AD-761 968

AN INTERACTIVE CONTINUOUS SIMULATION LANGUAGE

Russell Louis Hagen

Utah University

Prepared for:

Advanced Research Projects Agency

October 1968

PII Redacted

Technical Report 4-11

Russell Louis Hagen

AD 761968

1

AN INTERACTIVE CONTINUOUS SIMULATION LANGUAGE

October 1968

COMPUTER SCIENCE
Information Processing Systems
University of Utah
Salt Lake City, Utah

Advanced Research Projects Agency · Department of Defense · ARPA order 829
Program code number 6D30

47

ABSTRACT

During the recent history of computer science, a new class of programming languages has evolved. These languages are known as simulation languages. They were developed because of the great need to use simulation as a technique for problem solving and research. Computer simulation has come into increasingly widespread use to study the behavior of systems of which the state of the system changes over time. There have been two main types of simulation languages developed to study these systems, continuous simulation languages to study continuous change models, and discrete simulation languages for the analysis of discrete change models. The models used for analysis with a continuous simulation language are usually represented mathematically by differential or difference equations that describe rates of change of the variables over time. ICSL (Interactive Continuous Simulation Language) falls into this category as a programming language. ICSL not only has the capability for approximating the solutions of continuous change models, but also provides for interaction between man and machine during the course of the simulation. This interaction is in the area of computer graphics.

INTRODUCTION

ICSL is a FORTRAN V program for the digital simulation of continuous system models. A continuous simulation language such as ICSL touches into the realm of the analog computer programmer. Physical systems usually modeled and studied on an analog computer now are frequently simulated on a digital computer using one of the available simulation languages, or done first on the analog system and then checked on a digital system.

The usual method involved in doing a physical system analysis starts with the use of a block diagram of the system in terms of its separate functional components, or with a mathematical model in terms of differential equations. ICSL has a defined input language with which to describe the model and functional components of the system. Many problems faced by the analog computer programmer are nonexistent when the programmer turns to digital simulation as a means of analysis. These are problems involving time scaling, amplitude scaling, and of course the tedious job of wiring and potentiometer setting. Also for very large systems the analog computer programmer may need many more high gain amplifiers than he has at his facility, and without a digital system may be forced to abandon his project.

One basic criticism of the use of digital simulation by the use of the analog computer programmer is that using digital simulation techniques seem to "far remove" the programmer from the physical system he is studying, and he doesn't develop his normal intuitive feelings for the system. This criticism seems to come most from those who have been exposed to a continuous simulation language for the first time. There may be some merit to this criticism as complex problems can be described in ICSL with very little effort. The same problem if done conventionally on an analog system would entail much more involvement by the analog computer programmer and of course he would probably know the system he was studying much more thoroughly when completed.

This paper describes in detail the information required for programming in ICSL. The language was implemented almost entirely in the FORTRAN language to give it an air of machine independence [1]. It utilizes the existing facilities at the University of Utah graphics laboratory. The equipment in the graphics laboratory includes the following.

Access to the University of Utah's Univac 1108 computer.

A PDP-8 computer on-line to the Univac 1108 which controls the graphics equipment and serves an an information link between the two.

A model 35 teletype.

An IDI display scope.

These basic tools give the programmer the capability of interaction during an active user run. ICSL carries with it provisions for man machine interaction during the execution of a simulation program.

This capability is due to the work done in the area of interactive computer graphics at the University of Utah [2].

GENERAL DESCRIPTION OF ICSL

The ICSL system like other continuous simulation systems has a repertory of acceptable program statements with which to describe the input model and run conditions. These statements can be separated into two groups. First, those statements which describe the input, output, and execution control of the program, and second, those which describe the structure or configuration of the model to be simulated. These general classes of program statements are called EXECUTION CONTROL STATEMENTS and BLOCK EXPRESSIONS, respectively. The BLOCK EXPRESSIONS, or statements which describe the model, closely resemble FORTRAN arithmetic assignment statements [1]. FORTRAN was chosen because of its renown as a programming language. ICSL also carries another FORTRAN feature, that of the arithmetic statement function [1]. Functions are used widely in a simulation language and the statement function greatly facilitates function generation.

ICSL has a built in set of functions from which the components of a continuous system may be built. It also contains a large set of system error messages which when printed are excellent debugging aids for the user.

In ICSL, input and output is very easily accomplished. The programmer is free from format and data labeling responsibilities. The different types of output include a printer listing, printer plot, a plot by a digital plotter, and a visual graphical display. The routines for digital plotting are written for the CalComp Model 570 plotter [5].

The ICSL system also takes advantage of the existing facilities at the University of Utah graphics laboratory for man machine interaction during the course of the program execution [2].

Constants, parameters, and initial conditions for variables used in the simulation model can be described very easily. Also two types of functions can be described. The first type of function called TABLE FUNCTIONS are input in tabular form. These can be constant functions or parameter functions. Constant functions are functions which remain constant during the entire course of the simulation problem execution. Parameter functions have more than one set of tabular data and different sets are used for different phases of the problem execution. The second type of function is the forementioned ARITHMETIC STATEMENT FUNCTION feature from FORTRAN [1].

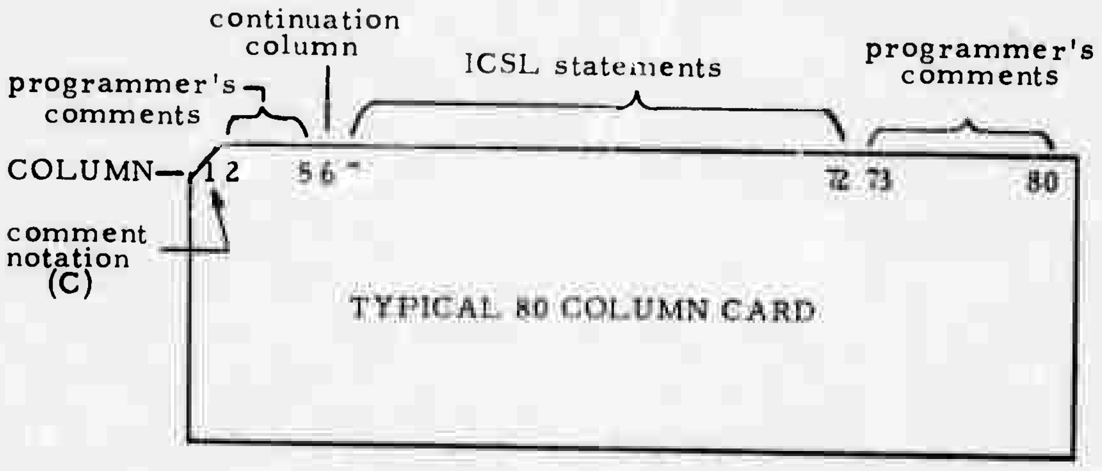
The programmer can also specify such things as integration step size, automatic statement sequencing, and other optional items which are all explained in detail in this paper.

