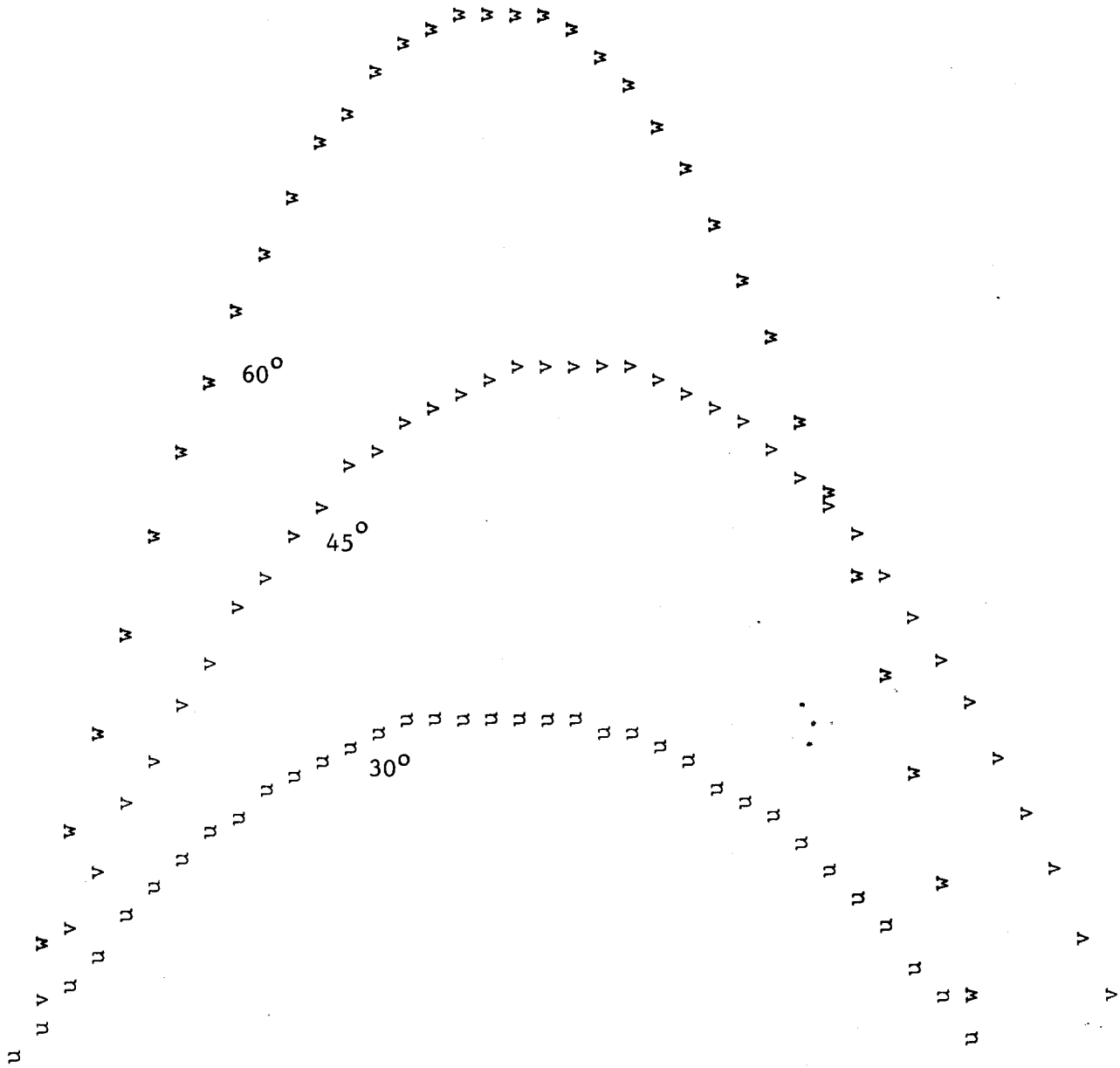
EERIE

SEPTEMBER 1, 1962

etc.

Robert A. Sharpe



EERIE

INTRODUCTION

This EERIE description applies to an interpretive system for the CYCLONE DIGITAL COMPUTER with a 16,384 word memory. The 1961 EERIE description applied to the CYCLONE when its memory was limited to 1024 words. It may also be noted that the present input tape preparation equipment is Friden FLEXOWRITER equipment instead of the previous TELETYPE equipment. This change has involved a shift to 8 level paper tape from the previous 5 level paper tape. From the standpoint of the user this means a rather significant convenience of tape preparation for EERIE programs since the keyboard is now essentially equivalent to a conventional typewriter.

**Rather significant changes have been made in EERIE itself. As a consequence the task of writing EERIE for the new machine was far from a trivial one. Some of the important changes are:

1. The new EERIE program uses floating binary arithmetic so that integer arithmetic may be performed exactly. This necessitated completely rewriting all of the arithmetic routines. It should be noted that this change also payed a dividend of about 30 percent faster oprating speed.

2. The ORDER structure of EERIE was approximately doubled and now includes trss1, trss2, tan, sinh, cosh, tanh, lxa, sxa, intgr, copy, swap, clear, tab, punch, crlf, space, trz, and other orders.

3. ALL ROUTINES were rewritten to take advantage of the order structure, expanded memory and block facilities.

4. A new organizational philosophy was used in the EERIE program so that the program could be rapidly and conveniently checked for accuracy and so that faulty student programs could not in general cause any harm to EERIE.

5. A NEW METHOD of addressing was added using ASTERISK notation.

6. ADDITION of an automatic debugging aid with the order FLAG.

7. ADDITION of constant listing facilities to improve program readability and minimize clerical effort.

**Mr. Paul Sampson programmed the arithmetic routines during the summer of 1961.

Mr. LaFarr Stuart wrote up, debugged, and tested the arithmetic routines during the fall of 1961. Mr. Stuart also aided the author in assembling and testing the complete EERIE program which used a total of 80 subroutines totalling more that 2800 machine language instructions and requiring more than 1 hour of FLEXOWRITER time to merely type out the complete program. The estimated total man time involved in the preparation of EERIE is 1000 hours.

Constant listing facilities were added by Mr. Robert Croft in early fall of 1962.

DESCRIPTION

Many additions have been made to the order structure of EERIE in order to make the program as versatile as possible. These are described in considerable detail in the section named EERIE ORDER CODES. At this point I wish to consider the changes involved in program tape preparation. First, note that each EERIE order must be typed as follows:

```
|margin |tab 1   |tab 2   |tab 3   |tab 4   |tab 5   |        |        |        |

        add      200;
```

The mnemonic operation code must appear at tab stop 1. It is NOT sufficient to set the margin over. The address must appear at tab stop 2. The address usually consists of a number giving the location in memory of the operand for the order. Less frequently an index is also specified. Still less frequently a decrement (or parameter) is required. Several examples are shown to illustrate how commas are used to control the decoding of the address information by EERIE.

```
add     200;          address only
add     ,5;           index only
add     ,,73;         decrement only
add     200,5,73;     address, index, and decrement
```

The ADDRESS specifies where an operand for the order is to be placed into or obtained from in execution of an order. The INDEX specifies one of 15 registers which may be used to modify the value of the address before the order is executed. INDEXING in EERIE is always SUBTRACTIVE. In other words, the order [add 200,5] will cause the computer to add the contents of location (200- present value in register 5) to the accumulator. Orders are available for setting, modifying, and testing these 15 registers. By convention index 0 is always maintained exactly zero. Thus, [add 200;] causes the machine to add the contents of location 200 to the accumulator. The DECREMENT is normally used to carry a parameter such as the size of an increment or the number of decimal places to be printed.

1. The address is limited to the range 0-4095. An address of 0 always references the accumulator.

2. The index is limited to the range 0-15. An index of 0 always causes the order to be obeyed without modification.

3. The decrement is limited to the range 0-4095.

Two types of information are typed on a program tape for EERIE. The FIRST type is simply an order which is to be stored directly in memory such as [add 200;]. The SECOND type is a PSEUDO order which is NEVER stored in the memory, but rather simply instructs EERIE what is to be done (e.g. [begin 200;] says to store subsequent orders at location 200 and following while [end 100;] says to stop storing orders and to start executing orders beginning at location 100).

NEW ADDRESSING METHOD

A method of addressing has been added to EERIE which facilitates greatly the construction of library subroutines. For this purpose an asterisk is used to denote the PRESENT LOCATION. Thus, if we wish to transfer unconditionally forward four orders we would write in the program merely [tru *+4;]. This addressing method also works very well for loading index registers since we may put 54 into index register 11 with the order [lxd *,11,54;] regardless of what location this order is to be stored in memory. Addresses may be formed as [*], [*+17], [*-25], or [37].

SAMPLE PROGRAM

Evaluate the polynomial that follows for 15 consecutive values of the variable x ranging from 0.00 to 0.14 in increments of 0.01. The polynomial is:

$$2.37 + 4.43\,x + 7.38\,x\uparrow 2 + 9.22\,x\uparrow 3 - 5.17\,x\uparrow 4 + 4.42\,x\uparrow 5 + 2.13\,x\uparrow 6$$

The first consideration in this example problem is a mathematical method. Polynomials are most conveniently evaluated using the method of nested factors---regardless of whether it is done by computer or by hand ---but this method has special significance for machine computation since it may be arranged to proceed iteratively. Thus, we arrange the evaluation in the form:

$$((((((2.13)x+4.42)x-5.17)x+9.22)x+7.38)x+4.43)x+2.37$$

Consider first how we would evaluate this polynomial in EERIE. What we would like to do is to pick up the highest order coefficient, multiply it by "x" then add the next lower coefficient, multiply it by "x", etc. until at the last step we add the lowest order coefficient. The address modification instructions are a natural for this operation. We will suppose that "x" is stored in location 1, the constant 2.13 in location 10, the constant 4.42 in location 11, etc. for the rest of the constants. Our program would then be:

```
lxd     *,1,6;          load index one with the constant 6
cla     10;             load the accumulator with 2.13
mul     1;              multiply the accumulated result by "x"
add     17,1;           add the coefficient "ai"
tix     *-2,1,1;        loop back to "mul" by "x" to repeat 6 times
```

After this sequence of orders the accumulator would contain the value of the polynomial.

Our complete program may be written as shown on next page. The double carriage returns have only been used for emphasis in separating connected sections of material and are not to be typed in the program---though they wouldn't hurt anything.

```
begin   100;            begin storing the program at location 100
lxd     *,1,8;          load index to input eight data values
inp     18,1;           input into location (18-value in index 1)
tix     *-1,1,1;        decrement index 1 and loop back 7 times

stz     1;              initialize value of "x" to zero
lxd     *,2,15;         prepare to loop a total of 15 times

lxd     *,1,6;          arithmetic loop shown above
cla     10;
mul     1;
add     17,1;
tix     *-2,1,1;        evaluation of polynomial complete

crlf    1;              punch a carriage return and line feed
out     ;               output the accumulator
cla     1;
add     17;
sto     1;              increment "x" by 0.01

tix     *-10,2,1;       completion of outside loop counting to 15
end     100;            pseudo order terminating input and storage
                        of orders
```

DATA TAPE:
2.13 4.42 -5.17 9.92 7.38 4.43 2.37 0.01

CONSTANT LISTING FEATURES

Quite frequently constants such as +2.0, +4.0, e = 2.7182818 and π = 3.14159265 are required in a program. For example, in finding the roots of a quadratic we need to evaluate a term of the form $(b^2 - 4ac)$. The constant of 4 in this expression is not connected with the equation, but nevertheless is required in the evaluation of the roots. Similarly, if we wish to find the circumference of a circle using the relationship $s = 2 \pi r$ we need the constant 2π. Other common constants required are 12 to convert feet to inches, 60 to convert minutes to seconds, and 5280 to convert miles to feet. All of these constants are important, since they are required in the solution of a problem, but they do not represent data since they remain the same for all problems of the same types.

