THE UNIVERSITY OF MICHIGAN

Memorandum 24

RAMP Architecture in a Utility Calculator System

David L. Mills

CONCOMP: Research in Conversational Use of Computers
F.H. Westervelt, Project Director
ORA Project 07449

supported by:

ADVANCED RESEARCH PROJECTS AGENCY
DEPARTMENT OF DEFENSE
WASHINGTON, D.C.

CONTRACT NO. DA-49-083 OSA-3050
ARPA ORDER NO. 716

administered through:

OFFICE OF RESEARCH ADMINISTRATION ANN ARBOR

May 1969

RAMP Architecture in a Utility Calculator System

ABSTRACT

This report describes an experimental multi-user utility calculator system similar to PIL, BASIC and FOCAL, but implemented as a subsystem in RAMP, a multiprogramming system described elsewhere. The features of this system include text editing, statement interpretation and expression evaluation as in the other systems cited. In addition, this system provides the capability of multi-tasking at the source language level. Thus each user can specify a program structure consisting of a number of asynchronous tasks which interact with each other in interesting ways.

This report was prepared using FORMAT, a computer program in MTS, the Michigan Terminal System. This program is described in: Berns, G.M., Description of FORMAT, a Text-Processing Program, Comm. ACM, 12, 3 (March 1969), pp. 141-146. The text was entered to this program partly in punched-card form and partly directly from a typewriter terminal and was printed on an IBM 1403 printer equipped with a TN print train.

RAMP Architecture in a Utility Calculator System

TABLE OF CONTENTS

1. Introduction 1

2. System Architecture 3

3. Language Specifications 7

3.1 Expressions 7

3.2 Statements 10

4. Operating Conventions 14

5. References 16

Appendix A. Error Diagnostic Dictionary 17

Appendix B. A Recursive Program to Calculate Factorials .18

Appendix C. A Number-Cruncher to Calculate Transmission Line Characteristics 19

1. INTRODUCTION

As an experiment not only in an application of the techniques discussed in the various RAMP reports (see references at the end of this report) but in the utility of carrying the multiprogramming operations to the statement level in an algebraic programming system, a utility calculator similar to PIL, BASIC and FOCAL has been implemented around the basic RAMP supervisor. Besides being capable of sustaining several users simultaneously, the system allows each user to define and execute a number of asynchronous tasks which can interact with each other in interesting real-time fashions.

The system is written for an 8K PDP-8 with an additional teletypewriter interface. This provides a two-user system which can be expanded without reprogramming by adding more teletypewriter interfaces to serve as many users as reasonable within memory-size and response-time limitations. In the current implementation a memory space of about 7000 characters is available for storage of user programs and data. This space, shared among all users, is used for the lines of the user's stored programs, for the data structures created by the various tasks, and for miscellaneous buffers and control blocks which are allocated and deallocated as multiprogramming operations proceed.

The facilities available to each user include the usual arithmetic-expression interpretation common in other similar languages. Variables are structured as scalars or arrays of any dimensionality. The SET statement is used to assign the value of an expression to a variable and the TYPE statement is used to print the value of an expression. Conditional branching among the lines of a stored program can be specified with the GO and IF statements and the branch line numbers of these statements can be computed. Most of the common unary and binary operations and transcendental functions of one variable are implemented, as well as a complete set of I/O formatting and conversion functions.

The most significant departure from convention in this system is its multiprogramming capability. In the language in which the user writes his programs, a subroutine call (DO statement) is actually a task invocation. In addition, tasks may be invoked to list a user-defined stored program or to print the values assigned the currently allocated variables. The invoking task may be specified to proceed in parallel with the invoked task or may be specified to wait for it. Thus, it is possible to specify a number of "simultaneous" tasks which perform various functions and interact with each other in interesting ways.

Statements entered to the system via the teletypewriter keyboard are interpreted immediately in the order entered, while statements entered via the stored program are interpreted sequentially in the order of increasing line numbers, as modified by conditional branching statements. Each statement entered in either of these ways is treated as an independent task-scheduling operation, so that each user can be assured some processing even if another user hangs his stored program in a do-nothing loop.

Input/output queueing and line-editing operations are integral to this system in the same fashion as other RAMP systems and special device support can be added in the fashion popular with these systems. In particular, high-speed tape and data transmission equipment and display devices can be supported easily with only minimal reprogramming.