A BLOCK EXPRESSION in the ICSL language is a FORTRAN arithmetic assignment statement used to describe the structure of the model being simulated [1]. These expressions can reference any of the SYSTEM FUNCTIONS, TABLE FUNCTIONS, or STATEMENT FUNCTIONS in the program. Since ICSL was implemented chiefly in FORTRAN, new function routines can be added to the system very easily, or old ones removed.

The basic elements from which to construct valid ICSL program statements include variable names, constants, functions, operators, and special reserved words.

The mode of inputting an ICSL program is via punched cards. Valid ICSL program statements are punched in columns 7-72. Columns 2-5 and 73-80 are ignored by the system and may contain any punched information the programmer wishes to appear on his printed output. Column 6 is reserved for continuation notation. Any ICSL statement may be continued on as many as five cards. Any non-blank character punched in column 6 of a card denotes that the card is a continuation of the preceding statement. . The basic format for punched cards is shown in Figure 1, below. . More than one statement may be punched on a card. If this is done, statements must be separated by a semicolon (12-6-8) punch). A "C" in column 1 indicates a comment card and the card is ignored during the compilation phase of the system and merely printed out.

Figure 1



If only one statement appears on a card, then the use of a semicolon to terminate the statement is not necessary, but may be used if desired. The basic structure of a program is shown in Figure 2.

Figure 2

```
----------------------------------
----------------------------------
----------------------------------    input/output
---- Any number of execution ------       and
---- control statements -----------   execution
----------------------------------    control
----------------------------------

BEGIN
----------------------------------
----------------------------------
------ Any number of block --------   model
------ expressions ----------------   description
----------------------------------

END
```

The block expressions of the program are delineated by a BEGIN END pair. The special statement BEGIN is used to separate the block expressions of the program from the execution control statements. The END statement is used to terminate the program. The statements of the program can otherwise be in any order with the exception that a function declaration must precede the description of all table functions (see page 17).

THE ICSL LANGUAGE

VARIABLE NAMES

A variable name contains one to six alphanumeric characters. The first character must be alphabetic. Since all computations are done in the floating point (real) mode, all variables used by the programmer are considered to represent floating point (real values). Examples of variable names are:

```
X        X1DOT        I21        QL2
```

CONSTANTS

Constants may be written either in integer form or floating point form. Either form may be followed by an exponent denoted by the letter E followed by a signed or unsigned integer. The letter E denotes that the preceding constant is multiplied by the integer power of 10 which follows it. Examples of constants are:

```
-5.327        25        523E-6        .025E12
```

BLOCK EXPRESSIONS

A block expression as mentioned before strongly resembles a FORTRAN arithmetic assignment statement [1]. The general form of a block expression is:

```
V = EXPR
```

where V is a valid variable name and EXPR denotes an arithmetic expression. The expression may contain functions, constants, and other variable names. Also the operators +, -, *, /, and ** denote the same meaning as in FORTRAN (addition, subtraction, multiplication, division, and exponentiation, respectively). Any expression can be enclosed in parentheses to any depth, and function arguments are also enclosed in parentheses and separated by commas.

All functions with more than one argument cannot contain expressions as arguments, but they can contain variable names or constants. A function cannot contain another function as an argument.

Functions with a single argument can contain any expression as an argument as long as it does not contain another function.

As mentioned previously, any block expression can be continued on additional cards up to a limit of five. Examples of block expressions:

```
X= 2*(A**2 + B*(C + D))
ZSQ = SQRT(X**2 + W)
X = INT(XDOT, INIT)
AB1 = 3.0E-6 * (B+C) + 5E-2 *D/E
Y = FCNSW(VARI,25,.035,43E-8)
```

ICSL SYSTEM FUNCTIONS AVAILABLE

In a digital simulation language a large number of functions are needed to facilitate operational elements similar to those of an analog computer. These include such items as integrators, function generators, pulse generators, etc. These items are called ICSL system functions. The language also has the standard FORTRAN library functions which can be used as aids in function generation. The system and library functions available are given in tables which follow.

The user may also write his own FORTRAN function subprograms for use by his ICSL programs. Two dummy functions, USER1 and USER2 are contained in the ICSL system program. If the user inputs a FORTRAN function subprogram with either of the above names, the corresponding dummy routine will be replaced by the user's routine, and the newly input routine can be referenced by any block expression in the ICSL program.

The routine USER1 expects five arguments, the first of which is determined by the ICSL system. The remaining four arguments are supplied by the user when referencing the function from a block expression. The reason the extra argument is added by the ICSL system, is that the user may wish to write his own function subprogram to implement an operational element. If this element was referenced by more than one block expression in the program, and the element had to save previously defined values in order to determine a correct output, then the user needs to have some method of determining which block expression is referencing the function subprogram each time it is called. A typical FORTRAN function subprogram declaration might be

```
FUNCTION USER1(I, X, Y, Z, W)
```

The value assigned to I will be 1 for the first block expression that references this function, 2 for the second, and etc. The values of X, Y, Z, and W will be supplied by the argument list of the function name USER1 in the ICSL program block expressions. An example might be:

```
Y = USER1(X1, X2, X3, X4)
```

The variables X1, X2, X3, and X4 of the above expression correspond to the variables X, R, Z, and W of the previous function subprogram declaration.

The routine USER2 expects only a single argument. This one argument is supplied by the user from a block expression. Examples are:

```
Y = USER2(X**2 + W)

W2 = USER2(A)
```

SYSTEM FUNCTIONS

NAME	GENERAL FORM	FUNCTION
INTEGRATOR	Y = INT(X, IC)	$Y = \int_0^t X\,dt + IC$ equivalent Laplace transform $1/s$
DERIVATIVE	Y = DER(X, IC)	$Y = \frac{dx}{dt}$ equivalent Laplace transform s
IMPLICIT FUNCTION	Y = IMP(X)	$Y = f(Y)$ $\lvert Y - f(Y)\rvert \leq 10^{-5}$
DEAD TIME (DELAY)	Y = DELAY(X, P)	$Y = X(t - P)$ $t \geq P$ $Y = 0$ $t < P$ equivalent Laplace transform e^{-ps}
ZERO-ORDER HOLD	Y = ZHOLD(X1, X2)	$Y = X2$ $X1 > 0$ Y = last ouput if $X1 \leq 0$ $Y(0) = 0$ equivalent Laplace transform $\frac{1}{s}(1-e^{st})$
MODE-CONTROLLED INTEGRATION	Y = MODINT(X1, X2, X3, IC)	$Y = \int_0^t X3\,dt + IC$ $X1 > 0$ $Y = IC$ $X1 \leq 0$ $X2 > 0$ Y = last ouput $X1 \leq 0$ $X2 \leq 0$ (continued)

SYSTEM FUNCTIONS (continued)