The constant listing feature of EERIE is especially important for representing constants such as those just mentioned. For example, EERIE permits the following specialized order construction:

```
cla     157;
add     +1.0;           notice this order
sto     157;
```

This simple section of program will cause the number in location 157 to be increased by +1.0 every time that this sequence of orders is obeyed. Similarly, location 157 could be modified by any other numerical value.

In order that you may use this facility to best advantage you should understand how the constant listing feature of EERIE works. During the loading of an EERIE program into memory actual address of 157 will be supplied to the first and last order in the sequence given on previous page. The middle order, however, will receive quite special treatment. When a number is discovered in the address of this order---by the first address character being a plus or minus sign---then this constant will be listed at the end of the memory, and the address where it has been listed will be supplied as the actual address of this order. If this is the first constant listed then the constant +1.0 will be stored in location 4095 and the address 4095 supplied as the address of the order. The listing process attempts to make maximum memory utilization by storing a given constant only once and supplying the address of the first listing to all subsequent references to any particular constant.

As a consequence of the previous considerations certain rules may be deduced. These basic rules of usage are:

1. Zero may never be listed as a constant.

2. Program storage and variable storage is not permitted in a space at the end of the memory equivalent in size to the number of distinct constants which have been listed.

3. A constant should never be listed as the address of an order which stores anything in memory.

4. Any constant to be listed must start at the regular address position and must begin with a plus or minus sign.

Certain problems require a series of constants, but the nature of the evaluation involves a looping process which prohibits the use of the address constant listing feature just described. For example, consider the series evaluation of some basic function such as the exponential:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \ldots + \frac{x^n}{n!}$$

If we suppose that 5 terms are sufficient for the accuracy we require then we could use the constant listing feature by completely writing out the evaluation of the series---nested factor form---as follows:

```
cla     +0.00833333333;      1/5!
mul     100;                 100 is assumed location of x
add     +0.0416666667;       +1/4!
mul     100;                 times x
add     +0.166666667;        +1/3!
mul     100;                 times x
add     +0.5;                +1/2!
mul     100;                 times x
add     +1.0;                +1/1!
mul     100;                 times x
add     +1.0;                value now in accumulator
```

Although this program is reasonably compact for only 5 terms it is still more than twice as long as the looping form shown in an earlier example. Also, quite clearly the address constant listing feature is not applicable when the program is revised to use looping.

However, another feature of the input program with EERIE permits constants to be conveniently listed with the program, but without the bother of including a series of input orders and placing the constants on the data tape. All that is required is to simply type the constants one per line starting at the first tab stop. Each constant must be signed. In looping form the series evaluation of the exponential could then be written:

```
relative
loc     lxd     *,1,5           initialize index 1 to 5
        cla     *+5;            initialize accumulator
        mul     100;            multiply by x
        add     *+9,1;          add next coefficient
        tix     *-2,1,1;        loop back until index exhausted
        tru     1,15;           assumed exit back to main program
        +0.00833333333;         listed constant
        +0.0416666667;          listed constant
        +0.166666667;           listed constant
        +0.5;                   listed constant
        +1.0;                   listed constant
        +1.0;                   both of these ones are required
```

SUBROUTINES

Subroutines, as implied by the name, are sections of program designed to perform a task subordinate to a complete program. Subroutines deserve special attention for at least two very important reasons:

1. Libraries of subroutines may be constructed for many of the common tasks. In this way the job of the programmer may be rather dramatically reduced since he can then simply coordinate the use of the subroutines rather than program all details of the entire program.

2. However, even if no library routines are available for his partic lar task, he may rather significantly reduce the effort in writing and debugging a program by careful decomposition of his program into a series of subroutines---each one of which may be rather rapidly and conveniently tested. If the subprograms have sufficient generality he may save them to use later and so avoid the necessity of duplicating his own effort.

A subroutine is in principle just a section of a more complicated and complete program. There is however, a special feature which distinguishes a subroutine from a subprogram. This special feature consists of a single order in EERIE, but the generality and convenience implied by its use far outweighs the almost trivial effort required to learn how to construct a subroutine.

A trivial example program will be considered in illustrating the construction of a subroutine so that emphasis may be kept on the subroutine formation rather than on the task to be performed by the subroutine. Since there is no cube root command directly in EERIE suppose that we construct a subroutine to extract the cube root. This task can be most conveniently performed using logarithms as follows:

$$\sqrt[3]{x} = (x)^{1/3} = e^{1/3 \ln (x)}$$

Assuming that x is in the accumulator we may find the cube root of x with the answer in the accumulator with the series of EERIE orders:

```
log     0;
div     +3.0;
exp     0;
```

Now as the orders are shown this is simply a section of a larger program. In order to convert it to a subroutine we simply have to add the single order as shown:

```
log     0;
div     +3.0;
exp     0;
tru     1,15;
```

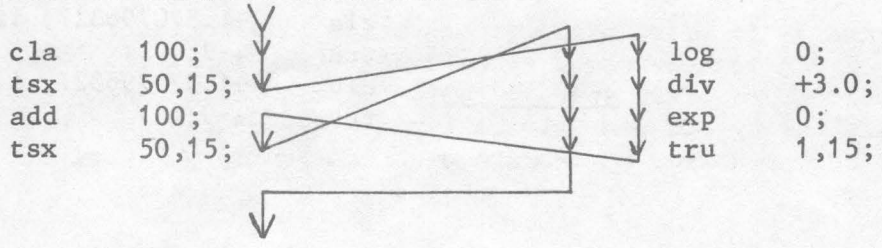
How do we use this subroutine? Simple. Suppose that inour main program that we are required to evaluate the expression:

$$\sqrt{x + \sqrt[3]{x}}$$

Our program would appear as follows in EERIE if we assume that x is in location 100 and that the cube root subroutine given previously begins at location 50:

```
loc 10)     cla     100;        load x into the accumulator
    11)     tsx     50,15;      take the cube root of x
    12)     add     100;        add x to the cube root of x
    13)     tsx     50,15;      take the cube root of (x plus cube root of x)
```

Now let us look at how the subroutine works. The order [tsx 50,15;] causes control to be transferred to location 50 and the <u>negative</u> of the location of the order to be placed in index register 15. After the cube root has been evaluated then the order [tru 1,15;] returns us to (1-[-location of order]) which is the location of the TSX order plus one---regardless of the location of the TSX order which branched to the subroutine. Thus, control passes through the sequence of orders as illustrated:



Insofar as the program is concerned it appears that the order [tsx 50,15;] is a cube root command. Similarly, a single such command can call upon a subroutine to perform any task which can be programmed.

An interesting and important saving in effort is produced in our program by virtue of the fact that the same coding for performing the cube root is used twice, but without having to repeat it twice. For commonly used subroutines this may save substantial memory space, minimize programming effort and significantly reduces the chance of a programming blunder.

Our illustration considers the simplest case where the subroutine operates on a single variable and produces a single answer. You might consider how several variables could be handled. Like most problems this one has many answers. One of the most elegant solutions is illustrated by a subroutine to convert from rectangular to polar form. The conversion formulas are as follows:

$$m = (x^2 + y^2) \quad , \quad \text{theta} = \arctan(y/x)$$

We will suppose that the values of x and y are always placed by the main program in the two locations immediately following the TSX order which transfers to the conversion subroutine. If the subroutine begins at location 50 then a single call on the subroutine would appear as follows:

```
tsx     50,15;
nop     ;
nop     ;
xxx     ddd;
```

```
cla     1,15;       obtain x
mul     1,15;       form square
sto     1;          temporary
cla     2,15;       obtain y
mul     2,15;       form square
add     1;          add square
sqrt    0;          square root
sto     1;          temporary
cla     1,15;       obtain x
trz     *+12;       zero is special case
idiv    2,15;       form y/x
itan    0;          form arctan (y/x)
sto     2,15;       replace y
cla     1,15;       test quadrant
trp     *+4;        result ok
cla     2,15;       no, must add π
add     +3.141592654;
sto     2,15;
cla     1;          obtain m
sto     1,15;       store in place of x
tru     3,15;       return to calling program
cla     2,15;       obtain y
trp     *+3
cls     +1.570796327; if neg. then -π/2
tru     *-7;
cla     +1.570796327; if pos. then +π/2
tru     *-9;
```

Notice that this program replaces the values of x and y carried into the program in the locations following the TSX order with the computed values m and theta. Also, the argument is returned in radians with the range $-\pi/2$ to $+3\pi/2$. Any other range could be selected by suitable program modification. The addresses following the TSX order are easily referenced:

1. first address is 1,15;
2. second address is 2,15;
3. third and return address is 3,15;

In order to avoid the machine hang up condition which prevails when x is zero in the evaluation of y/x a small program addition has been made to substitute $+\pi/2$ or $-\pi/2$ for the angle theta rather than attempting the division.

DATA HANDLING

Although a digital computer performs tasks at a very high rate, this should not cause the user to become careless and inefficient in his programming. For example, suppose that you can think of two ways to evaluate a given quantity; the first taking 12.4 milliseconds and the second taking 11.4 milliseconds. Even at a CYCLONE cost of $40.00 an hour this time savings doesn't seem worth the added effort. However, suppose now that this operation must be repeated over and over for tens of thousands of times so that the program requires several minutes computation time. If the first program required 12.4 minutes to run then it should be evident that the second would only run 11.4 minutes. This saving of 1 minute represents nearly a dollar and should not be lightly dismissed. If the first program should run 12.4 hours---and useful type programs might---then our 1 millisecond saving has mushroomed into an hour of time which is equivalent to a 40 dollar saving.

Now consider a subroutine for multiplication of two one-dimensional arrays together. Suppose that the x's are in location 3700-3799, the y's in 3800-3899 and the xy's are to be placed in locations 3900-3999. Then our program would be as follows:

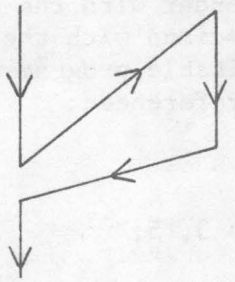
```
lxd     *,1,100;        load index for counting
cla     3800,1;         obtain x
mul     3900,1;         form xy
sto     4000,1;         store xy
tix     *-3,1,1;        loop back
```

If we wished to use such a section of program as a subroutine we would have to include a subroutine exit of the form [tru 1,15;] if we were to enter the program with the conventional subroutine linkage [tsx address,15;] . We could, however, generalize our program considerably by omitting the [lxd *,1,100; from the subroutine and placing it before the TSX transfer to the subroutine. In this way the program may be organized to only multiply together the desired number of elements.

The subroutine, as presently envisioned, is <u>not</u> very satisfactory or convenient because we have to move the data into the working locations 3700-3899 before calling on the subroutine and then move the data back from location 3900-3999 after using the subroutine. For example, consider use of our subroutine to multiply together x's from 250-299 with y's from 500-549 and place the xy's in location 350-399. Subroutine is assumed to begin in location 60.