2. SYSTEM ARCHITECTURE

The supervisory sections of this system include the buffer management, storage allocation, task scheduling, device interface, I/O utilities and initialization segments of the basic RAMP system (see reference). The operator's console teletypewriter service routines are scaled down somewhat, since some of the features are unnecessary in this system. The command language interpreter is restructured so that commands (or statement lines as they are called in this system) can be entered either directly from the keyboard or indirectly from the stored program. The stored program is implemented as an internal file indexed by line number. It is maintained in packed format, two characters per PDP-8 word, and can be edited by line number in the conventional fashion as in other similar systems. Figure 1 shows the structure of a line of the stored program.

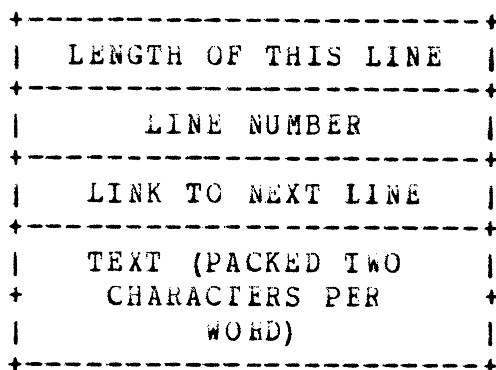

Figure 1. Stored Program Structure

A command task, called the real-time task, is associated with each teletypewriter console at system initialization. The parameters to this task include device table pointers for the command source and sink devices, control switches and dictionary pointers for the stored program and data structure associated with this task. Commands entered via a command source device can start other command tasks, called stored-program tasks, which take input from the stored program. Such tasks can be specified either to process the statements or to simply list the lines of the program together with their line numbers. The parameters to these tasks specify dictionary pointers for the data structures the task has created and the current line number.

A floating-point interpreter was coded specially for this system, since the DEC-supplied version was: a) too slow, b) too large, c) too inaccurate, d) could not operate

in an extended-memory environment. The current implementation includes recoded versions of the DEC transcendental functions and input-output conversion functions. The formats of floating-point instructions and operands are compatible with those of the DEC implementation.

The basic RAMP I/O input formatting utilities were modified for use in this system to provide interfaces to the floating-point input conversion and expression-scanner routines. The common interface to these routines provides a single subroutine call, with the exits providing information as to whether the input string was a constant, a letter string, or a special character. Constants are converted by this subroutine from the external character-string representation to the internal floating-point representation. Only the first and last letter in letter strings are retained and the resulting two-letter name may represent a command name, a variable or array name, or the name of an arithmetic function. An internal dictionary is associated with names in each of these classes and the entries in each class are assumed unique. The variable names created by each stored-program task in the system may either be associated with a dictionary unique to that task or may be associated with the dictionary belonging to the real-time task. The names of statements and arithmetic functions and the interpretation of special characters are associated with fixed dictionaries shared by all tasks (and therefore all users) in the system.

Variables and arrays are stored in this system as a data structure in the form of a downward-branching binary tree. At the apex of this tree and along the leftward-branching thread are the nodes associated with the variable names and their current values currently allocated to the task. A rightward-branching link to a node indicates a singly-subscripted reference (a vector), and the leftward-branching thread originating at that node contains the collection of all singly-subscripted elements currently defined together with their values. At each of these nodes in turn a rightward-branching link to a node indicates a doubly subscripted reference (a two-dimensional array) and the leftward-branching thread originating at the node contains the collection of all doubly subscripted elements, the first subscript of which is identified by the originating node. In this fashion a data-structure representing a collection of variable names each designating a scalar, vector, or array of indefinite dimensionality can be represented. A typical structure is diagrammed in Figure 2. Note that the subscripted elements of one dimension are disjoint from those of another, since they lie in different leftward-branching threads. Thus, no mapping function can

exist between, say, linear subscripts of one dimension and those of a higher dimension. Also note that a creation of an element of dimensionality n implies a creation of an element of all dimensionalities less than n.

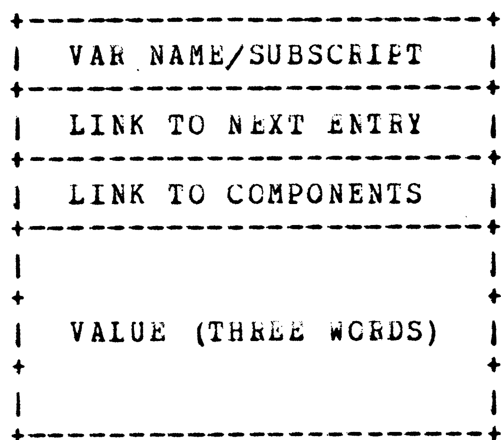

Figure 2. Data Structure

Integral to this system is an expression scanner modeled after that implemented in the MAD/1 compiler, as described in reference 3. This system uses a similar algorithm, but with the addition of several context-dependent transformations which provide machinery to parse exceptional syntactic cases. Unlike the MAD algorithm this version interprets the parse in real-time; that is, each atomic expression consisting of a single arithmetic operator and its operands is interpreted as it is identified in the left-to-right scan. Each call to the expression scanner causes the stack to be allocated (but not using the comparatively slow system storage-allocation facilities) and the value of the expression to be computed. These operations proceed to completion without involving a task-switching operation.