NAME	GENERAL FORM	FUNCTION
1st ORDER LAG (REAL POLE)	Y = REALPL (X, A, IC)	$A\dot{Y} + Y = X$ $Y(0) = IC$ equivalent Laplace transformation $\frac{1}{ps+1}$
LEAD-LAG	Y = LEDLAG (X, A, B)	$B\dot{Y} + Y = A\dot{X} + X$ equivalent Laplace transformation $\frac{AS+1}{BS+1}$
2nd ORDER LAG (COMPLEX POLE)	Y = CMPXPL (A,B,I,J,X) $Y(0) = I$ $\dot{Y}(0) = J$	$\ddot{Y} + 2AB\dot{Y} + B^2 Y = X$ equivalent Laplace transform: $\frac{1}{S^2 + 2ABS + B^2}$
FUNCTION SWITCH	Y = FCNSW (X_1, X_2, X_3, X_4)	$Y = X_2 \quad X_1 < 0$ $Y = X_3 \quad X_1 = 0$ $Y = X_4 \quad X_1 > 0$

FUNCTION GENERATORS

GENERAL FORM	FUNCTION
Y = FNAME(X)	FNAME IS THE NAME OF A TABLE FUNCTION WHICH IS IN THE PROGRAM. Y IS ASSIGNED THE VALUE THE FUNCTION ASSUMES AT THE POINT X.
Y = STEP(A, B, P, X)	$Y = A$ if $X < P$ $Y = B$ if $X \geq P$
Y = RAMP(IC,P,THETA,X) (Theta in degrees 0-90)	$Y = IC$ if $X < P$ $Y = (X-P)\tan\theta + IC \quad X \geq P$
Y = RAND(P) P any odd constant	RANDOM NUMBER GENERATOR (from 0 to 1.0)

LIBRARY FUNCTIONS

GENERAL FORM	FUNCTION
Y = EXP(X)	$y = e^x$
Y = ALOG(X)	$y = \ln(x)$
Y = ALOG10(X)	$y = \log_{10}(x)$
Y = ASIN(X)	$y = \arcsin(x)$
Y = ACOS(X)	$y = \arccos(x)$
Y = ATAN(X)	$y = \arctan(x)$
Y = SIN(X)	$y = \sin(x)$
Y = COS(X)	$y = \cos(x)$
Y = ABS(X)	$y = \|x\|$
Y = SORT(X)	$y = x^{1/2}$
Y = CBRT(X)	$y = x^{1/3}$

INPUT

Parameters, initial conditions, and constants can be described very easily in ICSL. A parameter is a variable name which is assigned more than one constant value. The use of parameters indicates that additional program executions are desired; one for each additional parameter value assigned. The general statement form for inputing these items is to assign constant values to a variable name. For example,

```
Y = 3.5
```

would define the value of 3.5 to be assigned as an initial condition or constant value of Y. If Y is recomputed by one of the block expressions of the program it is not a constant; otherwise, it is just assigned an initial condition. When inputing parameters the different parameter values are separated by commas. For example,

```
D12 = 4.6, 7.2, 8, 9.7, 10
```

The above statement would assign five parameter values to the variable D12. The maximum number of parameters given for any variable determine how many times the program will be executed. For the first execution the first parameter value is used, for the second the second is used, and etc. If for example the program is executing the nth time, then the nth parameter value of all program parameters is used. If a particular program parameter contains k parameter values, and $k < n$, then the kth one is used.

TABLE FUNCTION INPUT

Table functions are functions of a single variable the user wishes to reference from his program. Table functions are described similar to the way of constants and parameters. However, all table functions must be declared in a function declaration before they appear in the program. The general form of the function declaration is:

```
FUNCTION    F1, F2, F3, .......Fn
```

where F1, F2, F3, ...Fn are the alphanumeric names of the table functions to be input in the program.

Table functions are input by assigning a list of constant values to a function name. The general form for inputing table functions is:

```
ME = V1, V2, V3, .......Vn
```

where NAME is the alphanumeric name of the function and V1, V2, V3, ...Vn are constant values assigned to the function. The first value and every alternate or odd position value thereafter is assumed to be the values of the independent variable and these values must be in ascending order. The other, or even, position values are the functional values corresponding to each particular value of the independent variable to the left. For example:

```
FUNCTION   FUN, F21
FUN = 1, 1, 2, 4, 3, 9, 4, 16, 5, 25, 6, 36
F21  = 2.0, 0.0, 4.8, 8.6, 6, 8.2, 9.0, 7.2
```

would define the two table functions FUN and F21.

Parameter functions or functions containing more than one set of values can be input very easily also. Each set of parameter function values must be separated by a colon(:). Example:

```
FUNCTION   P
P = 2.5, 3, 3.5, 4, 4.5, 5:3.6, 2.0, 4.6, 7.0
```

The above function P is a table function with 2 sets of values. Since it has more than one tabular set of values it is a parameter function and for the first program execution the first declared set of values is used. For any additional executions the second set of data is used. A table function may contain a maximum of eight functional lists.

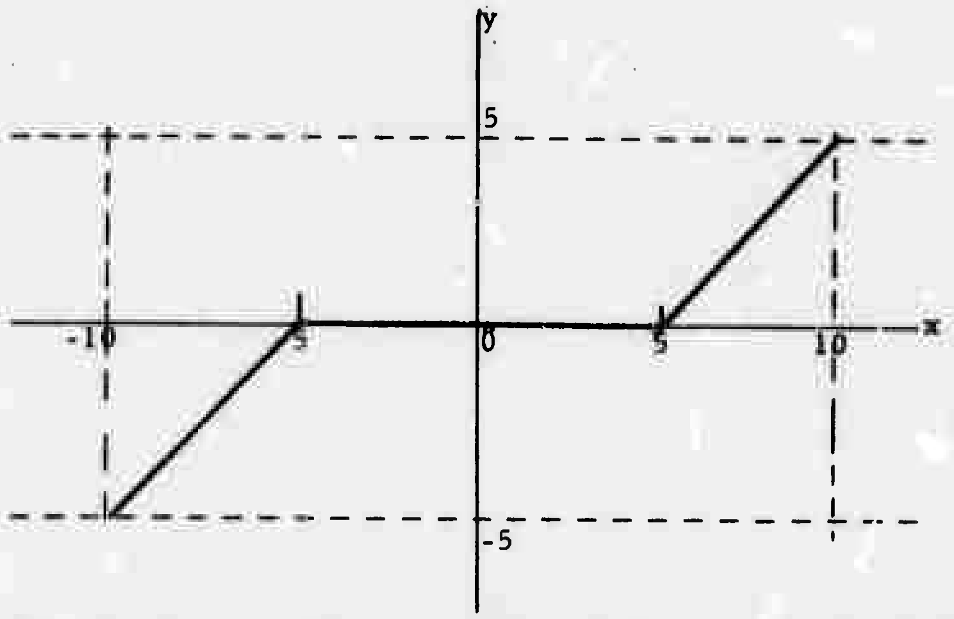
Any table function can be referenced by any block expression. For example:

```
Y = F1(X)
```

The above expression would assign to the variable Y the functional value the function F1 assumed at the point X. If the value of X is less than the minimum value of the independent variable of F1, then Y would be assigned the value the function assumed at the minimum value of the independent variable. If the value of X is greater than the maximum value of the independent variable, then Y would be assigned the value the function assumed at that maximum. Linear interpolation is used to evaluate all table functions. A more descriptive example of table function input would be as follows. Assume the following function was one of the inputs to be a functional component of the block diagram of a model to be simulated (Figure 3, page 19).

