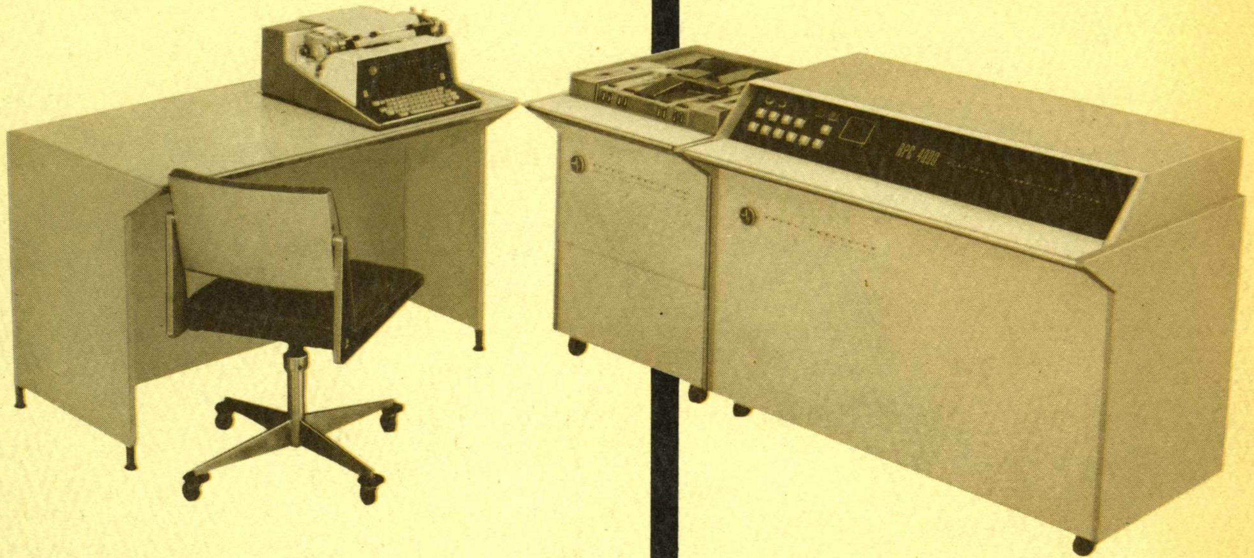
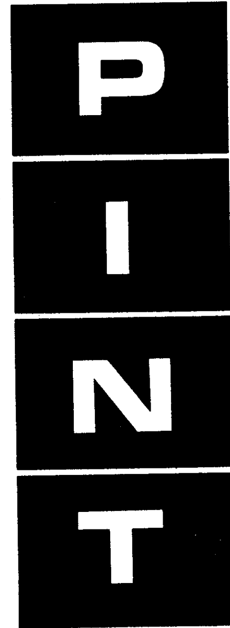
# RPC 4000

H1-02.0



# PINT program description

GENERAL PRECISION, INC.
**Commercial Computer Division**



PURDUE FLOATING POINT
INTERPRETIVE SYSTEM

# for the RPC 4000 General Precision Electronic Computer

Developed by
School of Electrical Engineering
Purdue University

**PROGRAM NO. H1-02.0**

© 1964, General Precision, Inc.

TABLE OF CONTENTS

SECTION 1 BASIC PROGRAMMING

1.1 Introduction
1.2 Available Functions
1.3 Memory
1.4 Numbers
1.5 Instructions
  1.5.1 Arithmetic Instructions
  1.5.2 Storage Instructions
  1.5.3 Functions
  1.5.4 Sequence-Changing Instructions
  1.5.5 Input and Output Instructions
1.6 Program Input and Operation
1.7 Programming Examples 1 and 2
1.8 Program Preparation
1.9 Post Mortem and Causes

SECTION 2 ADVANCED PROGRAMMING

2.1 Introduction
2.2 Floating Point Numbers
2.3 Additional Instructions
  2.3.1 Additional Arithmetic Instructions
  2.3.2 Additional Storage Instructions
  2.3.3 Additional Function
  2.3.4 Additional Sequence-Changing Instructions
  2.3.5 Additional Input and Output Instructions
2.4 Index Registers
  2.4.1 Description
  2.4.2 Instructions
  2.4.3 Address Modification
  2.4.4 Example
2.5 Alphanumeric Printing
  2.5.1 Additional Output Instructions
  2.5.2 ABC Character Codes
  2.5.3 Use of ABC
2.6 Decimal Memory Printout
2.7 Subroutines

SECTION 3 FURTHER PROGRAMMING

3.1 Introduction
3.2 Subroutine Construction
    3.2.1 Subroutine Instruction
    3.2.2 Example of Subroutine Instruction
3.3 Manual Intervention
3.4 Address-Changing Instruction
3.5 Additional Index Register Instructions
3.6 Additional Note on ABC
3.7 Coding Sheet Data Entry
3.8 Programs for General Locations
3.9 Additional Loading Code Words
3.10 PINT Stop Button
3.11 Compatible Output
3.12 Input Duplication
3.13 Errors Detected by Load and Input Routines
3.14 Basic Machine Language
3.15 PINT Summary
    Post Mortem Causes
    Program Loading Code Words
    Rules for Numbers
    Memory

SECTION 4 TECHNICAL CHARACTERISTICS OF PINT

4.1 Memory
4.2 PINT Instructions
4.3 PINT Numbers
4.4 Interpretation
4.5 Tagging
4.6 Comments
4.7 Speed

SECTION 5 EXTENDED MEMORY APPENDIX

SECTION 6 OPERATING PROCEDURES

6.1 To Turn the RPC-4000 ON and OFF
6.2 To Load the PINT Interpretive System
6.3 To Load a PINT Program
6.4 To Correct a Program
6.5 To Trace a Program
6.6 To Print Out a Program from Memory - DUMP
6.7 Post Mortem
    Error Printouts
6.8 To Load a Subroutine
6.9 Sense Switch Options

# SECTION 1

## BASIC PROGRAMMING OF PINT

### 1.1 Introduction

PINT, an interpretive system for the RPC-4000 computer, is designed to make the coding of problems for the RPC-4000 as simple as possible and to assist the programmer in carrying out many of the things which are often confusing or difficult to do. The system is quite flexible and yet is very easy to program and operate.

PINT is a program written for the RPC-4000 to enable it to understand an instruction code which is not its basic language. The program works with ordinary decimal numbers. The program carries out operations on these decimal numbers by interpreting the instructions which it is given.

### 1.2 Available Functions

PINT provides for:

1. Program loading
2. Data input
3. Data output (printing and punching)
4. Basic orders for
   a. Arithmetic
   b. Sin, cos, arctan, square root, $a^b$, $e^x$, $10^x$, $\log_e$, $\log_{10}$
   c. Logical decisions
   d. Input and output
   *e. Index Registers
   **f. Address modification (without index registers)
   **g. External intervention
5. Post Mortem
*6. Decimal memory printout
*7. Trace routine
*8. Alphanumeric output
**9. Subroutines
**10. Basic language connections

*Items discussed in section 2.
**Items discussed in section 3.

## 1.3 Memory

PINT has 1000 storage locations, numbered from 000 through 999. One instruction or one number occupies one PINT location, and certain programmed features make it impossible to accidentally use instructions for numbers or vice versa.

## 1.4 Numbers

The format for the input of numbers into the computer is the same as the format in which you would ordinarily write decimal numbers, with the following restrictions:

a. The number must start with its sign (+ or -).
b. The decimal point must appear within the number in its proper place.
c. No more than 8 actual digits may be used, but fewer are permitted.
d. The number must be followed by an asterisk (*), called a "stop code".

Examples of input and output formats are given below:

| Number | Input Format | Output Format |
|---|---|---|
| 2 | +2.0* | 2.000000 |
| -0.673 | -0.6730* | -.6730000 |
| 106.3725823 | +106.37258* | 106.3726 |

## 1.5 Instructions

Instructions are of the form ØØØxxx where ØØØ is the operation code and xxx is the address of the operand involved. When we define the individual instructions, it is handy to have some abbreviations for the quantities being processed. We will use xxx to refer to a general, three-digit address. We will use the symbol c() to mean "the contents of the location specified by the number in the parentheses." For example, c(132) means the number in location 132, and c(a) means the number in the accumulator. The following are the basic instructions for PINT. Other available instructions are discussed in sections 2 and 3.

### 1.5.1 Arithmetic Instructions

ADDxxx ADD c(xxx) to c(a) and leave in accumulator without changing c(xxx).

SUBxxx SUBtract c(xxx) from c(a) and leave in accumulator without changing c(xxx).

MULxxx MULtiply c(xxx) by c(a) and leave in accumulator without changing c(xxx).

DIVxxx DIVide c(a) by c(xxx) and leave in accumulator without changing c(xxx).

POS000 Make c(a) have a POSitive sign (absolute value).

### 1.5.2 Storage Instructions

CCFxxx Copy Contents From xxx into the accumulator without changing c(xxx).

CCIxxx Copy Contents of accumulator Into location xxx without changing c(a).

### 1.5.3 Functions

SQR000 Form the SQuare Root of c(a) and leave in accumulator [c(a) can't be negative].

SIN000 Form SIN c(a) and leave in accumulator [c(a) in radians].

COS000 Form COS c(a) and leave in accumulator [c(a) in radians].

ATN000 Form ArcTaN c(a) and leave in accumulator (principal value in radians).

PWRxxx Raise c(a) to the PoWeR c(xxx) and leave in accumulator [c(a) must be $>$ 0].

EXP000 Raise e to the EXPonent c(a) and leave in accumulator.

TEN000 Raise TEN to the exponent c(a) and leave in accumulator.

LNE000 Form the Natural log of c(a) and leave in accumulator [c(a) must be $>$ 0].

