U N I V E R S I T Y O F M I C H I G A N

Memorandum 27

MOMS: MICHIGAN'S OWN MATHEMATICAL SYSTEM

Robert W. Taylor, Editor

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

April 1970

PREFACE

This report describes an interactive mathematical system with graphical input and output capabilities. The system was programmed during the winter of 1968 by the members of the advanced systems programming course, Computer and Communication Sciences 673, under the direction of Professors B. Arden, B. Galler, and L. Flanigan. It runs on an IBM 360/67 computer with 2250 display operating under MTS, the Michigan Terminal System.

Shortly after completion of the project, the 2250 display was removed from the Computing Center. This accounts both for the lack of photographs and the delay in publication.

The members of the class wish to thank the Department of Computer and Communication Sciences and the University of Michigan Computing Center for the support necessary to carry out this project.

Persons wishing further details should contact Professor Bernard Galler at the University of Michigan Computing Center.

List of Class Members of
Communication Sciences 673: System Programming

Neil J. Barta
Bruce J. Bolas
Ronald F. Brender
Michael S. Feldberg
Daniel R. Frantz
Ross H. Hieber
Charles G. Moore
Robert E. Nicholls
Norman L. Schryer
Frances Stephenson
John S. Tripp
Robert L. Feldman
Jay A. Jonekait
Richard W. McHard
Ronald J. Srodawa
Robert W. Taylor
Pertr H. Wilcox

TABLE OF CONTENTS

Table of Contents, continued

1. INTRODUCTION

During the past several years, online mathematical analysis has received increasing interest [1,3]. These systems have been interpretive in nature, allowing a highly conversational approach to problem solving, and their usefulness in attacking complex mathematical problems has been demonstrated [2]. As stated in [1], these systems are generally characterized by a keyboard or pushbutton form of input and a graphical form of output. The output is sometimes also available in printed form. More significant is the fact that all systems allow a variety of data types, often including matrices, allowing a wide range of problems to be attacked. Finally, the languages employed by users of the systems are simple to learn but have a definitional facility which enhances flexibility.

During the 1968 winter term, the Computer and Communication Sciences 673 course, in order to understand more fully the internal workings of such a system, decided to build one of these systems as a class term-project. Michigan's Own Mathematical System—MOMS—is the result. It runs under the Michigan Terminal System, MTS, on an IBM 360/67, using an IBM model 2250 display console for both input and output. Because the project was to be completed in approximately fifteen weeks, it was necessary to restrict somewhat the allowable data types and certain other special

features. Nevertheless, a large variety of problems may still be attacked using MOMS.

2. HOW TO USE MOMS

A user at a 2250 console may use the MOMS system by issuing the MTS command

```
$RUN *MOMS
```

The MTS temporary file -MESSFILE will be created, and the user may examine this file to obtain a history of all error messages provided during his run.

When execution begins, the 2250 screen will appear as shown in Figure 1.1. Various words--the light buttons--appear along the vertical edges of the screen. These light buttons define the various commands that the user may issue to the MOMS system, and are analogous to the "function keyboard" of various other systems [3]. At the bottom of the screen is the so-called Virtual Keyboard. This contains the buttons A through Z, which may be used to store data, as well as the predefined buttons 0 through 9, and various other commonly used system buttons. Just above the Virtual Keyboard is the Echo Line. A recent history of various light buttons that the user has pushed will appear here. The rest of the 2250 screen is used to display the results of computations, both in graphical and

alphanumeric form. Messages to the user are displayed in the lower portion of this area.

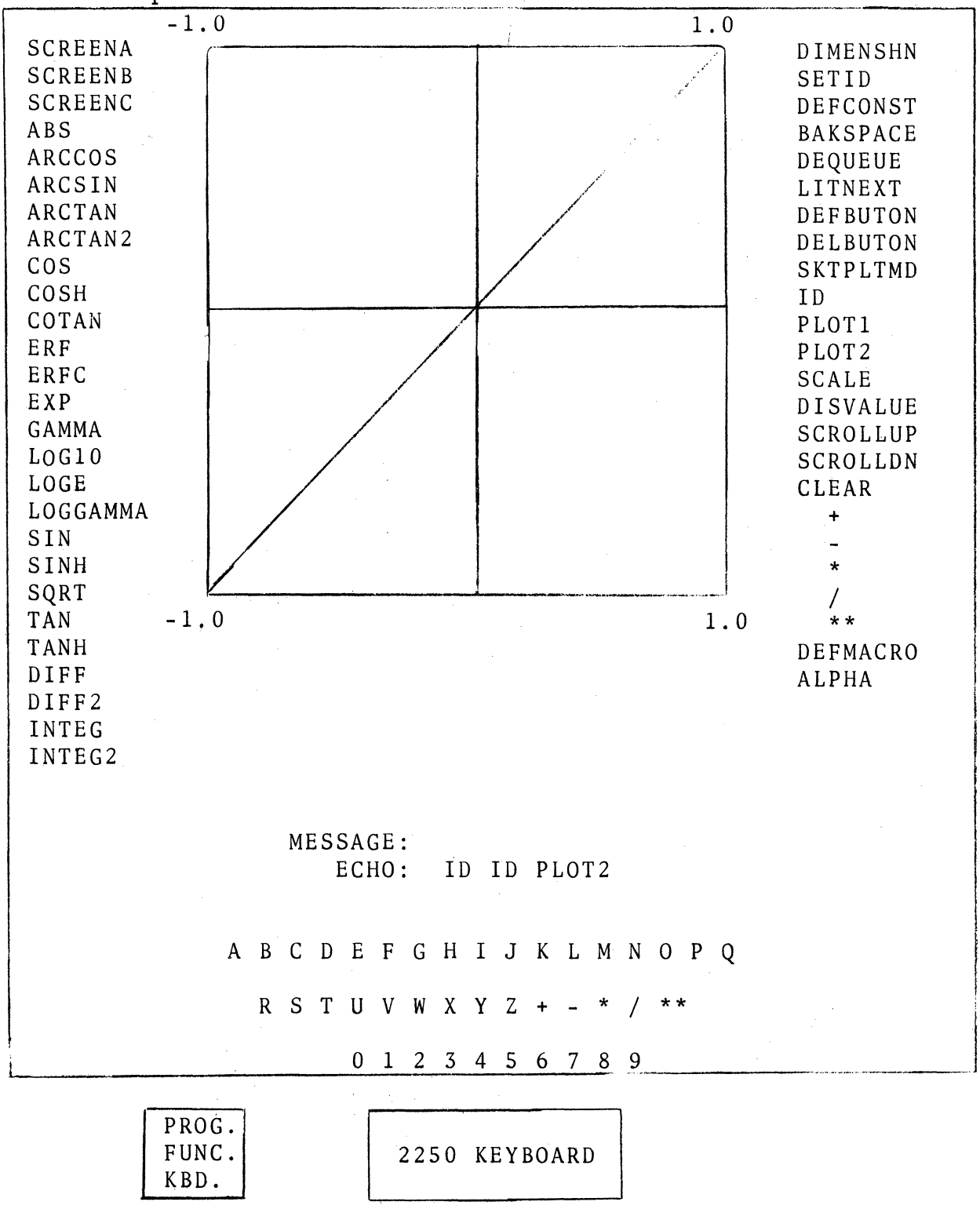


Figure 1.1

The user makes input statements to the MOMS system by pointing at various light buttons with the light pen. A result of a pointing sequence will appear on the Echo Line. The syntax of the input statements is operator postfix notation. Nesting is allowed. Thus, for example,

A B +

A B C + +

B B * 4 A C * * - D =

will add A to B ; add A,B, and C ; compute B^2-4AC and store it under button D , respectively. The user may edit his input statement using system operators and eventually request interpretation of the statement and display of results, either in graphical or alphanumeric form. Various declarative and definitional capabilities are provided to enchance the flexibility of the system. These are discussed in detail in Section 2.2.

2.1 DATA TYPES IN MOMS

The current version of the MOMS system has four data types: scalars (real*4), vectors (real*4, dimension=101), vector-pairs (two vectors concatenated), and macros. Provision was made in the design of the system for complex scalars, complex vectors, and complex vector-pairs (as well as real*8 and integer modes), but they are not yet

legal data types. The dimension of 101 for vectors is a default case and may be changed by the user using the DIMENSHN button as in Section 2.2.1.

There are currently several restrictions concerning the mode combinations acceptable to the various mathematical and arithmetic operators. In general, the user will not have to be concerned with mode incompatibilities, since almost all commonly used mode combinations are acceptable. However, should the default data types prove unacceptable for certain problems, care should be taken or unacceptable mode combinations will produce error messages.

The following facts summarize the default mode type conditions for operands.

1. All defined constants are scalar, real*4.
2. Vector pairs may only be created by using a displayed graph as an operand.
3. Vector pairs are acceptable to any operator which accepts vectors. Only the "range" part of the vector pair will be used. (This facility allows displayed results to be used in further calculations, and has been found extremely useful.)
4. The arithmetic operators + , - , * , / , and ** take two arguments, of which either both are vector, real*4 (or vector—pair), or one only may be scalar—real or scalar—integer. The result of these operations will be vector real*4.

5. All other operators, except = , accept only vectors or vector-pair.
6. = will accept any mode combination. The following conventions are observed:
   a. vector substituted into vector: clear; the dimensions of the two vectors must agree.
   b. vector substituted into scalar: the first element of the vector is copied to the scalar.
   c. scalar copied into vector: produces a constant vector as the result.
   d. scalar substituted into scalar: clear.

2.2 THE PREDEFINED BUTTONS

A more systematic description of the system buttons will now be given.

2.2.1 The Declaration Buttons

There are two major declarations which the user should make immediately after execution begins, if the default case is not acceptable. The first of these concerns the dimension of the vector data type. The default case for vectors is 101 entries. If the user wishes to change this standard dimension, he should point at the desired dimension (which may involve the definition of the constant, see Section 2.2.3) and then point at the DIMENSHN button. For example, the following sequence sets the standard dimension to 201 entries:

DFCONST 2 0 1 DEFCONST DIMENSHN

The only predefined operand in the MOMS system is the so-called ID vector. Stored under this button is the useful vector which ranges from -1 to +1 in 100 steps (i.e., 101 entries). If the user does not wish the default case, he may change the ID vector by using the SETID operator. This operation should be performed before any other operation is undertaken, even before changing of the standard dimension. The new definition of the ID vector is made as follows:

MIN MAX NUMBER OF ENTRIES SETID

Thus an ID vector from -1 to +1 with 200 steps (201 entries) would be established by the sequence

DEFCONST - 1 DFCONST DEFCONST 1 DEFCONST DEFCONST 2 0 1 DEFCONST SETID

2.2.2 Editing and Control Buttons

Editing of the input line is accomplished by using the BAKSPACE button. This button will delete the previous button pushed. Its effect is immediate. The user may also erase all button pushes back to the last interpreted equal sign by pressing the DEQUEUE button. The user is not allowed to use the BAKSPACE button or the DEQUEUE button past the last interpreted = because interpretation of that = will in general have changed data values.

In order that operator buttons may be used as data in special cases, the LITNEXT button is provided. Pushing this button causes the system to interpret the following button push as a data button unconditionally (See Sections 3.3 and 3.4 for examples.)

The START button may be invoked whenever the user wishes interpretation of its input sequence to start. Usually, however, the implicit starting of interpretation contained in the display operators (Section 2.2.4) will keep the use of this button to a minimum.

The RETURN button should be pointed at when the user wishes to terminate execution of MOMS and return to MTS. If for any reason he wishes to produce an error return to MTS, he may do so by depressing button Number 31 on the Programmed Function Keyboard of the 2250.

2.2.3 Definition of New Buttons and Constants

One uses the DEFCONST button to define a constant. For example, to define the constant 10.4 one would push

DEFCONST 1 0 . 4 DEFCONST

The user may define a new button using the DEFBUTON operation. For example, to define the button ALPHA, the user would push

DEFBUTON A L P H A DEFBUTON

Such a series of button pushes would cause the ALPHA button to appear in the vertical margin of the 2250.

Conversely, any user-defined button may be deleted using the DELBUTON button. Thus a user could press

ALPHA DELBUTON

to delete his button ALPHA. Only user-defined buttons may be deleted. If a user tries to delete a system button, for example A...Z , he will merely give it an undefined data type.

2.2.4 The Display Operators: PLOT1, PLOT2, SETPLTMD, SCALE, DISVALUE, SCROLLUP, SCROLDN.

A user may observe the results of a computation sequence defined by button pushes by using the various display operators. Graphical output may be in two forms. The first of these is a plot of the data on a coordinate grid. The second is a display of the numeric values of the data. Most often, a user will wish a plot of vector data types. The buttons PLOT1 and PLOT2 provide two means of obtaining such a display. PLOT1 plots the specified vector versus the system ID vector. This is the most common type of plotting. However, the button PLOT2 is available so that the user may plot two arbitrary vectors against each other. For example, suppose the user had pressed

ID ID * Y =

$Y = X^2$, $-1 \leq X \leq +1$,

thus the function is stored under the Y button and the button pushes

Y PLOT1

and

Y ID PLOT2

will produce exactly the same result.

A user also has control over the scaling of the grid, and the mode of the grid lines. To set the scale of the grid, the user should press the four operands which are the coordinates of the lower-left and upper-right corners of the grid. He should then press the SCALE button. If no scale is specified by the user, the system will use the domain and range of the first plot to determine default scale values. If a subsequent plot falls outside of this scale, the user will be notified via the message line.

The SETPLTMD button controls the type of grids on which graphs are plotted. The default case is Linear-Linear, but the user may specify other types of plots using the statement

TYPENUMBER SETPLTMD

The various options for TYPENUMBER are as follows:

Rectangular	0	Linear-Linear
	1	Linear-Log
	2	Log-Linear
	3	Log-Log
Polar	4	Angular-Linear
	5	Angular-Log

Thus for a Log-Log grid for his output, the user would make the statement

3 SETPLTMD

Normally, plots on a given screen (see Section 2.2.5) are cumulative. If a user wishes to have a replacement type of plotting, he may define a macro which first CLEARS the screen and then PLOTs the desired graph.

Numeric display of data may be obtained through use of the DISVALUE button. Thus, in our example, one might wish to determine how close to zero the approximation to X^2 is. Pointing at the buttons

Y DISVALUE

would display the first ten values of the vector Y . Since the vectors are usually 101 REAL*4 entries long, it is clear that we must examine more than the first ten locations to determine our answer. The SCROLLUP and SCROLLDN buttons are thus provided. By pointing at the numeric display and then pointing at the SCROLLUP button we successively scroll through the vector until we reach the desired entry. Up to three vectors may be displayed numerically on a single screen (see below).

2.2.5 Screen Viewing Operators

The MOMS system provides the user with three "working areas." Only one of these can be viewed at any one time by the user. He may switch views by pointing at

the SCREENA, SCREENB, or SCREENC button. The three views are independent of each other. Typically, therefore, a user will display vectors, etc., as plots on one screen, then change to another screen to examine numeric values, and perhaps return to the first screen for further inspection. Entities displayed on a screen may be selectively erased by pointing at the displayed entity and then pointing at the ERASE button. To erase all three screens the user need point only at the CLEAR button.

3. THE MACRO PACKAGE

The ability to group together a collection of "button pushes," and henceforth to treat that collection as a single unit, is a facility which is vital in serious computational problems. Other online mathematical analysis systems have this facility in varying degrees [1], and it was clear from the outset that the lack of such a facility in our system would seriously limit its capabilities. Thus, a definitional facility, called the Macro Processor, was implemented. This package gives the user the ability to define a string of button pushes and store this string with an associated name. Certain button pushes in the definition may be designated as formal parameters. When the user points at the macro name, the definition will be expanded with substitution of calling parameters in place of the formal parameters. Thus, a true macro facility exists in the MOMS system.

3.1 DEFINING A MACRO

A macro definition is a sequence of button pushes delimited by the DEFMACRO button. The general form is:

DEFMACRO NAME (D_1 M_1 T_1 ... D_m M_m T_m) B_1 B_2 ... B_r DEFMACRO

where NAME is the macro name, i.e., the button "under" which the macro is to be stored; D_i is the i-th formal parameter name; M_i is the (optional) mode of formal parameter D_i; and T_i is the (optional) type of the formal parameter D_i B_i is the i-th button push for the macro body. The mode of each formal parameter should be indicated by pointing at one of the four light buttons SCALAR (default), VECTOR, MACRO, OPERATOR; the type of each formal parameter should be chosen from the light button set REAL (default), and INTEGER. There may be at most 254 formal parameters within a macro definition.

The following rules must be observed in defining macros:

a. if m=0 , the form of the definition is DEFMACRO NAME (DUM) B_1 ... B_r DEFMACRO where DUM is used as place-holder but is never used in the macro body.

b. a macro definition must not appear within another macro definition.

c. a macro may call upon another macro, but it may not call upon itself (either directly of indirectly).

3.2 DELETING A MACRO

To delete a macro, press

NAME DELBUTON

where NAME is the name of the macro to be deleted.

3.3 CALLING A MACRO

A macro call can be one of the following forms:

(C_1 C_2 ... C_v) NAME

C_1 C_2 ... C_n NAME

where NAME is the name of the macro being called, C_i is the i-th calling parameter, n is the number of formal parameters for NAME, and $v \leq m$. The second form may be used only when the number of calling parameters equals the number of formal parameters. The first form allows for a variable number of calling parameters (v). If a macro has no formal parameters, then it may be called by pressing the button NAME. This should be distinguished from the case where no calling parameters are supplied, but formal parameters do exist in the macro definition. The following form should be used in the latter case:

( ) NAME

The following rules should be observed:

a. The number of calling parameters should not exceed the number of formal parameters. If this happens,

the extras are ignored and a warning message is displayed.

b. The number of calling parameters may be less than the number of formal parameters only if the undefined formal values were given actual values by a previous call. If this is not the case and such a call occurs, a fatal error will result.

c. Nesting of macro calls is allowed to a maximum level of 500. If this limit is exceeded, a fatal error occurs.

d. If a call upon a macro involved another macro or operator name as a calling parameter, the LITNEXT button must immediately precede that parameter. For example, if MAC is a macro name, a call upon another macro MAK with MAC as a calling parameter might look like:

(TI LITNEXT MAC ALPHA) MAK

The corresponding dummy parameter for MAC would have been given mode MACRO when MAK was defined.

3.4 DISPLAYING A MACRO

A macro definition may be displayed in the working area of the screen by the button sequence

LITNEXT NAME DISVALUE

where NAME is the button under which the macro definition

is stored. The macro definition will appear on the screen at a fixed position, and the formal parameters of the definition will appear within parentheses. However, the declared mode and type of the formal parameters are not displayed. Once on the screen, the display of the macro is treated as any other displayed entity and may be ERASEed and CLEARed in the normal fashion. Up to three macros may be displayed at one time, one on each of the three screens.

3.5 EXAMPLES

The following are permissible definitions of macros:

```
DEFMACRO A ( D VECTOR INTEGER E SCALAR REAL)
        D  E  =  DEFMACRO
DEFMACRO DEFBUTON A L P H A DEFBUTON ALPHA
        ( D E ) D E = DEFMACRO
        default modes and types for formal
        parameters
DEFMACRO  QED ( DUMY ) DEFCONST 3 . 1
        4 DEFCONST ARDVARK = DEFMACRO
        no formal parameters in the definition
DEFMACRO F ( M N MACRO O OPERATOR)
        M LITNEXT N O DEFMACRO
        illustrates a call upon a macro definition
DEFMACRO SUM ( M MACRO N MACRO )
         LITNEXT M LITNEXT M N DEFMACRO
        The following would be an illegal call,
        since it generates an infinite nesting
```

of the macro SUM.

(LITNEXT SUM LITNEXT SUM) SUM

4. THE MATHEMATICAL OPERATORS

Various standard mathematical operations are provided in the MOMS system. These include the standard arithmetic operations of +, -, *, /, **, and = , where ** is the exponentiation operator. In addition, those parts of the FORTRAN function library which accept REAL*4 arguments have been included, as well as facilities for integration and differentation. It is believed that these predefined operators will from a useful base from which more complicated operations may be constructed using the macro facility in MOMS.

4.1 THE ARITHMETIC OPERATORS

The arithmetic operators include those standard operations enumerated above. They will operate on all mode combinations (scalars, vectors, vector-pairs) and all type combinations (integer, real*4). User statements must, of course, be in the operator postfix notation, described in Section 2. The following table shows the equivalence between a FORTRAN-like notation and operator postfix notation:

FORTRAN	OPERATOR POSTFIX
A + B	A B +
A - B	A B -
A * B	A B *
A ** B	A B **

A / B	A B /
B = A	A B =

4.2 THE FORTRAN LIBRARY SUBPROGRAM OPERATORS

Table 4.1 lists those entries from the FORTRAN function library which were adapted for the MOMS system. Note, that when the argument type is listed as Real*4, the mode of the argument may be either scalar, vector, or vector-pair. If the mode is vector, the appropriate function will be applied to each entry in the vector. If the mode is vector-pair, the so-called "range" part of the vector-pair will be treated as if it were a vector, and the rest of the vector-pair will be ignored.

In general, an illegal argument to one of these functions will cause a value of zero to be returned. Thus, for example, X/0=0.

NAME	DEFINITION	ASSIGNMENTS		FUNCTION VALUE TYPE
		Number	Type	
ABS	$\|x\|$	1	real*4	real*4
ARCCOS	$\cos^{-1}(x)$	1	real*4	real*4
ARCSIN	$\sin^{-1}(x)$	1	real*4	real*4
ARCTAN	$\tan^{-1}(x)$	1	real*4	real*4
ARCTAN2	$\tan^{-1}\left(\frac{x_1}{x_2}\right)$	2	real*4	real*4
COS	$\cos(x)$	1	real*4	real*4
COSH	$\frac{e^{x}+e^{-x}}{2}$	1	real*4	real*4
COTAN	$\text{cotan}(x)$	1	real*4	real*4
ERF	$\frac{2}{\sqrt{\pi}}\int_o^x e^{-u^2}du$	1	real*4	real*4
ERFC	1-ERF(X)	1	real*4	real*4
EXP	e^{x}	1	real*4	real*4
GAMMA	$\int_o^\infty u^{x-1}e^{-u}du$	1	real*4	real*4
LOG 10	$\log_{10} x$	1	real*4	real*4
LOGE	$\log_e x$	1	real*4	real*4
LOGGAMMA	$\log_e\int_o^\infty u^{x-1}e^{-u}du$	1	real*4	real*4
SIN	$\sin(x)$	1	real*4	real*4
SINH	$\frac{e^{x}-e^{x}}{2}$	1	real*4	real*4
SQRT	$+\sqrt{x}$	1	real*4	real*4
TAN	$\tan(x)$	1	real*4	real*4
TANH	$\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	1	real*4	real*4

Table 4.1 FORTRAN Function Library Entries Adapted for MOMS

4.3 THE INTEGRATION AND DIFFERENTIATION OPERATORS

Four operators are provided for various cases of integration and differentiation. The light buttons INTEG and DIFF integrate and differentiate the one vector argument versus the standard domain, that is versus the ID vector. INTEG2 and DIFF2 use an explicitly provided domain, which is the second argument to the function. Both differentiation operators use the divided difference method to approximate the derivative. No attempt at smoothing is made; the last element of a vector is set equal to the second last element. INTEG uses trapezoidal integration to approximate the integral.

4.4 ADDITIONAL EXAMPLES

X Y ARCTAN2

In this function, the X argument is taken to be the abscissa, and the Y argument is taken to be the ordinate.

D V DIFF2

D V INTEG2

V is the vector (or vector-pair) being differentiated (integrated) versus the domain D.

5. A DETAILED EXAMPLE

Consider the problem of approximating $y(\Theta) = \sin(\Theta)$, $-\frac{\pi}{2} \leq \Theta \leq \frac{\pi}{2}$, with an n^{th} degree polynomial $P_n(x)$ written in

terms of Chebyshev polynomials, i.e.,

$$P_n(x) = a_0 + a_1 T_1(x) + \ldots + a_n T_n(x).$$

In the interval (-1,1) the coefficients can be expressed by

$$a_0 = \frac{1}{\pi} \int_{-1}^{1} \frac{y(x)}{1-x^2}\, dx$$

$$\vdots$$

$$a_i = \frac{2}{\pi} \int_{-1}^{1} \frac{y(x) T_i(x)\, dx}{1-x^2}$$

and the Chebyshev polynomials are recursively defined by

$$T_0(x) = 1$$

$$T_1(x) = x$$

$$\vdots$$

$$T_{i+1}(x) = 2xT_i(x) - T_{i-1}(x)$$

The problem is to construct a sequence of error fucntions $g_i(x) = y(x) - P_i(x)$ until a $g_i(x)$ is found which satisfies a predetermined maximum error criterion. A sample solution with comments follows.