Figure 3

$y = f(x)$



This function could be described as follows in the execution control section of an ICSL program.

```
FUNCTION   F

F = -10, -5, -5, 0, 5, 0, 10, 5
```

If the above function was referenced by the block expression.

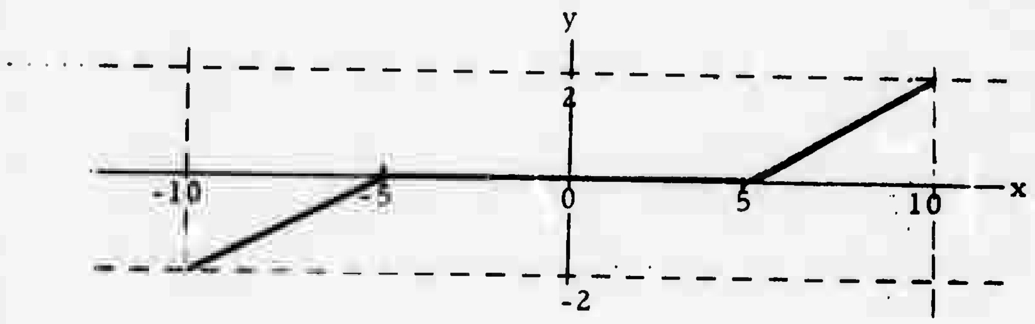
```
Y = F(X)
```

and the variable X had a value in the range $-5 \leq X \leq 5$, the corresponding value assigned to Y would be 0.0. If the variable X had a value in the range $-10 \leq X \leq -5$ then the corresponding value assigned to Y would be the value obtained using linear interpolation between the coordinates (-10, -5) and (-5, 0) to evaluate the function at the point X.

This table function can be made into a parameter function by the addition of another set of values. Suppose that a physical system was to be studied involving the function of Figure 3, but was to be analyzed for two cases. For the first case the function of Figure 3 was to be used, and for the second the function of Figure 4 was to be used.

Figure 4

$y = f(t)$



Both functions can be assigned by a single statement in ICSL as follows:

```
FUNCTION   F
F = -10, -5, -5, 0, 5, 0, 10, 5:-10, -2, -5, 0, 5, 0, 10, 2
```

The block expression

```
Y = F(-10)
```

would yield the value of -5 during the first program execution and -2 during the second and any additional executions.

STATEMENT FUNCTIONS

Statement functions are written as they are in FORTRAN [1]. The general form of a statement function is:

```
NAME(A1, A2, ... A5) = EXPR
```

where NAME is the statement function name being defined and A1, A2, ... A5 represent the dummy arguments of the function which are enclosed in parentheses. Every statement function defined must have at least one but a maximum of five dummy arguments. The dummy arguments must be valid variable names. EXPR is any valid arithmetic expression involving the dummy arguments. Statement functions can be referenced from any block expression. For example if the following statement function is defined:

```
ZEDL(A, B, C) = A*(B-C) + B**2
```

then one way the above function can be referenced from a block expression is as follows:

```
Y = ZEDL(2.0 X, W)
```

If the values of X and W were 3.0 and 2.0, respectively, then the value computed and assigned to the variable Y would be 11.0.

Two functional components of a block diagram, one which acted as an adder, and one which acted as a subtractor might be described by the following statement functions.

```
ADD(A, B) = A+B
SUB(X1, X2) = X1-X2
```

The following reference of the above functions by the block expression

Y = ADD(Y1, Y2) + SUB(Y1, Y2)

would yield the value of 10.0 if the values of Y1 and Y2 were 5.0 and 6.0, respectively, at the time this expression was computed.

OUTPUT

Output can be obtained in several modes. For a printer listing a LIST statement is required. Its general form is:

```
LIST      V1, V2, V3, . . . . . . . Vn
```

where V1, V2, V3, ... Vn are variable names used in the program. The variables are listed every dt units of time where dt is the independent variable step size.

A printer plot may be obtained by use of the PPLOT statement. Its general form is:

```
PPLOT      Vp, V2, V3, . . . . . . . Vn
```

where V1, V2 , V3, ... Vn are the variable names used in the program. The plot increment is also dt where dt is the independent variable step size.

A calcomp plot may be obtained by the use of the CPLOT statement. Its general form is:

```
CPLOT    V1, V2, V3, . . . . . . . Vn
```

where V1, V2, V3, ... Vn are variable names used in the program. The plot increment is the same as specified previously.

Output can also be displayed on the IDI scope by the use of a DISPLAY statement. Its general form is:

```
DISPLAY     V1, V2, V3, . . . . . . . Vn
```

where V1, V2, V3, ... Vn are the variable names whose values are to be displayed. The plot increment for each display is also dt where dt is the independent variable step size. All plot or display specifying statements will generate a single plot of each variable in the list for each program execution.

If the programmer is running more than one simulation on a particular model and he wishes to have a combined display or plot of how a single variable varies with time for more than one program execution, he may do so with the use of a MERGE statement. Its general form is:

$$\text{MERGE} \quad n \quad \left\{\begin{array}{l}\text{CPLOTS}\\ \text{DISPLAYS}\\ \text{PPLOTS}\end{array}\right\} \quad \text{OF} \quad V$$

where n specifies the number of program executions to be run before making a combined calcomp plot, printer plot, or display of how the variable V varies over time. Example:

```
MERGE   3   PPLOTS   OF   X1DOT
```

The above statement would cause the values assumed by the variable X1DOT to be saved and then plotted on the printer when the third program execution had been completed.

AUTOMATIC STATEMENT SEQUENCING

The block expressions of the program will be automatically sequenced by a sorting algorithm if the user includes a SORT statement in the program. Its general form is:

```
SORT
```

If the sort statement is not present in the program, the block expressions will be sequenced in the order they appear in the program. The statement sequencing algorithm considers that a block expression is ready to be sequenced next, if all of its inputs are available or have been previously defined. If the statement sequencing routine fails to sort the statements, the appropriate error message "SORT FAILURE" is printed on the user's program listing.