POOR WAY

```
lxd     *,1,50;
copy    300,1,3800;
copy    550,1,3900;
tix     *-2,1,1;
lxd     *,1,50;
tsx     60,15;
lxd     *,1,50;
copy    4000,1,400;
tix     *-1,1,1;
```



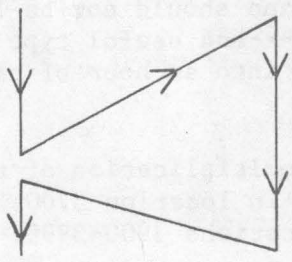
```
cla     3800,1;     obtain x
mul     3900,1;     form xy
sto     4000,1;     store xy
tix     *-3,1,1;    loop back
tru     1,15;       return
```

One feature which is poor about this program is that so much unnecessary data movement is required. For example, about 0.275 seconds data movement is required before entering the subroutine and about 0.175 seconds data movement is required after exiting from the subroutine---and only 50 points were moved. Another poor feature is that a maximum of 100 points can be multiplied together since only that much temporary working space has been allocated at the end of the memory.

Virtually all the poor features of the above program can be eliminated with one simple change in philosophy. Never move data if it can be avoided! This can be accomplished with great convenience in our example as illustrated by the following entry(on left) and subroutine (on right):

GOOD WAY

```
lxd     *,1,50;
lxd     *,2,300;
lxd     *,3,550;
lxd     *,4,400;
tsx     60,15;
```



```
sxa     *+3,2;
sxa     *+3,3;
sxa     *+3,4;
cla     0,1;
mul     0,1;
sto     0,1;
tix     *-3,1,1;
tru     1,15;
```

You should notice that index registers have been used to carry vital information to the subroutine. This is an especially important method since index registers may be loaded and unloaded faster than even the simplest arithmetic operations. The index register usage is as follows:

```
index 1     carries number of x's (or y's)
index 2     carries final data address plus one for the x's
index 3     carries final data address plus one for the y's
index 4     carries final data address plus one for the xy's
```

These four index registers are loaded before entry to the subroutine. Then the first business in the subroutine involves "planting" addresses in the corresponding orders. The whole process of loading and unloading the index registers requires only 0.010 seconds---regardless of the size of the arrays to be operated on. Compare this to 0.450 seconds required for a modest 50 points; the improvement is 45 to 1 even for this case.

Two more rather important considerations may be illustrated by this rather trivial example. First, notice that the size of the subroutine is relatively

unimportant compared with the size of the call on the subroutine when the subroutine is to be called on several times from several locations in the program. And second, notice that the GOOD WAY has the advantage of having an elegantly simple and clear entry; this feature is of fundamental importance when minimization of programming errors is considered.

PROGRAMMING TO AVOID EXCESSIVE DEBUGGING

One of the most perplexing problems facing the aspiring father of a program is that it doesn't work. Unfortunately, most programs of over 50 words do not work---at least when first tried. The errors may come from many sources. Those which result from miscopying and mistyping can be reduced by careful proofreading---but never seem to be entirely eliminated. It is not this type of error which is our present concern. We are concerned here with developing an overall philosophy of construction which tends to prevent the "unavoidable" errors of oversight---but even more importantly sharply reduces the time required by the programmer to eliminate the inevitable "bugs" from his program.

Probably the most serious mistake made by the novice programmer on a medium to large size problem is overestimating his ability to write error free programs. YOU WILL MAKE ERRORS! Anyone who claims otherwise either hasn't written any programs or is an outright liar. Now once you admit to YOURSELF that you program will have errors in it you have made a major step toward avoiding wasteful and expensive use of time on the computer to debug your program. The secret to success is careful organization of your program into sections and subroutines which can be individually tested and debugged. The situation is somewhat comparable to attempting to type up a 50 page manuscript---without error---on a continuous scroll of paper. As you can readily visualize this would be nearly impossible---even for the most proficient secretary. However, the task is relatively trivial in page form for a secretary of average ability.

It might pay at this point to estimate just how much is saved by dividing the task into smaller sections. Suppose that a secretary could type 50 percent of the pages perfectly the first time. The complete task then would require the typing of about 100 pages and this could be accomplished easily in three or four days. However, with the same degree of typing competence our secretary would likely have to attempt the task of typing an error free scroll a total of two to the fiftieth times---millions of millions---before producing an error free copy. ENOUGH SAID?

The task of arranging your program in subsections is not a trivial one, but the time spent is usually returned many fold. Not much help can be given with how and where to split a program since it depends so critically on the particular problem. However, the task of mechanizing the separation and testing of the subroutines can be simplified by considering the following experience of the author:

1. SIZE. Subroutines should be limited to approximately one page of EERIE orders. If necessary you can arrange to have subroutines of subroutines. The only caution in this case is to make certain that subroutine linkages to the various levels be made with different index registers---linkages on the same level can be made with the same index register since the saved information is always used before it is replaced with new information.

2. METHOD OF SECTIONING. Each tape should always be started with a carriage return and case shift---normally lower case. Each tape should always end in a carriage return and stop code. In this way each tape will stop without running out of the reader. The next tape may then be inserted in the reader and a BLACK SWITCH start will cause it to be read in the same manner. In general it is unwise to include BEGIN orders on subroutine tapes.

3. ADDRESSING. Avoid the use of absolute addresses wherever possible, since this makes it terribly time consuming and inconvenient to relocate the program. The examples in these notes have illustrated how * type addressing along with the constant listing features can permit subroutines to be constructed which do not depend in any way upon their location or the location of the data. The main program must use absolute addressing.

4. MEMORY ALLOCATION. A sound policy is to start your program at location 100. In this way you have locations 1-99 for temporary or working storage. Remember that location 0 is the accumulator and cannot be used for other temporary storage. Data for the program can usually be stored most conveniently from location 4000 back toward the front of the memory. Locations 4001-4095 have been avoided so that the constant listing feature can store its constants here without fear of storage overlap.

5. TAPE MARKING. Mark all tapes with program name, date, and number of the correction. The number of the correction helps avoid confusion as to which tape is the latest one. The importance of marking tapes immediately cannot be overemphasized. This simple task makes the difference between fun and frustration when you are working with your program.

6. TESTING. In general each subroutine should be thoroughly tested before incorporating it with the main program. Furthermore, do not just put all the routines together to form the main program. Rather, pick "core" routines and then keep adding to this nucleus a subroutine or two at a time until the whole program is assembled. Do not worry about the "wasted" time required to do this additional testing. Remember that you have localized the source of error in this fashion and that you will only have to change one short routine rather than to completely revise the whole program.

7. TAPE PREPARATION. Always rerun a tape that has been corrected to obtain an "untouched by human hands" copy of the program. If you do not do this you will end up hunting down "impossible" errors which exist because what is on the printed copy of the program does not agree with that read from the paper tape into the memory of the computer and executed.

8. FINAL PROGRAM. If you have used a stop code to halt each of your tapes you will find it very convenient to assemble the complete and final program when the errors have all been removed. The assembly is done by rerunning your tapes through the FLEXOWRITER in the same sequence as read into the computer. The stop code at the end of each tape will halt the FLEXOWRITER for convenient insertion of the next tape. TEST THIS COMPLETE TAPE BEFORE YOU ASSUME THE JOB IS DONE.

This list fairly well covers the basic items required in the production of a medium to large size program. One consideration with regard to testing the program in sections which is often overlooked is the possibility of replacing a complex subroutine by a dummy subroutine. In many cases the dummy subroutine can perform a trivial task---such as returning some special case such as zero or one. Furthermore, testing of routines can in many cases be done much more conveniently on special cases than on the actual data. Start out with very simple tests and work up to the complex tests. In this way gross blunders can be easily isolated and removed.

COMMON ERRORS

A list of common errors will be given at this point in an attempt to simplify your task of locating the errors.

1. Invalid mnemonics. When the computer halts at any point other than expected you have a mistake. DO NOT GO ON UNTIL YOU DISCOVER AND CORRECT THE DIFFICULTY. For some strange reason the computer is assumed to be a sort of extra-sensory-perception device which can read the mind of the user. NOT SO!

2. Address omission. An address other than zero is required for the orders TAB, SPACE, and CRLF. An index is required on all orders which contain an X in their mnemonic. A decrement is required on TIX, TNX, TXI, COPY, and SWAP.

3. Overstorage. If your program behaves very strangely and you become firmly convinced that the computer or EERIE is functioning incorrectly then look for orders which store data over program which is yet to be executed. For example, the following program will function most strangely and unpredictably:

```
begin    100;
inp      101;
out      101;
end      100;
+5.2713;
```

4. Incorrect transfer. Perhaps the most common error is made with transfer orders. This situation usually is produced by either miscounting or a correction which modifies the number of words in the program. In the latter case any transfer which occurs over the insertions or deletions must have its address changed. Infinite loops and other nonsense can easily be produced by a simple miscount of 1. A common error occurs on input when the TIX transfers one too early in the program:

```
lxd      *,1,5;
inp      200,1;
tix      *-2,1,1;
```

In this case the data tape will simply run through the reader without stopping and the program will go no further. This is because the index is reset every time and can never count down for exiting by the TIX order. The TIX must have a nonzero decrement or an infinite loop will also be formed.

5. Failure to initialize. When your program works fine for the first set of data and then fails for the second set of data it usually means that you have failed to initialize to zero all locations where results are accumulated. It is considered good programming practice to always initialize before using rather than initializing after use ready for the next use. For example, in computing the sum of the squares of the numbers in locations 100-199 we would write:

```
stz     1;          initialize temporary
lxd     *,1,100;    prepare to loop
cla     200,1;      obtain xi
mul     0;          form xi squared
add     1;          add temporary
sto     1;          return to temporary
tix     *-4,1,1;    loop back
```

Failure to include the STZ will not cause any difficulty the first time because EERIE clears the entire memory before loading a program, but a subseqent use would include the last sum of squares.

A COMPLETE EXAMPLE PROGRAM

At this point we shall consider the steps required to solve a typical computer problem. The example selected is that of determining the path of a baseball thrown at various velocities and angles. Air drag will be neglected in the computation so that our main attention may be directed to the method of organization rather than lost to the numerical details of the problem.

It will be assumed that the ball is thrown with an initial velocity of v0 feet/second and an initial angle to the ground of theta0. Our problem is then to determine the path of the ball described in the plane of flight. The equations of motion that we are concerned with are:

$$vx = v0 \cos(theta0)$$

$$vy = v0 \sin(theta0) - gt$$

Notice that the velocity in the x direction, vx, is constant until the ball strikes the ground. Also notice that the velocity in the y direction, vy, starts at its component velocity and linearly decreases because of the constant acceleration of gravity toward the center of the earth. The acceleration of gravity, g, is approximately 32.2 feet/sec/sec.

Our method of solution will be to choose a very small interval of time Δt and evaluate the distance traveled in the x direction as $\Delta x = (vx)(\Delta t)$ and the distance traveled in the y direction as $\Delta v = (vy)(\Delta t)$. Then x, y and t will be updated by these intervals and the process repeated again---and again. Although this method is only approximate---since in reality the velocity may change a small amount during the small time interval---we may make the accuracy sufficient for our use by simply choosing a small enough interval of time.

Another very basic consideration is how to present the data. While it could be presented as simply pages of numbers such an approach would turn out

to be both a waste of the time on the computer as well as a waste of your time in deciphering the significance of the results. A better way to present much computer output data is in the form of a graph. We shall choose to use this latter method of output. The particular method chosen is to draw a graph on the FLEXOWRITER by spacing over a distance on the page corresponding to the size of the number and print a distinctive character. We shall assume for the moment that we take care of this problem with an appropriate subroutine. Later we shall program the subroutine.

At this point we should precisely define the problem to be solved. Let us assume that we wish to find the relationship between the path of a ball thrown at the same initial velocity, but at different angles. One use of this comparison might be the selection of the optimum angle to throw a baseball if we wished to win at competition in a baseball distance throw. For definiteness we will assume that three different angles are to be considered for each initial velocity. We will also select the symbology on the plotted results as "u", "v" and "w". This presumably will be taken care of in the plot subroutine.