The basic statement scanner calls on the expression scanner to parse phrases of the form:

var(exp,exp,...)=exp,exp,...

In such cases the expression scanner is called one or more times to evaluate the subscript expressions indicated between the parens, and the indicated data structure is allocated and assigned to the particular task interpreting the phrase. Then the expressions after the equal sign are scanned each in turn and their values assigned to the structure at the indicated subscript element and to each

succeeding element (formed by incrementing the last subscript expression by one) in turn.

All operations involving program and data-structure storage make use of the system storage-allocation facilities. Storage is allocated to an element of a data structure only when necessary to store a value. Storage assigned a data structure element is deallocated when the allocating task terminates or by an explicit statement. Storage allocated the lines of the program is allocated and deallocated as line-editing operations require, but may be completely deallocated at any time by an explicit statement.

3. LANGUAGE SPECIFICATIONS

Following is a rudimentary "user's guide" to the language specifications and operation of the system. Although the facilities offered in this implementation are certainly adequate for the purposes here, namely to explore the multiprogramming utility, a more "userized" system would include several additional bells and whistles.

3.1 Expressions

A variable name is represented by a single letter followed optionally by either a letter or a digit. More than one letter or digit may follow the initial letter but in such cases only the last of these is significant. Variables are used in arithmetic, line number, and subscript expressions, and as task variables (see below). A variable is represented internally as a floating-point number with a value between about 10^{617} and 10^{-617} to a precision of about seven digits. An array name consists of a variable name followed by a list of subscript expressions separated by commas and enclosed in parentheses. The interpretation of variables and arrays corresponds to the usual MAD or FORTRAN interpretation. Note that, however, no explicit mappings between multi-dimensional notations are implied. That is, the elements at A(i) (i=1,2,...) and A(i,j) (i,j=1,2,...) are always disjoint and no mapping function can be defined between them. The dimension of an array name is the number of such expressions and no bound is placed on the number.

A constant is represented in conventional floating-point format and may take on values from 10^{-617} to 10^{617}. The external representation of the constant may contain any number of digits before or after the decimal point and the exponent may be of any value. However, the value of the constant, when converted to its internal representation, must lie within the specified limits and will be retained to only about seven digits in precision. Constants are used in arithmetic, line number, and subscript expressions, and the same conversion algorithm is applied in all cases.

An expression is represented in the usual manner as a list of variables, constants, operators, and punctuation marks. The permissible binary operators include "+" (addition), "-" (subtraction), "*" (multiplication), "/" (division), and "↑" (exponentiation). The unary "+" (no-operation), and "-" (negation) operators are differentiated from their binary counterparts in context. Certain special and transcendental functions of one variable are also included. These are summarized as follows:

Name	Function	Notation
ABS	absolute value	$\|x\|$
ATN	arctangent	$\tan^{-1}(x)$
COS	cosine	$\cos(x)$
EXP	exponential (base e)	e^{x}
ITR	extract integral part	entier(x)
LOG	logarithm (base e)	$\ln(x)$
NEG	negation	$-\|x\|$
SIG	extract sign part	sign(x)
SIN	sine	$\sin(x)$
SQT	square root	$\sqrt{x}$

Operator-precedence techniques (see references) are used in the parsing of the expression; and, in fact, the parser algorithm itself is a scaled-down version of that used in MAD/1 (see references) and the PDP-8 assembler resident in MTS. In this version, however, only a single precedence function is used, rather than two as in the other systems, so the parser really has more of the characteristics of the MAD compiler for the 7090. The precedence of the various operators is as follows:

unary +	6
functions: ABS, etc.	5
↑	4
unary -	3
* /	2
binary + -	1
punctuation: (,) etc.	0

Error recovery is facilitated by comparing the class codes assigned each identifier in context, a procedure used extensively by the MAD/1 compiler. It is easily shown that these error checks are exhaustive.