Button Pushes	Comments
DEFBUTON N E X T T E R M DEFBUTON	Define a button called NEXTTERM
DEFBUTON P I DEFBUTON	and a button called PI
DEFMACRO NEXTTERM (X)	Define a macro NEXTTERM

Button Pushes	Comments
2 ID U * * V - T =	with 1 formal parameter (not used) which computes the next term in the approximation and adds it to the polynomial which is stored under button P.
U V =	T_i from above is stored under button U. T_{i-1} is stored under button V. The macro also updates these values when T_{i-1} is computed.
T U =	
Y D * INTEG PI *	
T * P + P = DEFMACRO	
DEFMACRO DEFBUTON D I F F	Define a macro to compute the next term and take the difference between Y and the polynomial.
DEFBUTON DIFF (X)	
NEXTTERM Y P -	
DEFMACRO	
DEFCONST 3 . 1 4 1 5 9 DEFCONST	
B = 2 B / PI =	Store $\frac{2}{\pi}$ under button PI
1 PI / ID * SIN Y =	Construct vector Y=sin Θ, $-\frac{\pi}{2} \leq \Theta \leq \frac{\pi}{2}$ let up T_0 in V set up T_2 in U.
1 V =	
ID U =	
1 ID * - SQRT A =	Compute $\frac{y(x)}{1-x^2}$
Y A / D =	where $-1 \leq x \leq 1$ and store under D.

Button Pushes	Comments
D INTEG PI 2 / * P =	Compute a_0 and store under P
D U * INTEG PI * U * P + P =	Compute a_1T_1 and add to P; store under P.
DIFF PLOT1 . . . DIFF PLOT1	Compute successive approximations and display error function.

Note that this would display an approximation to any function stored under the button Y.

6. DESIGN CONSIDERATIONS

MOMS runs under the MTS system, which in turn is under the control of the UMMPS (University of Michigan Multiprogramming System) supervisor. From the outset, it was decided that this total dependence on system I/O support was the only reasonable way to proceed, since bugs in the MOMS system could not be allowed to disrupt other users. Moreover, the time and effort necessary to write the low-level I/O code would have made the project difficult to complete in one semester. However, idiosyncrasies of the IBM 2250 display made some special adjustments necessary. Specifically, it was necessary to process attention interrupts from the 2250 in something approximating real-time,

in order to keep the screen lit and capable of accepting button pushes. (An attention interrupt is caused by a light pen, or by pushing a function keyboard button; in the former case, the display stops.)

Since the IBM Graphics package GRAPHLIB, described in IBM Document No. C27-6909 was available in MTS, it was decided to use the routines with slight modifications by Computing Center staff programmers. The routine ANALS, which previously polled for an attention interrupt, was changed to call a user-specified subroutine in "real-time" when an attention interrupt arrived. The display would be restarted and the user task resumed when the subroutine returned. It should be stated that

a. the user-specified subroutine and any subroutine it called had to be written to handle recursive calls, or had to be mutually exclusive from any code which could possibly be executing at the time of the attention interrupt.

b. "Real-Time" does not mean at the time of the actual attention interrupt, while all other tasks in the system are stopped. Instead, the interrupt causes the status of the task running MOMS to be saved, and that task is initialized to run the attention interrupt "real-time" routine. When the attention routine returns, the original status of the task is restored, and it continues. This can be thought of as a push-pop situation where the status of the task is saved on a push-down stack, and the interrupt is transparent

to it, except for any desired effect of the interrupt routine, such as building queues or setting status bits.

7. THE HIERARCHY OF THE SYSTEM

The 2250 display hardware is controlled by the 2840 display controller through standard I/O operations. Thus, via channel control word commands it is possible to

a. write the display buffer,

b. read the display buffer,

c. read information such as function button pushed, and coordinates of a light-pen detect,

d. control display starting location, etc.

The supervisor, UMMPS, is ultimately in charge of all operations concerning I/O devices. For example, I/O devices are allocated to tasks by the supervisor, I/O devices are "started" by a supervisor call whose operands resemble the start I/O operation, and asynchronous attention interrupts from a device are fielded by the supervisor, which either ignores them, or passes them to a "real-time" attention routine designated by the user.

MTS provides system subroutines which interface with the supervisor, and

a. acquire devices from the supervisor when requested by a program or command, and to release devices acquired by a user when he is finished with them, or when he signs off.

b. provide a general interface to the appropriate device support routine which can operate the device as a standard terminal.

GRAPHLIB is an IBM package for controlling the 2250. It has been modified to run under MTS.

a. It contains subroutines for generating 2250 order programs to display graphs, functions, etc.,
b. It contains subroutines to perform I/O operations on the display. It does this via supervisor calls.
c. It contains a subroutine to handle attention exits. This has been modified to call user-specified subroutines in "real time."

The MOMS code may therefore be considered as being devided into two sections. Part of MOMS consists of code for interfacing with GRAPHLIB; the other part of MOMS is completely independent of the 2250. The code which interfaces with GRAPHLIB does the following:

a. Allocates space within the 2250 buffer,
b. Uses GRAPHLIB to build order programs, which are subsequently displayed.
c. Uses GRAPHLIB to write the order programs to the allocated space in the 2250 buffer.
d. Uses GRAPHLIB to start the display.
e. Uses GRAPHLIB to call a subroutine which builds the queue of button pushes in real time.

8. THE INTERNAL ORGANIZATION OF MOMS

This section contains detailed documentation concerning the various parts of the MOMS system. In general, it will not be of interest to the MOMS user, but is included for completeness and future reference.

The major subparts of the MOMS system are as follows: initialization, interpretation, 2250 buffer management, light pen management, function display, numeric and keyboard display, the macro processor, symbol table management, the mathematical operators, and numeric conversion. The major connections between the various subparts are shown in Figure 8.1.

MTS enters the MOMS system at the initializer. This in turn calls other parts of the system in order that they might initialize themselves. The major portion of all initialization consists of building the system entries in the symbol table and the corresponding display representation of the system operations in the 2250 buffer. Clearly, each section must access the symbol table during this time. In fact, the symbol table, which may be considered the ultimate store of knowledge in the system, is accessed constantly by all sections, although this is not shown in Figure 8.1. When initialization is complete and the initial 2250 image is displayed on the screen, control is transferred to the interpreter, which retains ultimate control throughout the remaining execution. When a user points at

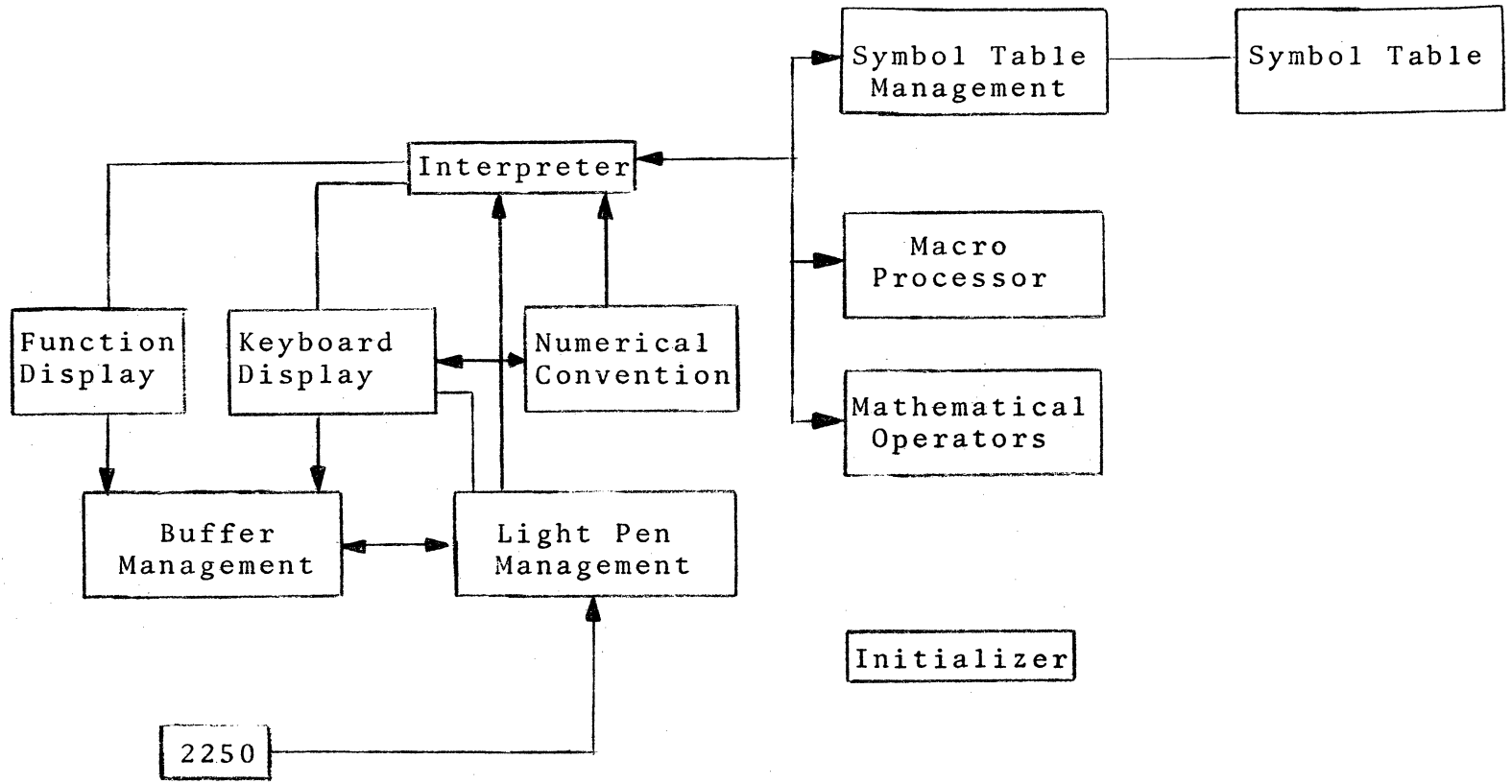


Figure 8.1 Major Connections between Subparts of MOMS.

a "light-button," the corresponding interrupt is fielded by the light pen management routines. They determine which light button was pointed at and communicate this fact both to the keyboard display section, so that it may be displayed on the echo line, and also to the interpreter. The interpreter handles all editing and definitional operations, except macros. It records the various button pushes in a queue, also called the stack. When a button push requiring the start of interpretation is encountered, the interpreter starts calling the various operators in this queue, collapsing the stack where appropriate. The system ultimately returns to the idling state when all interpretation is complete. The function of the various other parts may be briefly summarized as follows:

a. function display is responsible for constructing 2250 graphic order programs from the various vectors and vector-pairs supplied it. It is also responsible for handling the plotting mode and scaling operators for the grid.

b. keyboard display constructs 2250 graphic order programs which contain alphanumeric and other character information only. Thus it is responsible for the 2250 graphic order programs of all light buttons, the display of all numeric values, the upkeep of the echo line, the output of messages, and the display of macros.

c. the 2250 buffer management section is responsible

for actually writing the 2250 buffer and linking graphic order programs into the display regeneration loop. It also keeps track of the SCREENA, SCREENB, and SCREENC operations.

d. symbol table management controls all accesses and all editing of the symbol table.

e. the macro processor is responsible for the storage of all macro definitions and interacts closely with the interpreter when a stored macro is called.

f. the mathematical operators section and the numerical conversion section are self-explanatory.

8.1 INITIALIZATION R.W. Taylor

Name:	STINIT
Purpose:	To initialize MOMS
Calling Sequence:	OS(I) R type
Entry:	No parameters
Return:	Calls ERROR indicating a system error if the Interpreter ever returns control to it.

Functional Description:

The initialization section serves as the principal subroutine with respect to MTS.

1. Call initialization sections of

Symbol Table Management	SYMINIT1
Buffer Management	BUFINIT1
Interpreter	INTINIT

Light Pen Management	LPMINIT
Function Display	FDINIT
Macro Processor	MACROFIX

in the above order (STLOOP1)

2. Use the SYMODEFS macro to put the buttons SCALAR, VECTOR, REAL, INTEGER onto the screen.
3. Enter A...Z,.,(,) into the symbol table with default parameters (see Section 8.3, Symbol Table Management Routines) and save their STEPS* in a list to be passed to the display virtual keyboard routine (STSYMLP)
4. Create the symbols +, -, *, /, = and save their STEPS for display on the virtual keyboard. (The actual operators +, -, *, /, = are initialized into the symbol table by UINIT using the SYMODEFS macro.) (STSYMLP2)
5. Create the symbol table entries 0...9 and give them the attributes OPERAND, SCALAR, REAL, SYSTEM, CONSTANT (STSYMLP1)
6. Move the real values 0...9 into the appropriate place in the symbol table (that place pointed to by the VALUE pointer)
7. Call UINIT to define the mathematical operators and put them on the screen.

(Note: UINIT is called near the end so that

*STEPS = Symbol Table Entry Pointers

the mathematical operators will be at the bottom of the function button list.)

8. Call BUFINIT2

   Buffer Management has two entries so that light pen interrupts will not be accepted until initialization is complete.

9. Transfer control to the Interpreter.

8.2 THE INTERPRETER R. Srodawa, K. Moore, and J. Tripp

The Interpreter within the MOMS system is broken physically and logically into six sections, each of which is described briefly below:

Initialization

The initialization section of the interpreter is broken into two ports. The first port is called during the initialization of MOMS at the entry point INTINIT. It presets the global dimension to 101, the global mode to VECTOR, REAL*4, and creates and presets symbol table entries for the buttons DIMENSHN, ID, LITNEXT, BAKSPACE, DEFCONST, DEFBUTON, DEFMACRO, SETID, DELBUTON, DEQUEUE, and START. The second port of initialization is called at the entry point INTERP when all of MOMS has been initialized. This section initializes the three stacks kept within the interpreter (STACK, MSTACK, and BUTQ) and then transfers to the section of the interpreter which interprets button pushes (the Button Queue Processor).

Interpretable Operators

The interpretable operators section of the interpreter contains the operator definitions for several of the buttons.

START The START operator is an immediate operator which requires no operands and returns no results. Its only effect is to cause interpretation of the contents of the button queue to begin. Normally interpretation begins only when a button is pushed whose interpretation causes new information to be displayed. START is normally used to start interpretation because macros have been invoked which contain buttons that change the display.

SETID The SETID operator is used to specify the standard domain. It computes new values for the scalar button DIMENSHN and the vector button ID. The SETID operator must be the first operator to ever be interpreted.

RETURN The RETURN operator simply returns to MTS. It is the standard exit from MOMS back into MTS.

DELBUTON The DELBUTON operator accepts one operand and causes something to be deleted for it. If the operand currently has something displayed (a macro definition, graph, numeric value), that

is erased. Otherwise, if the operand currently has a macro definition, that is deleted. Otherwise, the button name is deleted from the screen and its STE removed, unless it is a system symbol, in which case the STE is changed back to undefined status. These deletions are performed by calling the subroutine appropriate for deleting the type of display represented by the operand (DSPDELNM for a button name, DSPERSPR for a macro definition display, DSPERSEV for a numeric vector display, FDERASE for a graph, and MACDELET for a macro definition). These routines are each called for the operand until one of them succeeds in removing it. The order of the calls is determined by the mode and structure of the operand, and is such that appropriate display deletes are attempted before the macro definition or button name is deleted.

DEQUEUE The DEQUEUE operator clears the three interpreter queues (BUTQ, STACK, and MSTACK) so that all past interpretive history, except items which have been stored in symbol tables entries , is forgotten.

Operator Call Processor

The operator call processor is called at the entry point INTKIP by the stack manipulation section of the interpreter. The operator call processor then searches for the appropriate definition of the operator, depending upon the modes of the operands, builds a parameter list for the operator, calls the operator definition, reduces the operand stack by the number of operands used, releases any temporaries that were used, and returns a result to the stack. The operator call processor is the interface between the operator definitions and the interpreter.

SYMOPDEF Subroutine

The SYMOPDEF subroutine is called by every occurrence of the SYMODEFS and SYMODEFD macros. This subroutine creates a new symbol table entry or instance for the new operator definition.

Button Queue Processor

This processor is called at the end of initialization by port two of the interpreter initialization. It processes the buttons which have been pressed by the operator and interprets the BAKSPACE, DEFCONST, DEFBUTON, and DEFMACRO buttons. Usually the symbol table entry pointer for a button is simply placed on the button queue (BUTQ). Whenever a button which requires immediate interpretation (DISPLAY, START, etc.) is encountered the stack manipulation section of the interpreter is called to interpret

all the button pushes since the last interpretation. The BAKSPACE, DEFCONST, DEFBUTON, and DEFMACRO buttons require more involved processing.

Stack Manipulation Processor

The stack manipulation processor is called at its entry point, INTJOHN, whenever the button queue processor receives a button which forces interpretation. The stack manipulation processor processes the STEPs from the button queue (BUTQ) and maintains the two stacks STACK and MSTACK. It calls the operator call processor every time it processes a button which is an operator or macro. The stack manipulation processor also obtains the buttons comprising a macro definition from the macro interpretation when appropriate.

8.2.1 Operator Call and External Specifications

This part of the documentation describes the function of the operator call part of the interpreter in just enough detail to allow those writing operators to set up appropriately the operator symbol table entry and know what to expect (and what is expected of them) when the operator is actually called.

The document is in three parts: the first indicates what information may be passed to operators if requested; the second gives something about the structure of operator symbol table entries; and the third describes two macros useful for putting operators into the symbol table.

*What Can Be Passed To Operators

Operators are called with a standard OS type I(S) calling sequence. If the operator wishes to return something to the stack, it can return only a single thing (which may be an operand or an operator), and it is expected to leave the STEP for that operand in GR0 on return. The parameter list for an operator may contain several different things, depending on the operator symbol table entry, including all or some of the following:

(1) A fixed, or variable, number of stack operands. These are always passed as a pair of parameters, one being the STEP, and the other being the VALUE (from the STE).

(2) Any fixed number of temporaries needed by this operator. These are passed as pairs of parameters, as in (1), and may be of any specified mode and structure, as needed by the operator. Temporaries are given the global dimension if they are vectors.

(3) The number of stack operands passed. Obviously of use only to operators which accept a variable number of operands, it is passed as a 4-byte integer.

(4) The standard dimension. This too is passed, if requested, as a 4-byte integer.

(5) The standard domain, or ID vector. This will always be passed as a REAL*4 vector if requested, and is passed as a pair of parameters as in (1) and (2).

(6) The operator STEP. This is passed directly in the parameter list (i.e., the STEP is in the list itself.)

As mentioned above, what subset of the above is passed to the operator, and in what order, depends on the STE for the operators whose description follows.

*What Operator Symbol Table Entries Look Like

An operator symbol table entry is really a rather empty (or at least devoid of much information) STE, coupled with a string of dope vectors, each of which describes an instance of the operator. 'Instances' of operators are different versions of the operator for operating on different kinds of operands. For instance, one version of the COS operator might operate on vectors, and another might operate on vector pairs. Each version would be described by a dope vector linked to the STE for the COS operator.

Some parts of the operator symbol table entry are relevant to the operator; these are given below.

The symbol table entry itself (the relevant parts):

(1) Name - gives 8-character name of this operator, as displayed on the screen.

(2) Dope Vector Pointer - points to first link in chain of dope vectors which describe instances of this operator.

(3) Display Buffer ID - gives the buffer ID for the operator (i.e., where it is displayed on the screen).

(4) Symbol Class - Will indicate macro or operator, depending on which it is (macros are treated nearly like operators by this section of the interpreter).

The rest of the STE for an operator is not used by anyone for anything of interest, so far as we know.

The dope vector contains all sorts of good information about this instance of the operator. Part of this information is fixed (in location within the dope vector), and that is given below. Figure 8.2.1 shows where and how this information appears in the dope vector.

(1) A full-word link to the next dope vector (if there is one, otherwise it is =0).

(2) Address of the routine which 'is' this instance of the operator (i.e., subroutine address).

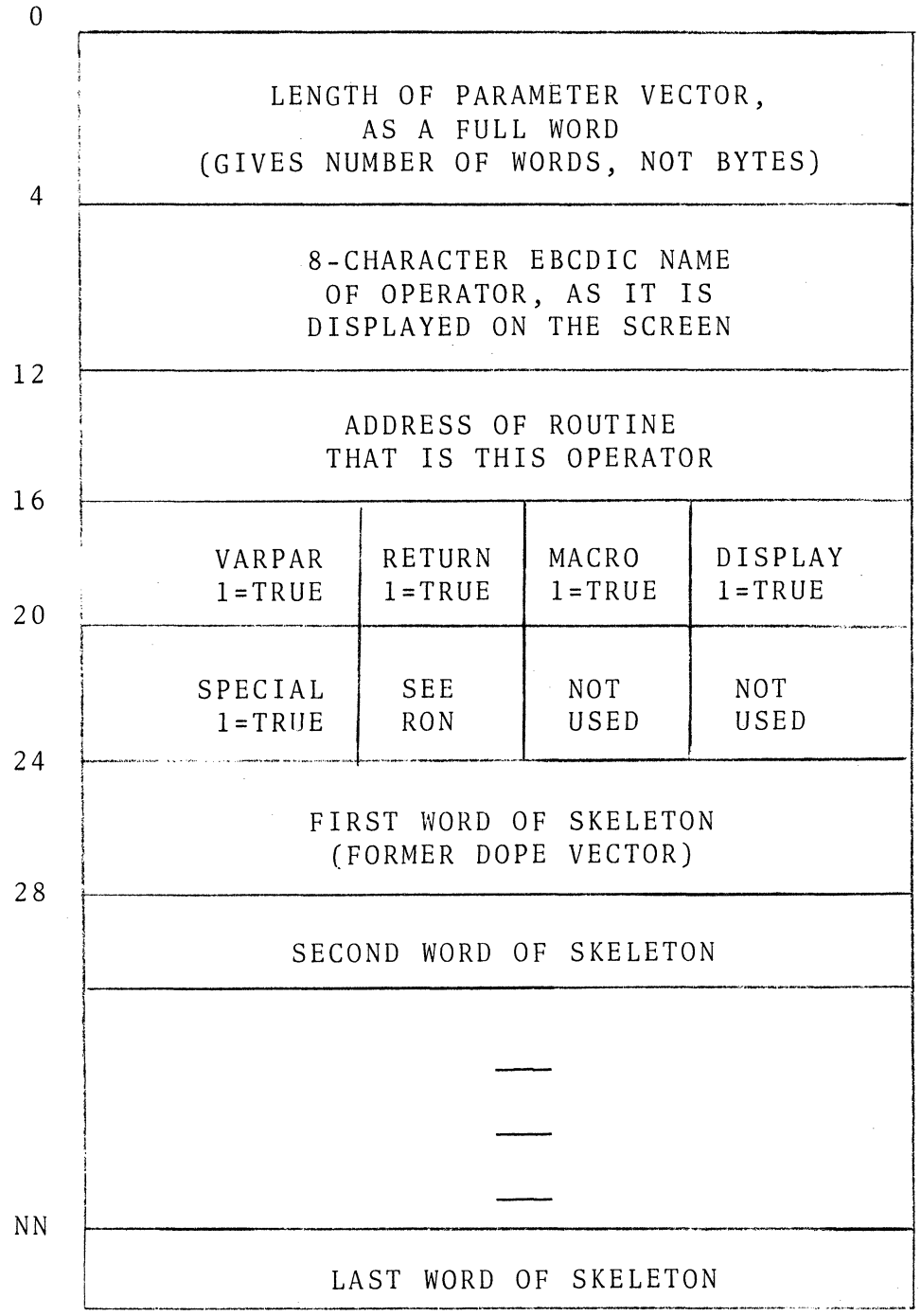
(3) Flags indicating (A) if the operator returns a result, and (B) if the operator will accept a variable number of operands.

(4) The length of the dope vector, in words (as a half-word).

(5) A flag indicating if this operator is special, meaning it must be either interpreted immediately or changes a symbol table entry. (Which of these is the case is also indicated if the operator is really special.)

The skeleton part of the dope vector, which follows the fixed information described above, is used to specify

Figure 8.2.1 Form of Parameter Vector for Macro: SYMODEFD



just what parameters should be passed to this instance of this operator when it is invoked, and also specifies what mode of operand is acceptable to it for each operand it takes. The length of the dope vector is determined when the instance of the operator is placed in the symbol table. The skeleton itself consists of some number of full words, each word specifying a parameter (or sometimes, a pair of parameters), to be passed to the operator. The first byte of each word indicates the kind of parameter to be passed, and the rest of the word is used to specify subsidiary information. The possible values for byte 0 (the first byte in the word) and the meaning of each value follow:

0 Means pass next stack operand, as a pair of parameters, checking the mode of the operand (on the basis of bytes 1 and 2) to see that it is suitable for this operator. In this case, the actual parameters (words in the parameter list) generated are STEP and VALUE (from the STEP), in that order, for the stack operand.

1 Means pass a temporary. In this case, a temporary of the mode indicated by bytes 1 and 2 is created and passed as a pair of parameters (STEP and VALUE as in (1)) to the operator. The temporary will be destroyed when the operator returns unless (1) the operator returns the particular temporary to the stack, or (2) the operator increments the use count for the temporary. In case (2) the operator is responsible

for seeing that the temporary gets destroyed eventually.