The next order of business is to allocate memory space for our program and working data storage. We shall assume that our main program begins at location 100 and that our plotting subroutine begins at location 200. Furthermore, we can assign data storage quite arbitrarily as follows:

```
location  3000)   v0
          3001)   theta1
          3002)   theta2
          3003)   theta3

          3004)   Δt1
          3005)   Δt2
          3006)   Δt3

          3007)   v0 sin (theta1)
          3008)   v0 sin (theta2)
          3009)   v0 sin (theta3)

          3010)   t1
          3011)   t2
          3012)   t3

          3013)   y1
          3014)   y2
          3015)   y3

          1)      position1
          2)      position2
          3)      position3

          4)      position-variable
          5)      position-end
```

You may notice that no space allocation has been made for x1, x2, and x3. We can avoid direct consideration of the x's in this special case because the velocity is constant in the x direction and so the x position is linearly connected with the time t. Thus, we choose an interval of time for each case, the Δt's, such that we advance a fixed amount in the x direction during each time

interval---we have chosen +0.5 feet for this purpose. For the normal range of velocities encountered in throwing a baseball---less than 140 feet per second---this corresponds to time intervals in the 5 millisecond range and should be small enough to permit reasonable accuracy in the results.

Rather than give any more prelude we shall directly consider the main program to perform the task described thus far. Explanatory material appears on the line with each order and should permit the program to be followed rather easily.

MAIN PROGRAM

```
begin   100;          begin storing orders at location 100;
inp     3000;         input v0 into location 200;
inp     3001;         input theta1
inp     3002;         input theta2
inp     3003;         input theta3
lxd     *,1,3;
cla     3004,1;       pick up thetai
div     +57.3;        convert to radians
sto     3010,1;       store temporarily, but still in acc.
cos     0;            cos(thetai)
mul     3000;         v0 cos(thetai)
idiv    +0.5;         +0.5/(v0 cos(thetai)) = Δti
sto     3007,1;       store Δti
sin     3010,1;       sin(thetai)
mul     3000;         v0 sin(thetai)
sto     3010,1;       store
tix     *-10,1,1;
clear   3010,,6;      clear 3010 to 3015, ti's and yi's
lxd     *,3,60;
lxd     *,1,3;
intgr   3016,1;       pick up integer portion of yi
sto     4,1;          store in temporaries to be used by plot
tix     *-2,1,1;
crlf    1;            punch a carriage return and line feed
tsx     200,15;       plot the three yi's as u, v, and w
lxd     *,2,10;
lxd     *,1,3;
cls     +32.2;        -g
mul     3013,1;       -g(ti)
add     3010,1;       v0 sin(thetai) - g(ti) = vyi
mul     3007,1;       vyi(Δti) = Δyi
add     3016,1;       yi + Δyi = new yi
sto     3016,1;       store new yi in place of old yi
cla     3013,1;       ti
add     3007,1;       ti + Δti = new ti
sto     3013,1;       store new ti in place of old ti
tix     *-9,1,1;
tix     *-11,2,1;
tix     *-19,3,1;
halt    100;
end     100;
```

Notice that a plot of the yi's for a horizontal displacement of only +0.5 ft. would yield a very long graph for typical values of initial velocity and angle. Consequently, an inner loop repeats the computation 10 times so that calls on the subroutine for plotting occur only every +5 ft. This is much better---insofar as accuracy is concerned---than to take time intervals corresponding to a step of +5 ft.

At this point we can use some of our previous suggestions and begin testing this program before we even bother writing the plotting subroutine. In order to do this we would simply write a dummy routine to replace plot for the time being. For this purpose a simple printout of the three integers in locations 1, 2 and 3 would be sufficient. Such a dummy subroutine could be whipped right out as:

```
begin   200;
lxd     *,1,3;
out     4,1,40;          output positions as 4 place integers
tix     *-1,1,1;
tru     1,15;            return to calling program
```

In order to test the program we also need a data tape with the constants v0, theta1, theta2 and theta3 typed in that order. This order is forced by the sequence of input orders in our main program. Suppose that we choose the initial velocity as 100 ft/sec and the angles to be 30, 45 and 60 degrees, respectively.

We then load our subroutine tape and it will stop on the stop code punched immediately after the last carriage return---a carriage return following the last order is essential. Then we load the main program and it will stop following the END pseudo order---not on the stop code in this case, but on the terminal carriage return and line feed. We then place the data tape under the reader and BLACK SWITCH START the program. When the computer stops, as determined by no sound from the loudspeaker, then remove the output tape and print it up on a FLEXOWRITER.

Providing we haven't made a blunder somewhere in the programming or typing we will get answers which we can hand check for accuracy. If they are incorrect then we know that the main program must be checked for errors. If they are correct then we can proceed to write the subroutine for plotting the results. Suppose that the main program is correct.

PLOTTING SUBROUTINE

The subroutine for plotting the three data points is supplied with 3 integers---because positions across a page are integer---and is expected to print the three characters u, v and w at positions matching these three integers. For example, if the three integers were 5, 23 and 45 we would want to space over 5 positions print a u, space over 17 more positions---note that the letter occupies one position---and print a v and finally space over 21 more positions and print a w. An important consideration arises at this point: Shouldn't tabulation codes be used where possible to minimize the computer time required to punch out the result? If we assume the tabulation stops are their normal 8 spaces apart then the savings in time is quite significant. For example, suppose that we consider a 66 line page with u, v and w a constant 64 positions from the left margin. The time required to punch this out with spaces would be over 1 minute as compared with about 8 seconds when tabs are used. Furthermore, the time required to type the spaced copy would be 6 minutes as contrasted with less than 1 minute for the tabbed copy.

One interesting feature shown in the plotting subroutine is the fact that orders can be moved with the COPY and SWAP orders. It is also possible to use the sequence CLA and STO to move an order---no other arithmetic sequence will work, however. Another interesting fact concerns the manner in which tabulation occurs on the FLEXOWRITERS. If the position of the carriage is within 1 position of a tab stop then a tab code will cause that tab stop to be skipped. Consequently, a sneaky little bit of coding was required in the plotting subroutine to advance the position counter in the manner that the carriage on the FLEXOWRITER would advance. We may calculate the resultant-position of the carriage from the initial-position of the carriage by the formula:

$$\text{resultant-position} = 8\ (\ 1\ +\ [\frac{1\ +\ \text{initial-position}}{8}])$$

We have used the notation [x] to mean the integer portion of x. Thus, if initial position is 6 then the formula gives:

$$\text{resultant-position} = 8(1\ +\ [\frac{1\ +\ 6}{8}])\ =\ 8(1\ +\ [\frac{7}{8}])\ =\ 8(1+0)\ =\ 8$$

And if the initial position is 7 then the formula gives:

$$\text{resultant-position} = 8(1\ +\ [\frac{1\ +\ 7}{8}])\ =\ 8(1\ +\ [\frac{8}{8}]\ =\ 8(1+1)\ =\ 16$$

Now consider how we would test this subroutine. In this case we would not use our main program given earlier, but would instead prepare a simple test program. In the test main program we would probably choose to have u and v zero and let w range from 0 to 20. In this way we would check to see that the tabbing is functioning correctly and would avoid all the swap orders in the sequence arranging prelude. Our test program might thus be:

```
begin   100;
stz     3;
lxd     *,2,20;
stz     1;
stz     2;
tsx     200,15;
cla     3;
add     +1.0;
sto     3;
tix     *-6,2,1;
end     100;
```

If the results of this test program---a straight line of w's---then you should make a test with all three variables changing. For example, you could have u start at zero, v start at 10 and w start at 20. Then u could be advanced +1 each time, v left alone and w advanced by -1 each time. An even better test would let u follow a sine wave, v be constant and w follow a cosine wave. If each had a reference value of 30 then the curves would cross each other in all combinations.

Notice that locations 1 and 2 must in general be restored each time that plot is used since it rearranges the values. Also notice that index 1 cannot be used in the test program since it would be destroyed in value by the use of index 1 in the subroutine.

3 VARIABLE PLOTTING SUBROUTINE

```
begin   200;
copy    *+54,,11;
copy    *+54,,12;       move punch commands to working area
copy    *+54,,13;
stz     4;              set position back to zero
cla     2;
sub     1;
trp     *+3;
swap    1,,2;
swap    11,,12;         this portion of the program
cla     3;              arranges the values of the
sub     2;              positions for u, v and w
trp     *+3;            into numerical sequence,
swap    2,,3;           and at the same time arranges
swap    12,,13;         the punch commands for u, v and w
cla     2;              into a matching sequence
sub     1;
trp     *+3;
swap    1,,2;
swap    11,,12;
lxd     *,1,3;
cla     14,1;           place i'th punch command in position
sto     *+27;
cla     4,1;            place i'th position into position temporary
sto     5;
sub     4;
trn     *+27;           skip printing if left of present position
cla     4;
add     +1.0;
div     +8.0;
intgr   0;
add     +1.0;           continue tabbing until the present
mul     +8.0;           position would exceed the position
sub     5;              of the character to be printed; this
trz     *+2;            is complicated by the fact that a
trp     *+5;            tab is skipped if within one space
punch   62;             before a tab stop
add     5;
sto     4;
tru     *-11;
cla     4;
add     +1.0;
sub     5;              then after tabbing as far as possible
trz     *+2;            space over until the present position
trp     *+5;            would exceed the position of the
punch   48;             character to be printed
add     5;
sto     4;
tru     *-7;
punch   52;             punch the i'th character u, v or w
cla     4;
add     +1.0;           advance position by one to account
sto     4;              for printing the i'th character
tix     *-32,1,1;
tru     1,15;           return from subroutine to main program
punch   52;
punch   53;
punch   54;
```

EERIE ORDER CODES

The EERIE order codes will be described in this section with the aid of specific examples for each order. The examples are typed in exactly the same form as they would be typed in a program, treating the left margin on each page as the left margin on the program sheet itself.

```
cla     127;
```

CLear and Add. Clear the accumulator to zero. Add the value in location 127 to the accumulator and leave the result in the accumulator. If the number in location 127 were -3.3372 then the final value of the accumulator would be -3.3372 after performing the order. This order does not disturb the value in location 127. This order is indexable and does not require a decrement.

```
cls     7;
```

CLear and Subtract. Clear the accumulator to zero. Subtract the value in location 7 from the accumulator and leave the result in the accumulator. If the number in location 7 were -3.3372 then the final value of the accumulator would be +3.3372 after performing the order. This order does not disturb the value in location 7. This order is indexable and does not require a decrement.

```
add     4091;
```

ADD. Add the value in location 4091 to the accumulator and leave the accumulated sum in the accumulator. If the accumulator originally contained +2.73 and the number in 4091 were -1.32 then the final value of the accumulator would be +1.41 after performing the order. This order does not disturb the value in location 4091. This order is indexable and does not require a decrement.

```
sub     133;
```

SUBtract. Subtract the value in location 133 from the accumulator and leave the accumulated result in the accumulator. If the accumulator originally contained +2.73 and the value in location 133 were -1.32 then the final value of the accumulator would be +4.05 after performing the order. This order does not disturb the value in location 133. This order is indexable and does not require a decrement.

```
mag     121;
```

MAGnitude. Place the magnitude of the value in location 121 into the accumulator. If the value in location 121 were -2.73 then the final value of the accumulator would be +2.73, regardless of its previous content. This order does not disturb the value in location 121. This order is indexable and does not require a decrement.

```
nmag    72;
```

Negative MAGnitude. Place the negative magnitude of the value in location 72 into the accumulator. If the value in location 72 were +3.2143 then the final value of the accumulator would be -3.2143, regardless of its previous content. This order does not disturb the value in location 72. This order is indexable and does not require a decrement.

```
mul     106;
```

