subroutines may be accessed interactively through the following pair of operations:

ENTER

Define Query Parameters DQP $(\underline{mch}, \underline{a}, \underline{q_t}, \underline{n_a}, \underline{a_1t}, \underline{a_2t}, \ldots, a_{n_a}t)$

or

DQP $(\underline{md}, \underline{a}, \underline{q_t}, \underline{l_a}, \underline{r_a})$ (39)

EXECUTE

ENTER

⋮

Query Call: (40)

QC $(\underline{mch})$

⋮

EXECUTE

The answer $\underline{a}$ from $\underline{mch}$ is returned to the calling procedure during execution; $\underline{a}$ is used in this procedure to determine the option selected. Examples of use are given in Chapter VII.

Once an object procedure is executed, the primary output is displayed for immediate inspection. A procedure (e.g., a data treatment such as regression) may have several output displays; the first one is designated as primary. A keyboard "PAGE" function causes display of each succeeding output, back to primary. Procedure outputs are generated into bulk-resident common scratch areas (interactive analysis procedures should be limited to unique priority levels, such as one of the two lowest, to make efficient use of scratch storage). These areas are defined in the system matrix definition parameters. Therefore, if output is in the form of a floating point or data-parameter matrix, the standard display matrix operation (CRTM) may be used for output. However, a most valuable characteristic of the CRT in interactive application is its graphic capability; outputs would, therefore, be graphic where possible.

The required graphic capability for this application need not be highly sophisticated; for example, full graphic input, which requires most of the software sophistication and complexity (1), is not essential since the application is "discovery" rather than "design." Three-dimensional output capability, while applicable, requires far too many complexities for the benefits it holds. Two-dimensional graphics (parametric plots) are in wide use for physical systems; and any graphic method loses its effectiveness beyond three dimensions anyway. Hence, the great majority of our graphic requirements may be summarized into:

1. Plots of variables versus time (an xy - plot),
2. Plots of y versus x_1, parameterized by x_2, x_3, ...,
3. Two or more plots superimposed for comparison.

In each case, alphanumeric identification of the plots would be necessary, and in each case, modification of variables and/or parameters in the analysis environment is often necessary.

The data for plotting may exist in any system data matrix, or may have to be generated from a model (g). A plot of data from a system matrix superimposed upon model predictions is a highly desirable capability in verification. Using procedures for executing g on arbitrary inputs (using b_0 and b_1 parameters for ranging), and sampling and storing history, one may obtain model outputs for various values of inputs, and, therefore, data matrices for plots using these previously defined procedures.

Scaling of plots may be accomplished using either $\{b_0, b_1\}$, or $\{g, h, \dot{n}_c\}$ parameters of <u>SDP</u> (Chapter III), as well as the magnitude of the data itself in some cases. With these alternatives and interactive response, it is not necessary to go through the tedium of scaling; a plot may be displayed and then adjusted if necessary using interactively entered offsets and scale factors. Some display hardware may also have linear vertical and horizontal adjustments useful for limited scaling. Scaling thus becomes a system function using internal parameters.

Reference (3) outlines the various hardware methods for command decoding and generation of lines and symbols on the CRT. With its limited system resources, and process and data analysis software, the process computer cannot provide very extensive direct character and line display generation. Therefore, hardware character generators, a combination random scan mode display format for line generation, and a typewriter mode for character generation, with appropriate control codes for switching modes, are necessary. In this manner, a single block of display commands may be transmitted to the CRT control logic (through the DKI or XDØ buffer) for any desired display, including superimposed plots.

The following procedure provides the basic functions for generating xyz - plots:

Plot from a Matrix s_1 versus s_2 by s_3, according to η:

$$\text{PMP } (\underline{S}, \eta, s_1, s_2, s_3) \tag{41}$$

$\underline{S}$ = any system matrix containing all the variables,

η = code for plotting style: point-by-point, solid line, or dotted line,

s_1 = system symbol for the y-axis variable,

s_2 = system symbol for the x-axis variable,

s_3 = optional system symbol(s) of desired parameter(s).

An example of a point-by-point display from this procedure for the variables s_1 = yobj, s_2 = xfrr, and s_3 = sfra is shown in Figure 7.

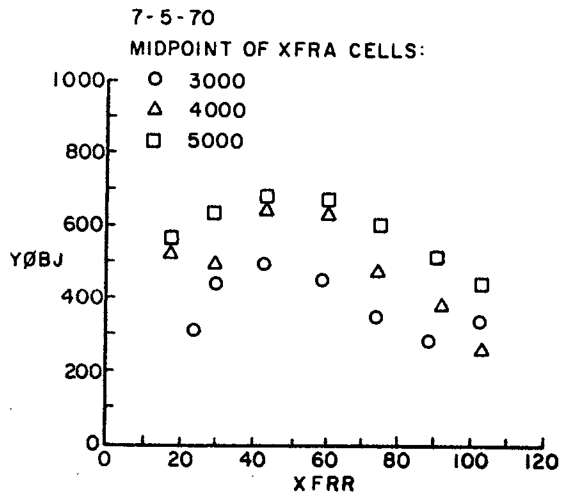


Figure 7. Point Plot of Yobj Versus XFRR with XFRA as Parameter

To obtain superimposed plots, an "Add" option is added to the above procedure and one has

"Add Parametric Matrix Plot: APMP $(\underline{S}, \eta, s_1, s_2, s_3)$. (42)

As indicated above a limited amount of interaction with plots is desirable:

1. Function keys to cause tracking and display of light pen trace across the CRT face.
2. Keyboard additions to plots for inserting additional information before printing a hard copy.
3. Light pen function keys to allow moving or deleting specific points on the plot. A different symbol (or color) for moved points would be used. The matrix $\underline{S}$ data would be changed so that modified input to analysis of variance, regression, etc., could be accomplished, but the original system matrix (from which $\underline{S}$ was built) would not be changed.
4. Adjustments to plot scale factors and offsets by function key and light pen manipulation of ordinate, abscissa, or groups of points having same symbol (for s_3 parameter scaling).