2 Means pass the standard domain vector (always REAL*4) as a pair of parameters. Bytes 1 and 2 are ignored.

3 The number of stack operands passed (total) will be generated as a 4-byte integer and passed.

4 The standard dimension is passed as a 4-byte integer. Bytes 1 and 2 ignored.

5 The operator STEP is passed as a 4-byte integer. Bytes 1 and 2 ignored.

6 Indicates end of dope vector (and end of parameter list, in a sense). Bytes 1 and 2 are ignored.

Byte 1 is used to specify what mode(s) of operand are suitable for this operator (if byte 0 = 0) or to specify what mode of temporary should be generated (if byte 0 = 1). Otherwise, it is ignored. The meanings of the bits are as follows:

Bit 0 - X'80' - scalar (if bit is ON)

Bit 1 - X'40' - vector

Bit 2 - X'20' - complex (when implemented)

Bit 3 - X'10' - vector-pair

Bit 4 - X'08' - operator

Bit 5 - X'04' - macro (console program)

Bit 6 - X'02' - undefined (i.e., operand may be of class undefined)

In the case where byte 0 = 0, byte 1 is used as a mask to see what is acceptable as an operand, e.g., both bits 0 and 1 could be ON indicating that the operator will accept a scalar or a vector in this operand position.

Vector-pair here is taken to mean a pair of vectors, stored one after the other in memory.

If an operator will accept operands of undefined class, they are simply passed to it "as is." If an undefined operand appears elsewhere, it will be given the global mode before being passed to the operator (if it will accept that mode).

Note that the operand specifications in the skeleton are listed in the order of things coming off the top of the stack. That is, the first specification in the dope vector applies to the operand on the top of the stack.

Operators may accept operands that are of the class OPERATOR or MACRO, and buttons of this class can be placed on the stack for use as operands by the LITNEXT button. The operator DELBUTON is an example of an operator using this feature.

Byte 2 is used in the same way as byte 1, but specifies the structure of operands allowed (or to be generated), as follows:

Bit 0 - X'80' - INTEGER*2 (not implemented within system)
Bit 1 - X'40' - INTEGER*4
Bit 2 - X'20' - REAL*4
Bit 3 - X'10' - REAL*8 (not implemented within system).

*A Simple Example

Consider the SIN operator. It might, as mentioned above, have several instances. Suppose we have a FORTRAN subroutine called RSIN which computes the trignometric sine of a scalar, and is smart enough to do it for arguments which are INTEGER*4 or REAL*4. The dope vector entry for this instance of the SIN operator would probably look something like this:

Flags would be present to indicate that the operator would accept only a fixed number of operands, and did return a result to the stack. But the operator is not "special" as it does not require immediate interpretation, or change the symbol table.

The skeleton part of the dope vector would contain

Word 0 - XL4'04000000'

Word 1 - XL4'00803000'

Word 2 - XL4'01802000'

Word 3 - XL4'06000000'

Word 0 indicates that the standard dimension should be passed as the first parameter to RSIN (it doesn't really need it, but then this is just an example).

Word 1 requests that the operand on the top of the stack be passed, and that it must be scalar, but may be REAL*4 or INTEGER*4.

Word 2 requests that a temporary scalar, REAL*4, be created

and passed as the next pair of parameters. Presumably, this is what will be returned to the stack.

Word 3 indicates end of dope vector, and puts nothing in the parameter list.

The subroutine RSIN could now be written something like this:

```
Subroutine RSIN (IVDIM,INSTEP,WRDIN,OUTSTE,WRDOUT)
```

...Where IVDIM will have the integer value equal to the standard dimension. INSTEP could be used as an array name to address the STE for the input operand (similarly for OUTSTE, and the output operand). WRDIN is the name of the variable whose sine is to be taken, and WRDOUT is where the result should be stored.

There would have to be a special function written to place the STEP for the temporary directly into GR0 on return, so that it could be placed on the stack.

*What Do the (Operator) STE Macros Look Like?

There are two macros, SYMODEFS and SYMODEFD, available to make putting operators into the symbol table a little easier. One, SYMODEFS, is a "static" macro, in that it assumes you know what you are doing when you write the macro instruction. The other one, figuratively speaking, assumes you don't know what you are doing, but have acquired information about the operator you wish to enter in the symbol table dynamically (such as from a table). Both

macros make all and exactly the assumptions made by the other symbol table management macros.

*SYMODEFS

Has several positional and keyword parameters with the following descriptions:

Positional Parameters

(1) The name of the operator, as a CL8 character constant.

(2) An ADCON giving the address of the subroutine which is this instance of the operator. May be A or V type ADCON.

(3) An operand sublist-type parameter, giving the dope vector for the entry, as a series of constants suitable for use in a DC-type statement.

Keyword Parameters

(1) RETURN=0 Operator does not return anything to the stack.
RETURN=1 Operator returns something to stack (default).

(2) VARPAR=0 Operator takes fixed number of operands (default).

VARPAR=1 Operator takes variable number of operands.

(3) DISPLAY=0 Don't try to display operator on screen.

DISPLAY=1 Display operator on screen (default). (If operator is already on screen, it will not be displayed again.)

(4) MACRO=0 Operator is really a hard code operator (default).

MACRO=1 Operator is really a macro (in which case the subroutine MACROINT is called, instead

of the macro itself, when the macro is invoked).

(5) SPECIAL=0 Operator need not be executed immediately. (It does not change screen, nor does it change the symbol table or anything in it.) (Default.)

SPECIAL=1 Operator does do one of the above.

If operator is SPECIAL=1 type, the keyword RONHEX must be present, and be =04 if the operator should be interpreted immediately and =02 if it modifies the symbol table.

Example. A macro-instruction which would make the appropriate symbol table entry for the subroutine RSIN described in the previous example follows:

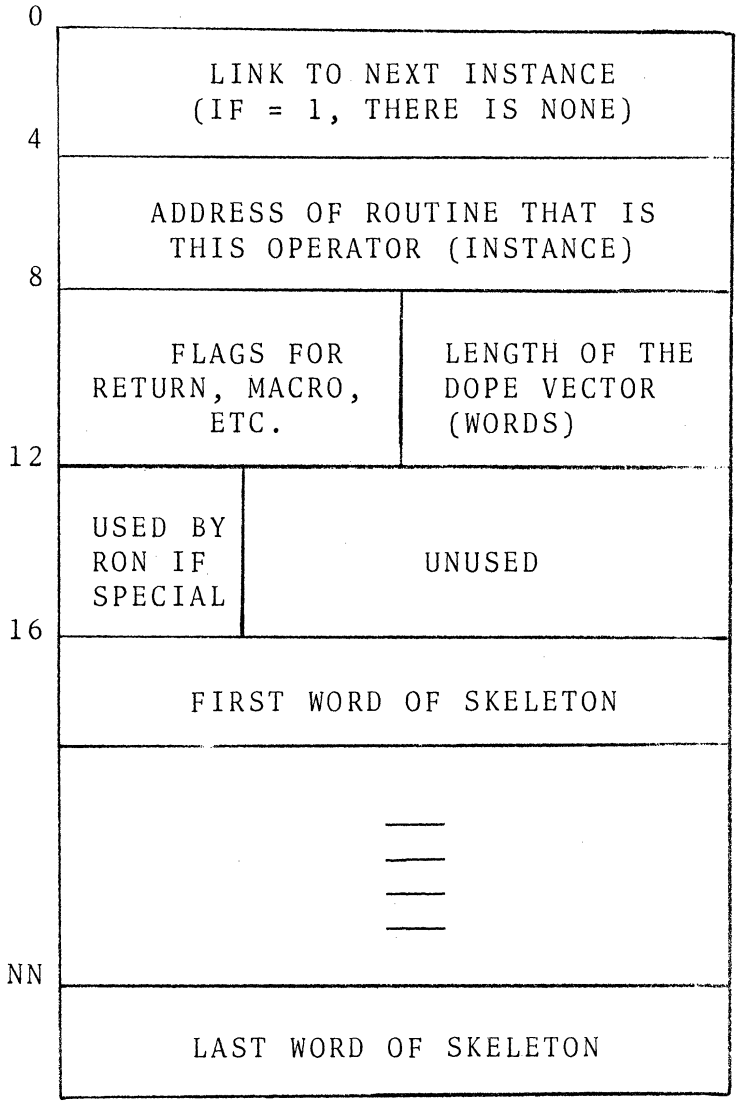
```
SYMODEFS  CL8'SIN',A(RSIN),(XL4'04000000',XL4'00806000',
            XL4'0180200',XL4'06000000'),RETURN=1
```

*SYMODEFD

This macro assumes you have built a table in memory providing the information to build a dope vector for an instance of the operator. The information is essentially the same as given in the SYMODEFS macro-instruction. The exact format is given in Figure 8.2.2.

The macro instruction takes one optional positional parameter which is the name of a general register containing a pointer to a parameter vector of the form given in Figure 8.2.2 (which must be on a full-word boundary). GR1 is assumed if no parameter is given on the macro-instruction.

Figure 8.2.2 Form of Dope Vector for Instance of an Operator (for Purposes of Documentation)



8.2.2 Internal Specifications and Operator Calls

This section of the interpreter (CSECT name - INTKIP) is executed whenever an operator in the button queue is to be invoked. It performs the following functions:

1 Searches for a suitable instance of the operator. (See external specifications for what this means.)

2 Builds a parameter list for the operator (or the macro processor, if the operator is really a macro), as specified by the operator's dope vector.

3 Calls the operator or macro with an OS type I(S) calling sequence.

4 Reduces the operand stack by the number of operands used by this operator, releases temporaries that were on the stack and used, and returns a result to the stack (if the operator produces one).

The above processing is performed in three logical "passes" over the operator dope vectors and/or operand stack: the search for the suitable instance of the invoked operator; the building of the parameter list for the selected instance; the release of used temporaries and return of result to stack.

*The Suitability Search

The linked list of dope vectors attached to the operator STE is searched, looking for the first suitable instance

of the operator. The instances are linked in such a way that the last instance placed in the symbol table will be the first one checked for suitability. This allows you to replace a macro definition by simply defining a new one -- the "latest" one will always be called.

Suitability is checked by matching each request for an operand (type 00 entry in the skeleton) against successive operands from the top of the stack. An instance will be found unsuitable in any of the following cases:

1 The operator (instance of it) will not accept some operand on the stack, e.g., the operator requires a real vector as its operand, and there is an integer scalar on top of the stack.

2 Some operand is of undefined mode and the operator will not accept the current default mode at this operand position.

3 There are not enough operands on the stack to satisfy this operator, e.g., it requires two operands and the stack is empty.

4 This operator will not accept a variable number of operands. The operator was preceded (on the button queue) by a parenthesized list explicitly defining the number of operands to be passed, and the numbers do not match.

5 The operator will accept a variable number of parameters, but there are not enough operands to meet its

minimum demands, e.g., the mode-defining operator will accept a parenthesized list of operands, but must always have at least one operand.

*Building the Parameter List

When a suitable instance of an operator is by some unlikely chance found, a parameter list for the operator is built, according to the specifications given in the skeleton of the dope vector for that instance. This process is described in the external specifications, except for one special case.

If the operator has said it would accept something of the global mode in a particular operand position, but that operand is at present of undefined class, the operand is given the global mode by the interpreter before it is passed to the operator. If the operator has said it will accept something of undefined class at this operand position, then the operand is untouched and is simply passed as undefined.

When the parameter list has been built, the operator is simply called with a standard OS calling sequence. It is expected to return (the only exception to this being, surprisingly enough, the operator RETURN, which gets you back to MTS).

*Reducing the Stack

When an operator has returned to the interpreter,

the operands it used are removed from the stack. This also involves checking to see if any of the operands are temporaries. If they are, their use-count is reduced by 1 (and the temporary will be destroyed if the count has gone to zero).

If the operator returned a result, the result is placed on the top of the stack, and its use-count is incremented by 1. A return is then made to the stack management part (INTJOHN) of the interpreter.

If the thing returned to the stack was an operator, it will be interpreted immediately at this point.

8.2.3 Button Queue Processor (INTRON)

The button queue processor requests buttons from light pen management via the entry LPMDEQLP and takes the appropriate action. This action can be

1) Stack the button on the button queue (BUTQ).
2) Edit the current contents of the button queue.
3) Gather together the buttons comprising a macro definition and call MACRODEF to define the macro.
4) Gather together the buttons comprising a new button name and then create a symbol table entry for this new button and have it displayed (by calling DSPNEWNM).
5) Gather together the buttons comprising a constant definition and then create a symbol table

entry for this constant and place it into the button queue as an operand.

The button queue processor actually is written as a finite state machine with four states and nine classes of input symbols. The four states are:

0. Normal state, not in any definition.
1. Inside of a DEFMACRO...DEFMACRO construction.
2. Inside of a DEFBUTON...DEFBUTON construction.
3. Inside of a DEFCONST...DEFCONST construction.

The nine input symbol classes are:

0 Button of operand class
1 Button of macro name class
2 Button of undefined class
3 Button of operator class (does not force interpretation)
4 DEFMACRO
5 DEFBUTON
6 DEFCONST
7 BAKSPACE
8 Button of operator class (does force interpretation)

Besides this information, the index into the button queue at the start of a macro, button, or constant definition is saved while inside a definition to aid in the processing of these definitions.

The button queue processor, when it receives a new button, references the appropriate entry in a state transition

matrix which tells what state to go to next and gives the address of a routine to take any necessary action with the current button. This state transition diagram is given in Figure 8.2.3.

Besides the actions noted in Figure 8.2.3, any BAKSPACE while in state zero causes a bit to be set which causes the stack manipulation processor to update the working stack from the master stack and begin interpretation from the beginning of the button queue. This is necessary because the backspaces could have deleted operators which have already been interpreted and have left results in the stack. Every unstacking of items from the button queue, either because of a macro, button, or constant definition or because of a BAKSPACE, causes DSPECHOZ to be called to correct the echo line. Use counts are also incremented on every symbol placed on the button queue and decremented on every symbol removed from the button queue.

8.2.4 Brief Description of Stack Manipulation Processor (INTJOHN)

No parameters in calling sequence.

INTJOHN is called by INTRON whenever an operator which forces interpretation is placed on the Button Queue-BUTQ. INTJOHN places each Button Push –BP– from BUTQ onto STACK, determines whether the BP is a macro or operator, and if so calls INTKIP for execution of that operator. Otherwise, BPs

Figure 8.2.3 State Transition Diagram

Current State

Current Button Class	0 Normal State	1 Inside DEFMACRO	2 Inside DEFBUTON	3 Inside DEFCONST
0 Operand	0 Stack the button	1 Stack the button	2 Stack the button	3 Stack the button
1 Macro Name	0 Stack the button	1 Stack the button	2 Stack the button	3 Stack the button
2 Undefined	0 Stack the button	1 Stack the button	2 Stack the button	3 Stack the button
3 Normal Operator	0 Stack the button	1 Stack the button	2 Stack the button	3 Stack the button
4 DEFMACRO	1 Stack & note location	0 Unstack defn and send to MACRODEF	1 Error, ignore button	1 Error, ignore button
5 DEFBUTON	2 Stack & note location	2 Error, ignore button	0 Unstack defn, create symbol, display it	2 Error, ignore button
6 DEFCONST	3 Stack & note location	3 Error, ignore button	3 Error, ignore button	0 Unstack defn, create symbol, convert constant, stack it

Figure 8.2.3, continued

7 BAKSPACE	0 Remove previous button	1 or 0 Remove previous button, If DEFMACRO go to state 0 instead of 1	2 or 0 Remove previous button, If DEFBUTON go to state 0 instead of 2	3 or 0 Remove previous button, If DEFCONST go to state 0 instead of 3
8 Operator which forces interpretation	0 Stack the button Call INTJOHN	1 Stack the button	2 Stack the button	3 Stack the button

continue to accumulate on STACK from BUTQ until an operator or macro is finally encountered. When BUTQ is empty, INTJOHN returns to INTRON.

INTJOHN maintains two stacks: a working stack, STACK, and a master stack MSTACK. Normal interpretation is handled via STACK. However, when any operator is called which modifies symbol table entries, STACK is copied into MSTACK. In turn, whenever editing takes place in BUTQ, MSTACK is copied into STACK before interpretation begins! This feature permits the user to edit a string of input BPs back to the last symbol table changing operator (such as =). After an operator of this type has been executed, the echo display and BUTQ are collapsed. That is, the next element of BUTQ above the operator becomes the bottom of a new BUTQ and echo line. The displaced part of BUTQ is sent to the history queue.

INTJOHN increments the use-count of every STEP which is placed on STACK, and decrements the use-count of every step which is sent to the history queue. Whenever STACK is copied into MSTACK, the use-count of each BP in STACK is incremented, and the use-count of each BP in the old version of MSTACK is decremented. Use-counts are modified similarly when MSTACK is copied into STACK.

When the LITNEXT operator occurs, the BP following it is placed on STACK as an operand without regard to its class; the LITNEXT operator thus allows the user to use an an operator as an operand.

When macros are expanded, the BPs comprising the macro definition are obtained from MACROGET routine and treated in exactly the same fashion as BPs obtained from BUTQ. The initial call on a macro expansion is accomplished by INTKIP. Subsequent calls to obtain BPs are done within INTJOHN; once a macro expansion has begun, all BPs are obtained from MACROGET until it returns a 0 in GR1, signaling the end of the macro definition. Nesting of macros is monitored by the macro processor, and INTJOHN does no pushdown on macro calls.

INTJOHN does no checking for syntax except for the use of parentheses. It is assumed that parentheses will be used only to delimit the operand stream for an operator, and hence every right parenthesis, ")" must be followed by an operator. Furthermore, a count is maintained of the number of left parentheses minus the number of right parentheses. This count may never go negative. If either condition is violated, interpretation is halted, and an error comment is presented to the user. Additional checking for proper use of parentheses is done within INTKIP. Note that parentheses may not be used to delimit compound operands, unless followed immediately by an operator.

Examples of illegal syntax:

