EQUATIONS FOR TIMING CALCULATIONS FOR
TINY BASIC AND PET BASIC PROGRAMS

A Thesis Presented to
The Faculty of the College of Engineering and Technology
Ohio University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by

Krishnamraju V. Alluri
June 12, 1981

THER
M
1981
ALL

OCT 16 1984

OHIO UNIVERSITY
LIBRARY
I owe special thanks to Dr. Harold F. Klock for his guidance and assistance in the preparation of this thesis.
To My Father and Mother
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION.</td>
<td>1</td>
</tr>
<tr>
<td>2. TIMING SIMPLE PROGRAMS.</td>
<td>6</td>
</tr>
<tr>
<td>3. EMPIRICAL FORMULAS FOR TIME CALCULATIONS.</td>
<td>18</td>
</tr>
<tr>
<td>Timing for PET BASIC statements.</td>
<td>18</td>
</tr>
<tr>
<td>Formula table for PET BASIC.</td>
<td>37</td>
</tr>
<tr>
<td>Timing for Tiny BASIC statements.</td>
<td>38</td>
</tr>
<tr>
<td>Formula table for Tiny BASIC</td>
<td>50</td>
</tr>
<tr>
<td>4. VERIFICATION OF EMPIRICAL FORMULAS.</td>
<td>52</td>
</tr>
<tr>
<td>Verification for BASIC programs.</td>
<td>56</td>
</tr>
<tr>
<td>Verification for Tiny BASIC programs.</td>
<td>65</td>
</tr>
<tr>
<td>5. HARDWARE AND SOFTWARE FOR THE ELEVATOR.</td>
<td>72</td>
</tr>
<tr>
<td>6. SUMMARY AND CONCLUSIONS</td>
<td>98</td>
</tr>
<tr>
<td>BIBLIOGRAPHY.</td>
<td>100</td>
</tr>
<tr>
<td>APPENDIX DETAILS OF THE TIMER.</td>
<td>101</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>DESCRIPTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Keyboard Interface with the Heathkit</td>
<td>8</td>
</tr>
<tr>
<td>2.</td>
<td>Keyboard Interface with User Port</td>
<td>13</td>
</tr>
<tr>
<td>3.</td>
<td>Schematic View of the Elevator</td>
<td>54</td>
</tr>
<tr>
<td>4.</td>
<td>Block Diagram for the Elevator</td>
<td>57</td>
</tr>
<tr>
<td>5.</td>
<td>Heathkit Interface with the Elevator</td>
<td>80</td>
</tr>
<tr>
<td>6.</td>
<td>IEEE-488 Interface with the Elevator</td>
<td>88</td>
</tr>
<tr>
<td>7.</td>
<td>User Port Interface with the Elevator</td>
<td>93</td>
</tr>
<tr>
<td>8.</td>
<td>Block Diagram for the Timer</td>
<td>105</td>
</tr>
<tr>
<td>9.</td>
<td>Block Diagram for the Timer (cont.)</td>
<td>106</td>
</tr>
</tbody>
</table>
CHAPTER 1.

Introduction

Systems based on microcomputers are dominating today's computer market. All types of systems beginning from small calculators to industrial control systems are using microcomputers. Every year new types of microcomputers appear on the market. This makes it very difficult for microcomputer based system engineers to keep track of new instruction sets of these microcomputers. In most cases, there is no similarity between two different generation microcomputers, made even by the same manufacturer. For example Motorola 68000 is completely different from Motorola 6800 and so is Z8000 completely different from Z80. One way to overcome this problem, is to use only one type of microprocessor. But with so many new ones coming on the market, and problems becoming more and more complex, it is impossible to ignore the additional features the new one's offer.

High level languages are independent of the machine code. Some of the high level languages available for the microcomputer systems are BASIC, PASCAL, FORTRAN, COBOL, AP/L and ADA. Modern day computers allow the user to run these high level languages by a software interpreter. Many software packages (programs for different applications) are available in the high level languages.
The basic role of an interpreter (1) is to accept input segments of code that represent programmer supplied instructions, analyze that code for content and intent, and cause machine processes to fulfill that intent. The significance of the high-level languages is twofold. First, they are artificial; that is, they are independent of any machine code in their syntactical structure. Second, the advantage to the programmer is in being able to concentrate on solving problems without specific regard for the internal operations of the computer.