Data and Procedure Modification

Several of the operations to accomplish modification of data and procedures have been discussed above under data and procedure definition. Of interest here are some of the mathematical manipulations useful in applying data analysis programs to experimental data.

Many data modifications needed for data analysis are transformations on entire rows (variables) of matrices. For example, in fitting linear regression models to process variables, nonlinear terms such as x^2, log x, x_1, x_2, etc., are often used to obtain satisfactory linear descriptions of process responses. Because difficulties arise in matrix inversion using dependent variables, such variables are often coded by normalizing; i.e., by subtracting the mean and dividing by the standard deviation of each variable. These modifications would be in addition to simple linear transformations, including exchanges of variable positions. Such transformations change the matrix definition; therefore, modified matrices should be built from "scratch" areas not used by on-line programs. Thus, modification begins with a transfer or sampling of the matrix into the scratch area, followed by step-by-step execution of the desired modify operations; e.g.,

```
ENTER
S ← MDP
CRTM (S)                                    (43)
EXECUTE
```

to move and display the matrix from its scratch area. Definition of each new variable within $\underline{S}$ makes use of the procedure

$$\text{Define New Variable} \quad \text{DNV } (s_1, r, \text{OPN } (s_2)), \tag{44}$$

where s_1 is any symbol for assignment to the new variable row position r. OPN(s_2) is a defined operation for obtaining each value of s_1 from the variable(s) listed as OPN arguments. Note that r may be a previously defined variable position, in which case that variable is replaced and redefined by $s_1 \leftarrow$ OPN(s_2). Following each execution of DNV, the modified $\underline{S}$ is displayed. Other matrices may be modified by operations more general than DNV, but these would be system generation procedures involving too much risk of the uncontrolled consequences of human error for use in on-line interactive analysis with a process control computer. If an erroneous DNV is executed, all that is necessary is to repeat the sequence in (43).

Another modification necessary for data treatments such as analyses of variance (AOV), as well as for parametric plotting, retrieval, or "just looking" at data, is a proper sequencing, or classifying, of matrix columns (observations) according to the values of independent variables. Data input to the AOV are normally responses only, but the method requires formal information about the independent variable settings, or treatment groups, for correct analysis. The sequence according to independent variables depends upon the number of independent variables, and the number of levels of each variable. The former must be obtained from the experimenter, but the latter can be determined from the data, given the system parameters g, h, and n_c, (Chapter III, sdp parameters). An auto-classification technique to accomplish this task for any matrix is given with an example in Appendix A. Again, for minimizing effects of human error, the technique would normally be limited to a scratch matrix $\underline{S}$, but in this case moving the matrix to the scratch area may be included in the auto-classify procedure. The auto-classify specification is:

$$\text{Auto-classify } \underline{\text{MDP}} \text{ to } \underline{S} \quad \text{ACS } (\underline{\text{MDP}},\ x_1,\ x_2,\ \ldots), \qquad (45)$$

where x_1, x_2, ... are the names of the independent classification variables in $\underline{\text{MDP}}$. The auto-classify parameters (see Appendix A and the Example, Chapter VII) are displayed upon execution of ACS for verification of a correct auto-classify. The auto-classified ACS provides a base matrix from which various dependent variables may be input to the analysis of variance, as

$$\text{AØV}\ (y) \tag{46}$$

where y is the name of the dependent variable, and all other parameters needed by AØV are provided in the auto-classified $\underline{S}$, which is an internal parameter. The auto-classify technique is also used in parametric plot routines for cross classification of the plot symbols.

Other steady-state analysis or optimization data treatments require similar data handling to that above and would be added to this open-ended procedural structure as necessary. For the present purpose, add the data treatment (on $\underline{S}$):

$$\text{Linear Regression LRG}\ (y,\ x_1,\ x_2,\ \ldots) \tag{47}$$

where y is the dependent variable and x_1, x_2, ... are the independent variables. The output from this data treatment is a model g, with responses line-plotted on the CRT, superimposed upon observed responses, (as, e.g., in Figure 8). In this manner, immediate visual observation of the quality of the model would be possible. Further, g may be defined as a procedure in response to an interactive query response upon keyboard stepping to the second "page" of the LRG output displays.

Verification of Results

It is clear from the above that independent verification of the results of interactive analysis can be accomplished by use of the same tools required to obtain initial results. Because the system is on-line to the physical process which is being described, immediate