MULtiply. Multiply the accumulator by the value in location 106 and leave the product in the accumulator. If the accumulator originally contained -2.4 and the value in location 106 were -2.0 then the final value in the accumulator would be +4.8 after performing the order. This order does not disturb the value in location 106. This order is indexable and does not require a decrement.

```
div     2134;
```

DIVide. Divide the accumulator by the value in location 2134 and leave the quotient in the accumulator. If the accumulator originally contained -3.63 and the value in location 2134 were -3.00 then the final value in the accumulator would be +1.21 after performing the order. This order does not disturb the value in location 2134. This order is indexable and does not require a decrement.

```
idiv    2134;
```

Inverse DIVide. Divide the value in location 2134 by the value in the accumulator and leave the quotient in the accumulator. If the accumulator originally contained -3.63 and the value in location 2134 were -7.26 then the final value in the accumulator would be +2.00 after performing this order. This order does not disturb the value in location 2134. This order is indexable and does not require a decrement.

```
sqrt    170;
```

SQuare RooT. Place the square root of the value in location 170 in the accumulator. If the value in 170 were +3.00 then the final value in the accumulator would be +1.7320508 after the operation, regardless of the original content of the accumulator. This order does not disturb the value in location 170. This order is indexable and does not require a decrement.

```
sin     24;
```

SINe. Place the sine of the value in location 24 in the accumulator. If the value in 24 were -0.785398163 then the final value in the accumulator would be -0.70710678 after the operation, regardless of the original value in the accumulator. Angles are assumed in radians and may be in any quadrant as well as greater than one revolution in magnitude. This order does not disturb the value in location 24. This order is indexable and does not require a decrement. Accuracy is lost if the angle is greater than 2 pi radians.

```
cos     24;
```

COSine. Place the cosine of the value in location 24 in the accumulator. If the value in 24 were -0.785398163 then the final value in the accumulator would be +0.70710678 after the operation, regardless of the original content of the accumulator. Angles are assumed in radians and may be in any quadrant as well as greater than one revolution in magnitude. This order does not disturb the value in location 24. This order is indexable and does not require a decrement. Accuracy is lost if the angle is greater than 2 pi radians.

```
log     1022;
```

LOGarithm. Place the natural logarithm (i.e. base e=2.71828...) of the value in location 1022 in the accumulator. If the value in location 1022 were +0.36787944 then the final value in the accumulator would be -1.0000000 after the operation, regardless of the original content of the accumulator. Natural logarithms may be taken of numbers in the full machine range of about $_{10}$-150 to $_{10}$+150. If logarithms to any other base are desired then this result may be modified by division by the logarithm to the base e of the desired base. The most common base change to base 10 is accomplished by multiplication of the natural logarithm result by +0.4342944819. Conversion of the natural logarithm result to base 2 is accomplished by a division by +0.6931471806. This order does not disturb the value in location 1022. This order is indexable and does not require a decrement.

```
exp     2170;
```

EXPonentiation. Place the value obtained by raising e (i.e. e=2.71828...) to the value found in location 2170 in the accumulator. If the value in location 2170 were +2 then the final value in the accumulator would be +7.3890561 after the operation, regardless of the original content of the accumulator. Exponentiation to any other base may be determined by standard combination exp[power*log(base)]. This order does not disturb the value in location 2170. This order is indexable and does not require a decrement.

```
itan    243;
```

Inverse TANgent. Place the inverse tangent of the value in location 243 in the accumulator in radians. If the value in location 243 were -1.0000000 then

the final value in the accumulator would be -0.78539816 after the operation, regardless of the original content of the accumulator. Since only one value is carried into this routine the result is limited to a determination of the principal value of the angle. Conversion of this result to degress may be done by a multiplication by +57.2957795. This order does not disturb the value in location 243. This order is indexable and does not require a decrement.

```
sto     2157;
```

STOre. Store a copy of the accumulator in location 2157, destroying the previous contents of location 2157, but leaving the value in the accumulator unchanged. If the value in the accumulator were -7.31 before the operation then after execution of the store order location 2157 would contain -7.31, regardless of its previous content, and the accumulator would still contain -7.31. This order is indexable and does not require a decrement.

```
stz     3152;
```

STore Zero. Store zero in location 3152 without affecting the value in the accumulator. This order would set the value in location 3152 to zero, regardless of its previous content. This order is indexable and does not require a decrement.

```
inp     250;
```

INPut. Input one number from paper tape and store in location 250, destroying the previous contents of location 250, but leaving the value in the accumulator unchanged. The number may be typed on tape in either fixed point form (e.g. +2 2.000000 3147 .00314672 2130000000) or in floating point form (e.g. $6_{10}3$ $-6.102_{10}-19$ $9.107_{10}-31$ $-_{10}7$ $3.1415926536_{10}+00$ $0.0031415926536_{10}3$). The rules for typing are so flexible that it is quite difficult to make an error. However, for definiteness we shall point out the following rules to be inflexibly followed:

1. No spaces, commas, or special characters may appear within a number.
2. Each number must be terminated in a space, tab, comma, semicolon, or carriage return and line feed---don't forget that the last number needs to be terminated too!
3. The number need not be signed plus if it is positive; the exponent need not be signed plus if it is positive; leading zeros are ignored before the decimal point; the decimal point is assumed to the right of the number if none is typed; numbers are limited to the range between $_{10}-150$ and $_{10}+150$ on input.
4. Each number, regardless of size or number of digits is compacted internally in floating point binary notation, accuracy is truncated to approximately 8 decimal digits, and the whole result stored in compacted form in exactly one CYCLONE word.

This order is indexable and does not require a decrement.

```
out     102,,124;
```

OUTput. Output the value in location 102 as a floating point number with two places before the decimal point and 4 places after the decimal point. This order does not affect the value of the accumulator, nor does it affect the value contained in location 102. The decrement portion of this order is used to specify the format of the printing. If we think of the decrement in terms of units, tens, and hundreds digits then we have:

1. hundreds digit — 1 for floating point, 0 for fixed point
2. tens digit — 0 to 9 digits before decimal point
3. units digit — 0 to 9 digits after decimal point

The following conventions have been followed on output format:

1. A positive sign on number is replaced by a space, but a negative sign is always printed.
2. The decimal point is not printed if zero digits are called for after the decimal point.
3. Two spaces are printed out before floating point numbers, however no additional spaces are printed before fixed point numbers.
4. All leading zeros, except for the one immediately before a decimal point are suppressed and replaced by spaces.
5. Spacing between columns of fixed point output is most easily accomplished therefore by simply requesting about 2 more places to be printed out before the decimal point than is warranted by the size of the numbers.

It is unwise to ask for more significant digits to be printed than is warranted by the accuracy of the data. In any case, however, the maximum number of significant digits should be limited to 8 since the numbers have been internally truncated to this length. If more than 11 significant digits are requested then the printed results will be complete nonsense. Several examples of output in various formats is shown below:

FORMAT:	20	80	42	135
	0	0	0.00	000.00000_{10}-99
	-1	-1	-1.00	-100.00000_{10}-02
	20	20	20.00	200.00000_{10}-01
	0	0	0.48	479.23457_{10}-03
	1	1	0.96	958.31232_{10}-03

This order is indexable and does use a decrement for format specification.

```
tru     235;
```

TRansfer Unconditionally. Transfer unconditionally to location 235 and start executing orders sequentially from this location forward. This order does not affect the accumulator or memory. This order is indexable and does not require a decrement.

```
trp     417;
```

TRansfer on Positive accumulator. If the accumulator is positive (zero is

treated positive) then the next order to be obeyed will come from location 417, otherwise the next order in sequence will be obeyed. This order does not affect the accumulator or memory. This order is indexable and does not require a decrement.

```
trn     735;
```

TRansfer on Negative accumulator. If the accumulator is negative (zero is treated positive) then the next order to be obeyed will come from location 735, otherwise the next order in sequence will be obeyed. This order does not affect the accumulator or memory. This order is indexable and does not require a decrement.

```
trz     233;
```

TRansfer on Zero accumulator. If the accumulator is identically zero then the next order to be obeyed will come from location 233, otherwise the next order in sequence will be obeyed. This order does not affect the accumulator or memory. Care must be exercised in the use of this order since the test is for an exact zero. For example, if a process required increments of 0.01 up to a final value of 1.00 it would be unsatisfactory to increment from zero to one in one-hundredths increments and test for termination by subtracting one from the running value and testing for zero. This test would fail to work because 0.01 is not expressable exactly in binary, just as 1/3 is not exactly expressable in decimal. Consequently, the addition of 0.01 a total of 100 times would yield 0.999999991 instead of exactly 1.000000000 and so a subtraction of exactly 1.00000000 would not yield a zero result. Notice, however, that integers are exactly expressable in binary and so it would be quite permissible to count from 0 to 100 in unit steps and test for a termination by subtraction of 100 from the running value and testing for zero. This process is adaptable, and very much preferred, for the case just cited. Simply divide the integer value by 100 (which is exact) to obtain intervals of 0.01 very precisely. This order is indexable and does not require a decrement.

```
ainp    334;
```

Alphabetic INPut. Input an alphabetic chain of characters and store them, five to a location, beginning at location 334 and continuing sequentially upward for however many locations required. The alphabetic chain must be delimited by quote marks. In determining the number of locations of storage which will be occupied by any alphabetic chain it must be noted that the count includes all characters within the quotes---including such things as tape feeds, backspaces, spaces, tabs, carriage returns, etc.---as well as the terminal quote mark. It also must be noted that if the material is to appear in upper case then an upper case must appear after the first quote mark---even though the FLEXOWRITER is already in upper case at this point. Ending up in lower case is automatically taken care of by the alphabetic output routine which supplies a lower case after each alphabetic output. For example, in order to have the sequence P17a= be stored in memory

it would be necessary to type [uc]["][uc][P][lc][1][7][a][uc][=]["] on the data tape; note each character has been included within brackets and the abbreviations uc and lc have been used for upper and lower case, respectively. In counting the memory space required we would find 9 characters and so a total of 2 locations in memory would be used. For the order given in the example this would amount to locations 334 and 335. This order is indexable and does not require a decrement.

```
aout    334;
```

Alphabetic OUTput. Output an alphabetic chain of characters from memory starting at location 334 and continuing until the terminating quote mark is found. For the example input given above we would have the output [uc][P][lc][1][7][a][uc][=][lc]. Notice that the terminating quote mark is not printed out and that a lower case has been supplied automatically by the alphabetic output routine. This order does not affect the accumulator or memory. This order is indexable and does not require a decrement.

```
lxd     200,7,53;
```

Load indeX from Decrement. Load index 7 from the decrement of location 200. If location 200 happens to be the location of this order then 53 will be placed in index 7. This situation need not be true but is quite convenient since the decrement of this order (i.e. lxd) is unused. The value placed in an index is unsigned and is maintained in the range 0 to 4095 (i.e. modulo 4096). Only unsigned integers may be placed in an order. This order is not indexable and no decrement is required. This order does not affect the content of the accumulator or the content of location 200.

```
sxd     200,7;
```

Store indeX in Decrement. Store the contents of index 7 into the decrement portion of location 200. If location 200 happens to be the location of this order then the present value in index 7 will be placed in the decrement of this order---which is otherwise unused anyway. The value in an index is unsigned and is maintained in the range 0 to 4095 (i.e. modulo 4096). This order does not affect the accumulator, or the content of index 7. This order is not indexable and no decrement is required.

```
lxa     200,5;
```

Load indeX from Address. Load index 5 from the address of location 200. If the address of location 200 should be 173 then 173 would be placed in index 5, regardless of the previous content of this index register. The value in an index is unsigned and is maintained in the range 0 to 4095 (i.e. modulo 4096). This order does not affect the accumulator, or content of location