The fact that interpretive software systems are so commonplace in the world of computers implies that there are decided advantages to their use. Since interpreters are not used by every system we can infer that there are some disadvantages. The two main disadvantages of the interpreter system are time and memory. The cost of memory is going down day by day. Hence, time is the only disadvantage. The time taken by the interpreter language is greater than the assembly language because the interpreter is a software routine that, as time progresses, translates a stored program expressed in pseudo-code into machine code and executes the intended operations. Hence, execution takes more time than the assembly language program.

Computers based on interpreters are becoming very popular. Recently National Semiconductor (2) has come out with a single chip microcomputer which operates with an interpretive language (Tiny BASIC) that resides in 2.5k
bytes of ROM contained within the microcomputer chip. It has an additional 64 bytes of RAM for a scratch pad.

Since time is the main drawback of the interpreter systems, we shall, in this thesis, analyze the execution times of two high level languages. They are BASIC which runs on the Commodore PET and Tiny BASIC which runs on the Heathkit ET-3400. We shall find the time it takes to execute simple programs like scanning of keyboard, a decision making type of program and a control program that controls a five floor elevator model. The elevator has up, down, and car switches. The movement of the elevator is indicated by LED's. We get the magnitude of the time involved to run simple programs and then find the time for the individual statements. We now develop empirical formulas for time calculations. We then shall verify the accuracy of these formulas. We shall do this on both the PET computer and the Heathkit ET-3400.

We are mainly concerned with control type of problems. Control programs generally have a lot of I/O operations. We shall see how the I/O operations are implemented in both the systems. In Tiny BASIC I/O operations are executed by the USR function. A USR function transfers the control of the main program to a machine language subroutine. This subroutine can easily perform the I/O operations. This subroutine is a part of the monitor ROM. The I/O for PET BASIC is handled directly by the BASIC language through two ports. They are the IEEE-488 port and the User port.
We shall run the elevator control program off the 3 ports. They are: (1) The address bus and the data bus of the Heathkit, (2) the PET IEEE-488 port, and (3) the PET User port.

To time the operations of Tiny BASIC we have constructed a hardware timer. The details of the hardware timer are given in the appendix. PET has an internal clock, which can be used in all PET time calculations. The PET system uses a Mostek 6502 CPU and the Heathkit uses Motorola 6800. The clock frequency of both the CPU's is 1 MHz.

In the second chapter, we find the time taken by both Tiny BASIC and PET BASIC, to execute simple programs like scanning of a keyboard and a decision making program. This will give us an idea of the magnitude of the time involved in running simple programs off the interpreter systems. The experiments are done on both the systems. The experiments also point out the limitations of the interpretive languages with respect to time. The experiment compares the time taken by the interpretive language to the assembly language.

In the third chapter we shall time the actual statements of Tiny BASIC and PET BASIC. Empirical formulas for the time calculations, for both languages, are derived from the experimental results. The final results are put in two tables.

In the fourth chapter we verify these formulas for their accuracy. We take the program segments from the elevator control program and time them. The observed time
is compared with the calculated time. The worst error for the PET is 5\%, and for the Heathkit, is 2\%.

Chapter five details the hardware and software aspects of the elevator. It also covers the details of the I/O operations for the elevator control off, the Heathkit, the IEEE-488 port and the User port. It describes the interfacing circuitry between the control system (PET or Heathkit) and the elevator. It also covers the details of the IEEE-488 port and the PET User port. All the programs to control the elevator are actually tested on the hardware elevator model and all run correctly.

In chapter 6 we make a brief summary of what we have done and the conclusions we have derived from the experimental results.

An appendix gives the details of the hardware timer.
CHAPTER 2.

Time Taken by Simple Programs

Before we actually time the individual statements of the high level languages we shall first find the time it takes to run simple programs in Tiny BASIC and PET BASIC. This will give us an idea of the magnitude of the time involved in the execution of simple programs. These times will be compared with the time required by the assembly language for the same programs.

The two programs that we shall be timing are: (1) scanning of keyboard, and (2) a decision-making type of program. By the end of this chapter we shall have an approximate idea of the time involved and also the limitations of the interpreter in terms of time.

In order to measure the timing of Tiny BASIC on Heathkit, we need an external hardware timer. The details of this hardware timer are given in the appendix. It requires 3 control signals. These control signals are generated by decoding the address bus. The timer can count up to 999 milliseconds. This time is more than sufficient for most of the programs.