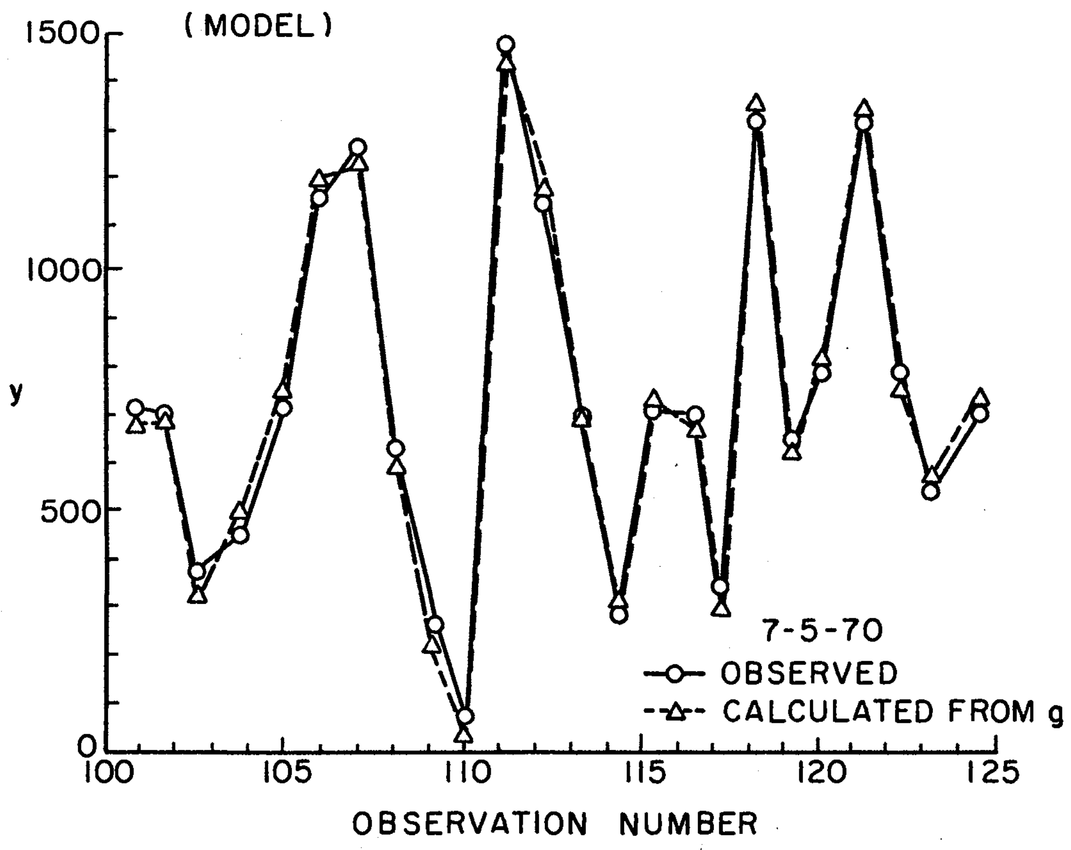


Figure 8. Model Verification LRG Display

repetition of experiments is possible to provide independent data by which the results of previous experiments can be verified or extended. Missing gaps in initial tests can be filled in. It is here that feedback of this outer analysis loop begins to make itself felt; not only in reduced time and costs of the analysis, but in direct improvements to the process using these quickly formalized results. The verification takes place, in summary, through:

1. Comparison of responses between experiments.
2. Superimposition of models upon their own and independent data observations.

CHAPTER VII

AN APPLICATION EXAMPLE

Refer to Figure 5. Assume that measurements from a chemical reactor are being sensed by the process computing system, and that two Data Acquisition operations (DAQ) operating on independent data are assigned to positions p_{11} and p_{21} as shown in the figure. The former DAQ is chained to a dynamic control program (p_{12}) operating on the same level. The latter DAQ provides for the conversion of all other process inputs on a lower frequency and priority, and is chained to a process calculations program (p_{22}), which generates the remainder of the calculated values needed ($\underline{Z} \subseteq \underline{X}$, see Figure 2). An on-stream process chromatograph measures feed stream compositions necessary for material balances yield calculations, and experimental control. The balances are typed out periodically to monitor the state of process calibration of the instrumentation on which responses are based. A process analyzer control program (λ03), using a data base similar to that of <u>YDP</u>, is required to sample and convert the data as it becomes available from this analyzer (programmed cycle). The remainder of the operating system programs are set up on priorities as indicated in the figure. The data treatments of this example will be the Analysis of Variance (AØV) and Linear Regression (LGR).

Define a data sample vector <u>EXPQ</u> from the measured and calculated values in X:

ENTER (In the remainder of this chapter, ENTER will be understood

wherever necessary)

DVX (EXPQ, XFR1, XFR2, XTR, YP1, C1R1, C2R1, C3R1, C4R1, C5R1)

EXECUTE. (See Equation (29).)

Let XFR1 and XFR2 represent system-sensed and calculated flow rates of primary and secondary feed streams to the hypothetical chemical reactor; let XTR and XPR be the temperature and pressure of the reactor, respectively. Let YP1 be the yield of the primary product, and C2R1, C2R1, ..., C5R1 represent measured compositions of R1 included as data for experimental control While EXPQ contains fewer data units than a typical real case, all the primary classes of variables are included: independent control, dependent (response), and experimental (environmental) control. The EXECUTE keyboard function causes storage of a new sv (matrix-name) septuple. The values of sv are:

$\underline{V}$ = $\underline{\text{EXPQ}}$;

m_v = 10[number of variables];

n_v = 1[number of observations];

c_v = a system-assigned link address of the sv containing the ordered set of indices of XFR1, etc., in $\underline{X}$;

d_v = null (∅), i.e., there is no column definition;

e_v = link to XFR1 in $\underline{X}$, assigned by system but not necessary for vectors resident in $\underline{X}$;

ξ_v = a retrieval code for $\underline{\text{EXPQ}}$; e.g., the binary check sum of the vector element symbols; and

Φ_v = code for floating point data.

Following storage of these system parameters, the interactive subsystem displays an acknowledgement, e.g.,

```
DVX COMPLETE

VECTOR SYMBOL EXPQ IN X

XFR1         FLOATING POINT DATA
XFR2
XTR          SINGLE PRECISION
XOR
YP1          Cv  = B100000
C1R1
C2r1         Dv  = Ø
C3R1
C4R1         Ev  = C1040
C5R1
```

This operation defines all the particular variables in $\underline{X}$ relevant to the study. Suppose a data point for this study is to be defined as an average of eight periodic samples from $\underline{X}$. Then, a sample matrix is defined by

DSM ($\underline{\text{SM}}$ ($\underline{\text{EXPQ}}$, 8))

EXECUTE. (See Equation (32).)