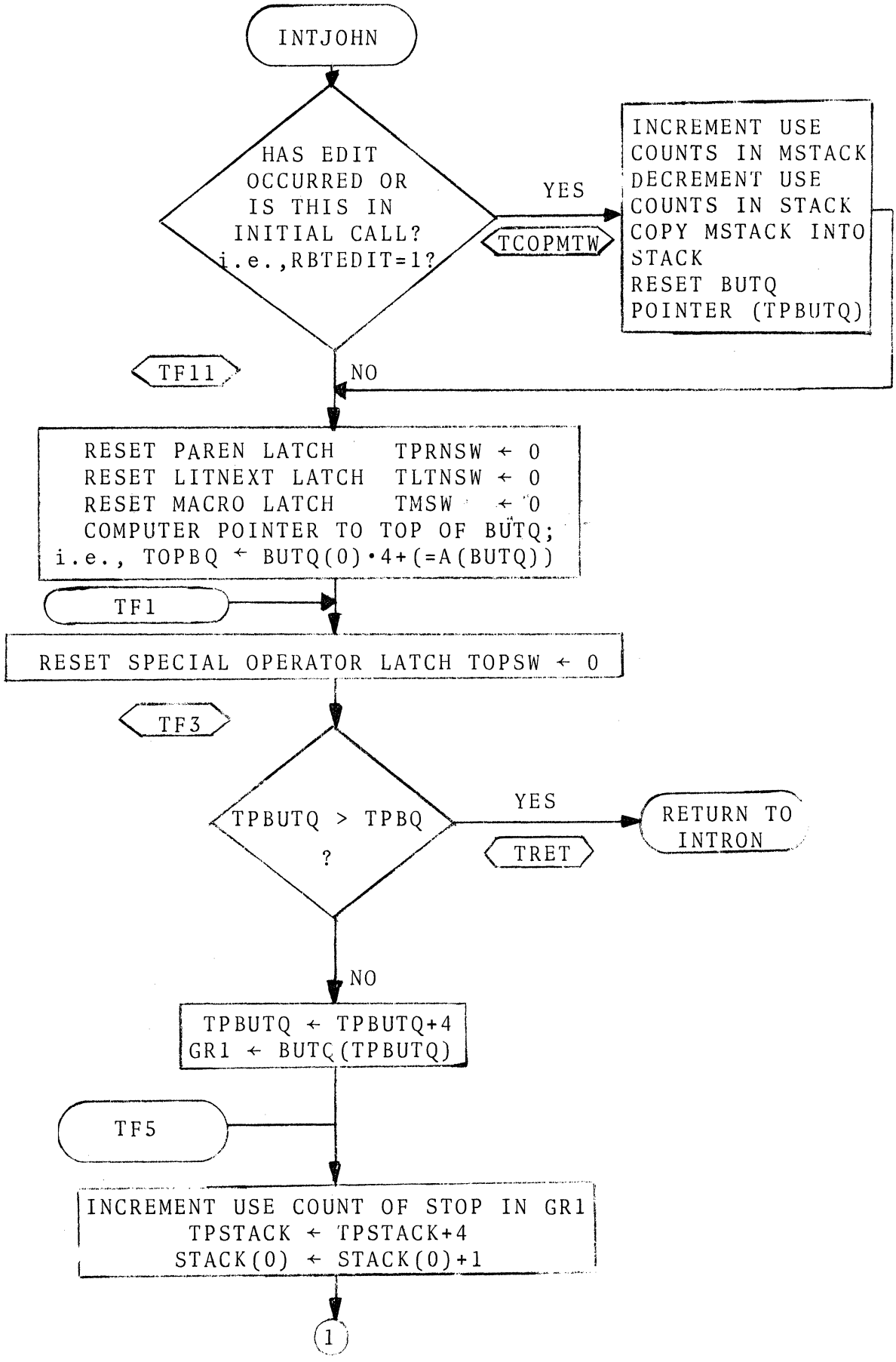
(A) B+

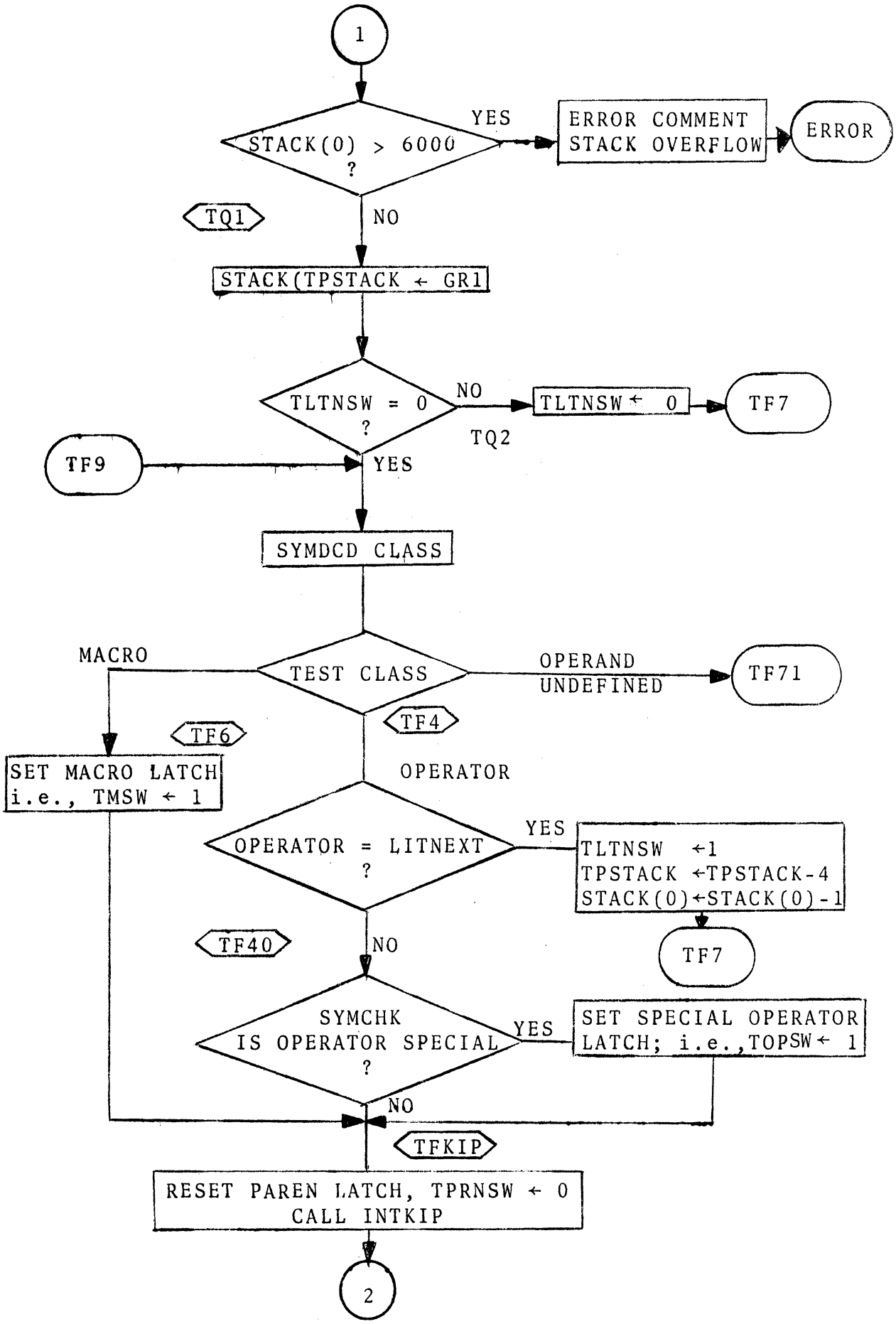
(AB-)C+

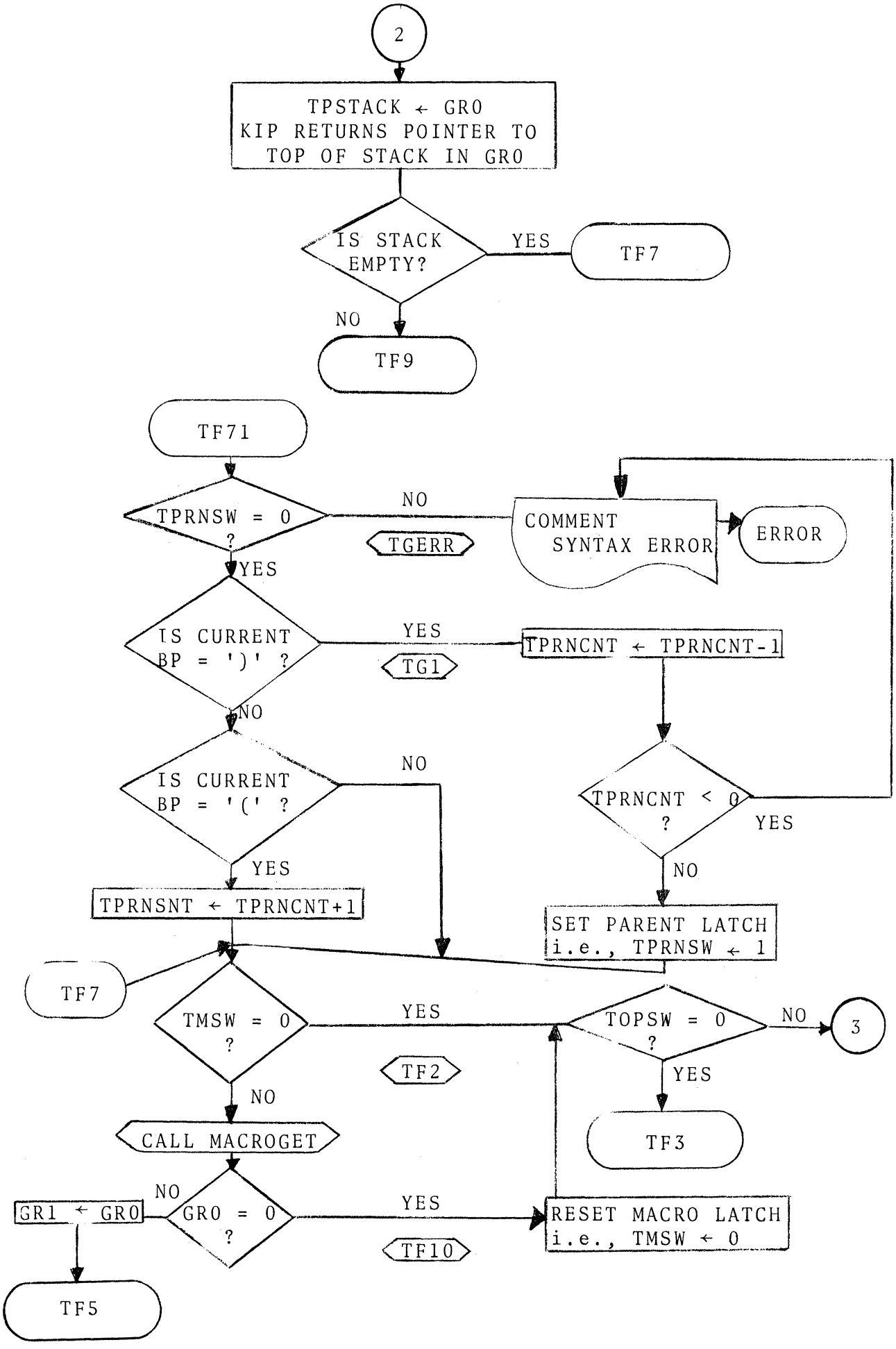
)AB*

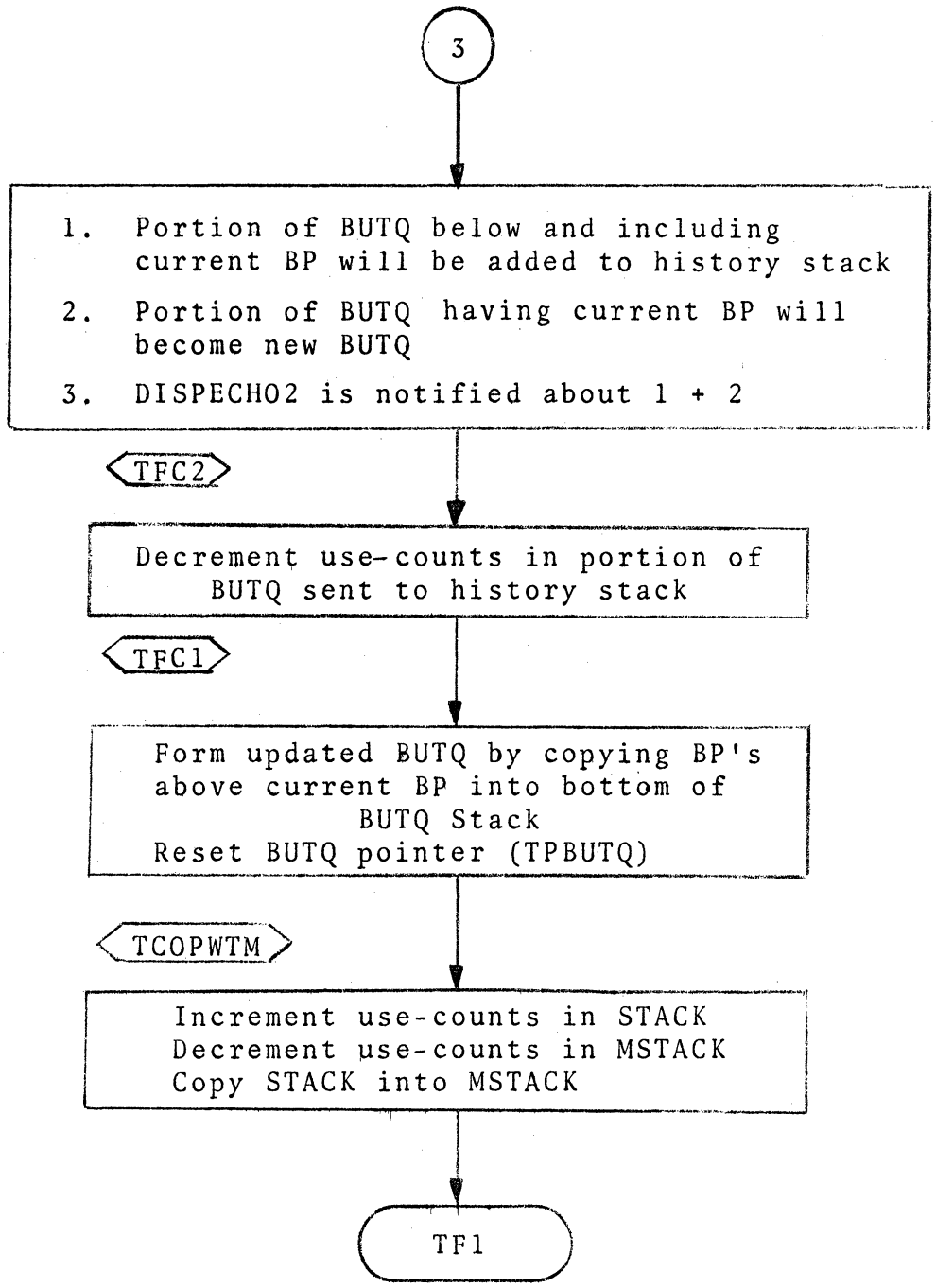
Examples of legal syntax:

```
A (B)+

C(AB-)+

(AB*
```


INTJOHN
HAS EDIT OCCURRED OR IS THIS IN INITIAL CALL? i.e.,RBTEDIT=1?
YES
TCOPMTW
INCREMENT USE COUNTS IN MSTACK
DECREMENT USE COUNTS IN STACK
COPY MSTACK INTO STACK
RESET BUTQ POINTER (TPBUTQ)
TF11
NO
RESET PAREN LATCH TPRNSW ← 0
RESET LITNEXT LATCH TLTNSW ← 0
RESET MACRO LATCH TMSW ← 0
COMPUTER POINTER TO TOP OF BUTQ;
i.e., TOPBQ ← BUTQ(0)•4+(=A(BUTQ))
TF1
RESET SPECIAL OPERATOR LATCH TOPSW ← 0
TF3
TPBUTQ > TPBQ ?
YES
TRET
RETURN TO INTRON
NO
TPBUTQ ← TPBUTQ+4
GR1 ← BUTQ(TPBUTQ)
TF5
INCREMENT USE COUNT OF STOP IN GR1
TPSTACK ← TPSTACK+4
STACK(0) ← STACK(0)+1
1


1
STACK(0) > 6000 ?
YES
ERROR COMMENT STACK OVERFLOW
ERROR
TQ1
NO
STACK(TPSTACK ← GR1
TLTNSW = 0 ?
NO
TLTNSW ← 0
TF7
TQ2
TF9
YES
SYMDCD CLASS
MACRO
TEST CLASS
OPERAND UNDEFINED
TF71
TF4
TF6
SET MACRO LATCH i.e., TMSW ← 1
OPERATOR
OPERATOR = LITNEXT ?
YES
TLTNSW ←1
TPSTACK ←TPSTACK-4
STACK(0)←STACK(0)-1
TF7
TF40
NO
SYMCHK IS OPERATOR SPECIAL ?
YES
SET SPECIAL OPERATOR LATCH; i.e.,TOPSW← 1
NO
TFKIP
RESET PAREN LATCH, TPRNSW ← 0
CALL INTKIP
2


2
TPSTACK ← GR0
KIP RETURNS POINTER TO
TOP OF STACK IN GR0
IS STACK EMPTY?
YES
TF7
NO
TF9
TF71
TPRNSW = 0 ?
NO
TGERR
COMMENT
SYNTAX ERROR
ERROR
YES
IS CURRENT BP = ')' ?
YES
TG1
TPRNCNT ← TPRNCNT-1
NO
IS CURRENT BP = '(' ?
NO
YES
TPRNSNT ← TPRNCNT+1
TPRNCNT < 0 ?
YES
NO
SET PARENT LATCH
i.e., TPRNSW ← 1
TF7
TMSW = 0 ?
YES
TF2
TOPSW = 0 ?
NO
3
YES
TF3
NO
CALL MACROGET
NO
GR1 ← GR0
GR0 = 0 ?
YES
TF10
RESET MACRO LATCH
i.e., TMSW ← 0
TF5


3
1. Portion of BUTQ below and including current BP will be added to history stack
2. Portion of BUTQ having current BP will become new BUTQ
3. DISPECHO2 is notified about 1 + 2
TFC2
Decrement use-counts in portion of BUTQ sent to history stack
TFC1
Form updated BUTQ by copying BP's above current BP into bottom of BUTQ Stack
Reset BUTQ pointer (TPBUTQ)
TCOPWTM
Increment use-counts in STACK
Decrement use-counts in MSTACK
Copy STACK into MSTACK
TF1

8.3 SYMBOL TABLE MANAGEMENT ROUTINES P.Wilcox J.Jonekait

8.3.1 Symbol Table Entries

Each symbol has one symbol table entry which contains various pieces of information about the symbol. The following is a description of the fields contained in a symbol table entry and their uses.

NAME (SYMNAM - 8 bytes): contains the 8-character EBCDIC name of the symbol.

VALUE (SYMVAL - 4 bytes): full-word pointer to the "value" of the symbol. It may be used for various purposes.

DOPE VECTOR (SYMDV - 4 bytes): full-word pointer to symbol's "dope vector" or "extended attributes." May be used for various purposes.

BUFFER ID (SYMBFRID - 4 bytes): full-word pointer reserved for the display buffer ID for the symbol.

USE COUNT (SYMUSC - 2 bytes): half-word "use count" for the symbol. This will be used to determine when and if a symbol should or may be removed from the symbol table.

DISPLAY LINK (SYMDLNK - 2 bytes): contains a half-word Symbol Table Index (STEI) which may be used to link symbol table entries together in a chain. When this field is referred to in a macro call, a full-word STEP must be given since the macro will convert it to or from the STEI. This field was originally requested by display people but they have since disowned it so it may be used for various purposes.

DIMENSION (SYMDIM - 2 bytes): half-word dimension of operands. Number of elements in a vector; number of elements in each of the two vectors of a vector-pair. Not used by scalars.

MODE (SYMMOD - 1 byte): The bits in the mode byte are used in conjunction with operands to indicate the form of the data associated with the symbol. The bits in this byte may be divided into three groups. The first group shows the mode of the symbol:

SYMREAL	symbol is real
SYMINT	symbol is integer
SYMCPX	symbol is complex

The second group gives the structure of the data:

SYMSCALR	scalar
SYMVEC	vector
SYMVPAIR	vector-pair

The third group is a single bit which gives the length or form of the individual elements:

SYMLNG	1=long, 0=short form

In each group, at most one bit may be set at any time. It is possible for none of the bits to be set; this is the undefined condition and is used when the symbol is first created.

CLASS (SYMCLAS - 1 byte): The bits in the class byte are switches which determine the basic nature of the symbol table entry. The following bits are defined:

SYMOPND symbol is an operand

SYMOPR symbol is an operator

SYMMAC symbol is a macro definition

SYMUDEF class of symbol is undefined

SYMUNU this symbol table entry is not in use

SYMSPC special attention

The first five bits listed are mutually exclusive, so only one (exactly) may be set at any time. SYMUNU is internal to the symbol table management routines and is not available to others through macros. The last bit is used by the interpreter.

TYPE (SYMTYP - 1 byte): The bits in the type byte indicate if a symbol may be changed or destroyed. The bits are:

SYMSYS system symbol

SYMUD user-defined symbol

SYMTMP temporary symbol

SYMRDO read-only symbol

ONE AND ONLY ONE of the first three bits should be set for each symbol in the symbol table. System symbols are never destroyed. If an attempt is made to do so, nothing is done unless the symbol is an operand, in which case it is reset to undefined class and mode. User-defined symbols are destroyed only when a request is made to do so. Temporary symbols are destroyed automatically when they are no longer being referred to anywhere. Read-only symbols should not be changed, but the symbol table routines do not do any checking for this.

EXTRA ROOM: At present, there are 32 bytes in a symbol table entry, of which all but three are being used. Currently the symbol table is aligned so that each symbol table entry pointer points to a 32-byte boundary and therefore has zeros for its five low-order bits.

8.3.2 Macro Descriptions

Parameters

Many of the symbol table management macros accept parameters which correspond to items in a symbol table entry (STE). These parameters may be divided into two classes:

Class I parameters refer to particular fields in a symbol table entry. For example, DOPE refers to the full-word pointer to the dope vector. When one of these parameters is used in a macro call, either the entire field is replaced (from GR0) or else the entire field is returned (in GR15), depending on the macro used. If the field is less than a full word long, it is right-adjusted in the register and high-order bits are either ignored or set to zero. Only one Class I parameter may be given in a macro call.

Class II parameters refer to particular bits in the TYPE, MODE, or CLASS bytes of the symbol table entry. When one of these parameters is used, only the individual bit associated with the parameter and possibly directly conflicting bits are affected. For example, if the para-

meter SYS is given with the SYMSET macro, then the system symbol bit will be set while the user and temporary symbol bits will be cleared, but no other bits in the TYPE byte will be changed. Any number of non-conflicting Class II parameters may be given in a macro call. Class I and Class II parameters referring to the same byte should not be used in the same call.

Class I Parameters

PARAMETER	FIELD	LENGTH
VALUE	value pointer	full word
DOPE, EXTATR	dope vector pointer	full word
DBUF#	display buffer ID	full word
TYPE	type bits	one byte
MODE	mode bits	one byte
CLASS	class bits	one byte
DLINK	STE link pointer	full word (STEP only)
DIM	dimension	half word
NAME	EBCDIC name	full-word pointer to 8-character name
USCNT	use count	half word

Class II Parameters

PARAMETER	BYTE	ATTRIBUTE*
OPER,OPERATOR	CLASS	operator
OPND,OPERAND	CLASS	operand
MACRO	CLASS	macro definition
UND	CLASS	undefined class
SPEC	CLASS	special attention
-SPEC	CLASS	not special
SCALAR	MODE	scalar operand
VECTOR	MODE	vector operand
VPAIR	MODE	vector-pair
INTEGER	MODE	integer
REAL	NODE	real

COMPLEX,CMPX	MODE	|complex
SYS,SYSTEM	TYPE	|system (permanent) symbol
USER	TYPE	|user-defined
TEMP	TYPE	|temporary symbol
CONST	TYPE	|read-only
VRBL	TYPE	|may be changed

*Vertical bars join mutually exclusive attributes.

8.3.3 List of Macros

SYMCRE

SYMCRE is used to create a symbol table entry and assign to it initial attributes. GR1 is assumed to point to an 8-byte region containing the name of the symbol to be created. If the symbol is to be temporary, the name may begin with the characters "#SYM". The last four characters of the name will then be supplied by the symbol table routines so that other programs will not have to worry about creating unique names for temporary symbols they use. If a name is given which is already in the symbol table, a return code of 4 will be given in GR15. Normally, the return code is zero. In either case, the STEP of the new or old symbol is returned in GR1. If there is no more room in the symbol table, a return code of 8 is used and GR1 is zero.

Initial attributes may be set by giving parameters acceptable to the SYMSET macro. Any fields not specified in this manner will be set to the value of the corresponding field in the global default entry, except for the mode byte which is set to zero (undefined).

```
PROTOTYPE:      label    SYMCRE  [par1.] [,par2,...]
EXAMPLE:                 LA      1,NAMEREG
                         SYMCRE SYS,OPND;REAL, VECTOR
```

SYMDES

SYMDES is used to destroy a symbol table entry. It calls upon the symbol table management routine SYMDS. If the STEP in GR1 refers to a system symbol, the symbol is reinitialized (in the same manner as when a symbol is created). If the STEP refers to a user or temporary symbol, the symbol is removed from the symbol table. Note that the symbol is not removed from the screen by the symbol table management routines; this must be done by the routine which wishes to destroy the symbol before SYMDES is called.

If the STEP in GR1 does not point to a valid STE, a return code of 4 is given; otherwise a return code of 0 is provided.

```
PROTOTYPE:      label    SYMDES
EXAMPLE:                 CALL BUFBOD remove from screen
                         L    1,STEP
                         SYMDES
```

SYMFIND

SYMFIND is used to search the symbol table for an STE with a known name. It calls upon the symbol table management routine SYMFS. GR1 is assumed to point to an 8-byte region containing the name of the symbol. If

it is found a STEP is returned in GR1 and the return code is 0. Otherwise a return code of 4 is given.

```
PROTOTYPE:      label    SYMFIND
EXAMPLE:                 LA     1,PLUS
                         SYMFIND
```

SYMDCD

SYMDCD decodes the CLASS, MODE, or TYPE byte of a symbol table entry and returns it in GR15 in a form suitable for use as a branch table index. The following are legal parameters only one of which may be given in one call: CLASS, MODE, TYPE, STRUCTURE (each may be abbreviated to its first letter). If more than one parameter is given in the call to this macro, all after the first are assumed to be statement labels to be used in a branch table which the macro will construct. If some of these labels are omitted, then the corresponding branch table entry will point to the first location after the macro expansion. If all the labels are missing, no branch table is constructed.

The branch indices generated are as follows:

GR15	TYPE	CLASS	MODE	STRUCTURE
0	system	operand	real	scalar
4	user	operator	integer	vector
8	temp	macro	cmplx	vector-pair
12		undefined	undefined	undefined

```
PROTOTYPE:    label    SYMDCD    parl  [,loc...]
EXAMPLE:               SYMDCD  MODE,REALLOC,,ERLOC
```

produces the same code as

```
                       SYMDCD  MODE
                       B    *+4(15)
                       B    REALLOC
                       B    *+8
                       B    ERLOC
```

Use Count Management

All symbols of user-defined or temporary types have associated with them a use count. This should be incremented by one at the beginning of each usage of the symbol and decremented at the termination of said usage.

SYMREF

SYMREF is used to increment the use count by one. No action is taken if the symbol is a system symbol.

SYMDLE

SYMDLE is used to decrement the use count. If the use count is found to be zero after decrementing the macro, SYMDES is called to destroy the symbol. As before, no action is taken for system symbols.

Attribute Manipulation

SYMSET

SYMSET is used to set an attribute(s) of a symbol table entry. Any Class 1 parameter and any combination of Class 2 parameters are legal. There is no error-

checking to see if contradictory parameters are specified. Parameters are processed left to right. If a symbol is an operand (or is changed to an operand in the same SYMSET call) and the mode byte is to be changed, the symbol table manipulation routine SYMSMU or SYMSMS is called and appropriate manipulations are performed to allocate space for the value of the symbol.

```
PROTOTYPE:    label    SYMSET  par1[,par2][,par3...]
EXAMPLE:               SYMSET  OPER,DLINK
```

SYMGET

SYMGET is used to get the value of any symbol table entry item. Any Class 1 parameter may be specified. The value of the item is returned in GR15.

```
PROTOTYPE:    label    SYMGET  par
EXAMPLE:               SYMGET  DIM
```

SYMCHK

SYMCHK is used to determine if a symbol table entry has certain specified attributes. Any Class 2 parameters may be specified. The last parameter must be the address to which a branch will be taken if any one of the attributes is not associated with the symbol table entry, i.e., if multiple attributes are specified success will be achieved if all of the attributes specified are present.

```
PROTOTYPE:    label    SYMCHK par1[,par2][,par3...],LABEL
EXAMPLE:               SYMCHK OPND,VECTOR,READ,BADPAR
```

SYMGINDX

SYMGINDX converts a half-word symbol table entry index (STEI) into a STEP. A parameter is accepted which specifies the register in which the STEI exists. The STEP is returned in the same register. GR1 is assumed if no register is explicitly specified.

SYMGPTR

SYMGPTR converts a full-word STEP to a STEI. Register conventions are as in SYMGINDX.

SYMERR
SYMDIAG
SYMTRACE
SYMTRAC

These macros are NOPS as anticipated demand for them did not materialize.

SYMSETG
SYMGETG
SYMCHKG

These macros perform the functions of the macros SYMSET, SYMGET, and SYMCHK on the global symbol table entry which contains current defaults.

8.3.4 Symbol Table Management Routines

The symbol table management routines which are normally used during execution are contained in two assemblies, each containing one control section. The first CSECT is named "SYMSTM1" and has the following entries:

SYMCS — create symbol

SYMDS — destroy symbol

SYMFS — find symbol

SYMINIT1 initialization

SYMTABLE PSECT and symbol table

The second CSECT is named "SYMSTM3" and has the following entries:

SYMSMU set mode to user byte

SYMSMS set mode to macro-formed byte

Also, there is a CSECT which contains entries to print-out symbol table entries or dump the whole symbol table. This will be described below. There is also a program to test the symbol table macros and programs, but it is not described here.

SYMCS

The entry SYMCS in SYMSTM1 is used to create a symbol table entry given the name of the symbol. GR1 must contain a full-word pointer to the first byte of the 8-character name. GRS 13,14, and 15 are used in the normal manner. On return, GR1 will contain a STEP, and GR15 will have a return code of 0 or 4. RC=0 indicates that a new symbol table entry (STE) has been created and the symbol table entry pointer (STEP) is in GR1. If RC=4, then a symbol with the same name already exists and the STEP in GR1 points to that old symbol.

The operation of the routine is as follows: If the name of the symbol indicates that it is not to be a temporary, that is, its first four characters are not the same as the four characters in location "SYMTPFX" (which is addressable relative to "SYMTABLE"), then a search

of the entire symbol table is made to see if the name is already used on an existing symbol. This search is made starting at the end of the symbol table and going toward the beginning. (The end of the symbol table means the last STE in use, not the end of the storage reserved for the table.) During the search, a check of each STE is made to see if it is used. If it is not, then its STEP is saved so that the STE may be used for this symbol if necessary. If a symbol with same name is found, its STEP is put in GR1 and a return with RC=4 is made. Otherwise an STE is formed for the symbol either at the end of the table or else in a previously unused STE found during the search (if one was found). This new STE is initialized by filling all fields from the global symbol table entry except the name, which gets the new name, and the mode and dimension which are set to zero.

If a temporary symbol is to be created, no check is made to see if it already exists, since this is impossible (hopefully) because this routine creates a unique name. A search is made starting from the beginning of the symbol table to find the first unused STE, which may be at the end of the table. In order that room for TEMPS may be found quickly, the first few symbol table entries are reserved exclusively for TEMPS. The exact number reserved is an assembly parameter and is the value of the symbol "SYMSTERT" which currently

is 5. When an unused STE is found, it is initialized just as other STEs except that its type is set to TEMP and the second four characters of its name are an index which is incremented each time a TEMP symbol is created. This index starts at 0001 and goes to 9999, at which point it returns to 0001. The index 0000 is reserved for the global entry which has the name "#SYM0000". This routine is called from the macro "SYMCRE".