3. Language Specifications

The validity of all arithmetic operations is checked at each step for overflows and underflows and, in addition, the following special tests are performed:

1. In division a divisor of zero is an error.

2. In the LOG function an argument less than or equal to zero is an error.

3. In the EXP function an argument greater than about 100 is an error.

4. In the SQT function an argument less than zero is an error.

5. The exponentiation operation " " involves calls on both the LOG and EXP functions using the identity A B = EXP(B*LOG(A)), so that operand errors may be found in these functions.

6. In several functions a conversion from floating-point to integer representation is required. In such conversions an integer representation greater than 2^{23} is an error.

The syntax of each expression is checked for errors at several points. These checks insure that the expression is well-formed, that the various symbols are defined, and that any array structures referenced in fact exist. In addition, certain types of expressions are checked for validity within certain statements such as whether the expression is valid when used in real-time, as against stored-program, mode.

If either an arithmetic or syntax error is found in a statement, interpretation of that statement is immediately terminated and a diagnostic message is printed on the operating console. This message takes one of the following forms:

xxxx FLOATING-POINT TRAP

xxxx INVALID SYNTAX

xxxx INVALID STATEMENT

xxxx INSUFFICIENT STORAGE

where xxxx is a four-octal-digit number identifying the location in the system at which the error was found. If interpretation was from a stored program, then the line number in which the error was found is printed. If an error is found, interpretation immediately stops and, if in

stored-program mode, the task is terminated. A list of errors and their current error codes can be found in Appendix A.

Expressions may be of three types, depending upon context; arithmetic, line number, and subscript. An arithmetic expression, representing a value of type real in the usual sense, is used wherever a reference to a program variable is intended and may take on positive and negative values of magnitudes from about 10^{-617} to 10^{617} to a precision of about seven decimal digits. A line number expression, representing a value used as a line number in the stored program, is used as an operand of the IF, GO, and other statements, and may take on positive integral values from zero to 4095. A subscript expression, representing a value of a subscript in an array name, is used in the construction of array names and may take on positive integral values from zero to 4095. Unless used as the first operand (left side of the assignment symbol "=") of a SET, DO, or LIST statement, all symbols and variable names in any expression of any type must be predefined; that is, must have occurred on the left side of a previous SET, DO, or LIST statement.

3.2 Statements

A _program_ is a sequence of _statements_ identified by _line number._ Each statement type is identified by name (see below) and consists of a sequence of arithmetic and line number expressions. Each line number is a positive integer of value less than 4096. Additions and deletions of statements in a program are done on a line-by-line basis at any time, whether tasks are running or not, and can be initiated either directly from the operating console or by the program itself. Statements are provided for these purposes as well as those to list portions of the program and to interpret such portions as commands.

A task is explicitly initiated by the interpretation of the appropriate statement and is terminated under one of several conditions. The program itself is unique to each user in the system and is not affected by the initiation and termination of the various tasks. At all times in which the user's operating console is active a _real-time task_ is operable in the system. This task operates with the console keyboard as its input. No line number is associated with this operation, so that the GO and IF statements are invalid when interpreted by the real-time task. Dependent tasks created by the real-time task are called _stored-program tasks_ and these operate with the program itself as input. By convention, tasks initiated by the real-time task are presumed to operate simultaneously, so that real-time

interaction is possible among them. On the other hand, a task initiated by a stored program task is presumed to continue until completion before the invoking task is resumed. In this sense the stored-program invocation is really a simple recursive subroutine call.

Each task invocation specifies a line number of the program at which interpretation is to begin, following which the program is executed by the task line-by-line as specified. Associated with each task is a variable name which is assigned a value equal to the current line number of the active task. The value of this task variable can be changed either by the task itself by means of the branch statements (GO and IF) or by other tasks. Thus the task variable assigned each task represents a kind of instruction location counter in a peculiar "central-processing-unit," and can be accessed and modified by other tasks running in the multiprogramming mode. By convention, if the value of the task variable exceeds that of the highest line number currently assigned in the program, execution of the task ceases. In all other cases, a branch to a program at any line number will cause the task to begin execution at the line number with a value at least that of the branch number. At any place in a program in which a line number is expected, the value can be generated by a variable, constant, or expression, so that branch points can be computed.

Each line of the program, each task and each of its allocated variables requires storage space in the machine and this space is hared among all users of the system (currently two). In order to conserve storage space on one hand and yet allow a reasonably unrestricted use of the available space on the other, the following algorithm is used:

1. Variables are associated with the task that created them and are released when that task terminates or by an explicit statement.