Here, another sv record is generated as above, making use of the sv-record of $\underline{\text{EXPQ}}$. Except here, e_v = a link address to separate system-assigned storage for $\underline{\text{SM}}$; and, storage for the column definition (observation numbers) is reserved and initialized in the d_v area.

A matrix for the averaged data points is now reserved by

DSM (DPEXPQ (EXPQ, 12))

EXECUTE. (See Equation (32).)

To store values into the sample matrix $\underline{\text{SM}}$ on-line, a system procedure is defined, called "Record Vector in SM," RVS, by

DSP (RVS). (See Equation (24).)

GCM ($\underline{\text{SM}}$). (See Equation (33).)

EXECUTE

This definition simply converts SM to an internal parameter for RVS so that RVS may be assigned to the real time lattice priority structure. Assuming the desired priority level $\ell = 2$ as in Figure 4, results in

ART (RVS, 2, 300). (See Equation (26).)

EXECUTE

which assigns RVS to a 300-second cycle on priority level two, and sets the RVS program state $\Psi_{(1 or 2)} \leftarrow \Psi_0$ to unlock the program and start the cycle execution. To define the procedure for recording the data point as the average of the rows of SM, enter

DSP (RDP). (See Equation (24).)

RAVE (SM, DPEXPQ) (48)

CRTM (DPEXPQ). (See Equation (27).)

EXECUTE,

where RAVE is a previously defined system operation for averaging rows of a matrix (SM) and generating the result into the current column of another matrix (DPEXPQ), as in the GCM, Equation (33). CRTM displays DPEXPQ. Now, whenever a steady-state data point, defined here as an average of eight samples over 40 minutes, is required, execute the procedure RDP:

RDP (56)

EXECUTE

Suppose now that a reactor experimental program is designed and executed with the independent variables XFR1, XFR2, and XTR; RDP being executed at each point. The matrix DPEXPQ contains the data for analysis of the experimental results. As soon as sufficient data are available for studying relationships between points, the data would be displayed and a sample matrix for analysis generated:

ENTER

CRTM (DPEXPQ). (See Equation (27).)

EXECUTE,

which displays the matrix:[1]

DATA MATRIX DPEXPQ

1. SQNØ	101	102	103	104	105	106	107	108	109	110	111	112
2. XFR1	325.4	300.2	360.1	323.2	304.1	356.2	326.1	302.4	359.8	320.9	298.5	357.5
3. XFR2	10425	11980	11860	10569	9422	9598	10328	12240	11946	10450	9600	9725
4. XTR	465	447	435	464	442	440	460	495	491	462	489	487
5. XPR	48.90	49.90	51.10	50.24	48.80	52.24	49.11	47.99	51.11	48.02	49.50	50.76
6. YP1	69.76	72.00	76.06	70.50	67.90	73.42	69.95	67.74	73.54	67.65	65.05	71.60
7. C1R1	2.34	2.36	2.35	2.33	2.34	2.35	2.36	2.30	2.33	2.36	2.35	2.35
8. C2R1	5.61	5.71	5.77	5.76	5.81	5.88	5.99	6.24	6.23	6.22	6.21	5.95
9. C3R1	10.44	10.40	10.37	10.10	10.07	10.00	10.00	9.76	9.50	10.12	10.15	10.23
10. C4R1	12.01	12.50	12.23	12.48	12.69	12.48	12.39	12.70	12.65	12.65	12.40	12.51
11. C5R1	70.02	70.11	71.24	69.50	70.11	70.15	70.16	70.07	70.11	70.98	70.90	70.11

The Build Matrix from Display operation is now used to select the data point number; the independent variables XFR1, XFR2, and XTR, and the dependent variable YP1; and the experimental points for this particular 2^3 factorial design:

BMD (S). (See Equation (28).)

EXECUTE

[1] In this example, the CRT window size is assumed large enough for all displays.

The light pen is pointed to the row numerals 1, 2, 3, 4, and 6; and the column numbers (SQNØ) 102, 103, 105, 106, 108, 109, 111, and 112, followed by another EXECUTE. The sampled matrix $\underline{S}$ is then displayed:

```
                         DATA MATRIX S

 1. SQNØ   102     103     105     106     108     109     111     112
 2. XFR1   300.2   360.1   304.1   356.2   302.4   359.8   298.5   357.5
 3. XFR2   11980   11860    9422    9598   12240   11946    9600    9725
 4. XTR      447     435     442     440     495     491     489     487
 5. YP1    72.00   76.06   67.90   73.42   67.74   73.54   65.05   71.60
```

To calculate an analysis of variance on YP1 for this experiment, auto-classify (see Appendix A) the sample matrix $\underline{S}$ into a standard sequence:

ACS ($\underline{S}$, XFR1, XFR2, XTR). (See Equation (45).)