LOG000 Form the base 10 LOG of c(a) and leave in accumulator [c(a) must be $>$ 0].

### 1.5.4 Sequence-Changing Instructions

JMPxxx Take the next instruction from xxx (instead of from the location following the locations of this JMP instruction). That is, JuMP to xxx.

JINxxx If c(a) is negative, consider this as a JMP instruction. If c(a) is positive, ignore this instruction. That is, Jump If c(a) is Negative to xxx.

HLTxxx HaLT, then jump to xxx if the START button is pressed.

### 1.5.5 Input and Output Instructions

INMxxx INput data tape into consecutive Memory locations starting at location xxx. Continue until the symbol END* is reached on the data tape. Then take the next instruction in sequence. If another INM is performed, the next set of data are used, since the tape has advanced.

PFA000 Print From Accumulator a number on the typewriter.

CAR000 Perform a CARriage return on the typewriter.

## 1.6 Program Input and Operation

PINT receives both programs and data from punched paper tape. Certain formats must be followed in the preparation of these tapes.

The program tape requires certain commands at the beginning and at the end to tell the computer where the program is to go and what is to be done with it. None of these commands require space in memory. The program must start with the command CLEAR* which causes the machine to clear the entire memory (but not to zero) so that if you should make an error the machine can detect it more easily. (However, if your work consists of several programs which must be loaded together, CLEAR* should be used only on the first one.) After this command comes the command LOADxxx*. The address xxx is the address of the first location in memory into which you want to have your instructions loaded. The computer will start loading your instructions at this address and continue in successive locations from there on. At the end of the program, after the last instruction, the command BEGINxxx* must be written to tell the computer that this is the end of the program and that it should go to location xxx, take its first instruction from there, and continue in sequence from that point executing the instructions which you have written.

Data is prepared in the format shown in paragraph 1.4. Do not omit any of the necessary symbols, and do not forget to end the data tape with the command END*.

## 1.7 Programming Examples

In the examples which follow, we have assumed that the program starts in location 000.

### 1.7.1 Example 1

Assume that a number a is in location 500, a number b is in location 501, a number c is in location 502, and a number d is in location 503. We desire to form x = a + b + c + d and place the result in location 504.

| | | |
|---|---|---|
| | CLEAR* | code word to clear all memory |
| | LOAD000* | program loading code word |
| in location 000 | CCF500* | copy a into the accumulator |
| 001 | ADD501* | a + b |
| 002 | ADD502* | a + b + c |
| 003 | ADD503* | a + b + c + d = x |
| 004 | CCI504* | copy x into 504 |
| 005 | HLT000* | stop the computation |
| | BEGIN000* | program starting code word |

### 1.7.2 Example 2

We desire to read two numbers from tape, find out which is larger, and print only this larger number. We then want to return the carriage and obtain a new pair of numbers from tape. We want to test 100 pairs of numbers.

| | | |
|---|---|---|
| | CLEAR* | code word to clear all memory |
| | LOAD000* | program loading code word |
| in location 000 | INM080* | input first number into 080, second into 081, etc. |
| 001 | CCF080* | copy first number into accumulator |
| 002 | SUB081* | subtract second number from first |
| 003 | J1N006* | if -, second is larger; jump to 006 |
| 004 | CCF080* | if not -, copy first number |
| 005 | JMP007* | jump to 007 |
| 006 | CCF081* | copy second number |
| 007 | PFA000* | print first or second number |
| 008 | CAR000* | return the carriage |
| 009 | CCF082* | copy counter into accumulator |
| 010 | ADD083* | add 1 to counter |
| 011 | CCI082* | store counter + 1 in original place |
| 012 | JIN000* | if result was -, return for more numbers |
| 013 | HLT000* | if result was +, halt |
| | BEGIN000* | program starting code word |

The first set of data for this program must contain the first two numbers to be tested and the two numbers necessary for setting up the counter. The rest of the sets of data contain only the two numbers to be tested. An example of the first three sets of data follows:

+2.0*+1.0*-99.5*+1.0*END*+1.0*+3.5*END*-1.2*-3.6*END*

Let us examine the operation of this program in detail. After the program has been loaded into the machine starting at location 000, the computer is told to take its first instruction from location 000. The first instruction is INM080, which takes the first set of data shown above and inputs +2.0 into location 080, +1.0 into location 081, -99.5 into location 082, and +1.0 into location 083. It then discovers the END command and goes on to the next instruction in the program.

The program then subtracts 1.0 from 2.0 and obtains 1.0, which is a positive number. The JIN instruction is therefore ignored and the instruction in 004 is performed, copying the number 2.0 into the accumulator. Then the computer jumps to 007, prints this number, and performs a carriage return.

The next instruction copies the counter into the accumulator, which at this point reads -99.5, adds one to it, making it -98.5, and returns this value to location 082. Since the result of this addition is still in the accumulator, and since the result is negative, the JIN instruction is taken as an active jump instruction and the program returns to location 000 for its next instruction.

This time the INM instruction is stopped after two numbers have been input, placing the number +1.0 in location 080 and the number +3.5 in location 081. Note that locations 082 and 083 are unchanged by this instruction. The result of the subtraction instruction (002) leaves a negative result in the accumulator, so the JIN instruction is active and causes the computer to jump to location 006, where it copies the number 3.5 into the accumulator. Then it prints the result and performs a carriage return.

The operation of the counter is the same as described above, except that the counter is -98.5 and is changed to -97.5. Hence the program returns again to location 000 for more numbers. A little thought will reveal that, after testing the 100th pair of numbers, the counter will be -0.5, and will be changed to +0.5, causing the JIN000 to be inactive and stopping the computer.

## 1.8 Program Preparation

The following is the tape for Example 2, with the first three sets of data:

```
CLEAR*LOAD000*
INM080*CCF080*SUB081*JIN006*CCF080*JMP007*CCF081*PFA000*CAR000*
CCF082*ADD083*CCI082*JIN000*HLT000*
BEGIN000*

+2.0*+1.0*-99.5*+1.0*END*+1.0*+3.5*END*-1.2*-3.6*END*
```

## 1.9 Post Mortem

Should your program fail for some reason which the computer is capable of detecting, the computer will print out a Post Mortem giving you enough information to determine what caused your program to fail. The Post Mortem will not catch all possible errors, but it will locate the ones most often made by users.

### Post Mortem Causes

a. xxx does not contain a number
b. divisor is zero
c. magnitude of exponent of number in accumulator is greater than +99
d. number in accumulator is negative
e. magnitude of number in accumulator is greater than 33554431.9
f. number in accumulator is zero
g. xxx does not contain an instruction
h. input program causes input post mortem if data tape is improper

# SECTION 2

## ADVANCED PROGRAMMING OF PINT

### 2.1 Introduction

PINT as it has been described so far is a rather limited system. Using the commands given, the user cannot deal with numbers which fall outside a rather narrow range. He cannot easily sort through a table of numbers, nor can he solve such problems as a general set of simultaneous linear algebraic equations. He cannot print anything except numbers, so that it is impossible for him to make columns of numbers with alphabetic headings.

PINT is capable of performing many other operations which have not yet been mentioned. It has a much larger range of numbers, provided by a technique known as "floating point". It can sort through tables by making use of devices known as index registers. It can print letters as well as numbers using an alphanumeric printing instruction.

### 2.2 Floating Point Numbers

Engineers have long used a system of notation in which the numbers are written in a form involving a number and a power of ten: $m \times 10^n$, where $m$ is a number and $n$ is an integer. The floating point portion of PINT makes use of this notation, thereby greatly extending the range of numbers which it can handle. In PINT, both $m$ and $n$ must be decimal integers, written without a decimal point. The range of numbers is limited only by the size of $n$. In PINT, $n$ cannot contain more than two digits; hence the exponent is limited to a range of $\pm 99$, which is more than enough for most practical problems.

The rules for writing floating point numbers are as follows:

a. The number $m$ is written with its sign first, followed by not more than 9 digits.
b. The sign may be omitted if it is +.
c. The number $m$ is followed by a stop code (*).
d. The number $n$ is written with its sign first, followed by no more than 2 digits.
e. The sign may be omitted if it is +.
f. The number $n$ is followed by a stop code (*).
g. No decimal point may appear anywhere in either $m$ or $n$.

The rules for writing fixed point numbers, which were first given in section 1, are repeated here with minor modifications:

a. The number is written with its sign first, followed by not more than 8 digits.
b. The sign may be omitted if it is +; if it is omitted, as many as 9 digits may be written.
c. The decimal point must appear in the number in its proper position.
d. The number is followed by one stop code (*).

Examples of input and output formats are as follows:

| Number | 2 | -0.673 | 106.3725823 |
|---|---|---|---|
| Input Floating | 2*+00* | -673*-03* | 106372582*-06* |
| " Fixed | 2.0* | -.6730* | 106.372582* |
| Output Floating | +2.0000000 +00 | -6.7300000 -01 | +1.0637258 +02 |
| " Fixed | 2.000000 | -.6730000 | 106.3726 |

## 2.3 Additional Instructions

### 2.3.1 Additional Arithmetic Instructions

RDVxxx Divide c(xxx) by c(a) and leave in accumulator without changing c(xxx). (Reciprocal DiVide)
NEG000 Make c(a) have a NEGative sign.
CHS000 CHange the Sign of c(a).

### 2.3.2 Additional Storage Instructions

XCHxxx EXCHange c(a) and c(xxx).
CNFxxx Copy Negative From xxx into the accumulator without changing c(xxx).
CZIxxx Copy Zero Into xxx without changing c(a).

### 2.3.3 Additional Function

SQA000 SQuare the number in the Accumulator.

## 2.3.4 Additional Sequence-Changing Instruction

JIPxxx If c(a) is positive, consider this as a JMP instruction. If c(a) is negative, ignore this instruction. That is, Jump If c(a) is Positive to xxx.