SPECIFYING CONTROL OVER THE INDEPENDENT VARIABLE

The independent variable can be given a name, an initial starting value, a final value, and a step increment value by the use of the STEP statement. Its general form is:

```
STEP      name = C1, C2, C3
```

where name is the name of the independent variable used for the program and C1, C2, C3 are constants representing the initial starting value, the final value, and the step size to be used, respectively. If a STEP statement does not appear in a program T is assumed as the name of the independent variable and 0.0, 5.0, 0.1 are assumed for C1, C2, and C3, respectively. For example:

```
STEP      TIME = 0,10,.2
```

The above statement would specify the following. First, that the variable name TIME was the name used in the program for the independent variable. Second, that the independent variable TIME would start at 0.0, and proceed in steps of 0.2 to the final value of 10.0 The independent variable step size determines the integration step size of all integrators in the program. Also the step size determines the plot and list increment of all output (page 22).

THE OPTION STATEMENT

The OPTION statement is used to input additional information describing the execution control of the program. Its general form is:

```
OPTION     O1,O2,.......On
```

where O1, O2, ...On can be any combination of the following options.

THE C OPTION

The C option is used to indicate that the user desires all plots specified in the program to be logarithmic plots and its form is C/Value where value is a constant representing the quiescent value to be used on the plot. If the C option is specified with no value, a value of zero is assumed.

THE T OPTION

The T option is used to control the time allotted for a program to execute. Its form is T/Value where value is a constant representing the maximum time in seconds to be used for the execution of the program. If the user specifies a maximum execution time with the T option, and the program exceeds this time, the execution of the program will be terminated.

FINDING MAXIMUM AND MINIMUM VALUES OF A VARIABLE

The maximum or minimum values a variable assumes during the course of the execution of a program can be found with a FIND MAX or FIND MIN statement. The general forms of these statements are:

```
FIND MAX        V1, V2, V3, .......Vn
FIND MIN        W1, W2, W3, .......Wn
```

Where V1, V2, V3, ...Vn and W1, W2, W3, ...Wn are variables used in the program. The maximum value of every variable appearing in a FIND MAX statement will be found and printed on the listed output of the program. The minimum value of every variable appearing in a FIND MIN statement will be found and also printed.

SPECIFYING THE INTEGRATION METHOD

The method of integration used for all integrators in the ICSL language may be arbitrarily chosen by the programmer with the use of the USE TYPE statement. Its general form is:

```
USE TYPE  n  INTEGRATION
```

where n is an integer constant and has the following meaning.

n	INTEGRATION METHOD
1	RECTANGULAR
2	TRAPEZOIDAL [4]
3	SIMPSONS [4]
4	MILNE 5th ORDER PREDICTOR-CORRECTOR [3]
5	RUNGE-KUTTA (4th ORDER) [4]

If no method is specified by the programmer, then the MILNE 5th ORDER PREDICTOR-CORRECTOR method is used. This method appears to be one of the best general numerical methods for the solution of ordinary differential equations.

MATHEMATICS FOR INTEGRATION METHODS

METHOD 1, RECTANGULAR INTEGRATION

$$Y_{t+\Delta t} = Y_t + \Delta t \cdot Y'_t$$

METHOD 2, TRAPEZOIDAL INTEGRATION

$$Y_{t+\Delta t} = Y_t + \frac{\Delta t}{2} \cdot (Y'_t + Y'_{t+\Delta t})$$

METHOD 3, SIMPSONS INTEGRATION

$$Y_{t+\Delta t} = Y_t + \Delta t \cdot (1/6 Y'_t + 4/6 Y'_{t+\frac{\Delta t}{2}} + 1/6 Y'_{t+\Delta t})$$

(continued)

METHOD 4, MILNE 5th ORDER PREDICTOR-CORRECTOR

$$\text{PREDICTOR } Y_{t+\Delta t} = Y_{t-3\Delta t} + \left(\frac{4 \cdot \Delta t}{3}\right)(2Y'_t - Y'_{t-\Delta t} + 2Y'_{t-\Delta t})$$

$$\text{CORRECTOR } Y_{t+\Delta t} = (Y_t + 7Y_{t-\Delta t})/8 + \Delta t(65Y'_{t+\Delta t} + 243Y'_t$$

$$+ 51Y'_{t-\Delta t} + Y'_{t-2\Delta t})/192$$

METHOD 5, RUNGE-KUTTA (4th ORDER)

$$Y_{t+\Delta t} = Y_t + (k_1 + 2K_2 + 2K_3 + K_4)1/6$$

$$K_1 = \Delta t \cdot f(t, Y_t)$$

$$K_2 = \Delta t \cdot f(t+\frac{\Delta t}{2}, Y_t + \frac{K1}{2})$$

$$K_3 = \Delta t \cdot f(t+\frac{\Delta t}{2}, Y_t + \frac{K2}{2})$$

$$K_4 = \Delta t \cdot f(t+\Delta t, Y_t + K_3)$$

All system integration routines are written so that centralized integration is performed. New routines can be added to the system very easily.

SWAPPING

Swapping may be arbitrarily specified by the use of the SWAP statement. Its general form is:

SWAP WHEN V = C

where V is a variable name in the program and C is a constant.

This statement has the following effect. When the variable V assumes a value greater than or equal to the value of the constant C during the execution of an ICSL program, then the program is transferred from memory to a reserved region on a FH-432 drum, and if a display has been specified will be shown on the IDI display scope. The user can then examine the display, and has the option of continuing the simulation program where execution was terminated, re-executing the last program execution, or terminating the program altogether. Also new values for any variable in the program may be input. These options are specified by the user via the model 33 teletype after swapping has occurred. This capability allows users to stop simulation programs at any point during the program execution, change any variables in the program, and then either continue or re-do the last program execution.

The commands on the model 33 teletype to specify these options are as follows:

```
RETURN
CONTIN
FINISH
```

The RETURN command brings the user's program back into memory and the last program execution is re-executed.

The CONTIN command also brings the user's program back into memory and the program begins where it left off at the time the swapping occurred.

The FINISH statement terminates the program.

The way new values are assigned to variable names is by putting a colon (:) after the command RETURN, or CONTIN, and then assigning a constant to each variable name to be overlayed. Example:

```
CONTIN:   A = 3.5, B = 7.2, C = 8.6
```

The above statement would first cause the values 3.5, 7.2, and 8.6 to replace the present values of A, B, and C in the program and then the program would continue executing where it left off when swapping occurred. If RETURN had been specified, the program would be re-executed for the last simulation. The return character on the teletype is used to terminate a command, and also is the signal that brings the user program back into memory after swapping has occurred. This must be the last character typed on any command given from the model 33 teletype.

SPECIFYING TITLES FOR OUTPUT

A title can be specified for any printed, plotted, or displayed output. The general form of specifying a title to appear with output is as follows:

$$\left\{\begin{array}{l}\text{LIST}\\ \text{CPLOT}\\ \text{PPLOT}\\ \text{DISPLAY}\end{array}\right\} \text{TITLE = '(any alphanumeric string)'}$$

As shown above, the user can specify any alphanumeric string enclosed in quote marks (4-8 punch), to be the title printed on a listing, printer plot, or calcomp plot, or the title displayed on the output of the IDI scope.

RESERVED WORDS

The ICSL language has the following reserved words which are not to be used as variable names in a program.