EXECUTE:

```
                 AUTOCLASSIFIED DATA MATRIX S

 1. SQNØ   109     103     112     106     108     102     111     105
 2. XFR1   359.8   360.1   357.5   356.2   302.4   300.2   298.5   304.1
 3. XFR2   11946   11860    9725    9588   12240   11980    9600    9422
 4. XTR      491     435     487     440     495     447     489     442
 5. YP1    73.54   76.06   71.60   73.42   67.74   72.00   65.05   67.90
                 8 TREATMENT GROUPS WERE GENERATED
                THE LENGTH OF EACH IS 1.  DESIGN IS:
 XFR1     4 4 4 4 0 0 0 0
 XFR2     2 2 0 0 2 2 0 0
 XTR      5 0 5 0 5 0 5 0
                          AOV PARAMETERS
          NO. VARIABLES 3  NO. LEVELS EACH:  2 2 2
```

The dependent variables in S (only one is considered in this example, YP1) are now in the correct sequence for input to the data treatment AØV, and the AØV parameters are stored in the S area for internal input to the AØV. The data treatment AØV may now be executed:

AØV (YP1). (See Equation (46).)

EXECUTE

which provides the AØV results on the display:

```
              AØV - YP1

VAR       D.F.     S.S.        M.S.
1.-XFR1    1     60.11561    60.11561
2.-XFR2    1     16.15961    16.15961
3.-XTR     1     16.38761    16.38781
12         1      0.06301     0.65051
23         1      0.95911     0.95911
13         1      0.55651     0.55651
123        1      0.06301     0.06301
TOTAL      7     94.85219
```

Query calls may be inserted in the AØV procedure for interactive flexibility. For example, stepping through the AØV output displays, one may encounter:

```
ENTER ERROR MEAN SQUARE FOR F TEST
1.0 END                 [Keyboard Response]
ENTER DEGREES OF FREEDOM
20 END                  [Keyboard Response]
SELECT CONFIDENCE LEVEL
 ☐ .99    [✓ .95]    ☐ .90
[Light Pen Response]
```

Upon entering an error mean square value, degrees of freedom, and selecting a confidence level with the light pen, an F - test is applied in the classical manner and the variables declared significant at the given confidence level, in their order of significance, are added to the AØV output results display:

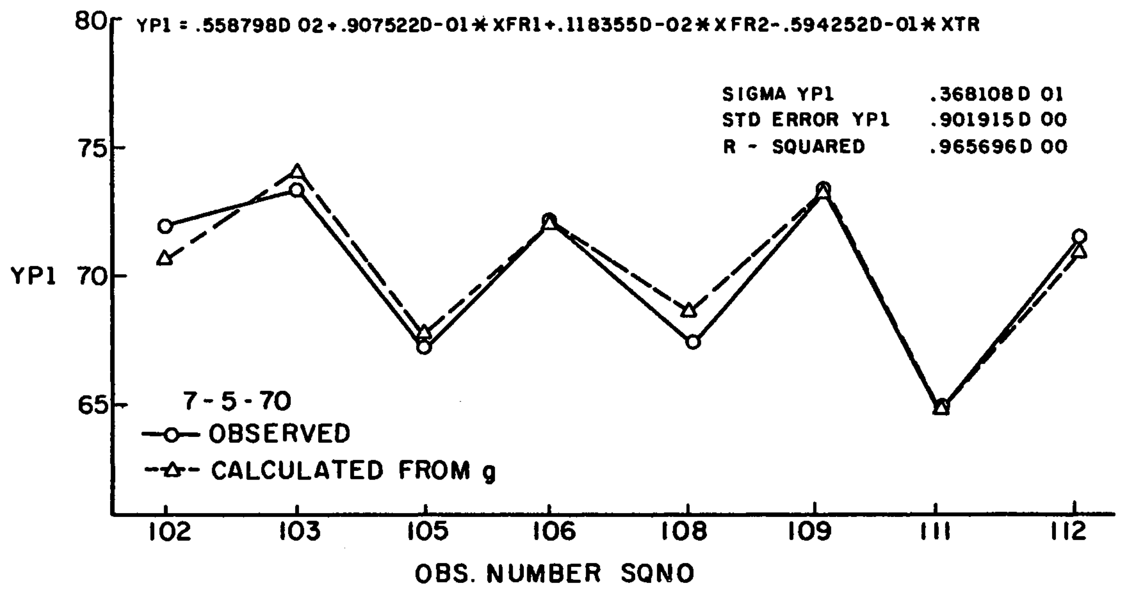
```
                     AØV - YP1
                                              SIGNIFICANT
  VAR      D.F.     S.S.          M.S.        VARIABLES
1.-XFR1     1     60.11561     60.11561 --- (1)
2.-XFR2     1     16.15961     16.15961 --- (2)
3.-XTR      1     16.38761     16.38781 --- (3)
12          1      0.61051      0.65051
23          1      0.95911      0.95911
13          1      0.55651      0.55651
123         1      0.06301      0.06301
TOTAL       7     94.85219
         CONF LEVEL = .95     EMS = 1.0
```

Entering the error mean square interactively in this manner allows independent, previously obtained estimates of experimental error to be used.

Taking the significant variables from the analysis of variance, one obtains a linear function g from the linear regression procedure (on $\underline{S}$)

```
LRG (YP1, XFR1, XFR2, XTR),
EXECUTE
```

which would produce the following outputs:



The keyboard paging function would step the LRG output display to other options or information calculated in the LRG procedure. A query of significance to the on-line system is:

ENTER DISPOSITION OF MODEL

☐ DO NOT RETAIN

[x] RETAIN
[Light Pen Response]

ENTER NAME OF MODEL G (<u>INPUT</u>, <u>OUTPUT</u>)

PDY (<u>X</u>, <u>X</u>) [Keyboard Response]

The procedure PDY (INPUT and OUTPUT are internal parameters) may be put on-line by:

```
CRTM (PQ)
EXECUTE
LEM  (PQ (XP, 22), 23).  (See Equation (34).)
EXECUTE
CRTM (PQ)
EXECUTE,
```

which:

1. Displays the PQ scheduler table for verification that PDY may be chained to the process calculations program (p = 22) and to determine the program number of PDY (e.g., 23);
2. Loads 23 into the xp field of the program 22 pq - record; and
3. Displays the updated PQ for verification of a correct entry.