The 3 control lines are to control the hardware timer are:

10 LET A=USR(7188,53248):clear the counter
20 LET A=USR(7188,53249):start the counter
30 LET A=USR(7188,53250):stop the counter

The comments to the right of the colons are only put there to make their meaning clear. They do not appear in the actual program. The USR function transfers the control to a machine language subroutine. The machine language subroutine here is a read subroutine. The body of the program is put between the second and the third statements.

The unencoded keyboard consists of 20 keys arranged in 4 rows and 5 columns. The circuit diagram for the keyboard is given in figure 1.

Tiny BASIC PROGRAMS.

The following program gives us the time taken for one keyboard scan when no key is pressed. The time is measured on the timer.

10 LET A=USR(7188,53248) ;Clear the counter.
20 LET A=USR(7188,53249) ;Start the counter.
30 LET I=0 ;Initialize the Row.
40 LET C=USR(7188,53252+I) ;Get column reading.
50 IF C<>31 THEN GOTO 100 ;Key pressed?
60 LET I=I+1 ;Next Row.
70 IF I <> 4 THEN GOTO 40 ;Last Row?
80 LET A=USR(7188,53249) ;Stop the counter.
100 END

Statement 40 is a USR function which transfers the control to a monitor read subroutine. The read subroutine is a machine language routine which reads the values from the location addressed by the USR function and places this value in the variable specified by the USR function.
Figure 1. Keyboard interface with the Heathkit.

The diagram shows a circuit with a NAND gate 7420, a 4 to 16 decoder 74154, and a 4 input AND gate 7421. The connections include signals such as A15, A14, A12, and A0, and the output signals D0 to D4 which represent the data bus of the Heathkit.
The value read is the column reading. The address bus is decoded to ground one row at a time. Here $I$ is a variable which contains the row number (0, 1, 2 or 3).

The actual time for one scan when no key is pressed is 289.6 milli seconds. This is quite a large time. At this rate we can get a maximum of 3 scans per second. If the switch is pushed for a very short time it is very likely that the scanning program will miss it. Since this is one of the shorter programs, we can safely conclude that scanning in Tiny BASIC is a slow process.

Decision making in Tiny BASIC.

Once a key is pressed and recognised some operation must be done, depending on the individual key that is pressed. The best way to implement this is to go to a different routine for different keys pressed. A big advantage in Tiny BASIC is that the line numbers can be used as variables. This makes the program jump directly to the key routine. The variable $I$ can have values from 0 to 3, and variable $C$ can have any one of the following values: 30, 29, 27, 23, 15. The following program makes use of only one jump instruction.

```
10 LET A=USR(7188,53248) ;Clear the counter
20 LET A=USR(7188,53249) ;Start the counter
30 GOTO I*100+(C-10)*5+500 ;Jump to the key routine
800 LET A=USR(7188,53250) ;Stop the counter
```

When $C=30$ and $I=2$, the time taken by the program to jump to the correct routine is 38.66 milliseconds. This
time is constant for all the keys. The time for decision making is quite small. The maximum length of the routines is not the same. The statement numbers for the starting of the key routines are 520, 560, 580, 590, 600, 620, 660, 680, 690, 700, 720, 760, 780, 790, 800, 820, 860, 880, 890, 900.

Since Tiny BASIC scan takes a long time, we can do a machine language scan and compare how fast the machine language program is for the key board. The following program scans the same key board in machine language.

```
E7 D000  STAA $DO00 ; Clear counter
E7 D001  STAA $DO01 ; Start counter
CE D004  LDX #$D004
5F  CLR B ; Clear row count
A6 00    AGAIN LDAA X ; Get the key
84 1F    ANDA #$1F
81 1F    CMPA #$1F
26 09    BNE KEYIN
08       INX
5C       INC B ; Get next row
C1 04    CMPB #4 ; Reached last row?
2D F2    BLT AGAIN
B7 D002  STAA $D002 ; Stop the counter
20 FE    HERE BRA HERE
```

The scan time when no key is pressed is 143 microseconds. This is a very short time. In applications where the time of a key closure is quite small or if the number of keys is more than 20, Tiny BASIC is not very fast. The machine language has a definite edge over Tiny BASIC. The time for a decision-making for the machine language is 32 microseconds. We can always use both the machine language and Tiny BASIC in the same program. We can jump into the machine language subroutine directly from the main Tiny BASIC program by the USR function. To return values to the
main program all we have to do is put the data in the preassigned locations (assigned by Tiny BASIC) of the variables.