LIST	MERGE
PPLOT	SORT
CPLOT	STEP
SWAP	OPTION

EXAMPLE ICSL PROGRAM
MASS SPRING DAMPER SYSTEM

The approach to use in modeling a physical system using ICSL is to obtain a solution using integrators instead of differentiators. If the equations describing the model are known, the highest derivative of any variable should be expressed as a function of the lower derivatives and any forcing functions. Then the block diagram can be constructed using integrators as the operational elements.

The classical example of a continuous system is a mass spring damper system. The differential equation describing this system is:

$$\frac{w}{g}\ddot{X} + C\dot{X} + KX = f(t)$$

where $\frac{w}{g}$ is the mass of an object suspended by a spring with constant K, and C represents the amount of damping by a shock-absorber type damper.

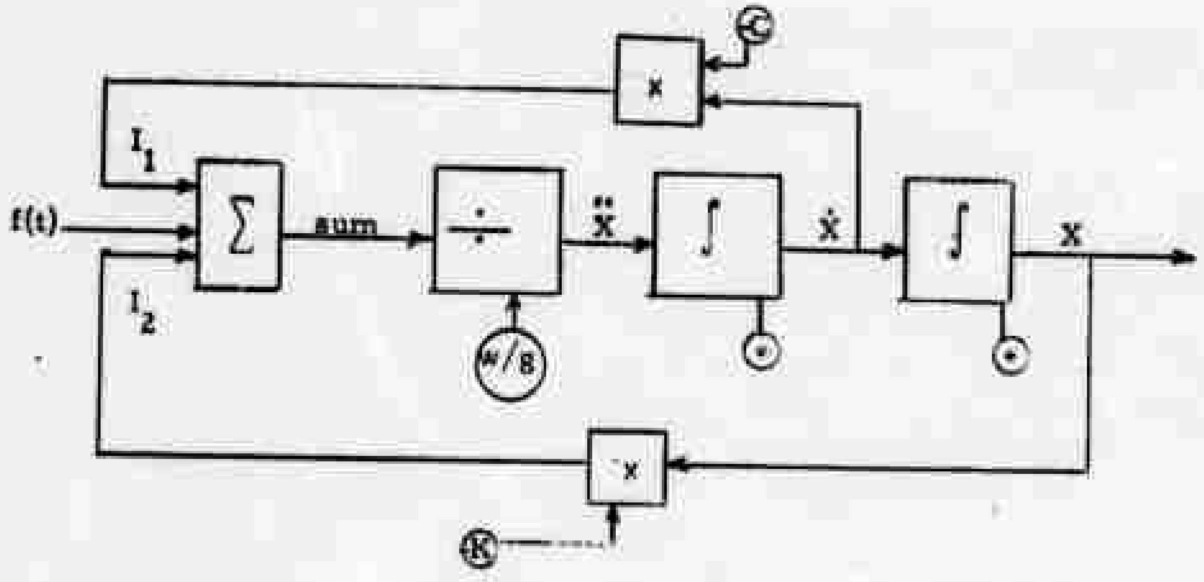
To study this system using ICSL one may choose the following method.

First rewrite the equations of the system in terms of the highest ordered derivatives of any variable. Doing this yields:

$$\frac{w}{g}\ddot{X} = -C\dot{X} - KX + f(t)$$

Next construct the block diagram of the system (see Figure 5, page 33).

Figure 5



Next the initial conditions for all integrators, or operational elements that require initial conditions must be determined. Assume that the system is at rest at time t=0 and the forcing function f(t) = Sin (t) is applied at time t=0, and w, g, and K are 32.0, 3.0, and 9.0, respectively. We may wish to study the behavior of the system for several values of C, say 8.0, 6.0, and 4.0. One ICSL program to do this system simulation would be as follows.

<u>line no.</u>

```
1       W = 32.0; G = 32.0; C = 8, 6, 4
2       K = 9.0; SORT
3       STEP TIME = 0.0, 20.0, 0.2
4       LIST TIME, X2DOT, XDOT, X
5       FIND MAX X;  FIND MIN X
```

(continued)

```
 6      OPTION  T/30
 7      LIST TITLE = 'MASS SPRING DAMPER SYSTEM'
 8      BEGIN
 9      I1 = -C*XDOT
10      I2 = -K*X
11      SUM = I1 + I2 + SIN(TIME)
12      X2DOT = SUM/(W/G)
13      XDOT = INT(X2DOT,0.0)
14      X = INT(XDOT,0)
15      END
```

DISCUSSION OF THE EXAMPLE PROGRAM

Line 1: Line 1 contains three statements. The first two assign the constant value of 32.0 to the variables W and G. The third statement of line 1 assigns three parameter values to the variable C. Since C is the only parameter of this program the model will be simulated three times; each time with a different value of C.

Line 2: Line 2 contains two statements; the first of which assigns a constant value of 9.0 to the variable K. The second statement is the SORT statement to cause the block expressions in the program to be automatically sequenced.

Line 3: Line 3 contains the STEP statement. This statement declares that TIME is the name of the independent variable. Also specified is that the independent variable will start at

0.0 and progress to the final value of 20.0 in steps of 0.2.

Line 4: Line 4 declares the names of variables of the program which are to be printed every 0.2 increments of the independent variable TIME.

Line 5: Line 5 contains two program statements; a FIND MAX statement, and a FIND MIN statement. Both statements have the variable X appearing in them. This will cause the maximum and minimum values the variable X assumes during each simulation to be printed out.

Line 6: Line 6 specifies the T option. If the program takes longer than 30 seconds to execute, execution will be terminated.

Line 7: Line 7 declares an alphanumeric title to be printed with the listed output.

Line 8: Line 8 contains the BEGIN statement and marks the end of the execution control section of the program and the beginning of the block expressions of the program.

Line 9-14: These are the block expressions of the program. The output of each block in the block diagram is written as a function of the inputs, to describe the model.

Line 15: The END statement signifies the end of the ICSL program.

Since no integration type was specified in the program, the Milne 5th order predictor-corrector method will be used by the system. A more compact way of writing the block expressions of this program is as follows.

```
BEGIN
X2DOT = (SIN(TIME)-C*XDOT-K*X)/(W/G)
XDOT = INT(X2DOT, 0.0)
X = INT(XDOT, 0)
END
```

In the partial program above, the intermediate variables I1, I2, and SUM were eliminated.