200. This order is not indexable and no decrement is required.

```
sxa     200,5;
```

Store indeX in Address. Store the contents of index 5 into the address portion of location 200. The value in an index is unsigned and is maintained in the range 0 to 4095 (i.e. modulo 4096). This order does not affect the accumulator or the content of index register 5. This order is not indexable and no decrement is required.

```
txh     205,3,15;
```

Transfer on indeX High. If the value in index 3 is currently greater than the decrement portion of this order then control is transferred to location 205, otherwise if the value in the index is less than or equal to the decrement of this order the next order in sequence is obeyed. Thus, control is given to location 205 if the value in index 3 is 16 or larger, but the next order in sequence is taken if the value in index 3 is between 0 and 15. This order does not affect the accumulator or memory or index value. This order is not indexable and a decrement is required.

```
txl     205,3,15;
```

Transfer on indeX Low. If the value in index 3 is currently less than or equal to the decrement portion of this order then control is transferred to location 205, otherwise the next order in sequence is obeyed. Thus, control is given to location 205 if the value in index 3 is 0 to 15, but the next order in sequence is taken if the value in index 3 is 16 or larger. This order does not affect the accumulator or memory or index value. This order is not indexable and a decrement is required.

```
txi     312,3,1;
```

Transfer with indeX Incremented. The value in index 3 is raised by the value in the decrement of this order and then control is transferred unconditionally to location 312. Thus, if the value in index 3 is 17 before this order is obeyed then index 3 will contain 18 (i.e. 17+1) after this order is obeyed. Values in indexes are contained modulo 4096. Thus, repeated use of this order would cause the sequence in index 3 to be 17,18,19,...,4094,4095,0,1,2,3,4,5, 6,7, etc. on successive passes. The value in the decrement determines the amount by which the index is incremented and may be any value in the range of 0 to 4095 (i.e. the decrement is also kept modulo 4096). Consequently, it is possible to count down with this order by the simple expedient of using a decrement of [4096-count]. Thus, a decrement of 4095 will have the effect of decreasing the count in an index by 1 each pass through (e.g. 17+4095 taken modulo 4096 is 4112-4096=16).

```
tix     222,5,1;
```

Transfer on IndeX. The value in index 5 is decreased by the decrement of this order (i.e. 1) if the value in index 5 is greater than the decrement of this order and control is transferred to location 222. If, however, the value in index 5 is less than or equal to the decrement of this order then the index is unaffected and the next order in sequence is obeyed. Thus, if index 5 originally contained 5 then it would be decreased to 4 on the first pass through and control returned to location 222. On the next pass the value in index 5 would be decreased to 3 and control again returned to location 222. This would continue until the value in index 5 reached 1. On the very next test it would be found that the value in index 5 was <u>not</u> greater than the decrement and so the transfer to 222 would be disobeyed and the next order in sequence executed.

```
tnx     314,5,1;
```

Transfer on No indeX. The value in index 5 is decreased by the decrement of this order (i.e. 1) if the value in index 5 is greater than the decrement of this order and the next order in sequence is obeyed. If, however, the value in index 5 is less than or equal to the decrement of this order then the index value is unaffected and control is transferred to location 314. This order is therefore identical to the "tix" order except for a reversal of control transfer. This order does not affect the value of the accumulator or memory.

```
tsx     233,5;
```

Transfer and Set indeX. Transfer unconditionally to location 233 and set index 5 to contain the value [4096-location of tsx order]. Thus, if this order were located at address 116 then control would be transferred to location 233 and index 5 would contain 3980 regardless of its previous value. This order is intended for subroutine linkages. The exit from the subroutine can be simply written as [tru 1,5;] and upon its execution control will be given to location 117 (i.e. 1-3980 modulo 4096 is -3979+4096 which is location 117). The important thing to note is that control is transferred one location down from location of the tsx order used to enter the subroutine---regardless of where the entry came from. Thus, a given sequence of orders can be used many different times from many different locations in the program without any concern about returning from the subroutine to the correct place in the program.

```
tan     225;
```

TANgent. Compute the tangent of the angle expressed in radians in location 225 and leave the result in the accumulator, regardless of the previous content of the accumulator. Thus, if location 225 contained -0.7853981633 then the accumulator would be -1.0000000 after the operation. This order is indexable and does not require a decrement. The value in location 225 is unaffected.

```
sinh    224;
```

SINe Hyperbolic. Compute the hyperbolic sine of the value expressed in location 224 and leave the result in the accumulator, regardless of the previous content of the accumulator. The value in location 224 is unaffected. This order is indexable and does not require a decrement.

```
cosh    26;
```

COSine Hyperbolic. Compute the hyperbolic cosine of the value expressed in location 26 and leave the result in the accumulator, regardless of the previous content of the accumulator. The value in location 224 is unaffected. This order is indexable and does not require a decrement.

```
tanh    153;
```

TANgent Hyperbolic. Compute the hyperbolic tangent of the value expressed in location 153 and leave the result in the accumulator, regardless of the previous content of the accumulator. This value in location 153 is unaffected. This order is indexable and does not require a decrement.

```
halt    103;
```

HALT. Halt and transfer control to location 103 on a black switch start. However, on a white switch start take the next order in sequence. This order is indexable and does not require a decrement. This order does not affect the value in accumulator or memory.

```
lxn     212,5;
```

Load indeX from a Number. Load index 5 from the number contained in location 212. Thus, if the number in 212 is +217.00 then index 5 will contain 217 after this operation is performed. The numbers are not rounded before they are placed in an index so +217.0001, +217.545, and +217.999 will all be loaded into an index as 217. Furthermore, the next smallest integer is always loaded so that -.00001 will be loaded as 4095 (i.e. -1 modulo 4096). Similarly, -1.53 will be loaded as 4094 (i.e. -2 modulo 4096). Also, the modulus operation will cause +4137.37 to be loaded into an index as 41 (i.e. 4137-4096). The accumulator is not affected by this order. This order is not indexable and no decrement is required.

```
sxn     212,5;
```

Store indeX as a Number. Store the value in index 5 as a floating point number in location 212. Thus, if index 5 contains 217 then +217.000000 will be stored

in location 212 by this operation. The number stored from an index is always positive. The combination of an "lxn" and an "sxn" may be used to retrieve the integer portion of any number---and also the fractional portion by subtraction. This order does not affect the accumulator. This order is not indexable and does not require a decrement.

```
intgr   453;
```

INTeGeR. The integer portion of the number in location 453 will be placed in the accumulator, regardless of the previous content of the accumulator. Thus, if location 453 contained 2143.317 then the accumulator would contain +2143 after this order had been executed. There is no rounding in determining the result in the accumulator. Thus, 2143.0001, 2143.5000, and 2143.9999 will all cause 2143 to be placed in the accumulator. Rounding may be easily accomplished by the user, however. One-half is merely added to the number before the "intgr" order is executed. Negative numbers are entered in a fashion equivalent to the positive numbers although it appears different. For example, -.0001 would be entered as -1.00000, -0.9999 would also be entered as -1.0000, and -1.000001 would be entered as -2.00000. This really amounts to applying the rule that the "integer" value is determined as the SMALLEST integer which is contained in the value in the memory location.

```
crlf    3;
```

Carriage Return and Line Feed. Punch 3 carriage return and line feed characters on paper tape. The address may range from 1 to 15. Addresses outside of this range will cause either a machine halt or the printing of another character. This order does not affect the accumulator or the memory. This order is not indexable and does not require a decrement.

```
space   4;
```

SPACE. Punch 4 spaces on paper tape. The address may range from 1 to 15. Addresses outside of this range will cause either a machine halt or the printing of another character. This order does not affect the accumulator or the memory. This order is not indexable and does not require a decrement.

```
punch   17;
```

PUNCH. Punch one character "a" on paper tape. Whether or not it will print in lower case as "a" or in upper case as"A" is determined by the state of the FLEXOWRITER at the time the code is read in and printed. It is the responsibility of the user to see that this case shifting is taken care of before the character is punched---by punching the appropriate case shift character on tape first. Characters other than "a" are printed with codes given in the table below:

0 or) =0	+ or * =10	a or A =17	j or J =33	s or S =50
1 or _ =1	- or = =11	b or B =18	k or K =34	t or T =51
2 or ⏊ =2	; or : =12	c or C =19	l or L =35	u or U =52
3 or Δ =3	, or " =13	d or D =20	m or M =36	v or V =53
4 or [=4	. or ' =14	e or E =21	n or N =37	w or W =54
5 or] =5	$_{10}$ or ↑ =15	f or F =22	o or O =38	x or X =55
6 or < =6	stop =32	g or G =23	p or P =39	y or Y =56
7 or > =7	space =48	h or H =24	q or Q =40	z or Z =57
8 or Σ =8	feed =31	i or I =25	r or R =41	back sp =61
9 or (=9	lower c =58	crlf =59	upper c =60	tab =62
? or / =42	p. off =49			

```
randu   205;
```

RANDom Uniformly distributed number. Generate one random uniformly distributed number in the range between -1 and +1 and store in location 205. This order does not affect the accumulator. This order is indexable and does not require a decrement.

```
randn   224;
```

RANDom Normally distributed number. Generate one normally distributed number and store in location 224. The mean of these numbers is zero and the variance is 1.00. This number is obtained by simply adding three uniformly distributed numbers together and so is a good, but not highly exact, uniformly distributed number. Better distributions can be obtained by adding two of these numbers together and dividing by sqrt (2.0). This order is indexable and does not require a decrement.

```
nop     ;
```

No OPeration. This order performs no operation. Thus, it may be used rather indiscriminately in a program to correct errors which leave extra words in a program. It is often found convenient to insert a "nop" order every 10 words or so in the program so that a logical omission in a program can be corrected without causing serious address modification problems. HOWEVER, notice that if addresses are specified in the relative form given in the operation description of EERIE then little effort is usually required to add or delete statements from your program.

```
trss1   205;
```

TRansfer on Sense Switch 1. If sense switch one is in the OBEY position then control is unconditionally transferred to location 205. But if sense switch one is in the DISOBEY position then the next order in sequence is executed. This order does not affect the accumulator. This order is indexable and does not require a decrement.

```
trss2   205;
```

TRansfer on Sense Switch 2. If sense switch two is in the OBEY position then control is unconditionally transferred to location 205. But if sense switch two is in the DISOBEY position then the next order in sequence is executed. This order does not affect the accumulator. This order is indexable and does not require a decrement.

```
clear   225,,23;
```

CLEAR. Clear locations starting at 225 and running to 225+22 = 247. The address thus specifies the beginning location to clear while the decrement specifies the NUMBER of locations to clear. This order is indexable and requires a decrement. This order does not affect the accumulator. The particular advantage of this order is its rapidity. It will clear locations at a rate which is more than ten times that of a [lxd ;][stz ;][tix ;] combination. In fact, the time required is about 1500+200[n] microseconds for [n] locations.

```
copy    225,,325;
```

COPY. Copy the contents of location 225 into location 325. If this order is indexed then the single index modifies both the "from" and "to" locations. This order does not affect the accumulator nor the contents of the "from" address, but replaces the "to" location with the "from" value regardless of its previous contents.

```
swap    225,325;
```

SWAP. Swap the contents of location 225 with location 325. If this order is indexed then the single index modifies both the "from" and "to" locations. This order does not affect the accumulator. This single order replaces the sequence [cla x;][sto t;][cla y;][sto x;][cla t;][sto y;].

```
tab     5;
```

TAB. This order causes 5 tabulation codes to be punched on paper tape. The address may range from 0 to 15, but notice than an address of 0 causes the order to behave as NOP. This order is not indexable and does not affect the accumulator.

```
eerie   24;
```

EERIE. This order causes an immediate switch from execution of orders to input of a new EERIE program. This order is the counterpart of the pseudo

order END1 which causes an immediate switch from inputting an EERIE program to the execution of the stored program. The address of this order is immaterial. This order is stored in memory. It may be noted that the input and execution programs in EERIE are completely separate so that during order execution the location counter of the input program is not disturbed and during order inputting the accumulator is not disturbed. Consequently, a section of program may be input, this program run to compute some information, and finally control returned to continue inputting tape without a pause. This process is called an interlude in computing parlance. It may be used, for example, to conveniently place headings on program output material. Thus, we might prepare our program as:

```
        begin   1;
        ainp    5;
        aout    5;
        eerie   0;
        end1    1;
"This would be the heading which is to be printed out on the output data"
        begin   1;
        lxd     *,1,5;
        inp     200,1;              portion of the main program
        ..............
        ..............
```

This tape would load into memory in the following way:

1. The orders AINP, AOUT, and EERIE would be input and stored in memory in locations 1, 2, and 3 as specified by the pseudo order BEGIN.