2. The program itself is common to all tasks and can be erased at any time by any task.

3. A daughter task may access the variables of its mother task, those of the mother of its mother, and so forth.

4. Except for (3) above, a task has access to no variables which it has not created.

These conventions imply some rather interesting recovery procedures in pathological cases, as might be expected.

3. Language Specifications

There are some logically consistent situations which can occur when a task bombs (i.e., goes into a loop or traps) in which the only recovery possible is to erase the program. For instance, consider the case where task A calls task B which in turn calls task C using a task variable I for C which is not known to A. If task C bombs, then it cannot be stopped or restarted from A, since its task variable is not known to A. In general, then, it seems to be good programming form to use task variables which are known by the real-time task and therefore to all the daughter tasks invoked by the user.

The following statements are recognized in the current implementation:

TYPE exp1,exp2,...,expn

Type in floating-point format the values of exp1,exp2,...,expn. A character string enclosed in quotation marks (") may be used in place of a comma, in which case the string is printed at the indicated place.

SET var=exp1,exp2,...,expn

Evaluate the subscript expressions (if any) of var. Then evaluate each of the expressions exp1,exp2,...,expn in turn and substitute their values for the components of var in turn, starting with the indicated subscript of var. See text for variants of the SET statement.

GO exp

Branch to the statement number given as the value of exp.

IF exp,exp1,exp2,exp3

Evaluate exp and perform a three-way conditional branch as follows: If the value of exp is less than zero, equal to zero, or greater than zero, then branch to the statements whose line numbers are the values of exp1, exp2 or exp3 respectively.

DO var=exp

Evaluate the subscript expressions for var and assign this component of var as a task variable. Then assign the initial value of the task variable as the value of exp and begin interpretation at the corresponding line number.

END

Stop interpretation and return to the invoking task.

FORMAT exp1,exp2

Set format constants used in floating-point output conversion. The value of exp1 becomes the tab-stop value; each floating point output conversion is tabbed so that its first character is at a line position which is a multiple of this value. The value of exp2 becomes the number of significant (nonzero) digits retained during the output conversion process. (The full precision is retained during all internal operations however.)

ERASE

Erase all lines of the stored program.

LIST var=exp

Evaluate the subscript expressions for var and assign this component as a task variable. Then assign the initial value of the task variable as the value of exp and begin listing of the stored program at the corresponding line number.

DELETE var1,var2,...,varn

Deallocate the variable structures indicated by var1,var2,...,varn.

4. OPERATING CONVENTIONS

The console operating protocol and line-editing conventions are identical with those of the Data Concentrator and other related RAMP systems. In summary, the following characters are interpreted as line-editing functions:

Character	Function
RETURN	end of line
RUBOUT	line delete
Control-H (CAN)	character delete (backspace)
Control-E (ENQ)	attention
Control-Shift-P (NUL)	leader-trailer (ignored)

All 63 USASCII printing graphics available on the Teletype Models 33/35/37/38 except "@" are available to the system; all other characters are ignored. In the case of Model 37/38 the lower-case alphabetic characters are mapped into their upper-case equivalents. (This machine requires, of course, a different interface clock.)

All characters typed at the console keyboard are return-echoed to the console printer, but it is possible to enter lines to the system while the console printer is typing some other information. In such cases the return-echo line is delayed until the end of the current output line, and then is output to the printer. If the keyboard or echo buffers overflow, the return-echo process is suspended. If the keyboard buffer overflows (always due to an input line which is too long), then either a character-delete or a line-delete operation will clear the condition. If the echo buffer overflows, then simply waiting for current printer activity to cease will clear the condition.

Statements may be entered to the system in two modes: The real-time mode, in which statements are interpreted as entered from the keyboard, and the stored-program mode, in which statements are interpreted in sequence from the internal stored program. Since the real-time task is always active, statements can be entered to the system at any time, whether output is being produced on the console printer or not and regardless of the configuration of the operating tasks and their attached variables.

Statements are entered to the system from the stored program by creating a task using the DO statement. A task

created by either the DO or LIST statements can be stopped at any time by changing the task variable (using the SET statement) to a number higher than the highest line number currently assigned in the stored program. Statements in the two operating modes are structured alike except that the GO and IF statements are semantically invalid if entered in the real-time mode.

The stored program itself is created and modified using conventional procedures. A line to be entered to the stored program is prefixed by a line number in the range from zero to 4095 and a single blank and followed by the end-of-line (RETURN) code. A line of the stored program can be erased by entering the line number followed immediately by the end-of-line (RETURN) code. Lines may be entered to the stored program which, when themselves interpreted, will cause other lines to be modified by using the following rule: during interpretation, first the line number and one space is stripped from the interpreted line. Next the line-update operation is done, preserving any residual leading blanks. Finally the resultant line is treated as if it came from the real-time task. The entire stored program can be erased at any time and by any task using the ERASE statement.