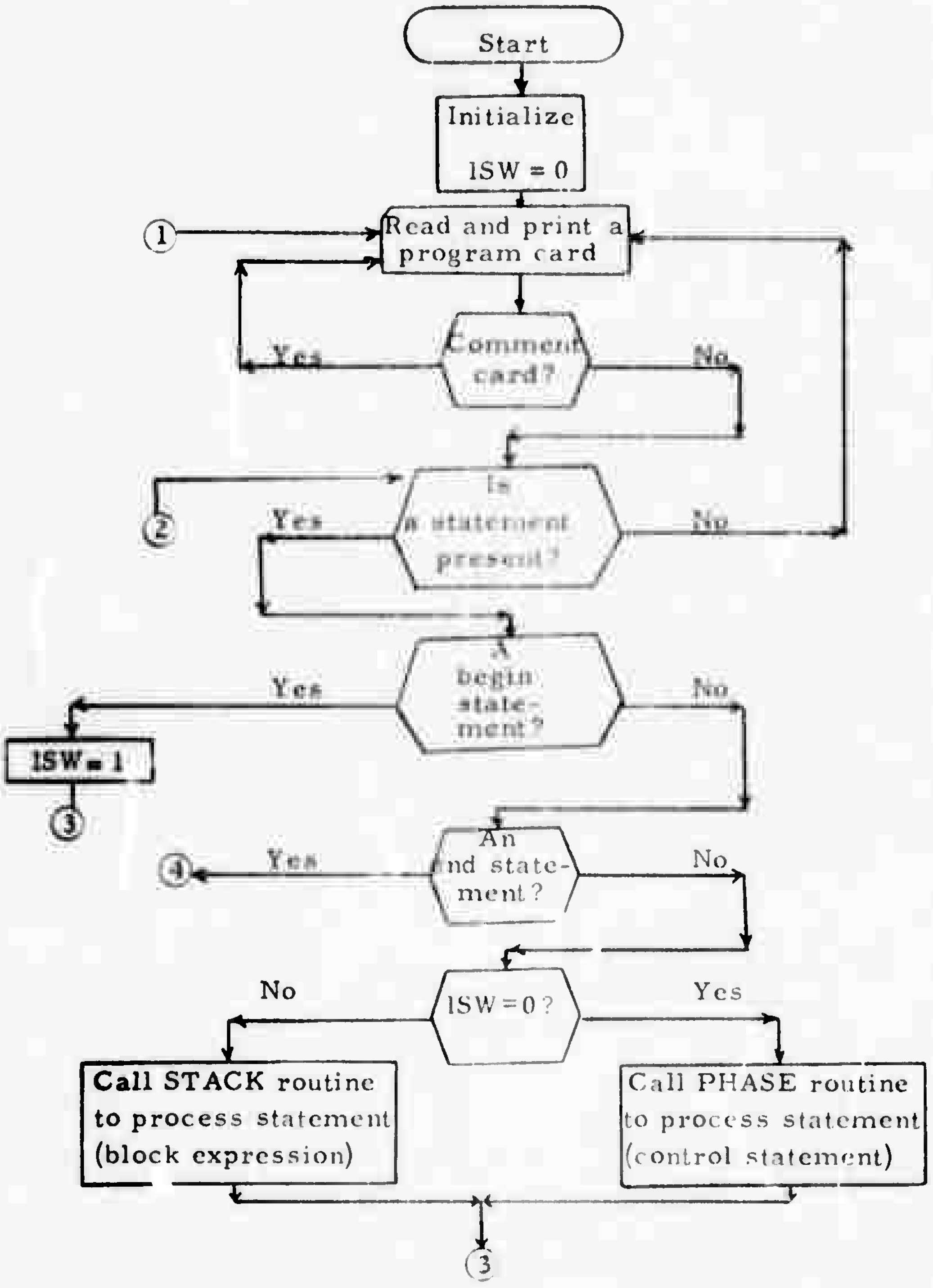
DESCRIPTION OF THE ICSL PROCESSOR

As mentioned at the beginning of this paper, the ICSL processor is almost entirely written in FORTRAN V. The processor in itself is complete with the exception of the present graphics software it utilizes for swapping and graphical display on the IDI scope, and for the specialized routines used for the CalComp model 570 plotter. In addition to these machine dependent features, the ICSL assembler generates executable code for the Univac 1108 Computer.

A continuous simulation language must approximate the solutions of ordinary differential equations, utilizing numerical methods. These methods usually entail a variety of iterative techniques, which when used effectively gively give satisfactory results to the user. Since most numerical methods require a large amount of iteration, it is generally desirable to generate efficient machine level code for those portions of the user program to be used repetitively. This greatly reduces the execution time needed for the solution approximation. It was for this reason, that 1108 code is generated for all block expressions of the user's program by the ICSL processor.

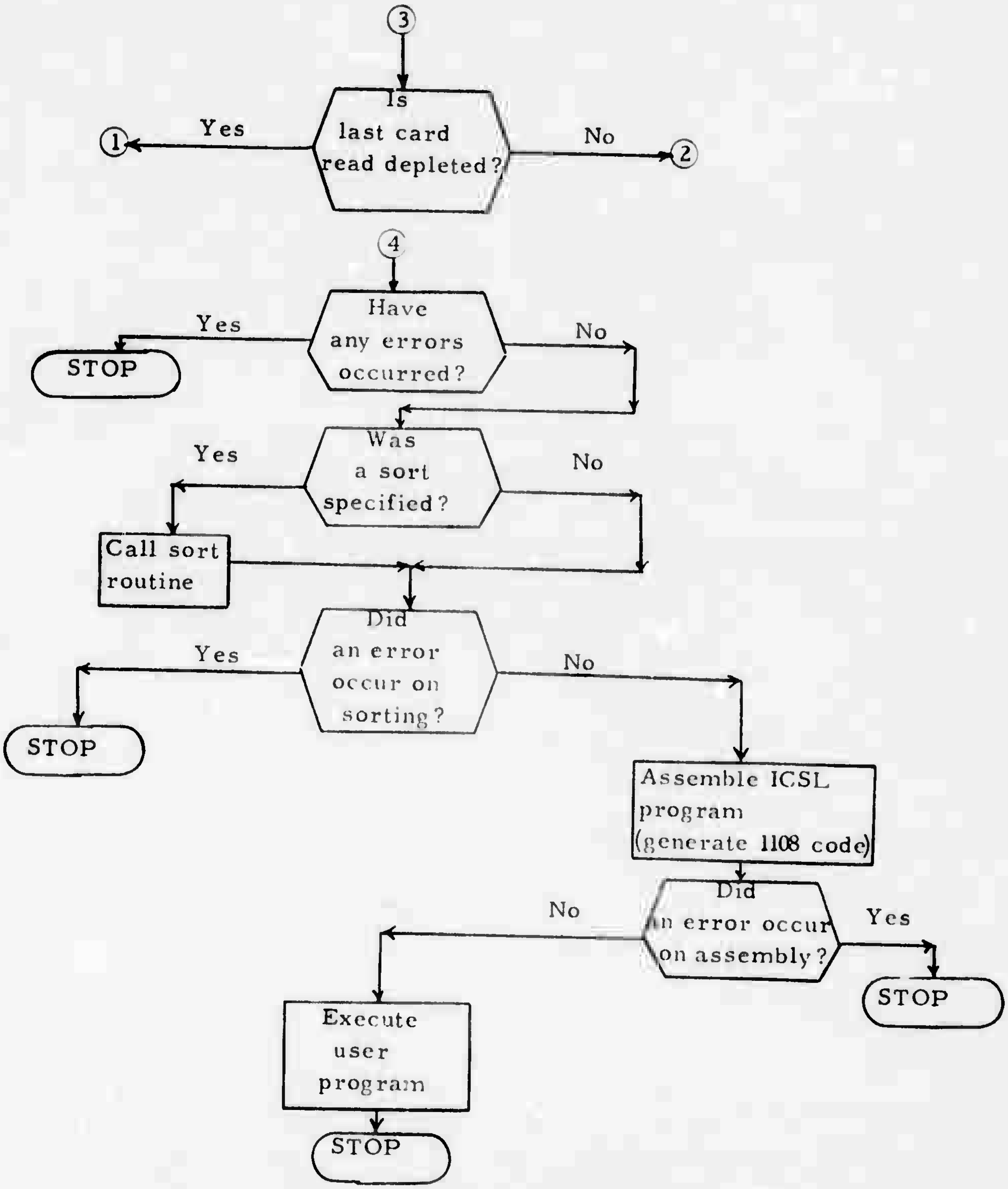
A generalized flowchart of how a user program is processed is shown in Figure 6, page 38.