SYMDS

The entry SYMDS in SYMSTM1 is used to remove a symbol table entry from the symbol table. GR1 must contain the STEI for the symbol, and GRS 13, 14, and 15 are used in the normal manner. First, a check is made to see if the STE pointed to is in use and if it is not, a return with RC=4 is given. If the symbol is a system (permanent) symbol that is not an operand, then nothing is done and a normal return is made. If the symbol is a system symbol that is an operand, then it is reinitialized (in the same manner as when a symbol is created) with its type set to system. Any other symbol has its class set to unused so that it may be reused later. Reinitialized symbols and destroyed symbols have their mode byte set to zero, thereby releasing storage assigned to their value pointer (see SYMSMU write-up).

If a symbol was removed from the table, a check is made and the pointer to the end of the symbol table is

set to point to the last STE actually currently in use. (If the number of TEMPS defined is small enough so that they all fit in the room reserved for them, and if no other symbols are defined, then the end pointer points to the first block available to non-TEMP symbols.)

This routine is called by "SYMDES" macro expansions.

SYMFS

The entry "SYMFS" in SYMSTM1 is used to search the symbol table for a symbol with a given name. GR1 must contain a full-word pointer to the first byte of the 8-character name of the symbol to be found. The search is straightforward, starting at the beginning of the symbol table and including the reserved TEMP region but not the global entry. If the symbol is found, a return is made with its STEP in GR1 and RC=0 in GR15. If the symbol is not found, GR1 is set to zero and RC=4.

This routine is called by the "SYMFIND" macro.

SYMINIT1 and SYMTABLE

The entry, "SYMINIT1" in SYMSTM1 is the initialization entry for the symbol table management routines. It must be called during initialization before any attempt is made to create symbols. The entry "SYMTABLE" is the PSECT name for all the symbol table routines. It contains pointers to the beginning of the symbol table, the end of the table, the global entry, and the symbol table management error exit routine (which does

not exist and is not used). There is also the temporary symbol prefix, some scratch storage, and two save areas. Following all this is the symbol table itself. The entries in the symbol table are forced to a 32-byte alignment. The calculation of the first such address available to the symbol table is done during initialization when the various pointers are also initialized. The global STE is physically the first entry in the symbol table although logically (to the various search routines) it is not contained in it. The global entry is initialized when it is created but it may be changed later, during or after initialization by other routines. All the STE's used exclusively for TEMPS as well as the first non-TEMP STE have their class set to unused during initialization.

SYMPTO

The CSECT "SYMPTO" contains two entry points: "SYMTPO" (which is the same address as SYMPTO) and "SYMSPO". The latter requires an STEP in GR1 and prints out on SPRINT the state of that STE in a readable form. The first entry dumps the whole symbol table by repeatedly calling on the second. Both entries establish their own addressability and require only GR14 to be set to the return address. A normal return preserves all registers except GR15.

Assembly Parameters

The following constants appear in SYMSTM1 and are used to establish various characteristics of the symbol table entries and programs.

The location "TPFX" contains the four characters which are initially used to denote that a TEMP symbol is to be created. During initialization, these four characters are moved into the location "SYMTPFX" in the PSECT which may be changed at any time.

The location "INIB" contains the initial values which are loaded into the global STE when it is created.

The symbol "SYMSTETL" has a value which is the total length of a symbol table entry (in bytes) and is currently 32. Changing this value implies that the symbol table DSECT and STE alignment calculation has been or should be changed.

The symbol "SYMSTEIL" has a value which is the initialization length for STE's. This is the number of bytes which are moved from the global entry to the other entries when they are initialized. Currently this is 24 bytes, since the name is not moved. If this symbol value is changed, probably other codes in SYMSTM1 will have to be changed to indicate which bytes are to be moved.

The symbol "SYMSTERT" has a value which is the number of STEs reserved exclusively for TEMP symbols at the beginning of the symbol table. Since TEMP symbols are

created and destroyed quite often during processing, this reserved room at the beginning makes it more likely that the CREATE routine can find space for a TEMP easily without looking through the whole symbol table for an unused STE. If this reserved room is filled up, then TEMP symbols will be placed in the first unused STE in the rest of the symbol table just as any other symbol. Currently, the value of SYMSTERT is 5, but experience may show a larger figure to be appropriate.

Currently, 16 pages of core are assigned for the symbol table and PSECT. This allows room for over 2000 symbols and is probably far more than will ever be needed. Contrary to what other write-ups may say, the CREATE routine (SYMCS) at this time does not check to see if there is room to add another symbol to the end of the symbol table.

SYMSMU and SYMSMS

The entries "SYMSMU" and "SYMSMS" in CSECT "SYMSTM3" are called by the SYMSET macro to set the mode byte of a symbol. GR1 must contain a STEP for the symbol to be changed and GRS 13, 14, and 15 are used in the normal manner. GR1 is not changed. The entry "SYMSMS" is used if the mode byte to be entered was generated by the macro as the result of the use of Class II parameters referring to the mode byte. "SYMSMU" is called if the Class I parameter "MODE" is used. The only difference between the

two entries is that SYMSMU checks the class of the symbol. If it is not an operand, then the given mode byte is stored and nothing else is done. If the class is operand or if "SYMSMS" is called, then the routine releases old storage associated with the symbol (value) and gets new storage in accordance with the new mode byte.

The operation of the routine is as follows. If the new mode byte is the same as the old, nothing is done. If the old value pointer is zero or if it points to the dope vector pointer field of the symbol, then no storage is released. Otherwise the storage pointed to by the value pointer is released using "FREESPAC". If the new mode byte is zero, the dimension, the value pointer, and the mode byte are set to zero and a normal return is made (RC=0). In order to assign storage, both the true mode (integer, real, or complex) and the structure (scalar, vector, or vector-pair) must be known. If the new mode byte indicates both of these, then that is used. If either field of the new mode byte is zero (undefined) then the corresponding field from the old mode byte is used. If this also is zero, then the field from the global entry is used. When this has been settled, the space needed for the symbol value is computed. The dimension field from the global symbol is always used if the symbol is vector of vector-pair and is stored in

the dimension field of the symbol. If the symbol is a scalar, the dimension is set to 1. If four bytes or less are needed, then the value pointer is set to point to the dope vector pointer field of the symbol. If more room is needed, GETSPACE is called to obtain it. The return code for this routine is the same as the return code from GETSPACE or zero if GETSPACE was not called. Note that there is no checking for validity of the new mode byte, and if too many bits are set, the room allocated depends on the particular sequence in which the program checks the bits, and may well not be the largest amount indicated.

8.4 LIGHT-PEN MANAGEMENT ROUTINES F. Stephenson R. Brender

Relationship to Graphic Support Routines

These asynchronous attention routines are required to be able to queue light-pen hits to be later processed on a first-in first-out basis. This is not possible with the graphic system as described in the MTS literature.

A new routine, SETANLZ, has been added to the graphics support. This routine is very like that described in IBM literature except that it is an initialization call only. Return is to the calling routine immediately after making interrupt connections with the supervisor. When an attention occurs, control passes to the device-handling routines (LPDETECT, KBFNCDET, ENDORSEQ, and ASYNCER) on a true interrupt basis with

respect to the main program. These routines must return to SETANLZ in normal fashion. By convention SETANLZ will start buffer regeneration (redundantly perhaps) before expiring.

This leaves only the problem of "task time—interrupt time" coordination via the queue. When the queue is empty on entry to routine LPMDEQLP a flag called FLAG is set non-zero and WAYT state (via a SVC) is entered. The last task of the interrupt routines when a STEP has been added to the queue is to post that FLAG (set FLAG to zero) and hence awaken the task-time routine.

Addressability

Common addressability is set up at all entry points with

REG 10 = address of graphic DCB

REG 11 = address of fake PSECT (also known as LPMPSECT)

REG 12 = current CSECT (named LPMCSECT)

Note on Documentation

A name in parentheses after a STEP number in the functional description of these routines gives the program label roughly corresponding to the STEP description.

LPM INITIALIZATION

Name: LPMINIT

Purpose: To initialize internal tables and data structure of the light-pen management routines.

Calling Sequence: OS (I) R type

Entry: No parameters.

Return: No parameters. RC = 0.

Functional Description:

1. SETANLZ is called to initialize the graphic support software for handling asynchronous interrupts.
2. Various error counts are zeroed.
3. The initial values of the queue of STEPs are set to zero.
4. The light pattern in the function keys is set up.
5. The names 'FUNC00 ',...,'FUNC29 ' for the function buttons are defined to the symbol table and the corresponding STEPs entered in the FUNCKEY table.

PROCESS LIGHT PEN DETECT

Name: LPDETECT

Purpose: To obtain STEI of an item identified by light pen and place the corresponding STEP into the light-pen queue.

Calling Sequence: OS (I) R type

Entry: Parameters in 8-byte region established by previous call on SETANLZ

Return: RC = 0 All okay.

RC = 4 Something wrong, e.g., could not find an STEI.

Functional Description: LPDETECT

1. (LPDETGRF) Read into core 2000 bytes from 2250 buffer starting at the next lower even buffer address location as given in OUTPUT+2.

2. (LPSCAN2) Scan looking for two-byte sequence = X'2ACO' which is a "GNOP4"

3. If not found, return to SETANLZ without further action. SETANLZ will start regeneration of display and total effect to user will be null.

4. (GOTNAME) If found, then interpret the next two bytes as the STEI of an entity. Convert to STEP to store into queue.

5. (GOTNAME2) If queue is full, then exit. Otherwise, place STEP in queue. Then POST the FLAG for the LPMDEQLP entry, and exit to SETANLZ.

PROCESS END-ORDER-SEQUENCE

Name: ENDORSEQ

Purpose: To process end-order-sequence condition from 2250 display.

Calling Sequence: OS (I) R type

Entry: Parameters in 8-byte region established by previous call on SETANLZ

Return: RC = 0 All okay.

Functional Description: ENDORSEQ

1. Increment the counter EOSCOUNT by 1.

2. When EOSCOUNT exceeds = F'256' call ERROR, else return to SETANLZ.

This action is intended for diagnostic purposes. An end-or-sequence command should never occur. If it does, regeneration will be forced 256 times allowing the user at least 10 seconds to examine the partial image.

PROCESS ASYNCHRONOUS ERRORS

Name: ASYNCER

Purpose: To process asynchronous error conditions in 2250 display.

Calling Sequence: OS (I) R type

Entry: Parameters in 8-byte region established by previous call on SETANLZ

Return: RC = 0 All okay.

Functional Description: ASYNCER

1. Record the buffer location where the error occured in DATACHK. Return to SETANLZ.

Comments: An error message via SERCOM will also be implemented.

PROCESS FUNCTION KEYBOARD DETECTS

Name: KBFNCDET

Purpose: To process keyboard detects from either function or (real) manual keyboard.

Calling Sequence: OS (I) R type

Entry: Parameters in 8-byte region established by previous call on SETANLZ.

Return: RC = 0 All okay.
RC = 4 Something wrong, i.e., END or CANCEL from keyboard.

Functional Description:

1. The keyboard sense data is read to determine cause of interrupt.

2. If interrupt from manual (real) keyboard, simply ignore and return to SETANLZ.

3. (FNCDET) Else from function keyboard so the keynumber is extracted and used to index into both FUNCKEY and KEYCODE tables.

4. If KEYCODE entry is zero, then FUNCKEY contains the STEP for the function depressed. Load this and go step 5 of LPDETECT description.

5. (PRECALL) If KEYCODE entry is = X'80000000', then FUNCKEY contains the address of an immediate routine. This routine is called. It must return before the display regeneration is set up since all 2250 interrupt-handling is suspended.

GET 'STEP' FROM LIGHT PEN ROUTINES

Name: LPMDEQLP

Purpose: To obtain STEP of item pointed to by user from a queue of such in FIFO order.

Calling Sequence: OS (I) R type

Entry: No parameters

Return: GR0 contains STEP
RC = 0 All okay.
RC = 4 Something wrong somewhere.

Routines Required: ANALYZ & SETANLZ from *GPAKLIB

DSPECH01 (Echo line entry) from display section

EXTERNAL SYMBOL - DCBADDR = DCB and STRTADDR from buffer management.

Functional Descriptions:

(Note: ECH is pointer to next entry not in echo, BEGIN is pointer to next entry not on interpreter stack.)

1. On ENTRY, compare ECH to END. If equal, the echo line is up-to-date, so branch to (ALLECH).
2. To update echo, branch to (FILLTAIL) to get linear vector of queue. Call DSPECHO1 with address of parameter list in GR1. (Parameter list is two full-words: first is address of full-word length of the vector; second the address of the first word of the vector.)
3. (ALLECH) Compare BEGIN and END. If equal, the queue is empty so the program goes into wait state until another interrupt occurs which makes a queue entry. The echo line is updated after this entry.
4. (NTINT) The STEP of the next queue entry is put in GR0, BEGIN is incremented, and the routine returns with RC = 0.
5. (FILLTAIL) Given the pointer of where to start in the queue, entries are moved into a linear vector. A full-word count is computed and stored, as required for the parameter list. The value of ECH is updated and control returned. Also used by LPMNEWEC.

GET ECHO LINE NOT YET PASSED TO INTERP AFTER AN EDIT

Name: LPMNEWEC

Purpose: To obtain linear vector of that part of the button queue not already passed to the interpreter.

Calling Sequence: OS (I) R type

Entry: No Parameters

Return: GR1 contains pointer to PARLIST
PARLIST = A (LENGTH OF VECTOR)
A (VECTOR)
GRC = 0 All okay.
GRC = 4 ERROR

Routines Required: None

Function Description:

1. Picks up BEGIN and branches to (LIST).
2. (LIST) If queue is empty, returns with zero put in vector length. If queue is not empty, branches to (FILLTAIL) as in LPMDEQLP to get linear vector, and returns.
3. RETURN to calling program with address of parameter list in GR1, GRC = 0.

8.5 KEYBOARD AND NUMERIC DISPLAY ROUTINES

M.Feldberg R.Taylor R.Nicholls

The mandate for this group was to translate alphanumeric information derived from various parts of the system into order programs to be passed to the screen management routines for display on the screen and to 'manage' the order programs thereafter; that is to say, the particular order programs were to be cataloged. These order programs were written in a standard manner and were identifiable via an index subscripted to each program.

DSPKBRD

This provides the keyboard display and the initialization call to the echo line. It also provides the func-

tion definition (via SYMODEFS) for the scrolling, numeric display, and screen-clearing routines. At the time of the call to this module, a temporary file,-MESSFILE, is created and is used to keep all messages which are displayed on the screen and the entire button queue history.

DSPMESS

This contains both the message display routines and the message erasing routines. Figure 8.5.1 is a flow chart of the subroutine DSPMESS.

DSPNUMV

This module contains a large number of entries but we will describe only two in detail as the rest manipulate data in a simple way after it has been set up by the two routines to be described. The two to be described are:

DSPNUMV
DSPORDER

Each numeric display comes under the control of a control block (with the dummy structure VECTATT). This has the structure:

STEP of variable displayed

Starting address of value vector

Address in value vector of start of present display

Last address in vector

Buffer ID of display

Figure 8.5.1 Flow Chart of Subroutine DSPMESS

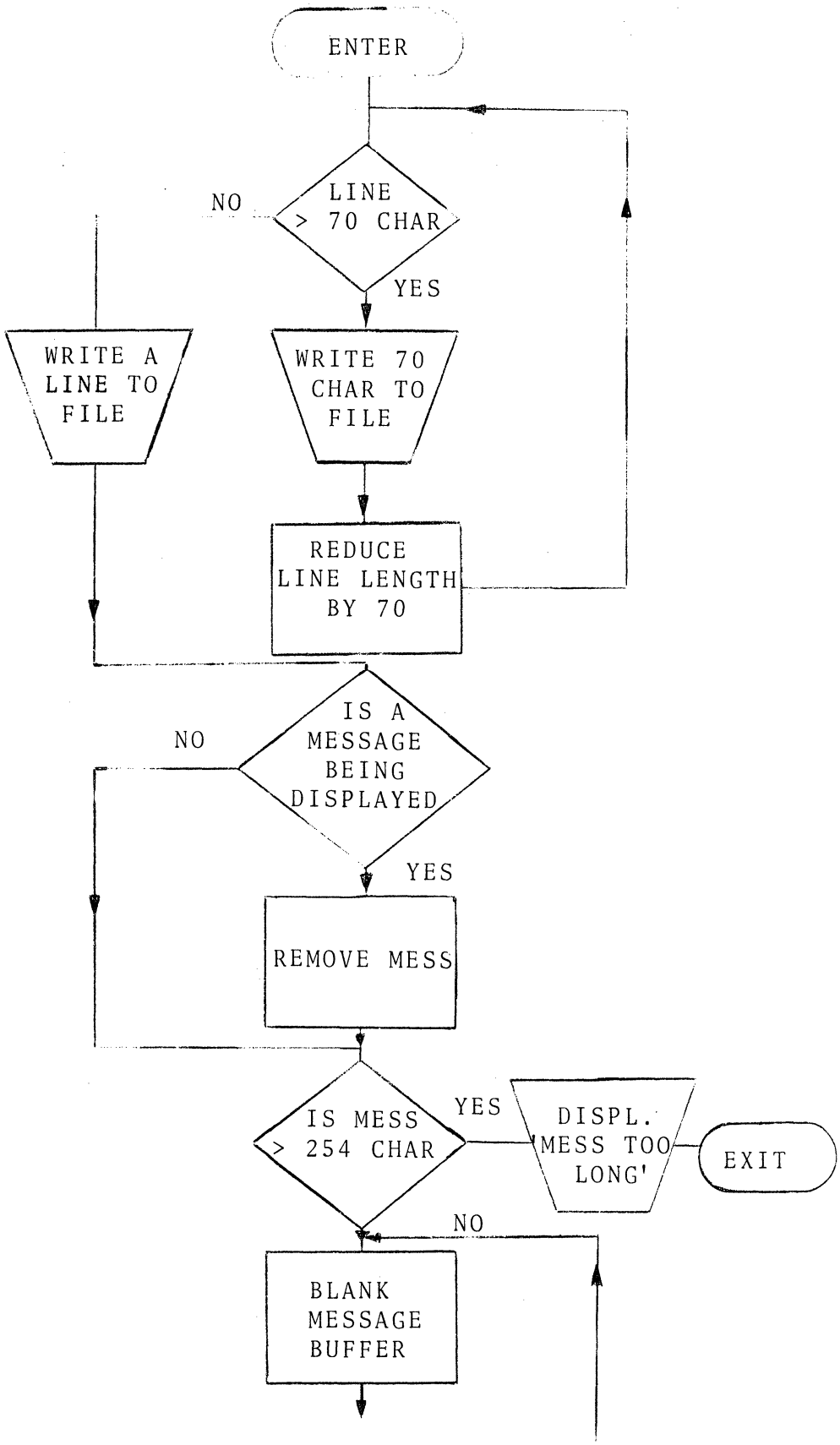
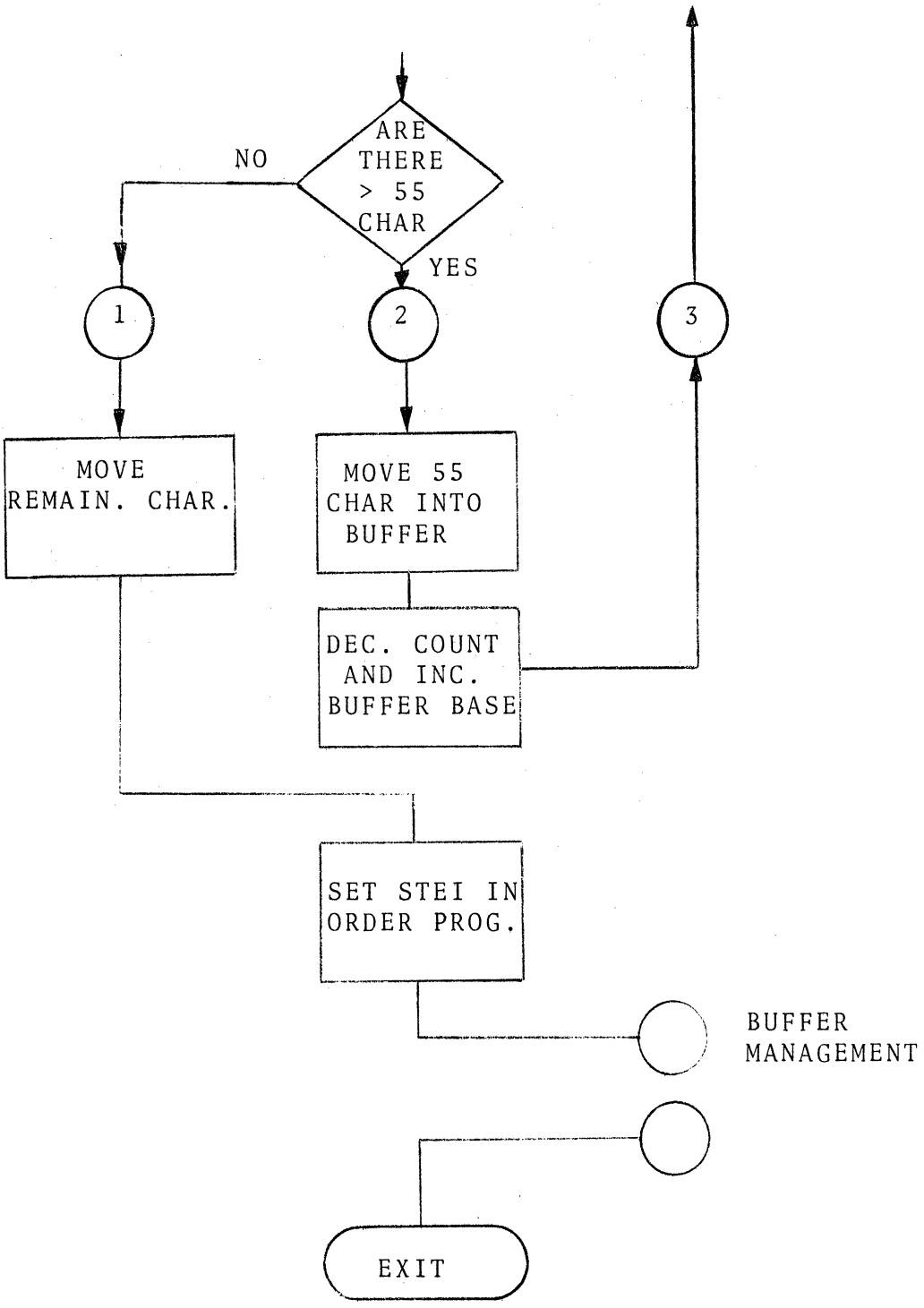


Figure 8.5.1, continued



Dimension of vector (1 if a scalar)

Pointer to head of list of a control block

Eight bytes of positioning orders for start of order program.

This structure is used for both vector and scalar displays. Three blocks are provided each kind of display; they are queued and used sequentially until all are in use. If at any time the display corresponding to a given block is erased, the block is placed on the back of the queue for later use. The queue structure is:

Byte index of first available pointer

Pointer to control block

"

"

F'-1'

The -1 limits the length of the list.

In order to obtain an order program corresponding to a given set of values (or more properly, to display the values corresponding to a given STEP), a control block is acquired, the relevant data are put into the block, and the block is passed to DSPORDER. To scroll a vector display, only the address of the head of the display in the control block need be changed before passing the address of the block to DSPORDER.

System Sub-button Description

Name: DSPLPROC (part of the DISVALUE button)

Purpose: To display a macro definition in the working area of the screen.

Calling Sequence: OS(I) S type

Entry: GR1 contains location of a list of 2 adcons

1. A(STE of the procedure)
2. A(Value of the procedure)

Return: RC=0 Procedure is displayed
RC=4 Procedure is not displayed

Instructions for use:

To display the definition of MACRONAME, the user should point at

LITNEXT MACRONAME DISVALUE

in that order with the light pen. The macro definition will be displayed in the current working area of the screen. The formal parameters of the definition will appear in parentheses.

Functional Description:

1. Pick up parameters using pointer in GR1.
2. Get the length (number of STEPs) from the type entry in the symbol table.
3. Go through the double entry table which contains the macro definition starting at A(Value+#dummy args*4) to get the STEPs of the definition. (See macro write-up for details.)
4. Place these STEPs in a local list. Record the number of STEP in the list.

5. Use a STEP to obtain an 8-byte EBCDIC name and build up a 2250 order program using these characters. (Trim all but one trailing blank and insert NEWLINE and spacing characters where appropriate.)
6. Loop on 5 until all STEPs processed.
7. Compute the length of the 2250 order program in bytes and call BUFBINGO.
8. Increase the use count of the macro.
9. Enter the STEP and Buffer ID of the displayed macro into the local table of macro definitions displayed.
10. Exit with RC=0.

Name: DSPERSPR (part of the ERASE button)

Purpose: To delete the display of a procedure from the working area.

Calling Sequence: OS(I) S type

Entry: GR1 contains the location of a list of 2 adcons

1. A(STE of the procedure)
2. A(Value of the procedure)

Return: RC=0 Procedure erased

RC=4 Parameter given was not a macro

Instructions for use:

The user should point at
MACRODEFINITION ERASE
in that order with the light pen. The displayed macro definition will disappear from the screen working area.

Functional Description:

1. Pick up arguments using pointer in GR1.
2. Use SYMDCD to see if operand is a macro. If not, exit with RC=4. If so, go to 3.
3. Find the entry pair for this macro in the table being kept of macro definitions displayed. Remove this entry pair (i.e., Buffer ID and STEP) and close up the rest of the table.
4. Call BUFBOD to remove the order program.
5. Decrease the use count of the macro.
6. Exit with RC=0.