A variable structure is created by the DO, LIST, SET, LOCAL and GLOBAL statements. When created by the LOCAL statement the variable is attached to the creating task, whether in real-time or stored-program mode. Thus the variable is known to all daughters of the creating task but none of its ancestral mother tasks. When created by the GLOBAL statement the variable is attached to the real-time task and is known to all tasks in the system. When created by the SET, DO or LIST statements, a search is made in the dictionary defining variables which belong to the creating task, then the mother of that task, then its mother, and so forth to the real-time task. If the variable name already exists in one of these dictionaries, then the reference in the SET, DO, or LIST statement is assumed to be to that instance, otherwise the variable is attached to the creating task.

The structure of the variable dictionaries in this fashion provides a parameterization facility in rather an interesting fashion and allows recursively structured stored programs with intercommunicating tasks as illustrated in Appendix B.

4. Operating Conventions

5. REFERENCES

1. Mills, D.L., RAMP: A Multiprogramming System for Real-Time Device Control, Concomp Project Memorandum 5, University of Michigan, May 1967.

2. Mills, D.L., I/O Extensions to RAMP, Concomp Project Memorandum 11, University of Michigan, October 1967.

3. Mills, D.L., The Syntactic Structure of MAD/I, Concomp Project Technical Report 7, University of Michigan, May 1968. Also in Procedings of International Seminar on Advanced Programming Systems, Hebrew University, Jerusalem, August 1968.

4. Mills, D.L., The Data Concentrator, Concomp Project Technical Report 8, University of Michigan, May 1968. Also in Procedings of University of Wisconsin Engineering Institute, December 1968, pp. 1-113.

5. Mills, D.L., Multiprogramming in a Small-Systems Environment, Concomp Project Technical Report 19, University of Michigan, May 1969. Also in Procedings of University of Michigan Engineering Summer Conference, June 1969.

6. Powers, V.M., Mills, D.L., and Laurance, N., An Assembly Language System for DEC Mini-Computers, Concomp Project Memorandum 20, University of Michigan, May 1969.

7.)u FOCAL Programming Manual, Form DEC-08-AJAC-D, Digital Equipment Corp., Maynard, Mass., 1968.

8.)u Pittsburgh Interpretive Language (PIL), in MTS - Michigan Terminal System, University of Michigan Computing Center, 1968.

9. Mills, D.L., and Powers, V.M., PDP-8 Program Relocation: Concepts and Facilities, Concomp Project Memorandum 17, University of Michigan February, 1968.

APPENDIX A. ERROR DIAGNOSTIC DICTIONARY

PDP-8 RAMP CONTROL SYSTEM (VERSION 58) 05-17-69

ASSEMBLY LISTING (ALL NUMBERS ARE OCTAL)

```
    PRINT ON,SHORT,NOREF
4017 XXX001 ERR001 INVALID VARIABLE ASSIGNMENT STATEMENT
5311 XXX018 ERR018 INTEGER OVERFLOW
5603 XXX002 ERR002 DIVIDE CHECK
6127 XXX003 ERR003 "SQT" ARGUMENT LESS THAN ZERO
6204 XXX004 ERR004 "LOG" ARGUMENT LESS THAN OR EQUAL TO ZERO
6337 XXX005 ERR005 "EXP" ARGUMENT TOO LARGE
5234 XXX006 ERR006 FLOATING-POINT UNDERFLOW
5243 XXX007 ERR007 FLOATING-POINT OVERFLOW
4473 XXX008 ERR008 INVALID OPERAND CONTEXT
4530 XXX009 ERR009 UNDEFINED SYMBOL
4517 XXX010 ERR010 INVALID BINARY OPERATOR CONTEXT
4561 XXX011 ERR011 INVALID UNARY OPERATOR OR "(" CONTEXT
4637 XXX012 ERR012 ILL-FORMED EXPRESSION
4727 XXX013 ERR013 STACK OVERFLOW
3747 XXX017 ERR017 INVALID BRANCH STATEMENT IN REAL-TIME TAS
4456 XXX014 ERR014 UNDEFINED SUBSCRIPT
3501 XXX021 ERR021 MESS ERROR.  IGNORE THIS COMMENT
3614 XXX024 ERR024 UNDEFINED STATEMENT TYPE
4140 XXX022 ERR022 VARIABLE DICTIONARY OVERFLOW
5333 XXX020 ERR020 INVALID LINE-NUMBER OR SUBSCRIPT EXPRESSI
3672 XXX019 ERR019 LINE DICTIONARY OVERFLOW
4133 XXX015 ERR015 UNDEFINED SUBSCRIPT
3663 XXX016 ERR016 INVALID LINE-NUMBER DELIMITER
4073 XXX023 ERR023 MISSING SUBSCRIPT EXPRESSION DEFINITION
4446 XXX025 ERR025 MISSING SUBSCRIPT EXPRESSION
4567 XXX026 ERR026 MISSING OPERAND OR EXPRESSION
```