## 2.3.5 Additional Input and Output Instructions

INA000 INput one number from tape into the Accumulator. Then take the next instruction in order. Appearance of END* on the tape will cause an error.

PRMxxx PRint from Memory location xxx a floating point number, unrounded, with 8 significant digits, preceded by 2 spaces.

PRA00n PRint from Accumulator a floating point number, unrounded, with one digit before the decimal point and n digits after, preceded by 2 spaces. $n < 8$. If $n = 0$, print 7 digits after the decimal point.

PFA00n Print From Accumulator a fixed point number, rounded, with n significant digits, preceded by 2 spaces. Print enough zeros to locate the decimal point. $n < 8$. If $n = 0$, print 7 significant digits.

TAB000 Perform a TAB. (Tabs are not usually necessary, since PINT automatically spaces before each number. Tab stops must be set by the user in advance.)

## 2.4 Index Registers

There are many times when the programmer has to count something during the operation of his program. Example 2 given in the last section presents one of these cases; the programmer must count the number of sets of data which have been read into the machine and stop the computer after 100 of them have been processed.

There is another type of operation which essentially involves counting and which the programmer often wants to do. Suppose, for example, that we wish to read into the computer a list of 100 numbers, sum the entire list, and print out the result. We might decide to read the list into the machine using the INM instruction, storing the numbers in the 100 locations from 800 through 899. Then we would take each number, add it into a sum, and print this result after all 100 had been done. One possible program for doing this is as follows:

```
CLEAR*LOAD000*
INM800*CCF800*ADD801*ADD802*ADD803* ..ETC.. ADD898*ADD899*PFA000*HLT000*
BEGIN000*
```

But this program has 103 instruction to add 100 numbers! It would be very nice if we could use only one ADD instruction and make it refer successively to each of the 100 numbers.

Index registers simplify both of the operations which have been discussed above. The index register makes it possible to set up a count and then test this count each time the program is executed to determine whether to continue repeating the program or to do something else. The index register also makes it possible to modify automatically the address in an instruction so that it refers to a different location each time it is executed.

### 2.4.1 Description of PINT Index Registers

PINT has seven index registers, numbered 1 through 7. They are all the same. Each has a count, an address, and an increment. The count is the portion of the index register which is used to determine how many times a particular piece of program has been executed. The address is the portion which is automatically added to the address of any instruction the programmer desires. The increment is the amount by which the address portion is increased each time the program is repeated.

### 2.4.2 Index Register Instructions

In the descriptions which follow, the letter j represents any number from 1 through 7, corresponding to any one of the seven index registers.

| | |
|---|---|
| jLDCxxx | LoaD the Count portion of index register j with xxx. |
| jLDAxxx | LoaD the Address portion of index register j with xxx. |
| jLDIxxx | LoaD the Increment portion of index register j with xxx. |
| jCIJxxx | Decrease the Count portion of index register j by one, increase the address by the amount of the Increment, and Jump to location xxx if the count is greater than zero. (Count, Increment, and Jump) |

### 2.4.3 Address Modification

The addresses of most instructions can be modified by a given index register simply by writing the number of the desired index register ahead of the letters of the instruction. Then the value of the address portion of that index register will be added to the address of the instruction before the instruction is executed. The instruction itself remains unchanged in memory; its address is changed only during its execution.

Any instruction which has a true address may be changed in this manner by an index register. Some instructions have special codes in their address portions (such as SIN000) and modification would be meaningless. Others require the number of the index register in front of them because they operate directly on index registers. The following is a tabulation of these three groups of instructions (those in parentheses are discussed in section 3):

| May Be Modified by Index Registers | | | | Cannot Be Modified By Index Registers | | | | Must Have Index Register Number | |
|---|---|---|---|---|---|---|---|---|---|
| ADD | CCF | PWR | INM | POS | SIN | EXP | INA | LDC | (AXA) |
| SUB | CNF | | PRM | NEG | COS | TEN | | LDA | (SXA) |
| MUL | CCI | JMP | | CHS | ATN | LNE | PRA | LDI | |
| DIV | CZI | JIN | (SRA) | | | LOG | PFA | | (CXF) |
| RDV | XCH | JIP | (CAI) | SQR | | | | CFI | (CXI) |
| | | (JOS) | (XIT) | SQA | | ABC | CAR | | |
| | | HLT | | | | | TAB | | |

## 2.4.4 Example Using Index Registers

Let us write the program for the example which was presented in paragraph 2.4, but this time we will use index registers. We will write the program beginning at location 200, input the 100 numbers, compute and print out their sum, and stop.

| | | |
|---|---|---|
| | CLEAR* | code word to clear memory |
| | LOAD200* | code word to load program |
| in location 200 | INM800* | input the 100 numbers starting at 800 |
| 201 | 3LDC099* | load 099 into count of index register 3 |
| 202 | 3LDA000* | load 000 into address of 3 |
| 203 | 3LDI001* | load 001 into increment of 3 |
| 204 | CCF800* | copy first number into accumulator |
| 205 | 3ADD801* | add on the next number |
| 206 | 3CIJ205* | decrease the count of index register 3 by one, increment the address, jump to 205 if the count is greater than 0 |
| 207 | PFA000* | if count = 0, print result |
| 208 | HLT200* | halt (jump to 200 if START button is pressed) |
| | BEGIN200* | code word to begin operation |

The use of the CIJ instruction does not change the number in the accumulator, so we are simply keeping a running sum of the numbers to be added. The computer uses the indexed ADD instruction in location 205 first as ADD801, then ADD802, then ADD803, etc., finally printing and stopping after using it as ADD899. Notice that the computation takes place only 99 times for the 100 numbers, since the CCF instruction is used to copy the first of the numbers into the accumulator.

The program which we wrote without using index registers contained 103 instructions, each of which was performed once during the operation of the program. Our program with index registers contains 9 instructions, which is considerably shorter than the original program, and much easier to prepare on paper tape for the computer. However, two of the instructions in this new program are executed 99 times each (ADD and CIJ). Hence we must execute 2 x 99 + 7 = 205 instructions to perform the job by using index registers. The operation of the new program will take about twice as long as the original one. This is a good example of the trade of time for space. We have reduced the amount of storage which our program took at the expense of increasing the amount of time which it took to operate. The programmer must often consider this trade, especially when both his time and the computer's time can be valued in dollars and cents.

## 2.5 Alphanumeric Printing

The programmer often wants to print alphabetic information on the output from his problem. For example, his results may appear as a number of related columns of figures. For ease in reading there should be column headings to describe what each column means. Sometimes it may be desirable to obtain the output in the form of an equation. In both cases, the required symbols may be printed by PINT.

### 2.5.1 Additional Output Instruction

ABCnnn Print AlphaBetiC characters from the next nnn locations. These characters must be specially coded as shown in the following table.

### 2.5.2 ABC Character Codes

| Operation | Code | Upper Case | Lower Case | Code | Upper Case | Lower Case | Code | Upper Case | Lower Case | Code | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Tape Feed | 00 | ) | 0 | 10 | G | G | 2G | W | W | 3W | |
| Car Ret | 01 | | 1 | 11 | H | H | 2H | X | X | 3X | |
| Tab | 02 | " | 2 | 12 | I | I | 2I | Y | Y | 3Y | |
| Back Space | 03 | # | 3 | 13 | J | J | 2J | Z | Z | 3Z | |
| | | | 4 | 14 | K | K | 2K | $ | , | 3, | |
| Upper Case | 05 | △ | 5 | 15 | L | L | 2L | : | = | 3= | |
| Lower Case | 06 | @ | 6 | 16 | M | M | 2M | ; | [ | 3[ | (36)* |
| Line Feed | 07 | & | 7 | 17 | N | N | 2N | % | ] | 3] | (37)* |
| Stop Code* | 08 | ' | 8 | 18 | O | O | 2O | | | | |
| | | ( | 9 | 19 | P | P | 2P | | | | |
| | | A | A | 1A | Q | Q | 2Q | ? | + | 3+ | |
| | | B | B | 1B | R | R | 2R | _ | - | 3- | |
| | | C | C | 1C | S | S | 2S | . | . | 3. | |
| | | D | D | 1D | T | T | 2T | SPACE | | 3D | |
| | | E | E | 1E | U | U | 2U | ÷ | / | 3/ | |
| | | F | F | 1F | V | V | 2V | | | | |

* if using Flexowriter

### 2.5.3 Use of ABC

The codes given above are written in groups of four and placed in successive locations following the ABC instruction. If the codes do not come out even in groups of four, fill the last group with the code FF so that there are four codes in the last group. An an example, suppose that we wish to print: Carriage return T h e E n d ! carriage return. We must manufacture the ! by using a back space. The program will be as follows:

| | | |
|---|---|---|
| | CLEAR* | code word to clear memory |
| | LOAD000* | code word to load program |
| in location 000 | ABC005* | the next 5 locations are ABC codes |
| 001 | 01052T06* | car ret, upper case, T, lower case |
| 002 | 2H1E3D05* | h, e, space, upper case |
| 003 | 1E062N1D* | E, lower case, n, d |
| 004 | 3.030518* | ., back space, upper case, |
| 005 | 0601FFFF* | lower case, car ret, filler, filler |
| 006 | HLT000* | halt |
| | BEGIN000* | code word to begin operating |

## 2.6 Decimal Memory Printout

The programmer often wants to know what is in a certain portion of the PINT memory. He may, for example, suspect that he has written an address or an instruction incorrectly and would like to verify what was written by printing it out of memory where he stored it. He may want to have a certain set of data printed out to see what operations his program has performed on it, especially when his program has operated incorrectly. The decimal memory printout provides the means for obtaining a print of what is in a section of memory.