2. Then a switch would be made from inputting orders to execution of orders beginning at location 1 by the pseudo order END1.

3. Then the orders AINP and AOUT would be executed and this would read the heading into memory and then immediately punch it out on paper tape.

4. Then the order EERIE would be executed and this would cause control to switch back to the portion of EERIE which inputs and stores orders. This would cause reading of the pseudo order BEGIN and then continuation of order storage at 1 and following.

You should notice that in this way we have managed to punch out our heading without using any memory space at least as far as our main program is concerned.

```
        flag    154,,1;
```

FLAG. This order instructs EERIE to flag the order in location 154 to a 1.

If the order is so flagged then automatic printout of this order will occur every time the order is encountered in execution of the program. This printout will include the location of the order, the order itself, the content of the index (if one is specified) after execution of the order, and the content of the accumulator after the order has been executed. If the decrement of this order is a 1 then the order will be flagged for printout, if the decrement is a 0 then the order will behave conventionally. It is thus possible to FLAG an order to be printed out and then to later FLAG the same order to inhibit printing. All orders are loaded with the FLAG a zero.

This order is indexable so that a sequence of orders may be flagged with a simple 3 word program. For example, suppose that we wished to FLAG to a one 35 orders beginning at location 123:

```
lxd     *,1,35;          prepare to loop 35 times
flag    158,1,1;         flag to a 1
tix     *-1,1,1;         loop back to flag
```

Later on in execution of the program we could FLAG these same 35 orders to a zero (i.e. so that they would not be printed on subsequent execution) with the order sequence:

```
lxd     *,1,35;
flag    158,1,0;         flag to a 0
tix     *-1,1,1;
```

CAUTION: Index 1 has been used here for simplicity. Care must be taken to insure the choice of index does not conflict with usage in your program.

You might wonder why provisions have been made to FLAG the orders back to the zero or nonprinting state. The answer is very simple, time. EERIE normally executes orders at an average rate of 500 per second. When the orders have been flagged for printout the execution time is limited by punch speed and is reduced to approximately 1 per second. Consequently, you must use very good judgment in the choice of orders to flag. Notice further that FLEXOWRITER types only about 1/6 as fast as the tape is punched so that only about 10 orders can be printed out per minute. However, since the order FLAG is stored in the memory as a conventional EERIE order it is possible to print out only essential material by proper timing of flagging to the print and nonprint states. No hard and fast rules may be given except possibly to say that if more than 1-2 minutes of debugging order printout is taken on an EERIE program you probably are wasting both your times---the time on the computer and the time on the flexowriter.

It is not necessary---nor particularly desirable---to prepare modifications to flag for printing or nonprinting as part of your main program. Instead, you should prepare them as corrections. In this way you will automatically end up with a final program tape---without the necessity for removing the debugging and without the possible errors caused by their removal. Suppose that you wish to FLAG location 137 to the printing mode between the orders in location 101 and 102 and FLAG location 137 back to the nonprinting mode between orders in location 141 and 142. Then the correction tape might look as follows:

```
begin   101;
tru     4000;
begin   4000;
xxx     ddd;        order in location 101 that was overwritten by tru  4000;
flag    137,,1;
tru     102;

begin   141
tru     4003;
begin   4003;
xxx     ddd;        order in location 141 that was overwritten by tru  4003;
flag    137,,0;
tru     142;
```

The effect of this correction tape is to replace the orders in location 101 and 141 by unconditional transfers to a section in memory where the over-written order is written followed by the FLAG order and then unconditional transfers back to locations 102 and 142, respectively.

CAUTION: Locations 4000-4005 have been used for simplicity. Care must be taken to insure that the location selected for the correction is clear. Also, the order transplanted in this way must not be one which is modified during execution of the program---or else the replacement unconditional transfers will be modified to nonsense.

A correction tape prepared as above may be entered into the computer any time after the program which it is to modify has been entered. For further details consult the section in the first half of the manual on general makeup and correction of program tapes.

PSEUDO ORDERS:

In addition to those orders in EERIE which must be stored in the memory for later execution we have also certain pseudo orders which merely instruct EERIE upon the handling of the program tape during its storage. These orders are NEVER stored in the memory.

```
begin   153;
```

BEGIN. This pseudo order instructs EERIE that it is to begin storing orders at location 153 and following. Consequently, the first order following [begin 153;] will be stored in location 153, the next order found on tape will be stored at location 154, etc. until this order is countermanded by a new order "begin". In many cases one "begin" instruction will be sufficient to store the entire program. As many "begin" orders as desired may be included in a program, however.

```
end     75;
```

END. This pseudo order instructs EERIE that it is to end the order storing phase and begin the order execution phase; taking the very first instruction to be EXECUTED from location 75. The END order must have an address specifying where the beginning of the program to be executed is located. This order is NOT stored in the memory. This order causes the machine to execute a black switch stop so that a data tape may be placed under the reader.

```
end1    75;
```

END1. This pseudo order is identical to END except that the computer does not stop, but rather goes immediately into execution of the program beginning at location 75. This pseudo order can be used to permit the inclusion of certain constants on the program tape---but after the program in the conventional fashion. A halt and transfer order can then be included in the program after the suitable number of input orders for inputting the data which has been included on the program tape. In this way it will seem as if the program and constants had been loaded as a complete entity.

```
pause   ;
```

PAUSE. This pseudo order causes the EERIE program to halt during the inputting of orders. This order has been included to permit the program to be assembled from several separate tapes, rather than requiring that all of the various tapes be reperforated as a complete package. You may achieve this same result using only a single stop code character if you obey the following rules:

RULES FOR USING STOP CODE AS A PAUSE:

1. Each section of program must begin with a carriage return.
2. Each section of program must be terminated in a carriage return.
3. This termination carriage return is immediately followed by a stop code.

PREPARATION OF PAPER TAPES:

```
|margin |tab 1  |tab 2  |       |       |       |       |       |

      cla     200,3,54;
```

Tapes containing orders for EERIE should be typed in the fashion shown above. The vertical bars at the top of the page have been typed at each of the preset tab stops on the FLEXOWRITER. The typing rules are simple but inflexible.

1. The operation code mnemonic must be typed at tab stop 1. It is absolutely not sufficient to set the margin right to the first tab stop.

2. The address must be typed starting at tab stop 2---leading zeros may be omitted from the address.

3. No spaces should occur within the address.

4. Address, index, and decrement must be separated with commas.

5. Every address must be terminated in a semicolon.

6. Comments may be typed to the right of the semicolon.

7. A carriage return and line feed MUST come after the "end" command on the program tape.

In actual tape preparation it is wise to follow the rules given below since from experience they lead to a minimum wasted time.

1. ALWAYS punch at least 6 inches of leader at the beginning of any tape using the TAPE FEED button. This key repeats and so punches the leader with a minimum effort.

2. NEVER advance the tape in the tape punch manually. The sprocket holes must be punched in the tape by the punch and this is done as the tape is advanced during punching.

3. NEVER manually interfere with the carriage on the typewriter. If during correction of a tape you find it necessary to position the carriage without wishing this information typed on the tape, then merely turn off the punch and use the SPACE, BACKSPACE, or CARRIAGE RETURN keys. BE CERTAIN THAT YOU TURN IT BACK ON BEFORE CONTINUING TYPING.

4. ALWAYS punch a carriage return and case shift at the beginning of a tape.

5. THEN start typing your program.

6. ALWAYS terminate your program in a carriage return.

7. ALWAYS punch at least 6 inches of tail at the end of a tape.

8. Tear the tape off at the punch, but DO NOT manually advance the punch---see 2 above.

CORRECTION OF PAPER TAPES IS MOST EASILY ACCOMPLISHED USING THE FOLLOWING RULES:

1. Try to avoid making errors by having the PROGRAM written very NEATLY and ORDERLY on a programming sheet.

2. Break up the typing into sections not more than one page long. Regardless of whether you think that you have made an error in typing a page or not, separate this material on the paper tape from the following typing by a section of TAPE FEEDS about 6 inches long.

3. Try to catch as many errors as possible at the time that they are made. At this time you can BACK UP the tape punch a suitable number of characters and DELETE the incorrect characters by using the TAPE FEED key which punches bottom 7 levels and is ignored on reading by the FLEXOWRITER. Then type the correct material manually and continue typing new material. This printed copy will be wrong, but the tape will be correct and a later reperforation may be used to delete the extraneous tape feeds if desired.

4. If some errors slip by do not worry about it at the moment. Simply continue typing up the program.

5. After the typing has been completed then take each section of tape---corresponding to no more than one page---and correct it in the following way:

   a. Set the FLEXOWRITER to ALL PUNCH and copy a new tape while typing a new copy of the program.
   b. As the error is neared depress the START READ key. This will halt the machine as long as it is kept depressed. Now advance the tape one character at a time by releasing the START READ key and IMMEDIATELY depressing again. When you are one character ahead of incorrect character hold down the STOP READ key as you release the START READ key.
   c. Advance the reader tape over the error manually and manually type in the correct information.
   d. Push the START READ key and continue duplication until you near the next error and then correct this error as before.

6. When each of the sections of tape has been corrected in the fashion indicated in step 5 then reperforate all of these tapes into one complete tape which should now be error free.

7. Proper use of the FLEXOWRITER should permit you to prepare an error free page of EERIE orders in about 15 minutes.

8. You need learn only three paper tape codes to permit you to rapidly find the error on the tape being read and bypass it. These are CARRIAGE RETURN AND LINE FEED, TABULATION, and SPACE. From experience you can easily find line and word by checking only these characters. Then delete and retype whole word if in doubt.

CYCLONE OPERATING PROCEDURE

1. ALWAYS bring a well marked copy of your program tape and a copy of the printed program to the CONSOLE of the CYCLONE. The printed copy of the program should be one which has come from printing the very LATEST program tape WITHOUT any MANUAL INTERVENTION on your part---the CYCLONE cares little for your intentions, it just does what you have placed on the tape.

2. BEFORE touching the computer you must SIGN IN on the LOG BOOK. You are to sign the beginning TIME to the nearest minute, your PROBLEM NUMBER---you are not to use the machine without having obtained one FIRST, your NAME, and whether the use is CC--code checking or P--paid.

3. YOU ARE NOT TO DISTURB THE MACHINE unless it is available for use. This can usually be determined from the log book. IF an ENGINEERING CODE is presently running then in general it is available for use---this is determined by the present problem number being preceded by EC (e.g. EC-111 or EC-117). IF THERE IS ANY DOUBT consult the MACHINE OPERATOR.