The reader experienced in experimental work, particularly in industrial plants geared for commercial production, will realize that events rarely take place orderly enough, and instrumentation is rarely good enough, to presume that the above example is typical. On the contrary, unforeseen events, instrumentation errors, and production requirements cause many puzzles and digressions from original goals. Unexpected behavior of responses will stimulate further experimentation, as will the need for verification. It is precisely this environment in which the interactive analysis system is of most value. In shortening the time lapse from data gathering to the analysis phase, problems may be recognized and solved in time to avoid or minimize the high risks of error, wherever they occur in the complex plant environment. Since there is a high psychological inertia working against experimental activities in the industrial plant, tests are usually temporary with long periods of more routine operation. With the interactive analysis system, and process calibration by computer, the success of an experiment, once run, may be more reliably judged before the test inertia relaxes, thus allowing quick repetition of portions of experimental work as needed.

CHAPTER VIII

CONCLUSIONS

The main objective of this study was to outline for evaluation functional requirements for applying a CRT display terminal to interactive data analysis using an on-line process computer. Such an outline has been given. This study has resulted in a better understanding of the software structural requirements, and some of the hardware requirements as well, for the interactive analysis system. Perhaps as important, a better appreciation of the benefits achieveable by more thorough and formal planning of software efforts has been realized.

Perhaps the most significant outcome of this work is the identification of a unifying data and procedural structure which enhances the efficiency of both the process and interactive analysis systems while allowing simplified specification at the man-machine interface. Using this data structure, a single set of matrix-manipulation operations satisfies most operating as well as interactive system data handling requirements.

Because the total effort required to implement complete systems is typically many man-years, it could not be within the scope of this study to demonstrate the approaches outlined. This actually turned out to have advantages. For example, interrelationships from the beginning to end of the study were more easily modified, and it is hoped, improved, as new problems arose in rounding out the study. And, hardware peculiarities were not allowed to dictate or influence techniques; thus more appropriate hardware may now be specified.

While the final level of detail has not been reached, it is apparent from this work that considerable power may be concentrated in a relatively few, but well-defined operations, which now can be written with most of the potential uses in mind. Among these are the masked scanning, insertion, deletion and other manipulations of matrix parameter and data tables, logging and typing of matrices, display of matrices, plots, and queries.

To facilitate interactive analysis, the operating system for scheduling and priority control may be simplified relative to extant systems by the formal specification of unifying data, parameter, and priority structure given here. Furthermore, a system may be completely specified and easily documented by adhering to these formal structures throughout the data acquisition, control, sampling, calculation, and interactive data analysis phases of industrial process studies.

Relatively simple parametric plots satisfy the large majority of requirements for graphic display in interactive analysis. While of great importance to efficient man-machine communication, these graphics do not replace tabular data display in interaction. An auto-classification technique was developed for powerful man-computer preparation of process data for analysis treatments, particularly for analyses of variance and parametric data plotting.

The interactive analysis console and software support may replace several hardware and software components of present systems; e.g., the operators console, input-output and alarm typewriters, and job-control languages for batch compilation and execution. Any such language becomes a natural part of the interactive language (as do the programming languages), and at the same time may be handled through other input-output devices as well as the terminal, if desired. However, interactive analysis increases the need for large bulk memories with fast access and transfer rates to store the data treatments, some of which may be large segmented programs. Associative memory, packed data instructions (fetch, store, and compare), and sophisticated indirect addressing are computer features lending higher efficiency to the main functions of the interactive analysis system.

Necessary CRT features include hardware image refresh, window techniques, symbol and vector generation, hard copy printing, and random, simultaneous alphanumeric-graphic instruction capability. Given more primitive display hardware, a small, inexpensive minicomputer should be used to provide these functions. Removable serial memory (e.g., magnetic tape cassettes) are desirable system components for handling large data matrices and storing past files for retrieval.

With formal structuring and appropriate hardware, interactive data analysis using the process computer is feasible, and can contribute significantly to the productivity and economic benefits of industrial process studies.

BIBLIOGRAPHY

(1) Coons, Stephan Anson. "An Outline of the Requirements for a Computer-Aided Design System." 1963 Spring Joint Computer Conference Proceedings, Vol. 23, p. 299. See also the four companion papers following this about the MIT work on graphics in computer-aided design systems.

(2) Coons, Stephan Anson. "The Uses of Computers in Technology." Scientific American, 215, 3 (September, 1966), pp. 177-188.

(3) Van Dam, Andries. Computer-Driven Displays and Their Uses in Man Machine Interaction, in F.L. Alt (Ed.), Advances in Computers, Vol. 7. New York: Academic Press, 1966, pp. 239-290.

(4) Aronson, B. L. "CRT Terminals Make Versatile Control Computer Interface." Control Engineering (April, 1970).

(5) Abraham, Frederick D., Laszlo Betyar, and Richard Johnston. "An On-Line Multiprocessing Interactive Computer System for Neurophysiological Investigations." Proceedings, 1968 Spring Joint Computer Conference.

(6) Lockemann, Peter C., and W. Dale Knutsen. "A Multiprogramming Environment for Online Data Acquisition and Analysis." CACM, Vol. 10, No. 12 (Dec. 1967), p. 758.

(7) Ball, N. A. et al. "A Shared Memory Computer Display System." IEEE Transactions on Electronic Computers, Vol. EC - 15, No. 5 (Oct. 1966), p. 750.