The special code word DUMP*, followed by the addresses of the locations to be printed and a stop code, initiates the printout. The special code word DONE* is used to return to the loading program. Complete instructions for operation of the printout are given in section 6.6.

## 2.7 Subroutines

There are often certain programs which are used by many people without change. These programs, called subroutines, are usually written by very good programmers and made available to all who want them. Examples are programs for the solution of simultaneous linear algebraic equations, matrix inversion, and least-squares curve fitting.

Subroutines are available in two forms. Some are in a form which require no additional programming by the user. When someone has a certain job to do, he simply obtains the tape for the program to do that

job, studies the write-up on the program to be sure that it does the right job and that he knows how to use it, and then uses the program. These routines usually require loading the program directly into the computer and then providing a data tape for its operation. Examples of these are programs for mean and standard deviation and for finding the roots of a quartic.

Other subroutines are written so that the user must include instructions in his program in order to make use of them. Routines of this type are those which are used for calculations which are likely to be required several times within one program. An example of a very simple operation which might be performed many times in one program is the evaluation of tan x and cotan x. Although we can easily write the coding for performing these operations, it might be handy to have a packaged routine for doing the job. The instructions for using the subroutine might appear like this:

> The subroutine will compute the value of tan x or cotan x to an accuracy of 0.0001%. The value of x is to be in the accumulator, and the result is left in the accumulator. The subroutine requires 14 locations. The calling sequence is as follows:
>
> | | For tan x | | For cotan x |
> |---|---|---|---|
> | a-1 | CCF (x) | a-1 | CCF (x) |
> | a | JMP $L_o$ | a | JMP $L_o$ + 7 |
> | a+1 | return | a+1 | return |
>
> Note: The instruction in a-1 may be omitted if the value of x is already in the accumulator.

The symbol a is used to represent an arbitrary location in the user's program. Hence a-1 is the location before a, and a+1 is the location following it. (x) means the address of the location containing the value of x. $L_o$ is the first location of the subroutine, which may be placed anywhere which is convenient to the programmer, provided that there are 14 consecutive locations available.

These points are best illustrated by a programming example. Suppose that we are making a computation which requires, at a certain point, that we take the cotangent of a number. Suppose that the instruction in location 143 of our program is an MUL416 and that we wish to form the cotangent of the result of this MUL instruction and then square the cotangent. We decide that we have space for our subroutine in locations starting at 800, so $L_o$ = 800. This portion of our program would look like this:

| | | | |
|---|---|---|---|
| in location | 143 | MUL416* | produce number to form cotan of |
| | 144 | JMP807* | jump to cotangent subroutine |
| | 145 | SQA000* | return point - square the resulting cotan |

Note that this subroutine gives us, in effect, a new instruction, for the use of the JMP807 actually means "form the cotangent of the number in the accumulator". Our program runs along executing instructions and reaches location 143. In 143 is our MUL instruction, which the computer executes and goes to 144. The JMP807 in 144 causes the computer to jump to 807 to compute the cotangent. After computing the cotangent, the subroutine is designed to jump to the location right after the JMP807 instruction, no matter where that JMP807 instruction is. Hence this time it jumps back to location 145, squares the result, and carries out our program instructions from there on. We may compute the cotangent any number of times we wish by writing the JMP807 instruction when we want to compute it. This is exactly what we do when we write SIN000 except that we do not have to load the subroutine for computing the value of the sine, since this routine is stored as part of PINT.

# SECTION 3

## FURTHER PROGRAMMING OF PINT

### 3.1 Introduction

At the end of the last section we pointed out that there are still some operations, such as the construction of subroutines, which you cannot yet perform with the PINT instructions you have learned. There are several more instructions available which will help you do this, and there are also some features of PINT which will make programming large problems somewhat easier. We will present some of the material in this section without giving programming examples, since by this time you have gained enough knowledge to be able to understand the use of various features without examples.

### 3.2 Subroutine Construction

In paragraph 2.7 we discussed the use of a subroutine which computed the value of tan x or cotan x when the number x was left in the accumulator. One feature of this subroutine, and of many others for that matter, is that it was capable of returning to your program (to the location directly after the location in which you wrote the JMP instruction) no matter where you wrote this instruction. This feature is provided by a special instruction which will record in the subroutine the location right after the one in which the JMP instruction was placed.

#### 3.2.1 Subroutine Instruction

SRAxxx Set Return Address. Take the address of the location of the last jump instruction executed, add one to it, and place it in the address portion of the instruction in location xxx without changing the operation code in that location. This will work with all jump instructions except CIJ.

#### 3.2.2 Example of Subroutine Construction

We almost always write subroutines as if they were to start in location 000. Then the programmer can store them anywhere that he wishes by making use of the MODxxx loading code described in paragraph 3.8. Let us write as an example of a subroutine the one which was described as an example in paragraph 2.7. This subroutine is to compute the tangent of the number in the accumulator if the user jumps to location $L_0$ (the first location of the subroutine), and is to compute the cotangent if he jumps to $L_0 + 7$. In either case the result is left in the accumulator. The program is on the following page.

```
in 000    SRA006*    set return address in 006 - entry for tan x
   001    CCI013*    store x temporarily
   002    COS000*    cos x
   003    XCH013*    store cos x and obtain x
   004    SIN000*    sin x
   005    DIV013*    sin x divided by cos x = tan x
   006    JMP000*    leave the subroutine with tan x in accumulator
   007    SRA006*    set return address in 006 - entry for cotan x
   008    CCI013*    store x temporarily
   009    SIN000*    sin x
   010    XCH013*    store sin x and obtain x
   011    COS000*    cos x
   012    JMP005*    leave the subroutine with cotan x in accumulator
   013    (space)    temporary storage
```

Suppose that the user's program contains a JMP807 in location 144. He is jumping to the SRA instruction in location 807. (He has stored the subroutine starting at location 800.) This SRA instruction notices that the JMP was in location 144, adds one to that number, and stores it as the address in location 812. Hence the JMP instruction in our subroutine, which initially had a blank address portion (usually filled with 000), is changed to JMP145. When the subroutine is executed, the result is left in the accumulator and the subroutine jumps to location 145 to resume operation of the user's program.

## 3.3 Manual Intervention

There are times when it is useful to be able to intervene in the operation of the program. This is provided by an instruction which tests the condition of SENSE SWITCH 2. This instruction may be used, for example, to tell the program that you want to change some data for a particular problem.

JOSxxx  If SENSE SWITCH 2 is down, jump to xxx. If it is up, ignore this instruction. (Jump On Sense switch)

## 3.4 Address-Changing Instruction

There is an instruction which permits the programmer to change an address in his list of instructions without using index registers. One example of the use of this occurs in subroutines. Suppose, for example, that the programmer wants to write a subroutine which will first input a number n telling how many simultaneous equations there are and then input the coefficients for the equations and solve them. He must therefore have some method of taking the number n and setting appropriate index registers to count the equations properly.

CAIxxx Round off the number in the accumulator to make it a proper integer. Copy this number as an Address Into the address portion of the instruction in location xxx without changing the operation code in that location.

This instruction might be used as follows. Suppose that the instruction 3LDC000 is stored in location 431, and that the accumulator contains the number 3.6. The execution of the instruction CAI431 will round the number in the accumulator to 4 and place it in location 431 as an address, causing the instruction in that location to read 3LDC004. Note that this instruction is actually changed in memory by the CAI instruction, whereas the use of index registers did not change the instruction as it stood in memory.

## 3.5 Additional Index Register Instructions

The following instructions are provided to make the index registers more versatile.

- jCXFxxx Copy the value of the address portion of indeX register j From the address portion of the instruction in xxx. (xxx must not contain an LDC instruction.)
- jCXIxxx Copy the address portion of indeX register j Into the address portion of the instruction in xxx. (xxx must not contain an LDC instruction.)
- jAXAnnn Add the number nnn to indeX register j's Address portion.
- jSXAnnn Subtract the number nnn from indeX register j's Address portion.

Suppose that location 267 contains the instruction ADD246 and location 437 contains the instruction SUB357. Suppose further that we execute the following sequence of instructions:

4CXF267*4AXA002*4CXI437*4SXA002

The execution of the 4CXF instruction will cause the value 246 to be stored as the address value in index register 4. The 4AXA instruction will cause two to be added to the address portion of index register 4, producing 246 + 2 = 248. Then the 4CXI instruction will take this new value of the address portion of index register 4 and place it in the address portion of the instruction in 437, changing it from SUB357 to SUB248. Finally, the 4SXA will subtract two from the address portion of index register 4, changing it back to 246.

## 3.6 Additional Note on ABC

Although the information given in section 2 makes the alphabetic print instruction ABC appear to be a regular instruction of the PINT repertoire, it is not. It is merely a program loading code which tells the program loading routine to load the next nnn locations with the special alphanumeric code. This loading code takes one space in the memory to avoid confusing the beginner. It is simply skipped over during the actual operation of a PINT program. The special alphanumeric code words are themselves recognized by PINT and operate independently as instructions.

This means two things to the user. First, he does not need any special code in his list of characters to terminate the list (indeed, there is no such code given - FF is only a filler). Second, he may cause only a portion of the list of codes to be printed simply by giving a jump instruction to the location of the first code he wants and the printing will continue from there. For example, in the program in paragraph 2.5.3, if the programmer should give the instruction JMP004 from some other place in his program, the result would be the printing of . back space upper case ' lower case carriage return . The program would then halt.

Corrections can be made to alphanumeric code words stored in the computer only by reloading the entire sequence. Simply writing LOADxxx* and then giving the corrected word will result in an error on input. Attempting to do this by writing LOADxxx*ABCnnn* and then giving the corrected word will cause two words to be stored, the ABC instruction and the corrected word. Hence the only practical way to correct an alphabetic error is to reload the entire sequence.