Name: DSPCLRPR (part of the CLEAR button)

Purpose: To clear the "working area" of the screen of all macro definitions displayed there.

Calling Sequence: OS(I) R type

Entry: No parameters

Return: RC=0 Done
DSPCLRPR calls BUFBOD with a Buffer ID. If this ID is not legal, a system error is indicated by BUFBOD.

Instructions for Use:

When the user points at the clear button, the various sections which display information in the working section --graphs, numeric display of vectors, macro definitions-- are called and each deletes the order programs that it set up.

Functional Description:

1. Sequentially extract entries from the double entry table of Buffer IDs and STEPs.
2. Call BUFBOD with each ID.
3. Decrease the use count of each macro display deleted by one.
4. Exit when all entries processed.

The following constitutes a list and definition of the present set of entry points into the package provided by keyboard and function display. Given are the external symbols and the calling sequence required at each entry. In some cases, a return or return code is provided and in these cases it is noted.

Name:	DSPKBRD
Purpose:	To display the predefined virtual keyboard symbols on the screen.
Calling Sequence:	OS (I) S type
Entry:	GR1 contains the location of a list of full word entries. The first entry is a full word count of the number of following entries. The rest are STEPs of the predefined symbols.
Return:	RC=0 Screen displayed RC=4 Screen not displayed
Notes:	The format of the screen will be entirely determined by the order of the list of STEPs. The symbols will

be displayed in large characters across the top of the screen, double-spaced and staggered line-by-line.

Name: DSPNUMV

Purpose: To display a vector in numeric form.

Calling Sequence: OS (I) S type

Entry: GR1 will contain the location of a list of adcons which will be prepared in the standard function call format. The list of adcons will be:

1. A(STE of vector to be displayed)
2. A(Vector)
3. A(Temporary STE)
4. A(Temporary vector)

Exit:
RC=0 Vector displayed
RC=4 Vector not displayed

Notes: The value of the vector at the time of the call will be copied into the temporary vector, and the original STE will be linked to the temporary by the DLINK of the temporary. This will allow for the use of the EBCDIC name. The use count of both STEs will be increased and both will be decreased when the temporary is erased. A maximum of 4 vectors may be displayed at any one time with maximum display length of 20 values.

Name: DSPORDER

Purpose: To construct an order program corresponding to a given control block.

Figure 8.5.2 Flow Chart of Subroutine DSPNUMV

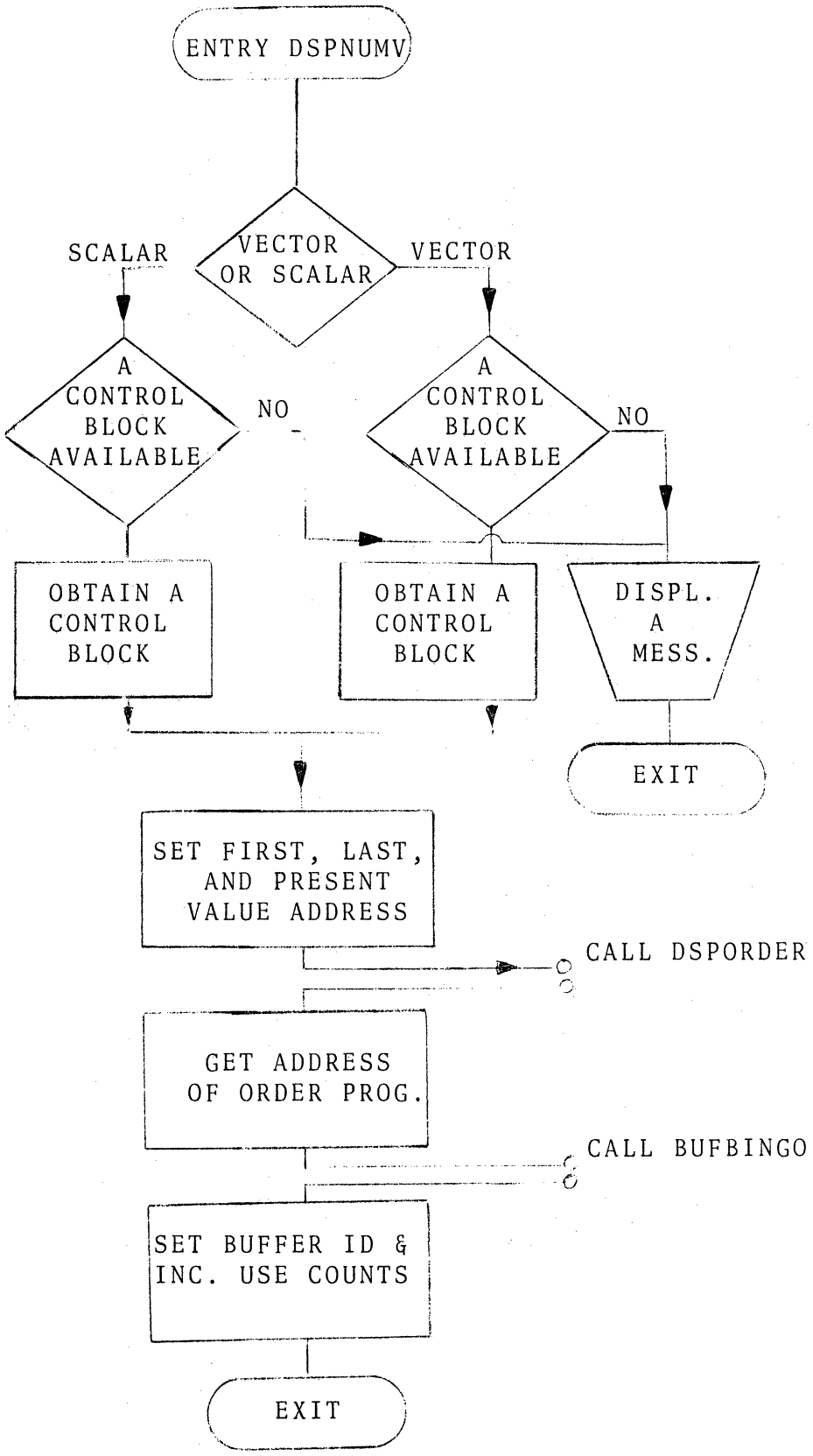


Figure 8.5.3 Flow Chart of Subroutine DSPORDER

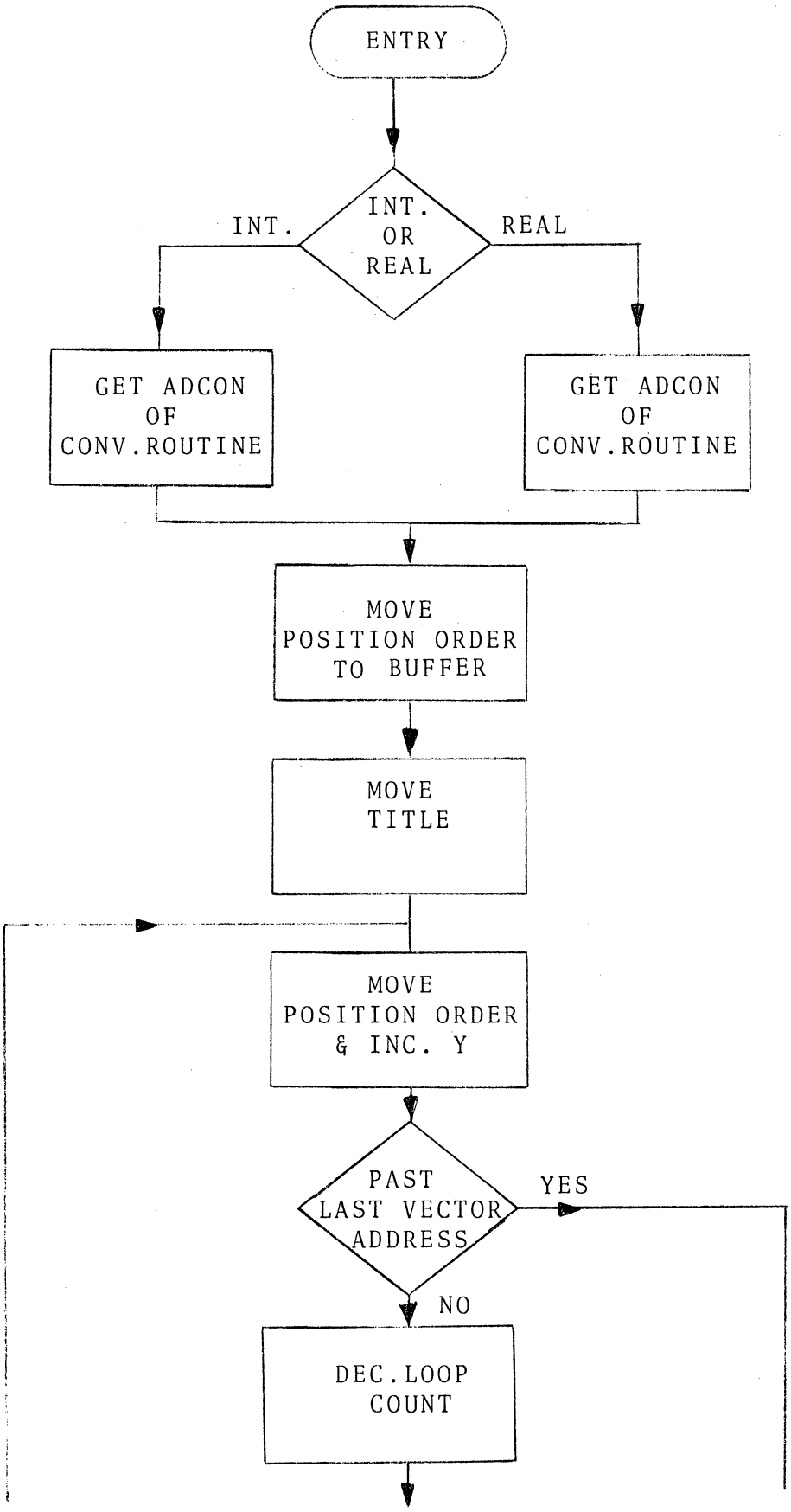
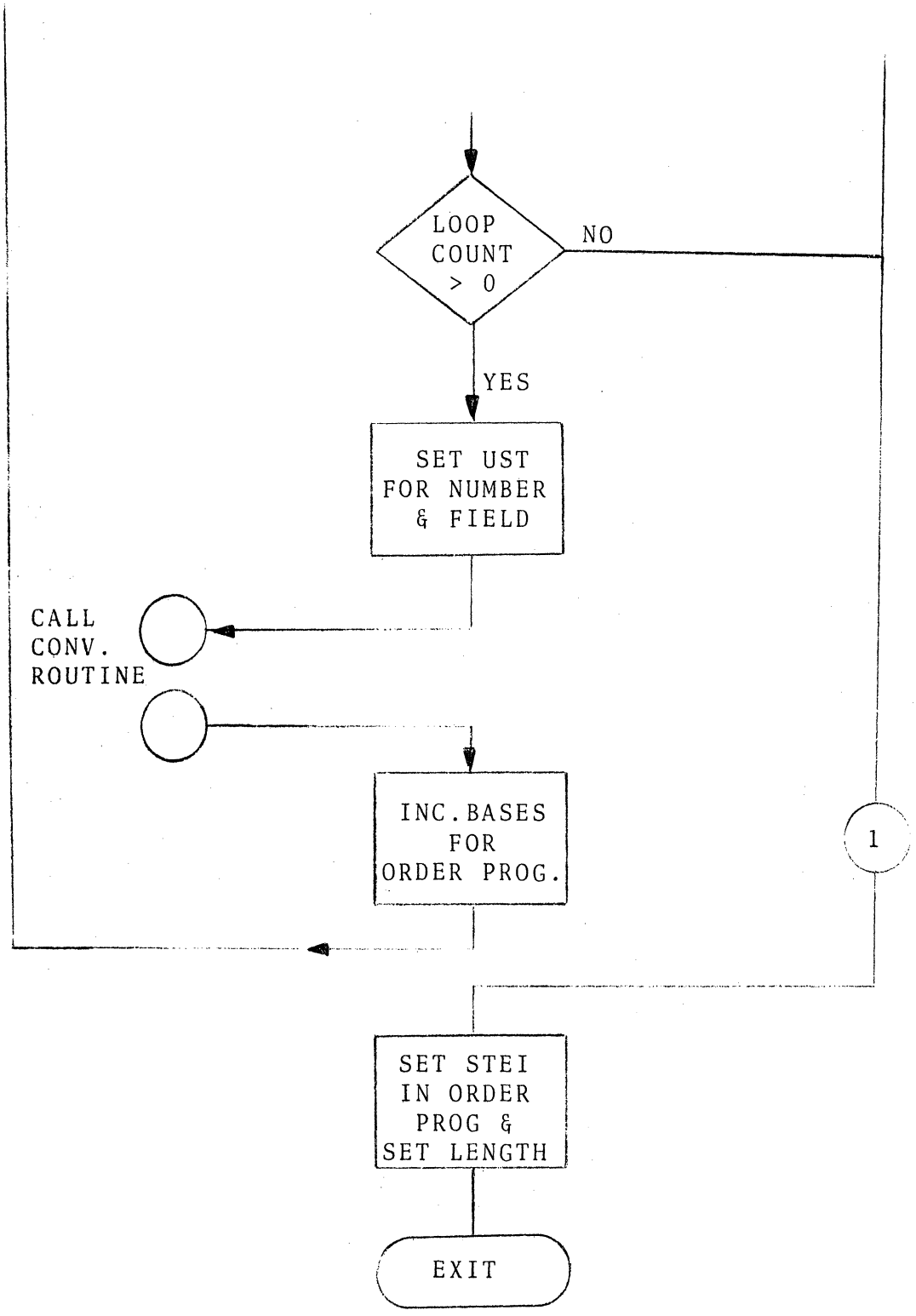


Figure 8.5.3, continued



Calling Sequence:	OS(I) S type
Entry:	GR1 contains the location of the control block.
Exit:	RC=0
Notes:	An order program is left in the PSECT with the count at the address COUNT. It is not an External Symbol.
Name:	DSPSCRLU
Purpose:	To scroll a displayed vector upwards.
Calling Sequence:	OS (I) S type
Entry:	GR1 contains the location of a list of 2 full word adcons: 1. A(STE of vector) 2. A(Value of vector)
Exit:	No return
Notes:	The vector will be scrolled upwards by 10 places for each call on the routine until the last 10 are displayed. No more scrolling may then be done.
Name:	DSPSCRLD
Purpose:	To scroll a vector downwards
Calling Sequence:	OS (I) S type
Entry:	GR1 contains the location of a 2 adcon list: 1. A(STE of vector) 2. A(Value of vector)
Exit:	No return
Notes:	The vector will be scrolled upwards

	by 10 places on each call until the first 20 values are displayed.
Name:	DSPERSEV
Purpose:	To erase a vector which has been displayed on the screen.
Calling Sequence:	OS (I) S type
Entry:	GR1 contains the location of a list of adcons: 1. A(STE of vector) 2. A(Value of vector)
Exit:	No return
Notes:	The display will be erased, the temporary entry will be destroyed, and the use count of the original STE will be reduced.
Name:	DSPNEWNM
Purpose:	To display a name on the function keyboard.
Calling Sequence:	OS(I) R type
Entry:	GR1 contains the STEP of the required STE.
Exit:	RC=0 Name displayed RC=4 Name not displayed
Notes:	The new name will be added to the list of names. This list will first cover the right-hand side of the screen and then the left-hand side.

Name:	DSPDELNM
Purpose:	To delete the name of a function from the function keyboard.
Calling Sequence:	OS (I) R type
Entry:	GR1 contains the STEP of the function to be deleted.
Exit:	RC=0 Name has been deleted RC=4 Name has not been deleted
Notes:	The name will be deleted and the whole keyboard moved up to remove the space left by the deletion.
Name:	DSPMESS
Purpose:	To display a message on the screen.
Calling Sequence:	OS (I) R type.
Entry:	GR1 contains the location of a half word count of an immediately following character string.
Exit:	RC=0 Message displayed RC=4 Message not displayed
Notes:	The message is displayed on the screen below the virtual keyboard. A limit of 254 characters may be displayed. The message will also be written in the file-MESSFILE, which is created during the initialization. Any previous message which is on the screen at the time of a call on this routine will be erased.

Name: DSPERSMS

Purpose: To erase a message from the screen.

Calling Sequence: No parameters are required.

Notes: This is used internally but it may be called by anyone who wants to.

Name: DSPECHO1

Purpose: To get those parts of the button queue which are kept by Light Pen management.

Calling Sequence: OS (I) S type

Entry: GR 1 contains the location of a list of 2 adcons:
1. A(Count of queue)
2. A(Queue)

Exit: No return

Name: DSPECHO2

Purpose: To pass those parts of the button queue which are kept by the Interpreter.

Calling Sequence: OS (I) S type

Entry: GR1 contains the location of a list of 4 adcons:
1. A(Count of button queue)
2. A(Button queue)
3. A(Count of history queue)
4. A(History queue)

Exit: No return

Name: DSPLPROC

Purpose: To display a procedure on the screen.

Calling Sequence: OS (I) S type.

Entry: GR1 contains the location of a list of 2 adcons:
1. A(STE of the precedure)
2. A('Value' of the procedure)

Exit: RC=0 Procedure is displayed
RC=4 Procedure is not displayed.

Notes: The call is a standard function call.

Name: DSPERSPR

Purpose: To delete the display of a procedure.

Calling Sequence: OS (I) S type

Entry: GR1 contains the location of a list of 2 adcons:
1. A(STE of the procedure)
2. A('Value' of the procedure)

Exit: RC=0 Procedure erased
RC=4 Something not quite right.

Name: DSPINC

Purpose: To increase the time count on a message.

Calling Sequence: None

Notes: A display is kept on the screen until one of the following has occurred:
a. A new message is displayed
b. Three additions are made to the echo line.
This routine is used to update the count from the echo line.

Name:	DSPFLDFN
Purpose:	To acquire a temporary MTS file called -MESSFILE.
Calling Sequence:	None
Exit:	RC=0 File acquired
	RC=4 File not acquired
Notes:	A temporary file called -MESSFILE is acquired and is used in order to store a record of the user's computations. The echo line is placed into the file whenever it is placed in the history stack. All messages that are displayed on the screen are also placed in the file. The FDUB of the file is kept under the external symbol of DSPFDUB, and this may be used by anyone who wishes to place anything in the file.

8.6 FUNCTION DISPLAY ROUTINES

General Description

All calls to display graphs in the shared area of the 2250 screen come to Function Display as operator calls from the interpreter. Operators are currently available to display and erase graphs (PLOT1, PLOT2, ERASE, and CLEAR) and to control the mode and scaling of displays (SCALE and SETPLTMD).

Buffer programs for the IBM 2250 are generated using the subroutines in *GRAPHLIB which are described in the

IBM Systems Reference Library manual, IBM System /360 Operating System Graphic Programming Services for the IBM 2250 Display Unit, Model I (Form C27-6921-0 file S360-30). These subroutines were used extensively, and the IBM manual must be used to understand the parameter tables that are set up to display graphs.

Four function display subroutines form the primitive basis for other function display programming. They are:

1. FDADDDS which adds a graph to the screen given two vectors, and the Symbol Table Entry Pointer (STEP) of a temporary vector-pair.
2. FDGRIDDS which puts a grid and labels on the screen after the appropriate parameters are set up.
3. FDMAXMIN which chooses corner points (scale values) for a grid given the x and y vectors to be displayed.
4. FDSETVAR which chooses increments between grid lines and sets up the vectors to be used as labels.

The following few pages describe the operator entries to handle PLOT1, PLOT2, and SCALE and the internal subroutines FDADDDS and FDGRIDDS.

Function Display: Operator Description

Name:

PLOT1 and PLOT2

Function:

display graphs that are composed of a vector-pair or a pair of vectors.

Operands:

PLOT1 takes one operand from the stack. It may be either a vector or a vector-pair. A vector-pair operand is displayed using its own domain and range. A vector operand is used as the y-vector of a graph and is displayed against the standard domain (the ID vector).

PLOT 2 takes two operands. If they are both vectors, the vector first entered onto the stack is used as the y-vector in the graph displayed versus the second vector entered.

If the first entered operand is a vector-pair, its y-vector is used as the y-vector in a display versus the second operand entered.

Results:

Neither PLOT1 nor PLOT2 returns an operand to the stack.

Notes:

1. In all cases the scaling used will be determined by the first graph displayed. If this is unacceptable, the SCALE operator may be used to set the scale.
2. The displayed graph is a floating-point vector-pair operand and may be placed on the operand stack with a light-pen hit on it. It may be used with any operator that accepts vector-pair mode operands.
3. The grid is displayed with the first use of PLOT1, PLOT2, or SCALE.

4. Transformations are performed on the x and/or y vectors if the current plotmode specifies that either one or both of the ordinates is to be of log mode.

Possible Messages:

1. DIMENSIONS OF PAIR OF VECTORS FOR DISP DISAGREE
   The two vectors which are to be displayed with PLOT2 must have the same number of elements. No action is taken if this error occurs, but the operands are removed from the stack.

2. SOME PTS OF LAST GRAPH OFF THE SCREEN
   One or more points of the last graph fell outside of the grid. All points within the grid boundaries are displayed.

3. INTEGERS IN LAST FP VECTOR FOR DISPLAY
   This means that there were some numbers that looked like integers that were supposed to have been floating point. Probably a programming error.

4. DISPLAY BUFFER OVERFLOW ERASE SOMETHING
   Too much is being displayed. Last display is ignored; operands are removed from stack.

Example:

Button pushes starting from the left

```
ID  ID  PLOT2
```

displays a straight line from (-1,-1) to (1,1).

Name:

SCALE

Function:

Sets the scale values to be used for the lower-left and upper-right corners of the grid and then displays

an empty grid. PLOT1 and PLOT2 will use these scale values for all subsequent displays.

Operands:

SCALE accepts four scalar floating point operands that describe the lower-left and upper-right corners of the grid to be displayed. The order which the operands are to be placed on the stack is:

1. x value of the lower-left corner.
2. y value of the lower-left corner.
3. x value of the upper-right corner.
4. y value of the upper-right corner.

Results:

No results are returned to the stack.

Notes:

1. Use of SCALE overrides any automatic scaling by PLOT1 or PLOT2.
2. If some graphs are already displayed they will not be rescaled. The grid will be changed under them. They may be rescaled by getting two light-pen hits, i.e., placing the graph on the stack twice, and then erasing and plotting again.
3. SCALE does not alter the plotting mode. The grid displayed will be of the current mode.
4. No messages are displayed by SCALE.

Function Display: Subroutine Description

Name:

FDDSVCPR

Function:

The entry point for all PLOT2 operator instances and the PLOT1 instance to display a vector against the standard domain.

Calling Sequence:

OS(I) S-type

There are six parameters, the STEP and VALUE of the x vector to be displayed, of the y vector to be displayed, and of the vector-pair temporary for the displayed graph.

Subroutines used:

1. FDMAXMIN: routine to set up the scale values for the corner points of the grid.
2. FDSETVAR: routine to set up grid and label increments and the label vectors.
3. FDGRIDDS: routine to display grid, labels, and the scale messages.
4. FDADDDS: routine to construct the buffer program for the graph to be displayed.

Description:

1. The dimensions of the x and y vector are checked for agreements.
2. The y and then the x vectors are moved to the vector-pair temporary.
3. If either or both of the vectors are integer they are converted to floating point.

4. If no grid is currently being displayed, FDMAXMIN, FDSETVAR, and FDGRIDDS are called to display the grid.

5. Return to caller.

Name:

FDDSFN

Function:

Entry point of PLOT1 when it operates on a vector-pair operand.

Calling Sequence:

OS(I) S type.

There are four parameters, the STEP and VALUE of the vector-pair to be displayed and of the vector-pair temporary used for the displayed graph.

Subroutines Called:

None; however code in FDDSVCPR is entered after initial setup.

Description:

1. The vector pair is moved to the temporary.

2. Variables are set up and FDDSVCPR is entered at REALY.

Name:

FDSCALE

Function:

Entry point for the SCALE operator.

Calling Sequence:

OS(I) S-type

There are eight parameters: the STEP and VALUE of the following four floating point scalars.

1. y value of the upper-right corner.
2. x value of the upper-right corner.
3. y value of the lower-left corner.
4. x value of the lower-left corner.

Subroutines Used:

1. BUFBOD: to erase a grid if there is one already up.
2. FDSETVAR: to set up the grid and labels parameters.
3. FDGRIDDS: to display a new grid.

Description:

1. If a grid is already up, it is erased.
2. The upper-right and lower-left corner scale values are set into the parameter tables.
3. FDSETVAR is called to set the grid and label increments and the label vectors.
4. The buffer program for the grid and labels is constructed with FDGRIDDS.
5. Return to caller.

Name:

FDADDDS

Function:

Adds one graph to the 2250 display buffer.

Calling Sequence:

OS(I) S-type

GR1 contains an address that points to a vector of four other addresses:

1. The address of the first element of the x-vector to be displayed.
2. The address of the first element of the y-vector to be displayed.
3. The address of a full word containing the number of points to be plotted.
4. The address of a full word containing the Symbol Table Entry Pointer for the temporary space used for the displayed graph.