(8) GEPAC 4020 RTMOS Manual, PCR - 114, GE Process Computer Department, Phoenix, Arizona, 1968.

(9) IBM Time Sharing Executive System, Concepts and Techniques, Form C26-3703, International Business Machines, 1967.

(10) SDS Sigma 2 Basic Control Monitor, Reference Manual 90 1064A March 1968, Scientific Data Systems Corporation (now Xerox Data Systems), Santa Monica, Calif.

(11) Knuth, Donald E. The Art of Computer Programming, Vol. I, Fundamental Algorithms. Reading Massachusetts: Addison Wesley Pub., 1968.

(12) Iverson, K. E. A Programming Language. New York: Wiley, 1962. See also a bibliography on APL available from IBM local representatives.

(13) Kent, William. "Assembler-Language Macroprogramming: A Tutorial Oriented Toward the IBM 360." Computing Surveys, I, No. 4 (December 1969).

(14) Minutes of the Third Workshop on Standardization of Industrial Computer Languages Part 1 - Narrative March 2-6, 1970, Purdue University, Lafayette, Indiana 47907. See also the minutes of prior workshops at Purdue.

(15) Dolotta, T. L. "Functional Specifications for Typewriter-Like Time-Sharing Terminals." Computing Surveys, Vol. 2, No. 1 (March, 1970).

APPENDIX A

AUTO-CLASSIFICATION OF DATA INTO TREATMENT GROUPS

Consider the m x n data matrix <u>SMX</u>, where the n columns represent different <u>observations</u> of the m <u>variables</u> listed in rows. Partition <u>SMX</u> into a k x n submatrix <u>X</u>, $k \leq m$, where ik = 1, 2, ..., k are indices of the (independent) classification variables for <u>SMX</u>. Define the transformation from <u>X</u> into <u>P</u>

$$P_{ij} = |a_i X_{ij} = b_i|_{int};\ i = 1, 2, \ldots, k;\ j = 1, 2, \ldots, m; \tag{1}$$

such that a k x n integer (int denotes "integer portion of") matrix <u>P</u> is formed with elements P_{ij}. $a_i \subseteq$ (belongs to) <u>A</u> and $b_i \subseteq$ <u>B</u> are fixed vector sets chosen such that the integer portion of P_{ij} is an element of a fixed finite subset of integers <u>S</u> for any value of X_{ij} (Set $P_{ij} = 0$ for $P_{ij} < 0$ and $P_{ij} = 0$ or h where $P_{ij} > h$; h = the highest integer in <u>S</u>). Call each element P_{ij} a <u>projection</u> of X_{ij} into <u>S</u>. Now concatenate all P_{ij} in each column to form a <u>composite projection</u> C_j over columns of <u>P</u> (one for one with <u>SMX</u>). Order the column indices of <u>P</u> according to the high-to-low numeric values of the composite projections. From this ordering, generate L sets of index subsets

$$N_{j'} = [\text{all } j\,|\,C_{j'} = C_j,\ j \text{ not already in } \underline{N_{j'}},\ j = 1, 2, \ldots, n,$$
$$j' = 1, 2, \ldots, L]\ . \tag{2}$$

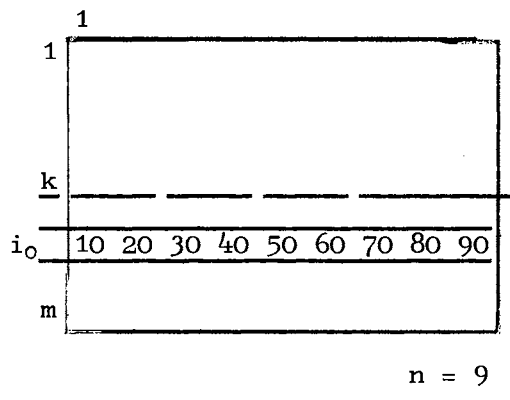
In other words, order the column indices of SMX in a sequence according to their composite projections and group together those with identical composite projections into L index subsets $\underline{N_{j'}}$ [j' = 1, 2, ..., L]. The index sets may then be used to rearrange the columns of SMX such that response treatment groups T(i_0) may be written directly from any row i_0 of the rearranged SMX:

$$\underline{T}(i_0) \left\{ \begin{array}{l} T_1(i_0) = X_{i_0 j},\ j = 1, 2, \ldots, n1 \\ T_2(i_0) = X_{i_0 j},\ j = n1 + 1, n1 + 2, \ldots, n1 + n2 \\ T_3(i_0) = X_{i_0 j},\ j = n1 + n2 + 1, n1 + n2 + 2, \ldots, n1 + n2 + n3 \\ \quad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\ T_L(i_0) = X_{i_0 j},\ j = n - n_L' + 1, n - n_L' + 2, \ldots, n \end{array} \right. \tag{3}$$

where $T_1(i_0)$, $T_2(i_0)$, ..., $T_L'(i_0)$ are the response treatment groups for the variable i_0, $X_{i_0 j}$ is an element of SMX, and n1, n2, ..., n_L' are, respectively, the number of elements in each response treatment group (i.e., the number which had identical composite projections). In general, $n1 \neq n2 \neq \ldots \neq n_L'$, but it may be convenient to allow a fixed block of memory for each group (block length = max [$n_{j'}$, j' = 1, 2, ..., L]), and allow for null elements. A list of response treatment groups may be obtained by resetting i_0 = [new variable index from a list] and reiterating through expressions (3) to generate a new $T_{j'}$, [j' = 1, 2, ..., L] set.