## 3.7 Coding Sheet Data Entry

Many programs use constants, which up to this point have been input by writing an INM or INA instruction at the beginning of the program. This is not necessary. The constants can be loaded as part of the program simply by writing them as proper numbers in either fixed or floating point format on the program tape. They must be placed in appropriate locations where the program would normally have data, but no special codes are required either to load them or use them. For example, suppose that we want to have a program which will input a number, add one to it, and print it out in fixed point format. Samples of how this can be accomplished follow on the next page.

| Correct | Correct | Incorrect |
|---|---|---|
| CLEAR* | CLEAR* | CLEAR* |
| LOAD000* | LOAD000* | LOAD000* |
| INA000* | 1.0* | INA000* |
| ADD004* | INA000* | ADD003* |
| PFA000* | ADD000* | PFA000* |
| HLT000* | PFA000* | 1.0* |
| 1.0* | HLT001* | HLT000* |
| BEGIN000* | BEGIN001* | BEGIN000* |

The number "one" in the above program is stored in the program right along with the instructions, but we must be careful never to get the constants into locations where they might be used as instructions. The same format rules which applied for INM and INA (see paragraph 2.2) apply for coding sheet data entry.

## 3.8 Programs for General Locations

The subroutine which we wrote in paragraph 3.2.2 was written for locations 000 through 013. Yet the user would like to place this program anywhere in memory. He does this by writing the loading code word MODxxx* after his LOAD code. This causes the value xxx to be added to the address of every instruction which follows until another LOADxxx* or MODxxx* is given. There are two exceptions to this: 1) if the instruction is preceded by an "x"; 2) if the instruction has no meaningful address portion, such as SIN000 (these are the instructions listed in paragraph 2.4.3 as "Cannot Be Modified by Index Registers").

For example, if we write the sequence LOAD100*MOD100*CCF200*xCCI500* x4LDC004* etc. and load it into the machine, the instructions will start in location 100 and will appear as follows: CCF300*CCI500* 4LDC004* etc. Note that this MOD code word modifies the addresses of instructions as they are loaded into the machine. It does not modify any of the loading codes such as LOAD or BEGIN.

## 3.9 Additional Loading Code Words

The code word CLEAR* has already been introduced in section 2, but its function has not yet been completely explained. This code word will cause the PINT system to clear all of the PINT memory when the code is received on input. This takes about two seconds. However, it does not set the locations to zero. Rather it places something in them which in the PINT system is taken to be "undefined". Hence an attempt to use a number from a location cleared by the CLEAR code word will result in a Post Mortem. The purpose of the CLEAR code is to assist the Post Mortem in catching common errors which beginning programmers often make, since

any attempt to use a number not previously stored or any attempt to jump to a location in which the programmer hasn't placed an instruction will produce a Post Mortem. CLEAR* clears all memory, so it must be used only at the beginning of the first program if more than one is to be loaded.

The code word WAIT* is designed to assist the programmer in operating the computer. For example, suppose that he has just given some instructions to the computer from the typewriter and now wishes to give it some from tape. He must push some buttons on the computer to make this change, but he cannot do it by merely pressing the proper button and then pressing the START button, since he will receive a notice of "Illegal Order". Typing the code word WAIT* will cause the computer to wait; he may then press the necessary buttons to change over to the reader and then press START without getting into difficulties with the loading program. This WAIT* code is used at the end of all subroutines, since it is assumed that the programmer wishes to load more routines or give a BEGINxxx* code from the typewriter.

## 3.10 PINT Stop Button

SENSE SWITCH 16 has been programmed to enable the user to stop the operation of his program in the middle if he has a feeling that something has gone wrong or if he wants to interrupt the operation of the program for some reason. This button will always stop the operation of PINT on the next CCF or CNF instructions and will then print a Post Mortem telling what particular instruction the computer was stopped on. Hence if the user simply wants to see what his program is doing, he may press SENSE SWITCH 16 and obtain a Post Mortem. If everything appears correct he may return to the operation of his program by getting to the beginning of the PINT loading routine and typing BEGINxxx* where xxx is the location of the CCF or CNF instruction on which the operation was just stopped.

Note that this makes it possible for a user with a short program to interrupt the user with a very long program in the machine. The user with the long program can stop his program without affecting any of his numbers. Then the user with the short program can place his program in the machine in a clear space in memory designated by the other user and can run his short program. After this short program is finished, the other user can transfer back to his long program and continue operating. Care must be taken that the user with the short program does NOT use CLEAR* and does not change any of the index registers which the other programmer is using. The short program can easily be modified to go into any portion of memory by using the MODxxx* code word, since most programs will be written for locations starting with 000 and the "x" code is rarely required.

### 3.11 Compatible Output

The decimal memory printout routine described in paragraph 2.6 may be used to produce a complete correct copy of the program which is stored in memory. The compatible output feature will print or punch both instructions and data in a form which can later be used as input for further machine operations. Depress SENSE SWITCH 8 to produce compatible output. It will place an x before any instruction which had an x before it when the program was loaded, and will place stop codes after every word.

This system is useful when a program has required a number of corrections after it has been loaded. These corrections may be loaded as patchwork, correcting an instruction here and there. Then the complete corrected program can be punched out when the program is working properly. If the program has been loaded into locations starting at 000, the result is a program which is relocatable using the MOD code word, provided the necessary x's have been included.

Compatible output may also be used to obtain data output from the computer on tape in a form that can be used by the data input routine in a later program. For example, the results of one problem may be used as the input data for the next.

NOTE: No memory address above F99 can be made compatible because locations are output modulo 1600.

### 3.12 Input Duplication

Material which is being loaded into the machine may be copied either on the typewriter or the punch by making use of the input duplication feature of the computer. If the button INPUT DUPLICATION - SELECT is down, the computer will output the information just input on the selected output device. The user must select the punch if he wants to make a tape copy, because the computer will attempt to select the typewriter.

### 3.13 Errors Detected by Load and Input Routines

The PINT loading and input routines contain checks to catch some of the errors which can be made on tapes, so that gross errors by the programmer can be caught more readily. In addition, the loading routine contains a feature which permits the loading of the correct instructions after the program tape has been loaded, so that corrections can be made very easily.

The PINT loading routine operates by examining the instruction presented to it and comparing it with a table of allowed instructions. If the operation code presented is not in this table, the PINT assumes that this must be a piece of data being loaded as part of the program, and proceeds to check this against the table of allowed symbols for data. If it finds that the word presented is not proper data either, it will print "Illegal Order xxx". It will then leave location xxx blank and will continue loading beyond that point. At the end of the program, after receiving a BEGINxxx, it will print "Load Corrected Orders". At this time the programmer must give the LOAD code word for each location to be corrected and the correct instruction taken from his coding sheet. He may then give the correct BEGINxxx code word to operate the program or he may check his program in the machine by having the computer dump (paragraph 2.6). Notice that, should the program tape contain a LOAD code word after the error has been detected but before the BEGIN code word, this system will not work, since the LOAD cancels PINT's recollection of the error.

The PINT loading routine will also catch errors caused by attempting to give improper characters in addresses in the same way that it catches improper orders. The same correcting procedure is provided as above.

The PINT data input routine will catch those errors which result from attempts to use improper characters as numbers. The data tape will stop and the machine will print "Incorrect Data". There is no way to correct this data as there was for instruction errors and the tape must be corrected before the operation can proceed. For example, attempting to write the number 1 as +1.t* or as +100*-.2* will be caught.

Note that the number "one" and the letter "ell" will both be accepted as "one" and that number "0" and the letter "o" will both be accepted as the number "0".

## 3.14 Basic Machine Language

There is perhaps a rare occasion when the PINT user wants to go into the basic machine language to perform some operations which are not part of the floating point system (for example, logical operations). Although doing this is difficult because of the rather weird structure of both PINT memory and PINT instructions, the user may get out of PINT with the following instruction:

XITyyy Transfer the basic machine control to the basic language instruction in yyy, where yyy is a special code for the machine address of the instruction.

yyy is converted to basic language as follows: Divide yyy by 64. Take the integer quotient and add 48; the result is the basic track number. The remainder is the basic sector number. For example, XIT463 will transfer basic machine control to the basic machine language instruction in location 05515.

Reentry into PINT location xxx is accomplished by the basic language instruction $\alpha$*07*tttss*02000* (decimal), where $\alpha$ is the basic machine location of this instruction and tttss is the basic machine location equivalent of the PINT location xxx. xxx is converted to basic machine language as follows: Divide xxx by 23. Take the integer quotient and add it to 48; the result is ttt. Take the remainder and add 1; the result is ss. For example, to transfer into PINT starting at location 463, the basic machine language instruction would read $\alpha$*07*06804*02000*. Improper reentry will usually cause a meaningless Post Mortem.