APPENDIX B. A RECURSIVE PROGRAM TO CALCULATE FACTORIALS

Following is a listing and a sample run of a program to calculate the factorial of a positive integer using the algorithm:

```
F(N)=1; if N=1
F(N)=N*F(N-1); if N>1
```

Program:

```
100 IF N,,,130
110 GLOBAL F=1
120 END
130 LOCAL X=N
140 LOCAL N=X-1
150 DO X=100
160 GLOBAL F=F*(N+1)
161 TYPE N+1,F
170 END
```

Sample run:

```
FORMAT 4,6
SET N=10
DO X=0
1    1
2    2
3    6
4    24
5    120
6    720
7    5040
8    40320
9    362880
10   3628800
```

APPENDIX C. A NUMBER-CRUNCHER TO CALCULATE TRANSMISSION LINE CHARACTERISTICS

The following program computes a summary of the electrical characteristics of loaded and nonloaded telephone cables of the type typically used in exchange loops. Calculations using this program have been made yielding data useful for the design of leased-line connected remote job entry equipment. The theory on which these methods are based can be found in the references following this appendix.

```
* COMPUTE LINE PARAMETERS
100 SET S=D*SQT(F)
105 IF .2718*S-1,,,114
110 SET E=1
112 GO 140
114 IF S-11.083,,,130
115 SET E=1+.08*(.2718*S-1)↑2
120 GO 140
130 SET E=.096*S+.26
140 SET R=R0*E
150 SET L=L0/E
160 SET G=G0*(F/1000)↑1.3
170 SET C=C0
180 IF QQ,190,,190
182 TYPE F,D*SQT(F),R,L,G*1E6,C*1E6,SQT(L*C)*1E3
184 END

* COMPUTE CHARACTERISTIC IMPEDANCE AND PROPAGATION CONSTANT
190 SET W=2*3.141593*F
220 SET T1=R*R+W*L*W*L
230 SET T2=G*G+W*C*W*C
240 SET T3=ATN(W*L/R)
250 SET T4=ATN(W*C/G)
260 SET Z=SQT(SQT(T1/T2))
270 SET Z1=(T3-T4)/2
280 SET H=SQT(SQT(T1*T2))
290 SET H1=(T3+T4)/2
292 SET A=H*COS(H1)
294 SET B=H*SIN(H1)
296 IF QQ-3,,496,496
300 IF QQ-1,310,,310
302 TYPE F,Z*COS(Z1),Z*SIN(Z1),8.686*A,B,B/W*1E3
304 END

* COMPUTE TRANSMISSION CHARACTERISTICS
310 SET T1=-N*B
330 SET T2=EXP(-N*A)
340 SET T3=T2*COS(T1+Z1)
```

```
350 SET T4=T2*SIN(T1+Z1)
360 SET T5=T2*COS(T1)
370 SET T6=T2*SIN(T1)
380 SET U1=RL*(1+T5)+Z*(COS(Z1)-T3)
390 SET U2=RL*T6+Z*(SIN(Z1)-T4)
400 SET U3=RL*(1-T5)+Z*(COS(Z1)+T3)
410 SET U4=-RL*T6+Z*(SIN(Z1)+T4)
460 SET E=2*RL*Z*T2/SQT((U1*U1+U2*U2)*(U3*U3+U4*U4))
470 SET E1=Z1+T1-ATN(U2/U1)-ATN(U4/U3)
480 IF QQ-4,,492
490 IF QQ-2,496,,496
492 TYPE F,8.686*LOG(E),E1/W*1E3
494 END

* COMPUTE EQUIVALENT-TEE PARAMETERS
496 SET A=K*A
498 SET B=K*B
500 SET T1=EXP(-A)
510 SET T2=1-T1*COS(B)
520 SET T3=1+T1*COS(B)
530 SET T4=T1*SIN(B)
540 SET M1=Z*SQT((T2*T2+T4*T4)/(T3*T3+T4*T4))
550 SET L1=Z1+ATN(T4/T2)+ATN(T4/T3)
560 SET T2=EXP(-2*A)
562 SET T3=1-T2*COS(2*B)
564 SET T4=T2*SIN(2*B)
570 SET M2=2*Z*T1/SQT(T3*T3+T4*T4)
580 SET L2=Z1-B-ATN(T4/T3)
600 SET T1=RX/2+M1*COS(L1)
610 SET T2=W*LX/2+M1*SIN(L1)
620 SET M1=SQT(T1*T1+T2*T2)
630 SET L1=ATN(T2/T1)

* COMPUTE TEE IMPEDANCE AND PROPAGATION CONSTANT
650 SET T1=2*M2/M1
660 SET T2=L2-L1
670 SET T3=1+T1*COS(T2)
680 SET T4=T1*SIN(T2)
690 SET T5=SQT(SQT(T3*T3+T4*T4))
700 SET T6=ATN(T4/T3)/2
710 SET T6=T6+SIG(T3)*(1-2*SIG(T4))*3.1415923/2
740 SET U1=T5*COS(T6)+1
750 SET U2=T5*COS(T6)-1
770 SET U3=T5*SIN(T6)
772 SET Z=M1*T5
774 SET Z1=L1+T6
780 SET A=LOG(SQT((U1*U1+U3*U3)/(U2*U2+U3*U3)))
790 SET B=ATN(U3/U1)-ATN(U3/U2)
800 SET B=B+(SIG(U1)-SIG(U2))*(1-2*SIG(U3))*3.1415923
810 IF QQ-3,302,302,310

* MAIN PROGRAM
```

```
1000 SET K=1
1010 SET D=.03589
1012 SET R0=.1095/(D*D)
1014 SET L0=.001
1020 SET G0=1E-6
1030 SET C0=.061E-6
1032 SET N=1
1040 SET RL=600
1042 SET RX=7.6
1044 SET LX=.088
1046 FORMAT 10,4
1047 TYPE K,D,L0,G0,C0
1048 TYPE N,RL,RX,LX
1050  SET P(1)=50,100,200,500,1E3,2E3,5E3,7E3,1E4,1.5E4
1052 TYPE "TRANSMISSION-LINE CHARACTERISTICS"
1054 SET QQ=0
1056 IF QQ-5,,1150
1058 TYPE "SET "QQ
1060 SET I=1
1070 IF I-10,,,1120
1080 SET F=P(I)
1090 DO X=100
1100 SET I=I+1
1110 GO 1070
1120 SET QQ=QQ+1
1140 GO 1056
1150 END
```