For example, suppose the matrix SMX

SMX



were partitioned at k and projected into $\underline{P}$:

$$\begin{array}{r|ccccccccc|}
1 & 5 & 5 & 7 & 5 & 3 & 5 & 3 & 5 & 7 \\
 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\
 & 2 & 2 & 4 & 4 & 4 & 2 & 4 & 4 & 4 \\
 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
K=5 & 7 & 7 & 7 & 3 & 3 & 7 & 3 & 3 & 7 \\
\hline
j = & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & n=9
\end{array}$$

The composite projections are the concatenated columns of P, i.e., 56247 for j = 1, etc. Sorting the column indices by composite projections gives the column sequence (high to low),

3, 9, 4, 8, 1, 2, 6, 7, 5,

and the groups of identical composite projections form the index sets

$\underline{N}_1 = 3, 9$	$(n_1 = 2)$,
$\underline{N}_2 = 4, 8$	$(n_2 = 2)$,
$\underline{N}_3 = 1, 2, 6$	$(n_3 = 3)$,
$\underline{N}_4 = 7, 5$	$(n_4 = 2)$, L = 4.

The response treatment groups from (3) for $i = i_0$ (see SMX above) is

$$\underline{T}(i_0) \begin{cases} T_1: & 30,\ 90 \\ T_2: & 40,\ 80 \\ T_3: & 10,\ 20,\ 60 \\ T_4: & 70,\ 50 \end{cases}$$

APPENDIX B

DATA PARAMETER NOMENCLATURE

Symbol	Member of Matrix or Set:	Meaning	Page No. Defined
a	MCH, MD	Answer to query	61
a_{1t}, a_{2t}, etc.	MCH	Multiple choice record of query	61
ad	WC	Starting address of working core area	38
b_0	SDP	Lower limit, engineering units	27
b_1	SDP	Upper limit, engineering units	27
b_2	YDP	Lower limit, signal n	29
b_3	YDP	Upper limit, signal n	29
c_v	SV	Link address, matrix name to row definition	30
d_v	SV	Link address, matrix name to column definition	30
d_1	YDP	Low limit alarm switch	30
d_2	YDP	High limit alarm switch	30
e_v	SV	Link address, matrix name to data storage area	30
g	SDP	Lower boundary for classification	26
h	SDP	Upper boundary for classification	26
$i(a_0)$	YDP	Link address to offset a	29
$i(a_1)$	YDP	Link address to scale factor a_1	29
i(b)	Q	Link address to scheduled program	35

$i(k)$	YDP	Link address to lag constant k	29
$i(xv)$	SDP	Link address to the triplet $\{g, h, n_c\}$	26
$i(\pi)$	YDP	Link address to conversion suboperation	29
k		Lag constant	29
ℓ		Priority level	35
ℓ_0		Current priority level	36
ℓ	SDP	Number of digits to left of decimal or alpha for printout	26
ℓ_a	MD	Same as l for direct answer to query	62
ln	WC	Length of working core area	38
mch	MCH	Name of multiple choice query	61
md	MD	Name of direct answer query	62
m_v	SV	Row dimension of matrix	30
n	N	Raw digitized analog input	29
n_a	MCH	Number of choices	61
n_c	SDP	Number of classification intervals from g to h	26
n_p	p	Name of program	35
n_v	SV	Column dimension of matrix	30
p	WC, (Implicit in P)	Number of program in ad, ln area	39
$p_0(\ell)$	(XPL)	Currently active program number	36
q_t	MCH, MD	Query text	61
r	SDP	Specifies whether data is floating point, integer, or alphanumeric and the number of digits to right of decimal for output of floating point data	
r_a	MD	Same as r for direct answer to query	62
s	SDP, RDP	Alphanumeric name of data unit	29

sc	Q	Current starting core location for n_p	35
tx	P	Time for next execution of n_p	35
$\underline{V}$	SV	Name of Matrix	30
w	YDP	Hardware address of n	29
xp	P	Number of program chained to program p	35
z_a	YDP	Alarm disposition code	29
τ	Q	Current relative entry to np	35
Δ	P	Time interval between executions of n_p	35
θ_v	SV	Currently active column number for matrix updating	30
λ_p	P	Priority level of np	35
ξ_v	SV	Compact abbreviation of the variable (row) definition of $\underline{V}$	30
Ψ	Q	Code for current state of n_p	35
Φ_v	SV	Code for data type: floating point, integer, alphanumeric, or parametric	30

VITA

Galen D. Stacy

Candidate for the Degree of

Master of Science

Thesis: MAN-MACHINE INTERACTION FOR INDUSTRIAL PROCESS ANALYSIS

Major Field: General Engineering

Biographical:

Personal Data: Born in Pittsburg, Kansas, September 19, 1934, the son of Mr. and Mrs. Carl E. Stacy.

Education: Graduated from Pittsburg High School, Pittsburg, Kansas, in 1952; received the Bachelor of Science degree with honors from Kansas State College of Pittsburg in 1956 with major in Physical Science and minor in Mathematics; graduated from U. S. Navy Russian language interpreter's school, Anacostia, D. C.; completed requirements for the Master of Science degree at Oklahoma State University in May, 1971.

Professional Experience: Junior and Senior Research Technician, Spencer Chemical Co., Summers of 1955 and 1956; Officer, U. S. Navy, 1956-1962, serving tours in industrial management, intelligence research, computer applications, and programming; instructor of computer programming, U. S. Department of Agriculture Graduate School, 1961-1962, abstractor and translator of Russian technical literature; Digital Systems Engineer and Project Engineer, Phillips Petroleum Co., 1962-1968; Manager of Systems Programming, Applied Automation, Inc., 1968-present.

Professional Organizations: Member, American Chemical Society, 1956-1965; member, Association for Computing Machinery, 1961-present.