## 3.15 PINT Summary

| Code | Name | Word Times* | Post Mortem if (see list below) | See Paragraph |
|---|---|---|---|---|
| ADD | ADD | 4 | A | 1.5 |
| SUB | SUBtract | 4 | A | 1.5 |
| MUL | MULtiply | 4 | A | 1.5 |
| DIV | DIVide | 4 | A, B | 1.5 |
| RDV | Reciprocal DiVide | 4 | B | 2.3 |
| POS | make POSitive | 2 | none | 1.5 |
| NEG | make NEGative | 2 | none | 2.3 |
| CHS | CHange Sign | 2 | none | 2.3 |
| CCF | Copy Contents From | 3 | A | 1.5 |
| CNF | Copy Negative From | 3 | A | 2.3 |
| XCH | eXCHange | 4 | none | 2.3 |
| CCI | Copy Contents Into | 3 | C | 1.5 |
| CZI | Copy Zero Into | 2 | none | 2.3 |
| CAI | Copy Address Into | 7 | D, I | 3.4 |
| SQR | SQuare Root | 9 | D | 1.5 |
| SQA | SQuare Accumulator | 3 | none | 2.3 |
| SIN | SINe | 14 | E | 1.5 |
| COS | COSine | 17 | E | 1.5 |
| ATN | ArcTaNgent | 17 | none | 1.5 |
| PWR | PoWeR | 28 | A, D, F | 1.5 |
| EXP | EXPonential | 11 | none | 1.5 |
| TEN | TEN exponential | 13 | none | 1.5 |
| LNE | Natural Logarithm | 15 | D, F | 1.5 |
| LOG | base 10 LOGarithm | 15 | D, F | 1.5 |

| Code | Name | Word Times* | Post Mortem if (see list below) | See Paragraph |
|---|---|---|---|---|
| JMP | JuMP | 2 | G | 1.5 |
| JIN | Jump If Negative | 2 | G | 1.5 |
| JIP | Jump If Positive | 2 | G | 2.3 |
| JOS | Jump On Sense | 2 | G | 3.3 |
| SRA | Set Return Address | 3 | none | 3.2 |
| HLT | HaLT and jump | 2 | G | 1.5 |
| XIT | eXIT | 1 | none | 3.14 |
| INM | INput to Memory | - | H | 1.5 |
| INA | INput to Accumulator | - | H | 2.3 |
| PRM | PRint from Memory | - | A, C | 2.3 |
| PRA | PRint from Accumulator | - | C | 2.3 |
| PFA | Print Fixed from Accumulator | - | C | 1.5, 2.3 |
| ABC | AlphaBetiC print | - | none | 2.5, 3.6 |
| CAR | CARriage return | - | none | 1.5 |
| TAB | TABulate | - | none | 2.3 |
| LDC | LoaD Count | 3 | none | 2.4 |
| LDA | LoaD Address | 2 | none | 2.4 |
| LDI | LoaD Increment | 2 | none | 2.4 |
| CIJ | Count, Increment, and Jump | 3 | G | 2.4 |
| AXA | Add to indeX Address | 3 | none | 3.5 |
| SXA | Subtract from indeX Address | 3 | none | 3.5 |
| CXF | Copy indeX address From | 3 | J | 3.5 |
| CXI | Copy indeX address Into | 3 | none | 3.5 |

* One word time in PINT is about 17 milliseconds (1/60 of a second). For some instructions, the word times given are approximate averages, since the times vary somewhat. An indexed instruction requires one additional word time.

Post Mortem Causes

A. xxx does not contain a number
B. divisor is zero
C. magnitude of exponent of number in accumulator is greater than +99
D. number in accumulator is negative
E. magnitude of number in accumulator is greater than 33554431.9
F. number in accumulator is zero
G. xxx does not contain an instruction
H. input program causes input post mortem if data tape is improper
I. number in accumulator is greater than 2047
J. xxx does not contain an address

## Program Loading Code Words (see paragraph given in parentheses)

| | |
|---|---|
| CLEAR | Clear all of PINT memory. This code word does not set the locations to zero, however, but to "undefined". (1.6, 3.9) |
| LOADxxx | Load the program in consecutive locations starting at xxx. (1.6) |
| MODxxx | Add the value xxx to all modifiable instructions not preceded by the letter x. (3.8) |
| WAIT | Remain in the loading program and wait for another code word. (3.9) |
| BEGINxxx | Stop, then go to location xxx for the first instruction. (1.6) |

## Rules for Numbers (see paragraph 2.2)

Floating point

a. The number is written with its sign first, followed by not more than 9 digits.
b. The sign may be omitted if it is +.
c. The number is followed by a stop code (*).
d. The exponent is written with its sign first, followed by not more than 2 digits.
e. The sign may be omitted if it is +.
f. The exponent is followed by a stop code (*).
g. No decimal point may appear anywhere.

Fixed point

a. The number is written with its sign first, followed by not more than 8 digits.
b. The sign may be omitted if it is +; if it is omitted, as many as 9 digits may be written.
c. The decimal point must appear in the number in its proper position.
d. The number is followed by one stop code (*).

## Memory

PINT has 1000 memory locations, numbered 000 through 999. There is no restriction on the use of any particular location. Each location holds one number or one instruction. Locations containing neither a number nor an instruction are considered undefined; attempts to use their contents either as instructions or numbers result in post mortems.

There are additional memory locations available, bringing the total to 1666, and the user is referred to the "EXTENDED MEMORY APPENDIX" (5.0) for the addresses of these locations.

## SECTION 4

## TECHNICAL CHARACTERISTICS OF PINT

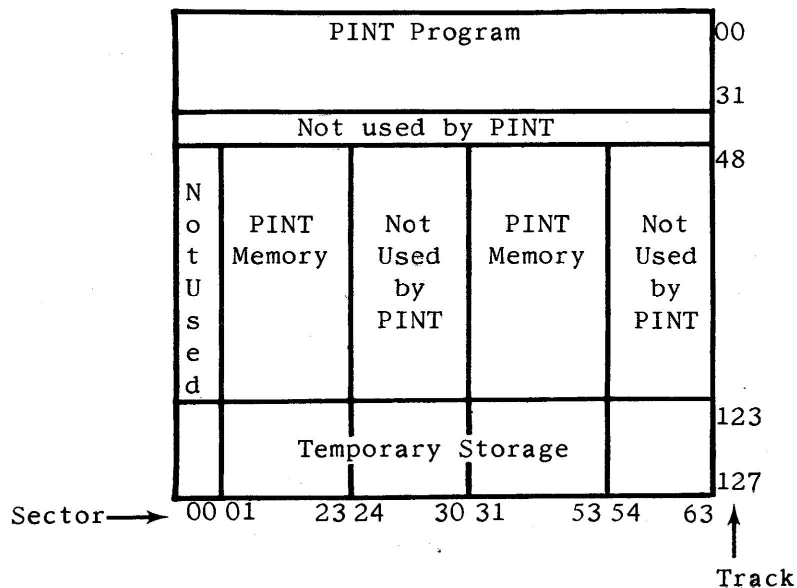
The following descriptions of the PINT program are not intended to present more than a superficial view of the general characteristics of the system from the programming standpoint. A detailed knowledge of the design of the system can be obtained successfully only by careful study of the coding of the routines which comprise PINT. To attempt to write down everything would be madness.

### 4.1 Memory

The PINT program occupies locations 00000-03163 on the drum and uses tracks 123-127 for temporary storage. It is written in "protected" form to prevent its accidental destruction by the user. The remainder of the drum is unprotected and is used for PINT memory consisting of 1666 PINT words. (See "EXTENDED MEMORY APPENDIX")

The PINT word occupies two sectors in one track and is arranged so that the second half of the word lags the first half by thirty sectors. These two halves are stored in sectors 01-23 and 31-53 respectively; thus, sectors 24-30 and 54-00 are not disturbed by PINT and may be used for permanent basic language programs.

The structure of PINT memory is illustrated below:



## 4.2 PINT Instructions

There are four basic types of PINT instructions:

Type A - Those which specify an address which may be modified by indexing (ADD, CCF, JMP, etc.)

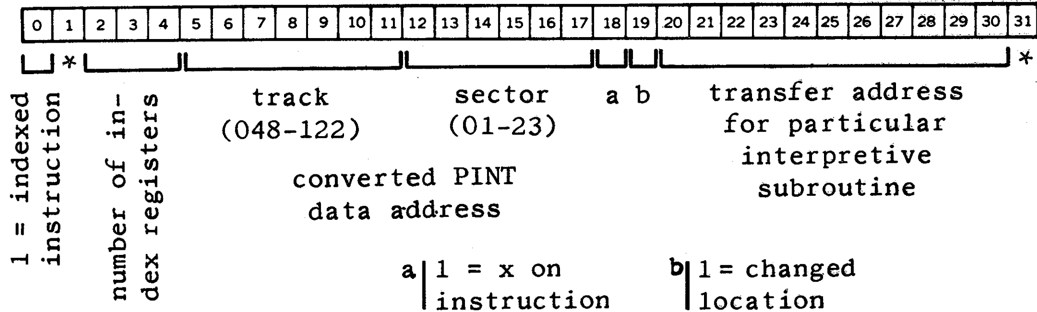
Type B - Those which specify an address and must be preceded by an index register number (jLDA, jLDI, jCIJ, jCXI, etc.)

Type C - Those which specify a count of some sort (PRA00n, PFA00n, jLDCnnn, XITnnn, etc.)

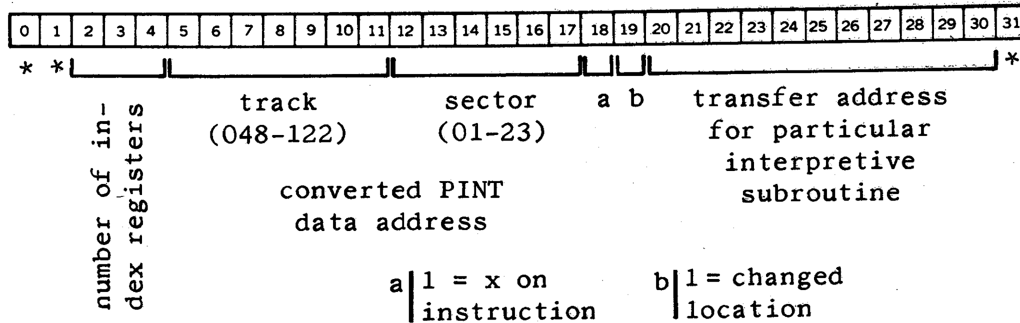
Type D - Those which have a meaningless address portion (SIN, POS, CAR, etc.)