Exit:

GR15 is set to zero.

Subroutines Used:

1. GETSPACE: used to get space for graphs to be displayed in one of the log modes.
2. ALOG10: used to transform functions displayed in one of the log modes.
3. DSPMESS: to display messages on the screen.
4. GSTOR: *GRAPHLIB routine used to add 2250 commands to the buffer program.
5. GSVPLOT: *GRAPHLIB routine used to construct the 2250 buffer programs for the graph itself.
6. BUFBINGO: used to write the buffer program into the 2250 buffer.

Format of the 2250 buffer program for graphs:

1. A 2-byte NOP (may be changed to "Defer Light Pen Detects" when used on 2250 model 3)
2. The buffer program for the graph. Absolute

vector mode plotting is used.

3. A 2-byte NOP (may be replaced by "Enable Deferred Detects" on the model 3).

4. A 4-byte NOP containing the Symbol Table Entry Index (STEI).

Subroutine Description:

1. The parameter tables for the *GRAPHLIB routines are initialized and set up for the vectors to be displayed.

2. If either or both of the vectors is to be displayed in log mode, space is obtained from GETSPACE and the $\log_{10}$ of the appropriate vectors is moved into this space.

3. Three calls are made on *GRAPHLIB to build the first three parts of the buffer program.

4. The STEI is obtained, entered into the 4-byte NOP and the two are added to the buffer program.

5. The buffer program is written to the 2250 with BUFBINGO.

6. The buffer ID is entered into the symbol table.

7. The STEP is entered into the table of displayed graphs.

8. The use count of the temporary is incremented.

9. Return to caller.

Name:

FDGRIDDS

Function:

Constructs the buffer program for the grid, labels, and scale messages.

Calling Sequence:

OS(I) R-type

No parameters.

Subroutines used:

1. GSTOR: *GRAPHLIB routine to add 2250 commands to the buffer program.
2. GCGRID: *GRAPHLIB routine to construct the buffer program for the grid.
3. GLABEL: *GRAPHLIB routine to label the axes.
4. GCPRNT: *GRAPHLIB routine used to add the scale messages to the buffer program.
5. BUFBINGO: routine to write the buffer program to the 2250.

Description:

1. The 2250 command to disable light-pen detects is placed in the buffer program.
2. Plot mode is checked and parameters for GCGRID are set up to give the correct mode display.
3. Parameters set up previously by FDSETVAR are used to set up two calls to GLABEL to label the x and y axes.
4. The scale messages are added to the buffer program.
5. The 2250 command to enable switch detects is added to the buffer program.
6. The buffer program is sent to the 2250 using BUFBINGO.
7. The buffer ID of the grid is saved locally.
8. Return to caller.

Function Display: General Organization

The routines previously described are contained in one assembly with two CSECTs (FDADDDS and FDGLOB). FDGLOB is the CSECT to be used as a PSECT and contains the function display global variables.

Entries to the Function Display Routines

The following pages describe the entries FDINIT, FDINTFLT, FDSETVAR, FDFLTBCD, FDMAXMIN, FDERASE, FDCLEAR, and FDSETMOD to the display section of the mathematical analysis package. Thsse routines perform data conversion, set plotting parameters, erase displays, and initialize the function display section.

FDINIT

This routine is called during initialization to enter four operators--PLOT1, PLOT2, SCALE, and SETPLTMD--into the symbol table (and onto the screen). In addition it calls the initialization point DSPECHO of the numeric and keyboard display section.

The characteristics of the four operators initialized by FDINIT are as follows:

PLOT1: Has two possible mode combinations:

1. A vector (or the first half of a vector pair) is plotted against the ID vector.
2. A function is plotted (i.e., the first half of the vector pair is plotted against the second half).

PLOT2: Has two possible mode-combinations:

1. A vector is plotted against another
2. The first half of a vector-pair is plotted against a vector.

SCALE: Requires four operands. These are the REAL*4 coordinates of the lower left and upper right corners of the graph (i.e., the values represented by these corners, which are positioned at fixed points relative to the screen).

SETPLTMD: Requires one REAL*4 or INTEGER*4 operand. This operand is the desired plotting mode number. The routine FDSETMOD, which interprets SETPLTMD, determines from the operand itself without reference to the symbol table whether it is of REAL*4 or INTEGER*4 mode.

The name of the PSECT for FDINIT is (predictably) FDINITPS. This PSECT contains the save area and all ad-cons for FDINIT.

FDSETVAR

This entry initializes or resets various parameters used by the *GRAPHLIB routines for displaying graphs. It examines U1, U2, V1, and V2, which are the scale values of the lower left (U1, V1) and upper right (U2, V2) scale values on the screen and sets the following variables:

ULABAX: the relative position in the V direction of the labels on the U-axis (=V1)

VLABAX: the relative position in the U direction of the labels on the V-axis (=U1)

UGRID: the scale increment between the grid lines in the V direction (= (U2 - U1)/8)

VGRID: the scale increment between the grid lines in the U direction (= (V2 - V1)/8)

ULABINC: the scale increment between successive labels on the U-axis (= UGRID)

VLABINC: the scale increment between successive labels on the V-axis (= VGRID)

USTRING: the EBCDIC characters comprising the U-axis labels (c.f., FDFLTBCD)

VSTRING: the EBCDIC characters comprising the V-axis labels (c.f., FDFLTBCD)

FDSETVAR is called without parameters, since all necessary information is contained in the control section FDGLOB. Note that the routine FDFLTBCD may reset ULABINC and VLABINC to a multiple of their values upon entry.

The PSECT for FDSETVAR is FDSVPSCT. It contains the save area and a four-fullword area used as a parameter list for the two calls to FDFLTBCD.

FDINTFLT

This is a "utility" routine which can be called to convert a vector of integers (INTEGER*4) to a vector of floating-point numbers. The REAL*4 results replace the integers. The calling sequence is

CALL FDINTFLT,(N,X), where

N is the INTEGER*4 number of elements in

X, the INTEGER*4 vector to be converted.

The PSECT for FDINTFLT is FDINTFPS; it contains only the save area.

FDFLTBCD

This routine is called by FDSETVAR to set up the strings of EBCDIC characters which comprise the labels for the function display.

The calling sequence for FDFLTBCD is

CALL FDFLTBCD,(X1,X2,DX,STRING,DIGITS),

where:

X1 is a "starting value" (i.e., the most negative or the least scale value in the U or V direction)

X2 is the "terminal value" (i.e., the most positive or the greatest scale value in the U or V direction)

DX is an increment (i.e., either ULABINC or VLABINC)

STRING is the location of the region into which conversation is to take place

DIGITS is the location of a three-byte region into which the exponent part of a scale-factor is to be placed (c.f., below).

The routine first checks to see whether the values X1, X2, and DX are consistent. If either

1. $X1 < X2$ and $DX \leq 0$

or 2. $X1 > X2$ and $DX \geq 0$

immediate return is made with RC=4. Then DX is reset to $n * DX$, where n is the least integer such that

$$\frac{56}{\left\lfloor \frac{X1 - XTOP}{DX} \right\rfloor + 1} \geq 6$$

Where XTOP = X1 + DX[|X1 - X2|/DX] and the brackets denote "integer part."

The routine then converts the floating-point numbers X1, X1 + DX, ..., XTOP to EBCDIC character strings of six characters each, with one of the following edit-patterns:

b±.ddd

b±d.dd

b±dd.d

b±ddd.,

where "b" denotes a blank, and "d" denotes a digit. The successive six-character groups are placed contiguously in STRING. In the field DIGITS the routine places three characters of the form ±dd. These will appear on the screen in the form

VERT SCALE:*10**(±dd)

HORZ SCALE:*10**(±dd)

to indicate that internal range and domain values are 10**(±dd) times the values appearing externally in the labels.

FDFLTBCD also returns with a return-code of 4 if it encounters floating-point values below 10^{-74} or above 10^{75}.

The PSECT for FDFLTBCD is FDFLBCPS. It contains the save area and all constants generated by literals.

FDMAXMIN

This routine examines two REAL*4 vectors and assigns values to the plot variables U1, U2, V1, V2 corresponding to the values of the maxima and minima of the vectors X and Y of the calling sequence, which is:

```
CALL FDMAXMIN,(N,X,Y),
```

where N is the INTEGER*4 number of elements in (each of) X and Y, and X and Y are the REAL*4 vectors to be examined. FDMAXMIN then sets

```
U1 = min(X) - 1/4 |max(X) - min(X)|
U2 = max(X) + 1/4 |max(X) - min(X)|
V1 = min(Y) - 1/4 |max(Y) - min(Y)|
V2 = max(Y) + 1/4 |max(Y) - min(Y)|
```

FDMAXMIN then calls FDSETVAR to reset various global parameters in accordance with the new scale values for the screen corners.

The PSECT for FDMAXMIN is FDMXMNPS. It contains the save area as well as the four-word adcon area for the call to FDSETVAR.

FDCLEAR

This routine is called by the routine which interprets the operator CLEAR, with no arguments. FDCLEAR inspects the list DSPTAB containing the STEPs of all current function-displays. For each non-zero entry, FDCLEAR:

1. Obtains the buffer ID of the display via SYMGET DBUF#
2. Deletes the display with a call to BUFBOD
3. Decrements the use-count of the temporary symbol via SYMDLE
4. Sets to zero the DSPTAB entry for the deleted display.

After all function displays have been deleted, FDCLEAR deletes the grid display by a call to BUFBOD and zeroes the contents of GRIDID in the control section FDGLOB.

The PSECT for this routine is FDCLPSCT. It contains the save area and all adcons.

FDERASE

The routine FDERASE is called by the routine which interprets the operator ERASE. General Register One must contain the STEP for the display to be erased. FDERASE then searches DSPTAB in the control section FDGLOB for the same STEP. If it finds the STEP of the ERASE operand, this entry in DSPTAB is zeroed, the macro SYMDLE is executed to decrement the use count, the graph is erased by a call to BUFBOD, and return is made with RC=0.

If the STEP of the operand cannot be found in DSPTAB, return is made with RC=4.

The PSECT for FDERASE is FDRSPSCT. It contains the save area and all adcons.

FDSETMOD

The chief function of this routine is to store a mode number for the current mode of plotting which is passed by the interpreter as a consequence of the SETPLTMD button-push. The call is

CALL FDSETMOD,(Z)

where Z is the REAL*4 or INTEGER*4 plot mode. FDSETMOD checks to see whether Z is integer or floating-point, and stores it as an integer in the location PLTMODE of the control section FDGLOB.

After storing the plot mode, FDSETMOD calls BUFBOD to erase the current grid, and FDGRIDDS to create a new one.

The PSECT for FDSETMOD is FDSMPSCT. It contains the save area and all adcons.

8.7 BUFFER MANAGEMENT ROUTINES B.J. Bolas, R.W. McHard

General Layout of Buffer

The initial buffer has a start regeneration timer (GSRT) followed by a transfer (GTRU) to the first order program in the regeneration loop. This generally branches around the "buffer full" message, which is next in the

in the buffer; however, when a call on BUFBINGO or BUFBORE fails due to insufficient space, this message is linked into the regeneration loop. Following this message are three initial GTRUs, one for each type of order program. Type 0 order programs are those buttons on the periphery of the screen which are always displayed; Type 1a order programs are those displays in the central region of the first screen given to the user; Type 1b order programs are those displays in the central volatile region of the screen given to the user when he hits the FLIP button for the first time. The initial GTRUs point to the first order program of the corresponding type. The last Type 0 order program points to the initial 1a or 1b GTRU, depending upon which screen is the currently active screen. The last Type 1a and 1b order programs both point to the GSRT. As indicated shortly, the buffer is always "packed"; i.e., there are no unused buffer locations between order programs in the buffer at any time. To establish properly the linkage within types of order programs, a GTRU to the next order program of the same type is tacked onto the end of the order program when BUFBINGO and BUFBORE are called. The address of this GTRU is updated by buffer management whenever it changes (due to a call on BUFBOD or BUFBORE).

Corresponding to each order program in the buffer is a control block with five full-word entries. The control blocks are linked with backward and forward pointers within each type, the linkage corresponding to the linkage of the corresponding order programs in the buffer. The format of these control blocks is as follows:

```
**************************************************
* 1 POINTER TO ITSELF FOR ERROR CHECKING,        *
**************************************************
* 2 BACKWARD POINTER TO PREVIOUS CONTROL BLOCK,  *
**************************************************
* 3 FORWARD POINTER TO NEXT CONTROL BLOCK,       *
**************************************************
* 4 ABSOLUTE BUFFER ADDRESS OF ORDER PROGRAM,    *
**************************************************
* 5 BYTE-LENGTH OF ORDER PROGRAM.                *
**************************************************
```

Following is a general description of the internal procedures used in the subroutines in the buffer management package.

BUFINIT

Purpose:

To initialize the image of the buffer during initialization of the system.

Description:

The image of the buffer is a region of storage of length 8K = 8192 bytes, the length of the 2250 buffer. At initialization, certain necessities, such as a start regeneration timer, a message indicating that the buffer is full (which is displayed only when appropriate), and

transfers to order programs to eventually be inserted, are put into the image.

Entry:

No parameters are passed.

Exit:

GR1 has the storage address of the buffer image.

GR15 has return code 0.

Note:

BUFINIT is called by BUFINIT1.

BUFBINGO

Since the buffer is always packed (see below), the decision as to whether or not the buffer has sufficient space for the order program reduces to a comparison of its length against the difference between 8192 and the next available buffer address. If there is sufficient room in the buffer, the order program is moved into the buffer at the next available buffer location, and is linked into the regeneration loop for order programs of its type. If there is not sufficient room in the buffer, then a message to that effect is displayed to the user. To guarantee that there is always room for this message, it is resident in the buffer after initialization, but is normally not linked into the buffer display-regeneration loop.

BUFBOD

When an order program is deleted, the buffer is "packed" by moving all order programs (if any) following the deleted order program in the buffer (with higher absolute buffer addresses) into the region vacated by the deletion of the order program. This was designed to alleviate the problem of deciding whether an order program passed by BUFBINGO or BUFBORE can be inserted into the buffer. See the section on recommendations for the future for an argument against using this scheme.

Note: One obvious implication of this buffer-packing scheme is that linkage of order programs within their respective types is monotonic with respect to absolute buffer addresses. In other words, each order program branches to an order program with a higher absolute buffer address (except the ones at the end of the loops, which branch to appropriate locations in the initial buffer).

BUFBORE

The new order program is inserted at the same absolute buffer address as the old order program, if there is room. If not, the message that the buffer is full is linked into the regeneration loop. To insert the new order program, first all order programs with higher buffer addresses are moved to leave a hole just the right size for the inserted order program. Thus the buffer remains packed.

Note: BUFBORE inserts the new program into the loop for the same type as the old program. Thus BUFBORE implicitly assumes that the new order program will be of the same type as the old order program. (It is speculated that all such calls will involve only Type 0 order programs, since BUFBORE is strictly a system-generated call; however, BUFBORE will work for any type, as long as both the inserted order program and the deleted order program are the same type.)

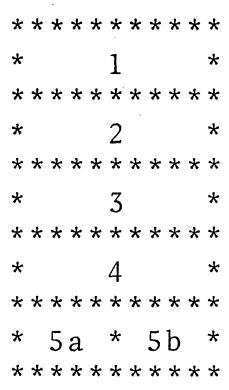
BUFFLIP

The last Type 0 GTRU (transfer instruction) points always to either the initial Type 1a GTRU or the initial Type 1b GTRU. A call on BUFFLIP (i.e., a light-pen hit of the FLIP button by the user) merely switches this last Type 0 GTRU to point to the Type 1b loop instead of the Type 1a loop, or vice versa.

It is now felt that the scheme of always packing the buffer is inefficient and wasteful of CPU time, and that instead the buffer should be packed only when there is no space in the buffer big enough for insertion of the order program. Then extra information is needed by buffer management in order to keep track of the portions of the buffer currently available. Given this information, the most efficient method of buffer management is to search the list (in whatever form kept) for the smallest space large enough for the order program, while

accumulating the total available space in the same pass. Then, if a space is found, the order program is moved into that space. If no space is found, the accumulated length is compared against the length of the order program (allowing, of course, for the 4-byte GTRU which buffer management tacks onto the end). If the order program is no larger than this accumulated length, then the buffer is packed (entirely) and the order program is inserted. If the total space is not large enough for the order program, then no packing takes place and the user is notified that the buffer is full. This scheme can obviously be extended to keep (again in a one-pass search) a record of partial cumulative space-lengths for spaces which are consecutive (no other space regions in between) in the buffer. Then, for example, even if no one free buffer region is sufficiently large for the insertion of the order program, there may be two regions separated by order programs which together may be large enough for the order program. Then only this block of order programs needs to be packed, combining the two small free regions into one larger free region. This eliminates the need to pack the entire buffer, and can easily be extended to three or more consecutive regions.

The new information needed could easily be incorporated into the current control-block structure. Restructure the control blocks as follows:



Here 1 points to itself if the control block represents an order program, and is 0 if it represents a region of space; 2 is a backward pointer as before; 3 is a forward pointer as before; 4 is a forward pointer to the control block to the next region (allocated or not, running through all types); 5a is the absolute buffer address; and 5b is the length of the region.

Then there are four linked lists--one for each type of order program, and one for blocks of available space. The linkage is accomplished by the backward and forward pointers in words 2 and 3 of the control block. For control blocks representing order programs, the linkage corresponds to the linkage of the corresponding order programs in the buffer; for control blocks representing unallocated regions in the buffer, any convenient scheme (such as monotonicity with respect to absolute buffer addresses) can be used. The main utility of the thread linking all control blocks in word 4 is that the linkage of order programs within their respective types is

not necessarily monotonic under this scheme, and updating the linkage in the buffer after packing is greatly facilitated by such information.

One last recommendation for future work on buffer management is that the FLIP concept can be generalized with very little difficulty. Allow any convenient number of screens to the user, acquiring each by GETSPACE as the user defines it (the first is by necessity defined by the system during initialization). When defined, the initial buffer is established. Only Type 0 programs are in any image besides the volatile type peculiar to the screen. This has the immediate advantage of not wasting buffer space with order programs which often lie dormant in the buffer, not being linked in the regeneration loop. A table is maintained which indicates the storage address of each defined image. To save the need for updating the Type 0 order programs in every image as they change, an "image" of Type 0 order programs only could be maintained. Then, when a user switches to a different screen, see if the new Type 0 order programs will fit in with the old Type 1 order programs. If not, link in the message to the user that this particular screen is full; otherwise, proceed as usual.

The user could specify a new screen by a hit of a parameter followed by a hit of the FLIP button, for example:

3 FLIP

If 3 was specified for the first time, this defines a new screen. If 3 was previously specified, switch to it as usual. If the user wishes to create several screens, it would be wise for him to be able to create a catalog of screens and their uses, and put that on a separate screen. (It is not clear that he can do this with the system as it now stands.)

Buffer Management

Name: BUFINIT1

BUFINIT2

Purpose: To initialize Buffer Management.

Calling Sequence: OS type I.

Entries: Neither entry takes parameters.

Exits: RC=0 only.

Subroutines called: BUFINIT, SYMOPDEF, GOPEN.

Other external symbols: BUFDCB, BUFPOLST. The locations of the graphic DCB and poll list; for use by Light Pen Management.

Usage:

BUFINIT1 must be called before calling any other Buffer Management entry (which implies that it must come early in the initialization sequence). It does not require that any other section be initialized. BUFINIT2 assumes that SYMOPDEF and the routines it calls are

already initialized, and that Light Pen Management is ready to receive interrupts.

Description:

BUFINIT1 turns off Light Pen Management's interrupt switch (LPMISW), calls GOPEN to open the 2250 display and initialize the data control block (DCB), and calls BUFINIT to initialize the rest of Buffer Management. BUFINIT returns the address of the main storage graphic output area (an "image" of the device buffer); this is saved for use by BUFWR.

BUFINIT2 calls SYMOPDEF to enter the FLIP operator, turns LPMISW on, and returns.

Name:	BUFBINGO
Purpose:	To insert an order program into the buffer to be displayed on the 2250.
Description:	The order program is linked into the buffer display-regeneration loop if there is sufficient room. If not, an error message resident in the buffer is linked into the regeneration loop. The linkage is in terms of the type of the order program to be inserted. System entities such as the virtual keyboard, the echo line, and system-defined buttons are

always displayed, as are all user-defined buttons. All other user-defined displays are displayed on the current screen, but disappear when the FLIP button is hit (and re-appear when the FLIP button is hit again).

Entry: GR0 = 0 if the order program is system-defined or a user-defined button. GR0 is nonzero otherwise.
GR1 has the storage location of the order program to be inserted, the first halfword of which contains the byte-length of the remainder ("body") of the order program.

Exit: GR15 = 4 if there is insufficient room in the buffer for the order program. In this case, the user is notified that the buffer is full.
GR15 = 0 if the order program was inserted into the buffer.
GR0 contains the buffer ID for the order program if it was successfully inserted into the buffer.

Note: BUFBINGO calls BUFWR to write the buffer from the image.

Name: BUFBOD

Purpose: To remove an order program from the buffer display-regeneration loop.

Entry: GR0 contains the buffer ID returned from the BUFBINGO call which inserted the order program into the buffer.

Exit: GR15 = 0 normally.
GR15 = 4 if the buffer ID does not turn out to be one. This indicates a snark in our system. In this case, nothing is done.

Note: BUFBOD calls BUFWR to write the buffer from the image.

Name: BUFBORE

Purpose: To replace an order program in the buffer with another order program.

Description: This is strictly a system-generated call. Since the new order program is likely to be longer than the old order program, it is important to know that the new order program will fit into the buffer before deleting the old order program. This is not done if consecutive calls on BUFBOD and BUFBINGO are used.

Entry: GR0 has the buffer ID returned from the BUFBINGO call which inserted the original

order program into the buffer.
GR1 has the storage address of the order program which is to replace the old order program. This order program has the same format as for the call on BUFBINGO.

Exit: GR15 = 4 if there is insufficient room in the buffer for the new order program. In this case, the original order program is not deleted, and the user is notified the the buffer is full.
GR15 = 0 if there was sufficient room in the buffer for the new order program.
GR0 contains the same buffer ID passed to it, which serves as the buffer ID of the new order program, if the operation was successful.

Note: BUFBORE calls BUFWR to write the buffer from the image.

Name: BUFFLIP

Purpose: To switch between the volatile graphic portion of the three screens available to the user.

Description: BUFFLIP supports the immediate operators SCREENA, SCREENB, SCREENC which are system-defined buttons on the screen. All system entities and user-defined buttons are always

displayed.

Entry: GR1 contains pointer to SCREENA, SCREENB, or SCREENC identifier.

Exit: GR15 = 0 always.

Note: BUFFLIP calls BUFWR to write the buffer from the image.

Name: BUFWR

Purpose: To write a composite order program to the 2250 display and start regeneration.

Calling Sequence: OS (I) R type.

Entry: GR0 contains the length of the order program to be written from the output area.

Exit: RC = 0 only.

Subroutines called: GIOCR.

Description: BUFWR uses the length given in GR0 unless this is zero, in which case it uses the last nonzero length given. BUFWR puts the necessary information into a data event control block (DECB), calls GIOCR to write the specified length from the output area to the display buffer, and returns.

Note: If the Mathematical Analysis Program is re-entered after an interlude in MTS, the "restart" sequence should call BUFWR with Length=zero. This will re-establish the correct order program in the display buffer, and start it.

Additional Notes

All Buffer Management initialization is completely dynamic. Therefore, a "restart" package could re-initialize Buffer Management by calling BUFINIT1 and BUFINIT2.

MTS support for the 2250 (*GRAPHLIB) uses modified routines from IBM OS Graphic Access Method (GAM). These routines have been further modified especially for this Communication Sciences 673 project. In particular, the SETANLZ routine has been added, primarily for Light Pen Management. One feature has been added for Buffer Management: When SETANLZ restarts the display after an interrupt, it takes the buffer restart address (BRSA) from the DCB, for use in the Set Buffer Address Register and Start channel command. This address is placed in the DCB by GIOCR for each Read & Start or Write & Start operation. This ensures that display regeneration will always begin where Buffer Management last specified.