4. DO NOT PUSH ANY OF THE BUTTONS UNLESS YOU HAVE BEEN INSTRUCTED IN THEIR USE. If you fail to heed this warning you may destroy programs which have been previously stored in the memory.

5. ALWAYS clean up your work space at the FLEXOWRITER and most especially at the CONSOLE of CYCLONE when you leave. SQUARE GREY BASKETS are NOT wastebaskets, but paper tape storage bins. Two large ROUND WASTEBASKETS have been supplied for your use.

COMPLETE LIST OF CODES, PAGE REFERENCES, AND TIMING:

add	ADDITION TO ACCUMULATOR	Page 1	3.6 ms
ainp	ALPHABETIC INPUT TO MEMORY	Page 6	*
aout	ALPHABETIC OUTPUT FROM MEMORY	Page 7	*
cla	CLEAR AND ADD TO ACCUMULATOR	Page 1	2.0 ms
cls	CLEAR AND SUBTRACT FROM ACCUMULATOR	Page 1	2.0 ms
clear	CLEAR MEMORY MACRO INSTRUCTION	Page 13	2.0 + 0.2n ms
copy	COPY	Page 13	2.4 ms
cos	COSINE OF MEMORY TO ACCUMULATOR	Page 3	13.6 ms
cosh	HYPERBOLIC COSINE TO ACCUMULATOR	Page 10	27.5 ms
crlf	CARRIAGE RETURN AND LINE FEED	Page 11	*
div	DIVISION OF MEMORY INTO ACCUMULATOR	Page 2	4.2 ms
eerie	EERIE. LOAD ORDERS WITHOUT HALTING	Page 13	1.0 ms
exp	EXPONENTIATION TO BASE E	Page 3	14.5 ms
flag	FLAG. DEBUGGING AID CONTROLLING ORDER PRINTOUT	Page 14	1.4 ms
halt	HALT. TRANSFER ON BLACK SWITCH START	Page 10	**
idiv	INVERSE DIVIDE(ACC. INTO MEMORY)	Page 2	4.2 ms
inp	INPUT A NUMBER FROM PAPER TAPE	Page 4	*
intgr	INTEGER VALUE TO ACCUMULATOR	Page 11	2.6 ms
itan	INVERSE TANGENT IN RADIANS TO ACC.	Page 3	16.6 ms
log	NATURAL LOGARITHM	Page 3	14.0 ms
lxa	LOAD INDEX FROM ADDRESS	Page 7	1.2 ms
lxd	LOAD INDEX FROM DECREMENT	Page 7	1.7 ms
lxn	LOAD INDEX FROM A NUMBER	Page 10	2.2 ms
mag	MAGNITUDE OF MEMORY TO ACCUMULATOR	Page 1	2.0 ms
mul	MULTIPLICATION	Page 2	4.2 ms
nmag	NEGATIVE MAGNITUDE TO ACCUMULATOR	Page 2	2.0 ms
nop	NO OPERATION	Page 12	0.8 ms
out	OUTPUT NUMBER ONTO PAPER TAPE	Page 5	*
punch	PUNCH A CHARACTER ON PAPER TAPE	Page 11	*
randn	RANDOM NORMALLY DISTRIBUTED NUMBER	Page 12	6.0 ms
randu	RANDOM UNIFORMALLY DISTRIBUTED NUMBER	Page 12	4.0 ms
sin	SINE OF ANGLE IN RADIANS TO ACC.	Page 2	13.6 ms
sinh	HYPERBOLIC SINE TO ACCUMULATOR	Page 10	27.5 ms
space	PUNCH A SPACE ON PAPER TAPE	Page 11	*
sqrt	SQUARE ROOT TO ACCUMULATOR	Page 2	10.1 ms
sto	STORE A COPY OF ACC. IN MEMORY	Page 4	1.4 ms
stz	STORE A ZERO IN MEMORY	Page 4	1.4 ms
sub	SUBTRACTION FROM ACCUMULATOR	Page 1	3.6 ms
sxa	STORE INDEX AS AN ADDRESS	Page 8	1.4 ms
sxd	STORE INDEX AS A DECREMENT	Page 7	1.9 ms
sxn	STORE INDEX AS A NUMBER IN MEMORY	Page 10	3.0 ms
swap	SWAP TWO NUMBER IN MEMORY	Page 13	3.4 ms

tab	PUNCH A TABULATION CODE ON TAPE	Page 13	*	
tan	TANGENT OF ANGLE IN RADIANS TO ACC.	Page 9	28.8	ms
tanh	HYPERBOLIC TANGENT TO ACCUMULATOR	Page 10	35.0	ms
tix	TRANSFER ON INDEX	Page 9	1.8	ms
tnx	TRANSFER ON NO INDEX	Page 9	1.8	ms
trn	TRANSFER ON NEGATIVE ACCUMULATOR	Page 6	1.1	ms
trp	TRANSFER ON POSITIVE ACCUMULATOR	Page 5	1.1	ms
trss1	TRANSFER ON SENSE SWITCH ONE	Page 12	1.0	ms
trss2	TRANSFER ON SENSE SWITCH TWO	Page 13	1.0	ms
tru	TRANSFER UNCONDITIONALLY	Page 5	0.9	ms
tsx	TRANSFER AND SET INDEX	Page 9	2.0	ms
txh	TRANSFER ON INDEX INCREMENT	Page 8	1.9	ms
txi	TRANSFER WITH INDEX INCREMENT	Page 8	1.6	ms
txl	TRANSFER ON INDEX LOW	Page 8	1.9	ms
trz	TRANSFER ON ZERO ACCUMULATOR	Page 6	1.1	ms

* These orders involve mechanical input-output equipment. Present times are approximately 60 characters per second. The situation is complicated on output, however, by the fact that a buffer memory of 64 words links the CYCLONE and the output punch. If the buffer is not full then outputting requires negligible time, but outputting is limited to 60 characters per second after the buffer has been filled. IT THEREFORE BEHOOVES THE USER TO ARRANGE HIS COMPUTING AND PUNCHING IN BLOCKS OF 64 CHARACTERS OR LESS (I.E. ONE PRINTED LINE) IF IT IS AT ALL POSSIBLE.

** Timing does not apply.

LISTING OF THE OPERATION CODES IN RELATED GROUPS:

ARITHMETIC:
add,sub,mul,div,idiv,mag,nmag,cla,cls,sto,stz;

SUBROUTINES:
sqrt,sin,cos,tan,sinh,cosh,tanh,log,exp,itan,randu,randn;

TRANSFERS:
tru,trp,trn,trz; trss1,trss2; tix,tnx; txh,txl; txi; tsx;

INPUT-OUTPUT
inp,out; ainp,aout;

FORMAT:
crlf,space,tab,punch;

INDEX MODIFICATION:
lxa,lxd,lxn; sxa,sxd,sxn;

MISCELLANEOUS:
copy,swap,clear,halt,nop,eerie;

DEBUGGING:
flag;

ISU
COMPUTING
CENTER

EERIE

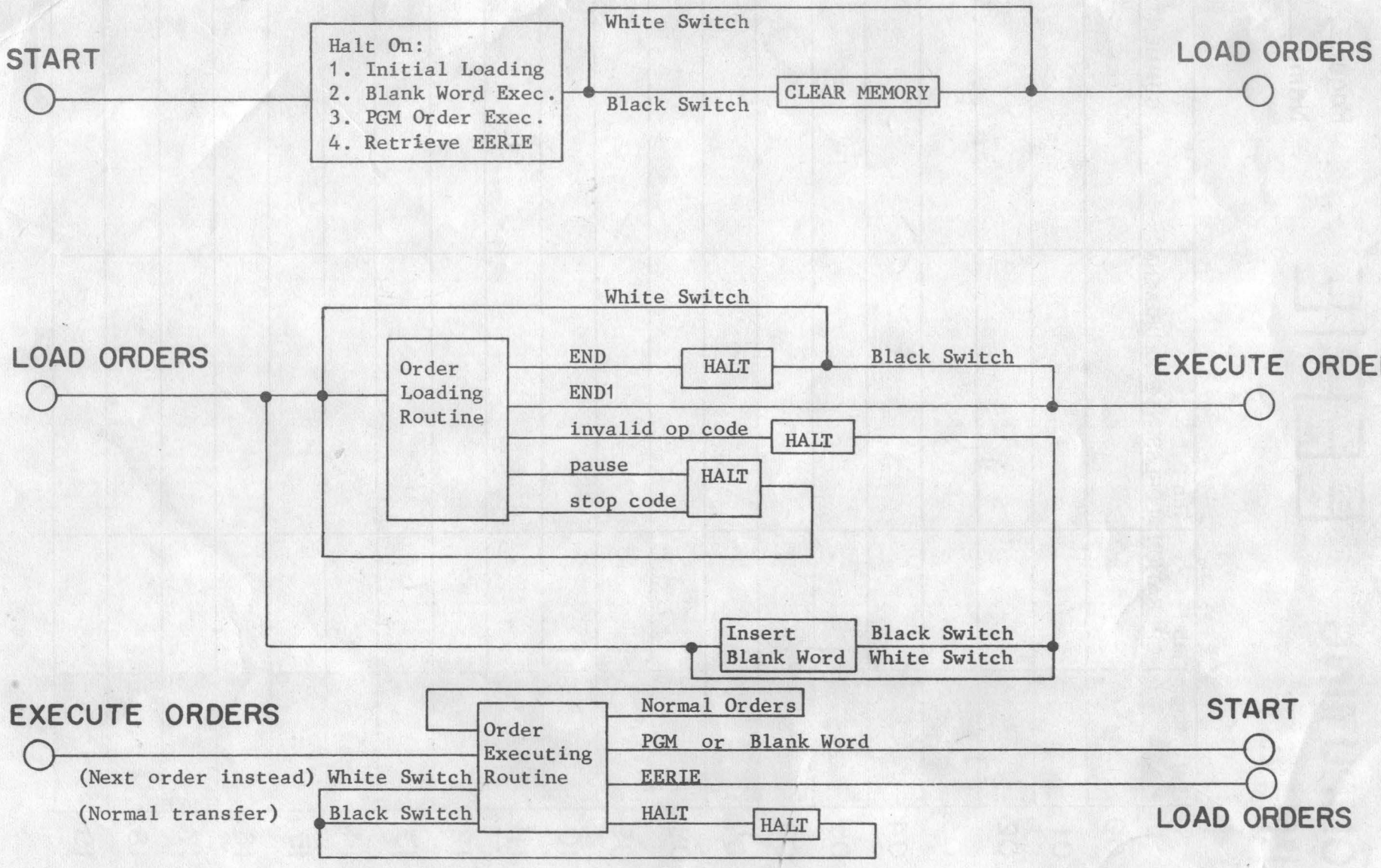
CYCLONE

Page _____ of _____

Name _____

	Left Margin	Tab 1 OPERATION	Tab 2 ADDRESS, INDEX, DECREMENT	COMMENTS
00				
01				
02				
03				
04				
05				
06				
07				
08				
09				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				

EERIE


START
Halt On:
1. Initial Loading
2. Blank Word Exec.
3. PGM Order Exec.
4. Retrieve EERIE
White Switch
Black Switch
CLEAR MEMORY
LOAD ORDERS
LOAD ORDERS
White Switch
Order
Loading
Routine
END
HALT
Black Switch
EXECUTE ORDERS
END1
invalid op code
HALT
pause
HALT
stop code
Insert
Blank Word
Black Switch
White Switch
EXECUTE ORDERS
Order
Executing
Routine
Normal Orders
PGM or Blank Word
START
EERIE
(Next order instead) White Switch
(Normal transfer)
Black Switch
HALT
HALT
LOAD ORDERS