The internal format of each type of instruction is given below:

Type A

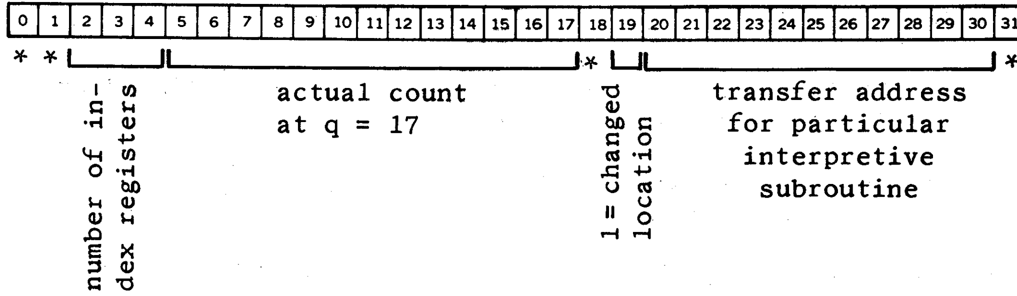



Type B

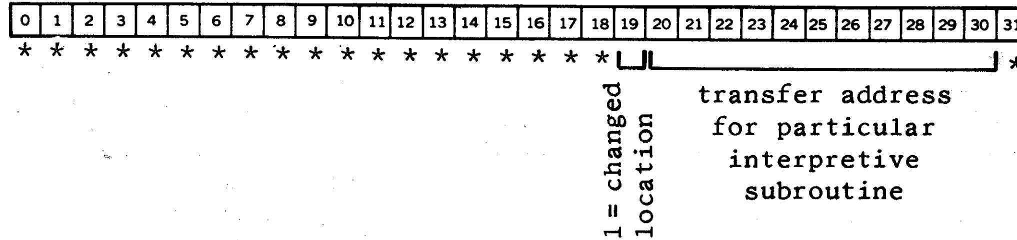



* indicates not used

Type C

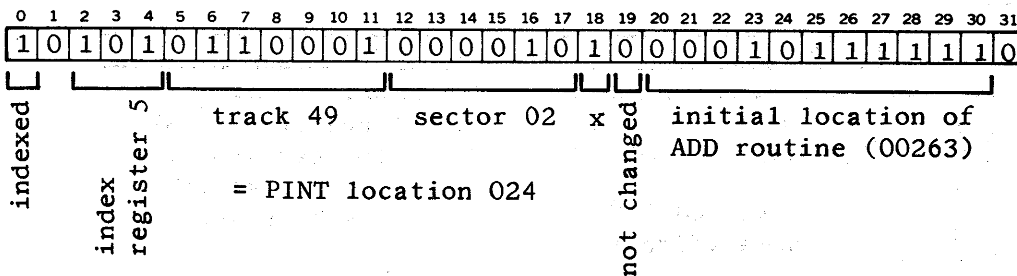



Type D




* indicates not used

The PINT instruction x5ADD024 would appear as follows:



## 4.3 PINT Numbers

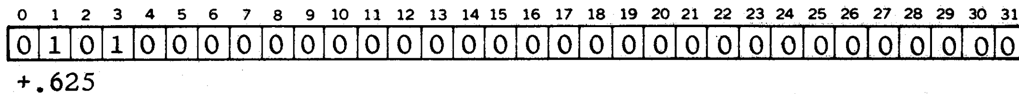
All PINT numbers are floating point and are stored in two parts as follows:

a. the characteristic, consisting of a sign and 31 magnitude bits, normalized, with the binary point at the high-order end.

b. the exponent, consisting of a sign and 17 magnitude bits, stored at q = 17, and complemented for ease of handling. Bits 20-30 contain the address of the post-mortem routine. Thus, if a number is used as an instruction, a post-mortem will result.

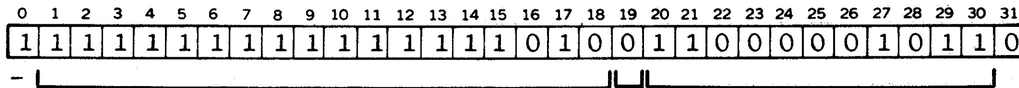
The PINT floating point representation of the number 5 is shown below:

Characteristic




+.625

Exponent




−

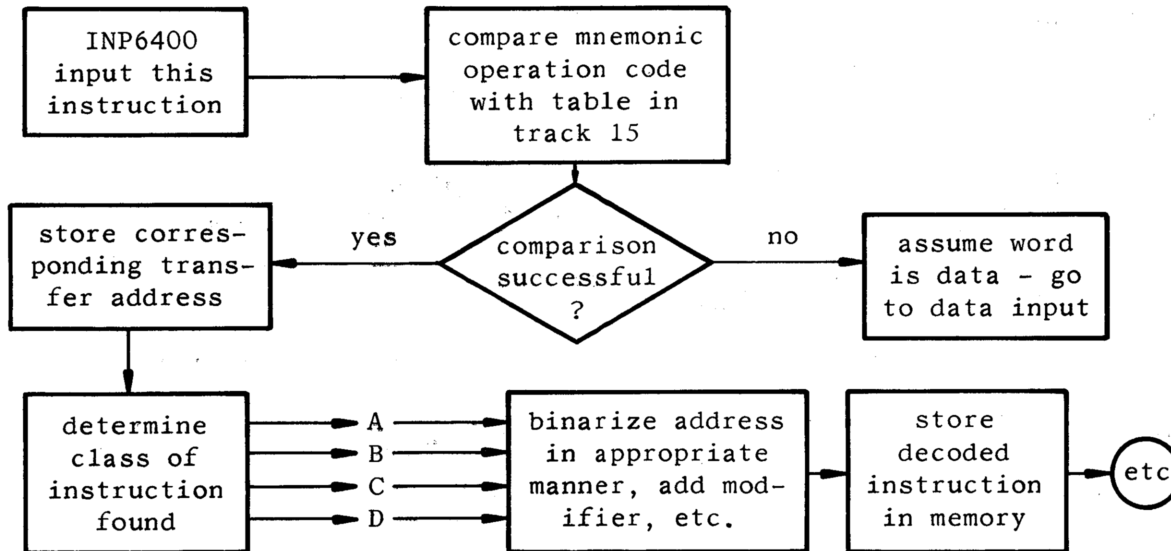
complement of exponent 3.

not changed tag

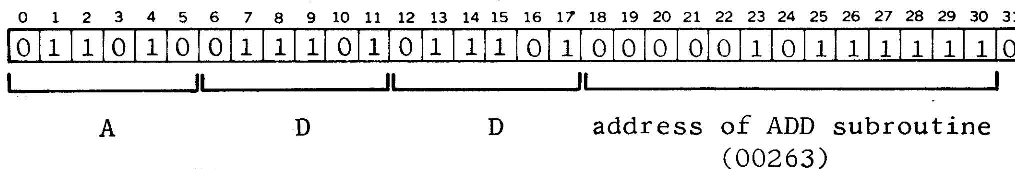
address of post-mortem routine (02411)

## 4.4 Interpretation

In the PINT system, interpretation is done during the program loading phase instead of during program operation. A simplified block diagram of the program loading routine will explain how this is done.




A typical mnemonic operation code table entry for ADD is as follows:




## 4.5 Tagging

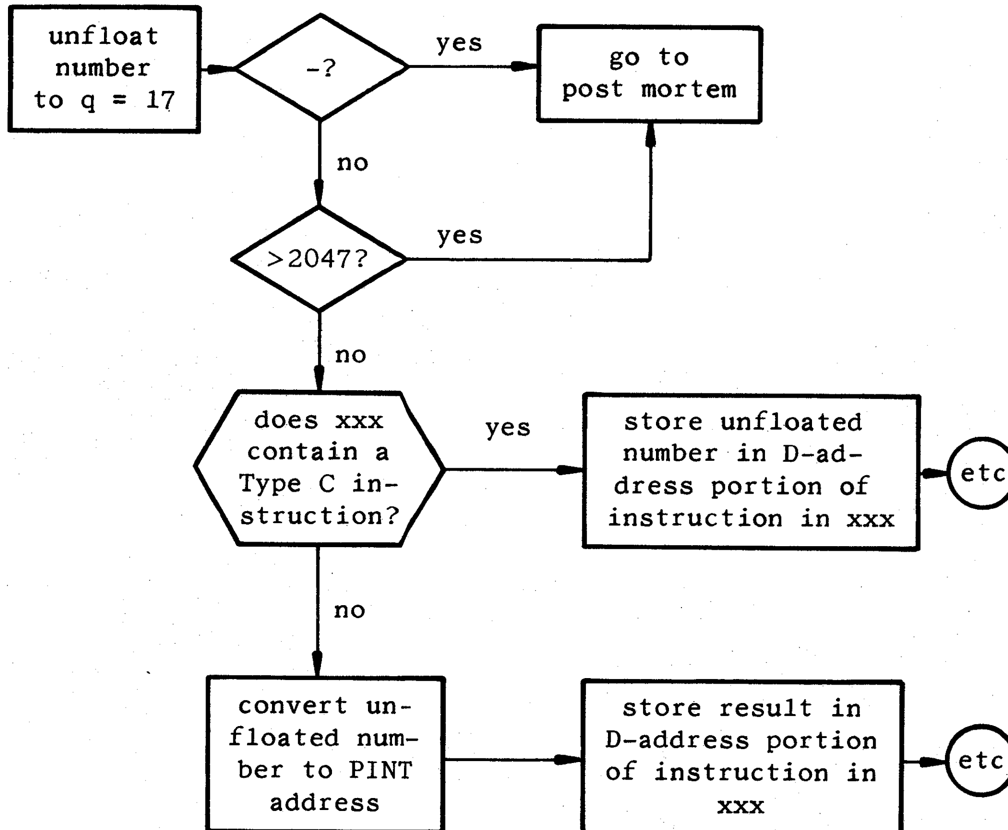
All PINT words contain a tag to indicate to the post-mortem routine whether or not the word was changed by program operation. This tag consists of a bit at q = 19 in the first half word (sectors 01-23). When post-morteming, PINT will detag and print the contents of all locations which have been changed and hence tagged with a 1 at 19. Using the repeat mode, the operation takes less than two seconds to check the entire memory, exclusive of printing time.