Figure 6



Start
Initialize ISW = 0
1
Read and print a program card
Comment card?
Yes
No
2
Is a statement present?
Yes
No
A begin statement?
Yes
No
ISW = 1
3
4
An end statement?
Yes
No
ISW = 0 ?
No
Yes
Call STACK routine to process statement (block expression)
Call PHASE routine to process statement (control statement)
3

Figure 6 (continued)


3
Is last card read depleted?
Yes
1
No
2
4
Have any errors occurred?
Yes
STOP
No
Was a sort specified?
Yes
Call sort routine
No
Did an error occur on sorting?
Yes
STOP
No
Assemble ICSL program (generate 1108 code)
Did an error occur on assembly?
No
Yes
STOP
Execute user program
STOP

As depicted in the flowchart of Figure 6, a user program is read in the same manner of reading data from a FORTRAN program. The user's program cards are read sequentially and as statements are recognized, a check is made to determine if a BEGIN or END statement is present. If a BEGIN statement is recognized, then the variable ISW is assigned the value 1. This variable is used as a switch in the logical control of the program to call one of the two statement processing routines PHASE1 and STACK. When a begin statement is recognized, it is assumed that all program statements to follow will be block expressions, and the subroutine STACK is called to process any additional statements. Since there are two major categories of statement types in ICSL, it seems natural to portion the statement processing in this manner. Following the recognition of an END statement, if no errors in the user's program were detected during processing, the block expressions are sorted and 1108 code is generated and executed. A brief description of the main program routines follows.

THE SUBROUTINE PHASE1

This subroutine processes all control statements of the user's program. It recognized the reserved words LIST, PPLOT, STEP, SORT, etc., and stores information specifying the proper routines to be called during the execution of a user's program. Variable and function names are recognized and stored for reference from the block expressions of the user's program. Also all input from the user is stored sequentially for later use.

THE SUBROUTINE STACK

The subroutine STACK processes the block expressions of the user's program. The expressions are analyzed for programming errors such as illegal variable names, undefined function calls, incorrectly nested expressions, and etc. If no errors have occurred in the processing of any statements of the user's program, then the block expression will be rearranged into reverse or postfix polish notation. For example a typical block expression might be

$$Y = A*(B+C)$$

This expression when rearranged in the postfix notation would be

$$YABC+*=$$

This form is the final block expression form used by the ICSL assembler for the generation of 1108 code.

STATEMENT SORTING

The block expressions of a user's program are sorted by the subroutine SORTM. In using various numerical techniques for the approximation of solutions of differential equations, the equation must first be of the form

$$y' = f(y, t)$$

where t represents the independent variable of the function y. The sorting algorithm used isolates and defines $f(y, t)$ hopefully from the user's block expressions. The purpose of the statement sort is two fold. First, for the purpose mentioned before and second to sequence all other block expressions in a proper computational order. For example, the two block expressions

```
X = Y*THETA
Y = COS(TIME)
```

are not in the correct order if the variable X is computed before the variable Y. Since X is a function of Y besides being a function of THETA, the variable Y must be computed first to insure that X will be assigned the proper value for a certain value of the independent variable TIME.

THE ASSEMBLER

The subroutine ASSEM generates the 1108 machine code which is executed in the event no errors occur during assembly. The subroutine takes the sorted block expressions in postfix polish notation and generates the appropriate machine code for the computations of the expression. The expression

```
Y = X*(A+B)
```

would be represented in postfix notation as

```
YXAB+*=
```

The order of computation of the expression is defined by examining the expression from left to right until an operator is found. This operator then operates on the two preceding operands (in this case the variables A and B are operated upon by the addition operator +).

Upon completion of the operation the resultant operand is placed in the expression and the operator and two operands just combined are removed. This process is continued until the = (store) operation has been completed. The 1108 code which would be generated by the subroutine ASSEM for the last expression follows. The example code given will be given using the 1108 assembly language mnemonic names for ease of reading.

Code			Meaning
LA	A0,	B	Load arithmetic register A0 with B
FA	A0,	A	Execute a floating addition of A to the contents of register A0.
FM	A0,	X	Executve a floating multiply of X to the contents of register A0.
SA	A0,	Y	Store the contents of register A0 in Y.

The actual computational part of the user's program (the block expressions) then are used to generate 1108 machine code. This code is executed repetitively during the simulation process.

REFERENCES

1. Sperry Rand Corporation. UNIVAC Division, 1108 Multiprocessor System FORTRAN V, Programmer's Reference Manual, UP 4060, New York, c 1966.

2. Copeland, Lee and Carr, C. Stephen, Graphics System, Computer Science Information Systems, University of Utah, Salt Lake City, Technical Report 4-1, Nov. 15, 1967.

3. Milne, W. E. and Reynolds, R. R., Fifth-Order Methods for the Numerical Solution of Ordinary Differential Equations, 9, Journal of ACM, Jan., 1962.

4. Ralston, Anthony, A First Course in Numerical Analysis, McGraw-Hill Book Co., New York, c 1965.

5. Using the Digital Plotter, University of Utah Computer Center, July 1966.

VITA

Russell Louis Hagen [redacted]

[redacted] He moved at an early age to Ogden, Utah where he attended high school.

After finishing high school, he enrolled at Weber College in Ogden where he graduated following a two year program in Electrical Engineering.

After graduation he was employed by Hercules Powder Co. at Bacchus, Utah where he worked up to the time of his entrance into the United States Army.

His military obligation was satisfied at Fort Hayes in Columbus, Ohio where he also attended classes at the Ohio State University.

Following an honorable dischargd from the Army, he returned to Utah, this time to Salt Lake City where he enrolled at the University of Utah. He graduated in June of 1967 with the degree of Bachelor of Science in Computer Science. Upon graduation he began his graduate study.

PII Redacted