8.8 THE MACRO PROCESSOR

The Macro Processor handles the definition and calling of macros. Its entry points are as follows:

Entry Point	Purpose	External symbols or macros referenced
MACDINIT	part of initialization, called internally by MACROFIX	SYMCRE
MACDSCT	simulated PSECT	

MACRODEF	define a new macro	GETSPACE, SYMREF, DSPMESS,SYMODEFD
MACDELET	delete a macro	FREESPAC
MACROGET	returns the next STEP in a macro expansion	DSPMESS, SYMGET
MACROFIX	initializes the Macro Processor	SYMCRE, DISPNEWNM, MACDINIT
MACROINT	initializes a call upon the MP	DSPMESS, SYMGET
MAC2PSCT	simulated PSECT	
MDEF	simulated PSECT	

Internal specifications

1. The interpreter calls MACRODEF with GR1 pointing to the beginning of the definition and GR0 containing its length (i.e., the number of button pushes, exclusive of the DEFMACROs):

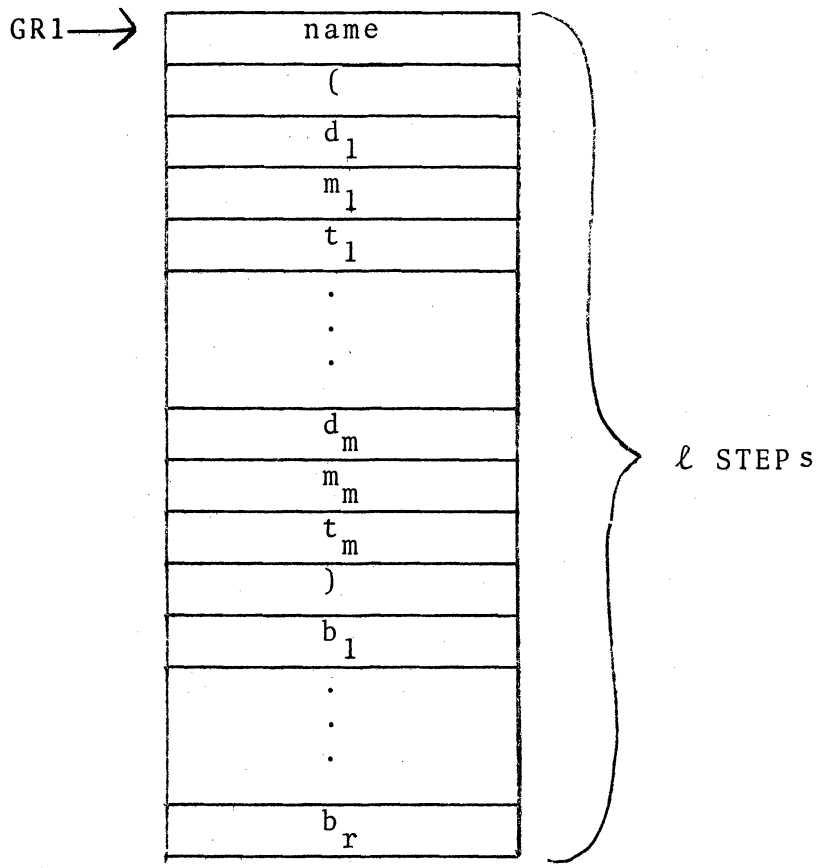


Figure 8.8.1 Storage of a Macro Definition by Interpreter

,GR0 contains the number of STEPs (ℓ)

All editing is done prior to this call by the interpreter.

2. The Macro Processor will create the DOPE vector for "name" in the same format as that for operators (see Section 8.2). The first parameter in the "skeleton" will indicate the number of calling parameters, then the macro STEP, then the operands:

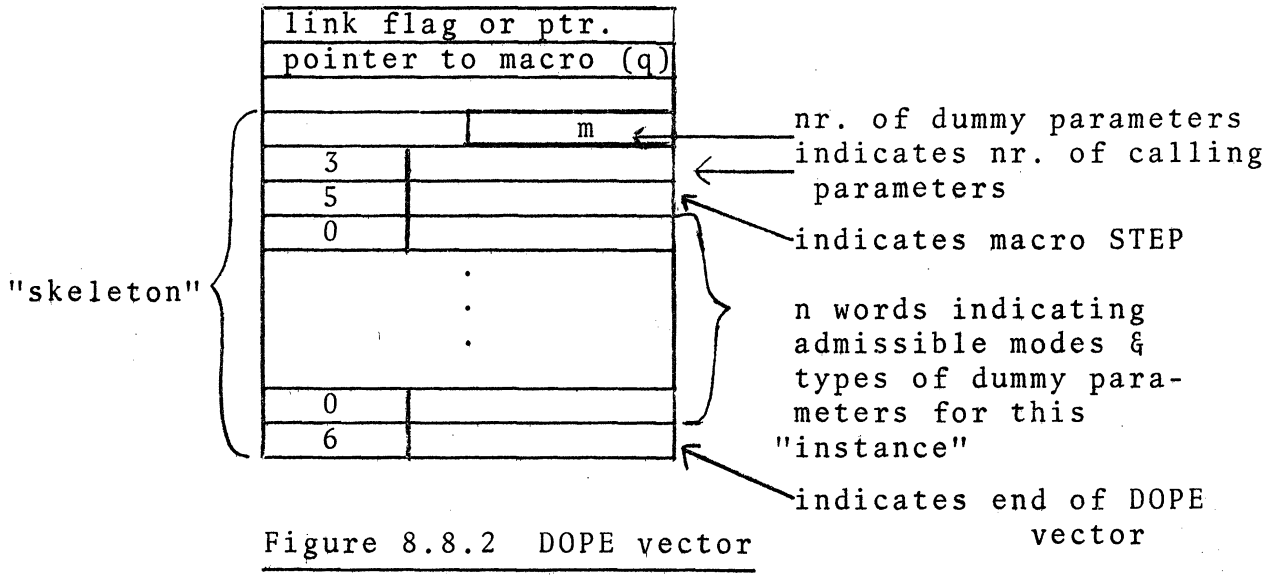


Figure 8.8.2 DOPE vector

3. The Macro Processor will increase the use count for each distinct dummy parameter STE and for the macro STE.

4. Using GETSPACE, the MP stores a macro as follows:

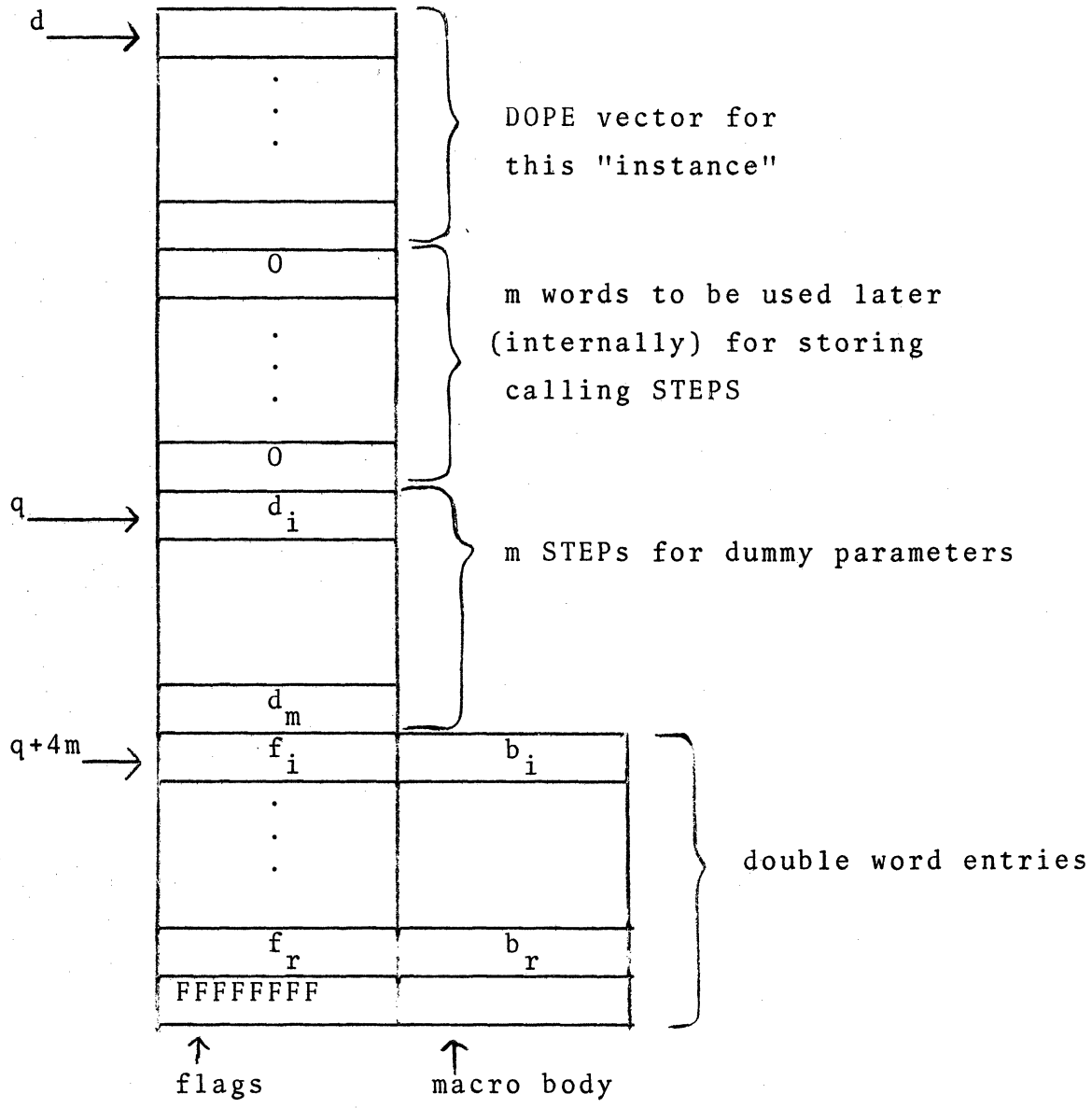


Figure 8.8.3 Final Storage of a Macro Definition.

where

b_i is the STEP corresponding to the button push b_i.

f_i is a flag for the ith STEP b_i:

when b_i is not a dummy STEP, $f_i=1$

when b_i is a dummy STEP, for d_k, then $f_i=4(k-1)$

the last entry is signaled by a flag of hexadecimal FFFFFFFF, as shown, and has no corresponding STEP.

5. Return Codes (in GR15)

If there is any error in the definition, an error message is displayed, and control is returned to the interpreter with a return code of 4. In this case, it is just as if the macro had never been defined. The user may then redefine the macro.

Displaying a Macro

Figure 8.3.3 indicates how a macro is stored. The m dummy STEPs start at location q. Both m and q can be obtained from the DOPE vector for the macro by noting that the second word of the dope vector contains q and the 12th and 13th bytes (i.e., one half-word) contain the number of dummy parameters plus 7, i.e., (m+7) (see Section 8.2, Interpreter). Thus, if GR15 had the pointer to the DOPE vector,

```
L   GR,4(GR15)
LH  GS,10(GR15)
```

would load GR and GS with q and M+7, respectively.

The macro body then starts at location q+4m as double word entries. The first word of each entry is a flag, the second a STEP. The only flag the display need consider is hexadecimal FFFFFFFF, which signals the end of the macro body (and which has no STEP).

Deleting a Macro

A. External specifications

A macro is deleted as follows:

name DELBUTON

where name is the name of the macro to be deleted. All "instances" of the macro will be deleted.

B. Internal specifications

1. The interpreter calls upon the MP via entry MACDELET. The MP then FREEs all SPACs associated with storage of the macro, including DOPE vectors. It also decreases the use count for the macro and for each distinct dummy parameter.
2. The interpreter will see to it that the macro button is no longer displayed.

Initializing the MP

The MP is initialized via entry MACROFIX, which requires no parameters (except a save area pointer in GR13). In addition to internal initialization, the MP creates the buttons MACRO and OPERATOR and indicates that they are to be displayed.

Calling a Macro

Internal specifications

1. The calling mechanism for a macro is very similar to that for an operator. After doing any necessary editing, the interpreter calls upon the operator processor indicating an operator or macro has been encountered. If the operator processor determines that it is a macro, it calls MACROINT to initialize the call, giving it a pointer in GR1 to a parameter list (specified in the DOPE vector skeleton), as follows:

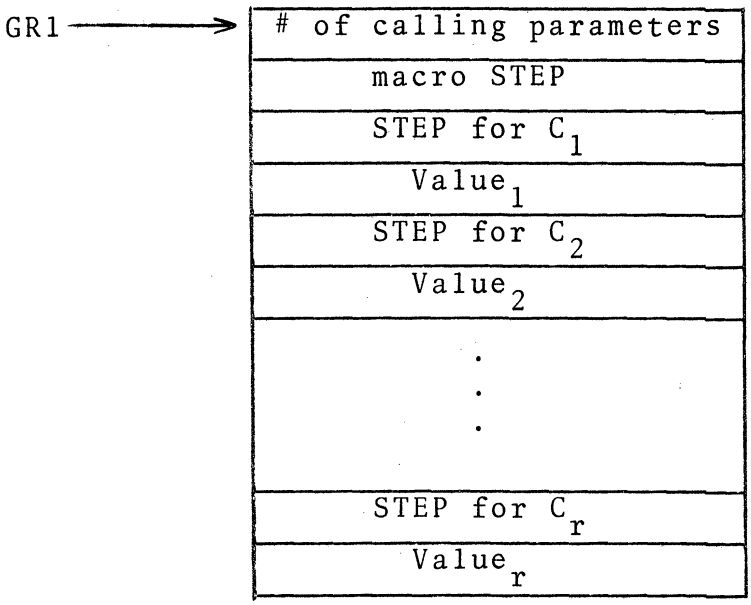


Figure 8.8.4 Parameter List to MACROINT

2. After the MP returns to the operator processor, the OP indicates to the interpreter that subsequent "button pushes" are to come from the MP, and returns control to the interpreter.

3. The interpreter calls entry MACROGET for subsequent STEPs, which are returned one per call in GR0. When the MP has no more STEPs to return, it returns a 0 in GR0. If MACROINT is called before a 0 has been returned by MACROGET, the MP will handle the appropriate nesting. A 0 is not returned until all nesting has been "popped" and the last macro has been returned.

4. Return codes (in GR15)

   a. 0 - normal return from MACROINT or MACROGET

   b. 4 - too few calling parameters, with undefined values (in MACROINT); fatal error. The MP displays the error message

   c. 8 - nesting level exceeds 500 in MACROINT; fatal error. The MP displays the error message

   d. 12 - debugging error for MACROINT or MACROGET: in the former, a negative number of dummy or calling parameters was encountered; in the latter, MACROGET was called when it had not STEPs to return; in either case, an appropriate message is displayed.

8.9 MATHEMATICAL OPERATORS Ross H. Hieber

8.9.1 Program Logic

The assembly code CSECTs are:

UINIT - The math operator's initialization section, containing the SYMODEFS. Completely reentrant-compatible, and uses OS macros.

UINIT# - The "PSECT" for UINIT.

UINIT2 - This is called by UINIT and is logically equivalent to a subset of UINIT, but is a separate CSECT to allow assembly of new SYMODEFS without reassembling everything; currently contains the scalar-vector mode combinations for the arithmetic operations. It uses UINIT# PSECT.

USTEP - Used as a function by FORTRAN functions to return in GR0 a STEP from the parameter list, needed to tell the operator call section of the interpreter what operand to put on the stack. No PSECT is needed.

The FORTRAN functions (subroutine for = since it needn't return a STEP in GR0) are intermediaries between the interpreter and FORTRAN library functions in most cases. Most of the intermediary FORTRAN functions are multiple entry, and there are several separate functions to allow addition of features without recompiling everything. These intermediate functions are needed to loop over the dimension, sometimes to check for arguments out of range with IF statements, and to return a STEP in GR0.

8.9.2 Nonfatal Handling of Arithmetic Errors

FORTRAN IF statements are used where appropriate to check for arguments illegal (too large, etc.) to FORTRAN library functions. A true zero is returned for any element of a vector which can't be calculated. Zero is used because i) this is most obvious on the graph, ii) it can never cause trouble in later operations. A few possible arithmetic errors not checked are exponent overflow and underflow, and $|\text{argument}| > 2^{50} \cdot \pi \sim 10^{15}$ to SIN & COS. The overhead for this approach of checking with IF statements would be intolerable in a batch FORTRAN program doing massive calculations, but it is quite practical where the arithmetic calculations are a minor part of the total time and storage.

8.9.3 Allowable Mode Combinations and Automatic Mode Conversion

Here we come to the weak point of the current implementation. Handling all possible mode combinations requires massive numbers of SYMODEFS and entries (e.g., 16 combinations each for +,-,*,/,** to handle vector-scalar and real*4-integer*4; and it goes up exponentially if we add vector-pair, complex, real*8, etc.). I have extended my original set to accept what now seems to be the vast majority of common needs, as described below.

1. Vector-pairs: Any vector input argument can also be vector-pair, with only the range used. No additional SYMODEFS or instances are needed. This allows using a graph as an operand.

2. In the rare cases where the other operators can't handle the modes, = can always be used first to convert since it will handle all 16. This takes 16 SYMODEFS, i.e., 16 instances, but fewer entries since (real real =) can be combined with (int int =), and (vec vec =) with (vec scalar =).
3. +,-,*,/,**: These take two arguments, of which either both are vector-real*4 (or vector-pair, of course), or else one only may be scalar-real or scalar-integer. I added the scalars by request since such operations as (3 ID *) are very common.
4. All other operators: Will accept vector(-pair) only.
5. **: FORTRAN's ** library functions will not accept (negative base) ** (≠0 real power) even though a real power of integer value is well-defined. Since DEFCONST produces a real scalar to use as exponent, and present vector exponents are all real, I've glitched the two applicable FORTRAN function entries. If the exponent is integer value within 1 part in 10^6 I use (negative real base) ** (integer exponent).

8.9.4 Meaning of Vector-Scalar Combinations

For +,-,*,/,**: Modes are for the 2 input arguments.

1. Vector vector: The calculations are done and stored in the temporary element by element (as scalars for

each element).

2. Vector scalar: The scalar is used as if it were a constant vector, with vector temporary output.

3. Scalar scalar: This could produce a scalar temporary output, but a constant vector output was preferred to avoid later mode problems with vector-only operators that use this temporary as an input argument.

For =:

Equals is unique in not producing a temporary; it gets its output operand from the stack with predefined mode.

1. Vector-vector: Obvious, just copied over.

2. Vector in, scalar out: The first element of the vector is copied into the output.

3. Scalar in, vector out: Produces a constant vector out, currently necessary as the only way to get constants to some operators.

4. Scalar in, scalar out: Obvious to the meanest intelligence.

In summary, the most commonly needed mode combinations are provided. A policy of always producing vector-real temporaries as output is followed to avoid mode problems with subsequent operators.

8.9.5 Notes on Other Operators (Not Calling FORTRAN Library)

1. INTEG, INTEG2, DIFF, DIFF2:

INTEG and DIFF integrate and differentiate the one vector argument versus the standard domain; INTEG2 and DIFF2 use the same routines but take the domain from the stack. DIFF uses divided differences between adjacent elements of the vectors, i.e., a 2-point formula, with no attempt at smoothing. Since this produces one less output element than the dimension, the last element is set equal to the second last; the expected jiggle on the last n points after $\frac{d^n}{dx^n}$ is no worse than the noise over the whole domain due to this crude method. Attempted division by ~0 gives 0 out for that elt. INGET uses trapezoidal integration, i.e., a 2-point formula; for the additive constant I set the first element = 0. The trapezoidal integration, though crude, works nicely within our resolution and has the advantage of being compatible with the differentiation.

8.10 UTILITIES Daniel R. Frantz

General Description

The basic idea behind the numeric conversion subgroup is to provide a limited subset of the IOH360 input/output operations so that the large core requirements of IOH360 can be reduced. To this end, three subroutines were written: floating point input, floating point output, and integer output. The input routine accepts a number in standard FORTRAN format E-type notation and coverts to a

REAL*4 internal representation. The output routines produce either an I14 or E0.6.14 formatted character string from an internal form of INTEGER*4 or REAL*4, respectively.

All routines use OS (I) S type calling sequences and do their best to simulate CSECT/PSECT reentrancy requirements to facilitate future conversion to TSS.

The current design of the system states that the input routine is called only by the interpreter and that the output routines are called only by numeric display, although this is not a limitation. These routines do not call any lower level subroutines.

Limitations and Future Work

The mathematical analysis system as designed by the Communication Sciences 673 class provides the option of expanding the type of numbers used to include REAL*8 and INTEGER*2. The current routines don't handle these formats explicitly in themselves, but obvious simple manipulations by the calling programs should be able to provide at least usable results (i.e., converting short integers to long before output, and zeroing the low-order word on short input to simulate long input).

If the decision is made to go ahead on the alternate number form, the calling sequences for output should probably be expanded to include a parameter indicating long or short forms. To make things neater, this parameter could

also be extended to be a four-way switch (simulating the interpretive abilities of IOH360) on the form of the output so that there would be only one output routine name rather than the two currently used. The output routines would have to be changed very little to provide this extra capability. The input routine currently converts numbers to the long form and then truncates before returning so that the changeover for this routine would be even simpler.

Mode and Structure Operators

The Numeric Conversion subgroup also programmed the operators VECTOR, SCALAR, REAL, and INTEGER. These operators enable the user to declare the mode (REAL or INTEGER) and structure (VECTOR or SCALAR) of a variable-length list of symbols. Additional operators of this sort for future expansion (e.g., COMPLEX or VCTRPAIR) can be added to the routine very simply by inserting a call to an internally defined macro. The label field or the call should contain the subroutine entry name, and the single macro parameter is the name of the symbol table mode bit to be set. The symbol table macro calls to set up the light buttons, operator definitions, and parameter list structures are made in the initialization section.

CONVERT A FULL-WORD INTEGER TO EBCDIC

Name:	NCINT
Purpose:	To convert a long-form integer (INTEGER*4) to an EBCDIC string of characters in format I14.
Calling Sequence:	OS (I) S type.
Entry:	GR1 contains the address of a list of adcons: 1. A(full-word integer to be converted) 2. A(14-byte region for I14 EBCDIC string output)
Exit:	No return code.
Note:	No other subroutines are called.
Algorithm:	The number is divided by successively smaller powers of 10 (decimal). The quotient is used as the next lower digit and the remainder is used in the next step. Leading zeros are converted to blanks.

CONVERT SHORT FLOATING-POINT NUMBER TO EBCDIC

Name: NCFPT

Purpose: To convert a short-form floating-point number (REAL*4) to a string of characters in format E0.6.14.

Calling Sequence: OS (I) S type

Entry: GR1 contains the address of a list of adcons:

1. A(short-form floating-point number to be converted)
2. A(14-byte region for E0.6.14 EBCDIC string output)

Exit: No return code.

Note: No other subroutines are called.

Algorithm: Reduce the number to one of form (hexadecimal exponent=0) (mantissa normalized) by multiplying by 10 or 0.1 if the hexadecimal exponent is less than or greater than zero, respectively, keeping track of the number of multiplications and the sign generated (negative and positive, respectively). This number is the tentative decimal exponent. The mantissa, M, is then of the form $1/16 \leq M < 1.0$. This number is converted to a decimal fraction by successive multiplications by 10(decimal) and taking the overflow past the decimal (rather, hexadecimal) point as the base ten number to the right of the point, in successively lower positions. If the first digit so generated is zero, decrease the decimal exponent by one.

DECLARE MODE OR STRUCTURE OF SYMBOLS

Name: NCREAL
NCINTGER
NCSCALAR
NCVECTOR

Purpose: To call symbol table management for the purpose of setting mode bits in the definition of a variable length list of symbols.

Calling Sequence: OS (I) S type.

Entry: GR1 contains the address of a list of adcons:

1. A(number of pairs of words in following list=n)
2. STEP (Symbol Table Entry Pointer) of the first symbol.
3. Value of the first symbol (from STE)
4. STEP of the second symbol
5. Value of second symbol

...

2n. STEP of last symbol in list
2n+1. Value of last symbol in list

Exit: No return code.

Notes:

1. The form of the user call on these operators during a run is to "push" the following succession of buttons (e.g., for SCALAR):
   (VAR1 VAR2 VAR3 ... VARn) SCALAR
2. The value words in the parameter list are not used or needed, but are provided as part of the generalized operator calling sequence.
3. The symbol table macro calls to set up the light buttons, operator definitions and paramater list structure are made in the initialization section.
4. Subroutine called: SYMSMS.

BIBLIOGRAPHY

1. Ruyle, A., Brackett, J.W., and Kaplow, R., "The Status of Systems for On-line Mathematical Assistance," Proceedings ACM National Meeting, 1967, pp. 151-167.

2. Karplus, W.J., On-line Computing, McGraw-Hill, New York, 1967.

3. Culler, G.J., "User's Manual for an On-line System," in Karplus, W.J., On-line Computing, McGraw-Hill, New York, 1967, pp. 303-324.

4. MTS: Michigan Terminal System, 2nd ed., Computing Center, and Concomp Project, University of Michigan, Ann Arbor, 1967.

5. IBM Document No. C27-6909.

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)*
UNIVERSITY OF MICHIGAN
CONCOMP PROJECT

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3. REPORT TITLE
MOMS: MICHIGAN'S OWN MATHEMATICAL SYSTEM

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Memorandum 27

5. AUTHOR(S) *(First name, middle initial, last name)*
TAYLOR, ROBERT W., editor

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
April 1970	160	5

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
b. PROJECT NO.	Memorandum 27
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	

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

MOMS--Michigan's Own Mathematical System--provides a facility for interactive, computer-aided mathematical investigations. The system runs on an IBM 360/67 under the Michigan Terminal System (MTS). An IBM 2250 Graphic Display is used for input and output. This report contains a user's manual as well as detailed documentation of the system organization and resulting modules. Sections on the relation between MOMS and the operating system are also included.

DD FORM 1 NOV 65 1473

Unclassified
Security Classification

Unclassified
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
computer-aided mathematics computer graphics IBM 360/67 IBM 2250						

Security Classification