A rather infrequently used tag is found in instructions in the first half word at q = 18. If this bit is a 1, the instruction was preceded by an "x" during input. The Dump routine picks up this tag and prints the "x" in front of the instruction, making it possible to dump a program in relocatable form.

4.6 Comments

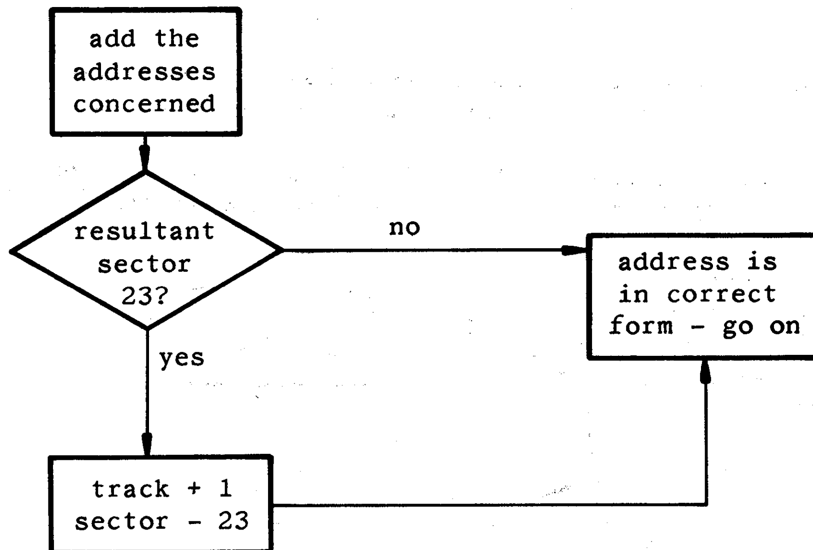
A few unusual aspects of the system should be mentioned here for they result from the unorthodox structure of the PINT memory.

The CAI operation (Copy Address Into) is the slowest non-function instruction and is illustrated in the following block diagram:




As a result of the structure of PINT memory, some instructions are limited. For example, jCXI may not be used to modify count-type (Type C) instruction. This could have been allowed only by approximately doubling the operation time.

The memory structure also causes the CIJ, AXA, and SXA orders (and the PINT executive routine itself) to count by 23's when stepping along through memory. This is usually done in the following manner, which requires only 1/4 drum revolution.




## 4.7 Speed

PINT was not designed to be a fast system; nevertheless, considerable effort was made to obtain this. It was found that the present memory structure provided a compromise between wasting drum space and reducing the access time during program execution. The programming difficulties introduced by not having a sequential memory were more than compensated for by the greater operating speed.

SECTION 5

## EXTENDED MEMORY APPENDIX

As has been previously pointed out, it is possible to extend the PINT memory to 166 locations. This is accomplished by the use of the alphabetic characters A through F in such a manner that a numerical digit never appears to the left of an alphabetic digit.

The following table gives the complete list of memory locations available:

| Locations | PINT Addresses |
|---|---|
| 0 -- 999 | 000 -- 999 |
| 1000 -- 1099 | A00 -- A99 |
| 1100 -- 1199 | B00 -- B99 |
| 1200 -- 1299 | C00 -- C99 |
| 1300 -- 1399 | D00 -- D99 |
| 1400 -- 1499 | E00 -- E99 |
| 1500 -- 1599 | F00 -- F99 |
| 1600 -- 1609 | FA0 -- FA9 |
| 1610 -- 1619 | FB0 -- FB9 |
| 1620 -- 1629 | FC0 -- FC9 |
| 1630 -- 1639 | FD0 -- FD9 |
| 1640 -- 1649 | FE0 -- FE9 |
| 1650 -- 1659 | FF0 -- FF9 |
| 1660 -- 1665 | FFA -- FFF |

TOTAL NUMBER OF LOCATIONS -- 1666

SECTION 6

## OPERATING PROCEDURES

### 6.1 To Turn the RPC-4000 ON and OFF

To turn the computer ON

1. Press POWER ON on computer
2. Press SYSTEM POWER on 4500
3. Press POWER on 4500
4. Hit MASTER RESET
5. Make sure nothing is selected off-line

To turn the RPC-4000 OFF

1. Raise POWER on 4500
2. Raise SYSTEM POWER on 4500
3. Press POWER OFF on computer

* If someone is going to use the computer within an hour DO NOT press POWER OFF on computer.

### 6.2 To Load the PINT Interpretive System

1. Load PINT tape into reader
2. Depress ONE OPERATION
3. Press SET INPUT MODE
4. Depress EXECUTE LOWER ACCUMULATOR
5. Select READER on line
6. Press START COMPUTE

About three inches of tape will be read.

7. Press START COMPUTE
8. Depress SET INPUT MODE
9. Raise ONE OPERATION
10. Press START COMPUTE

The entire tape will now be read.

11. Lock out track 00 - 31 by flipping the two switches under the scope, down.

The PINT CHECK SUM is used as follows:

1. Load tape into reader
2. Follow steps 2 through 10 above
3. After loading, the program will sum the entire PINT system and print O.K. If something else is printed, reload PINT.

## 6.3 To Load a PINT Program

1. Place program tape in reader
2. Depress ONE OPERATION
3. Press START COMPUTE
4. Raise ONE OPERATION
5. Press START COMPUTE

If program tape does not read in, execute the following steps:

1. Select typewrite input (on-line)
2. Depress ONE OPERATION
3. Press SET INPUT MODE
4. Depress EXECUTE LOWER ACCUMULATOR
5. Press START COMPUTE
6. Type 00000000* on typewriter (eight zero's)
7. Raise ONE OPERATION
8. Select reader input
9. Depress START COMPUTE to read program tape

## 6.4 To Correct a Program

If the typewriter prints "Illegal Order xxx" while you are loading your program, it will print "Load Corrected Orders" at the end, and then stop. Independent corrected instructions may now be loaded by the LOADxxx operation.

Note: If the program has more than one LOAD code PINT will not print "Load Corrected Orders".

In order to load corrections under these conditions:

1. Depress SENSE SWITCH 4
2. Depress ONE OPERATION
3. Press START COMPUTE
4. Raise ONE OPERATION
5. Raise SENSE SWITCH 4

The corrected instructions may now be entered as above.

## 6.5 To Trace a Program

The trace within the PINT system may be activated at any time by depressing SENSE SWITCH 1 which causes the following printout:

1. Location of each active jump instruction
2. Jump instruction executed
3. Contents of the Accumulator

Lift SENSE SWITCH 1 to inhibit the trace.

## 6.6 To Print Out a Program from Memory - DUMP

1. Depress SENSE SWITCH 4
2. Depress ONE OPERATION
3. Press START COMPUTE
4. Raise ONE OPERATION
5. Press START COMPUTE
6. Type DUMP*
7. To print one location type xxx*
   To print a series of locations type xxx xxx*
8. To inhibit printing depress, then raise, SENSE SWITCH 2
9. To print again, repeat step 7
10. To return to PINT loading program type DONE*

## 6.7 Post Mortem

The PINT system prints a Post Mortem if your program makes any of a number of errors:

"UND" means undefined, that is, it is neither a number nor an instruction.

### Error Printouts

ØØØ is an operation.

1. "ILLEGAL ORDER IN XXXØØØYYY in YYY..." This means that the contents of YYY are not consistent with the operation of the instruction. The contents of YYY are printed out for reference.

2. "IN XXXØØØYYY" ØØØ is a CCF or a CNF instruction and SENSE SWITCH 16 is down.

Following the identification, PINT produces a printout of all changed index registers and memory locations.

The Post Mortem may be terminated at any time by depressing SENSE SWITCH 32, and then raising it again.

Index Register contents are printed as follows:

A = Address, I = Increment, C = Count

## 6.8 To Load a Subroutine

1. If instructions for the subroutine say that it is self-loading, load as a program.
2. If it is not self-loading, proceed as follows:

   a. Depress SENSE SWITCH 4
   b. Go to PINT loading routine (6.3)
   c. Type LOADxxx*
      MODxxx*
      WAIT*
      where xxx is the initial location for the subroutine to be stored in.
   d. Raise SENSE SWITCH 4
   e. Place subroutine in reader
   f. Press START COMPUTE

## 6.9 Sense Switch Options

| | | |
|---|---|---|
| SENSE SWITCH 1 | UP: | No effect |
| | DOWN: | Trace program mode |
| SENSE SWITCH 2 | UP: | No effect |
| | DOWN: | a. Stop "DUMP" |
| | | b. Jump on JOS instruction |
| SENSE SWITCH 4 | UP: | Tape input mode |
| | DOWN: | Typewriter input mode |
| SENSE SWITCH 8 | UP: | Normal output mode |
| | DOWN: | Compatible output mode |
| SENSE SWITCH 16 | UP: | No effect |
| | DOWN: | PINT stop |
| SENSE SWITCH 32 | UP: | No effect |
| | DOWN: | Stop Post Mortem |

NOTE: EXECUTE LOWER ACCUMULATOR must be down at all times. Therefore, do not use ONE OPERATION button to interrupt a program; use SENSE SWITCH 16.

LIBRASCOPE GROUP

GENERAL PRECISION

COMMERCIAL COMPUTER DIVISION



LITHO IN U.S.A