REFERENCES FOR APPENDIX C

1. Hinderliter, R.G., _Transmission Characteristics of Bell System Subscriber Loop Plant,_ A.I.E.E. Trans. On Communications and Electronics, (September 1963), p. 464.

2. Johnson, W.C., "Transmission Lines and Networks," McGraw-Hill, New York, 1950, 361 pp.

3. "Reference Data for Radio Engineers," Westman, H.P. (Ed.), International Telephone and Telegraph Corp., 1959, 1121 pp.

4. Shaw, T., _The Evolution of Inductive Loading for Bell System Telephone Facilities,_ B.S.T.J., 30, 1951, p. 149.

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) THE UNIVERSITY OF MICHIGAN CONCOMP PROJECT	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP
3. REPORT TITLE RAMP Architecture in a Utility Calculator System	
4. DESCRIPTIVE NOTES (Type of report and inclusive dates) Memorandum 24	
5. AUTHOR(S) (First name, middle initial, last name) David L. Mills	
6. REPORT DATE May 1969	7a. TOTAL NO. OF PAGES 24 — 7b. NO. OF REFS 4
8a. CONTRACT OR GRANT NO. DA-49-083 OSA 3050 b. PROJECT NO. c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) Memorandum 24 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
10. DISTRIBUTION STATEMENT Qualified requesters may obtain copies of this report from DDC.	
11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Advanced Research Projects Agency

13. ABSTRACT

This report describes an experimental multi-user utility calculator system similar to PIL, BASIC, and FOCAL, but implemented as a subsystem in RAMP, a multiprogramming system described elsewhere. The features of this system include text editing, statement interpretation, and expression evaluation as in other systems cited. In addition, this system provides the capability of multitasking at the source language level. Thus each user can specify a program structure consisting of a number of asynchronous tasks which interact with each other in interesting ways.

UNCLASSIFIED
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
multiprogramming						
operator-precedence language						
text editor						
floating-point interpreter						
PDP-8						