Timing for PET BASIC programs.

In the PET system we have an internal timer which can be used in time calculations. This timer is updated every 1/60 th of a second. Since this timer is not accurate, we take the time to run the program in a loop, for 500 times and then find the average value. The details of this method is given in the next chapter where we do the actual timing of the program statements.

On PET BASIC we shall try similar problems that we have done in Tiny BASIC. The first program is to find the time for a keyboard scan when no key is pressed. In the Heathkit computer we had many address lines which were decoded and used for different controls. The PET user port has only 8 data lines and a control line CB2. We need 5 input lines to get the column readings and 4 output lines to ground one row at a time. The control line is complicated and takes a lot of time to decode it. The best way would be to program 5 lines for input and two lines for output. The output lines are connected to a 2 to 4 decoder. The decoder outputs are active low. Three lines of the user port are programmed for output and and five lines are programmed for input. The following statement does just that.

POKE 59459,224
The user port is addressed by the decimal address 59471. The circuit diagram is given in figure 2.

When we are reading the input (D0 to D4) by a PEEK command, lines D5 to D7, even though programmed for output, effect the input lines. The input value depends on the recent value poked into these locations. There are two ways to overcome this problem.

1) The 3 most significant bits of the input can be AND'ed with zeros. 2) The poked value can be added to the input and then compared.

The input is the column reading. The scanning program makes use of the second method because the second one takes less time.

Scanning program for PET BASIC.

```
5 A=TI
7 FOR I = 1 TO 500
10 FOR R=0 TO 192 STEP 64
20 POKE 59471,R ;Ground one row at a time
30 C=PEEK(59471) ;Get the column reading
40 IF C<>31+B THEN 80
50 NEXT R
60 NEXT I
65 B=TI-A
70 PRINT (B-44)/30 ; milliseconds.
80 END
```

TI is a variable of the PET computer which is updated every 1/60th of a second. The statement 10 FOR R=0 TO 192 STEP 64 controls two output lines. When R=0 is poked, the output from the lines D6 and D7 will be 00. When R=64 is poked, the output from these lines will be 01. When R=128 is poked, then the output from these lines will be 10. When R=192 is poked, then the output from these lines will be 11.
Figure 2. keyboard interface with the user port.
If these lines are connected to the decoder input to select 1 out of 4 outputs then, we can connect the decoder output directly to the four rows of the keyboard.

The time taken per scan when no key is pressed is 105.16 milliseconds. This means nearly 10 scans per second.

Decision making in PET BASIC.

We are assuming that variable $R$ contains the row number and variable $C$ contains the column reading. The following program is a decision-making program in PET Basic. Depending on the value of $R$ (row count) and $C$ (column reading) the program transfers control to a particular routine.

```
5    C=NOT C AND 31
8    R=R/32
10   IF R>=2 THEN 170
20   IF R=1 THEN 100
30   IF C>4 THEN 60
40   IF C<4 THEN 80
50   GOTO 540
60   IF C=8 THEN 560
70   GOTO 580
80   IF C=2 THEN 520
90   GOTO 500
100  IF C>4 THEN 130
110  IF C<4 THEN 150
120  GOTO 640
130  IF C=8 THEN 660
140  GOTO 680
150  IF C=2 THEN 620
160  GOTO 600
170  IF R=3 THEN 250
180  IF C>4 THEN 210
190  IF C<4 THEN 230
200  GOTO 740
210  IF C=8 THEN 760
220  GOTO 780
230  IF C=2 THEN 720
240  GOTO 700
250  IF C>4 THEN 280
260  IF C<4 THEN 300
```
R can take any one of the four values 0, 1, 2 or 3. C can take any one of the four values 30, 29, 27, 23 or 15. We negate C and AND it with 31 so that the new values of C are 1, 2, 4, 8 or 16 respectively. This program is based on the algorithm known as binary search. Even though this program is quite long, the time taken to jump to the corresponding routine is quite small. We are forced to use this algorithm, because in PET BASIC the line numbers cannot be used as variables. The jump table for the keys is given below.

<table>
<thead>
<tr>
<th>COLUMN</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROW 0</td>
<td>500</td>
<td>520</td>
<td>540</td>
<td>560</td>
<td>580</td>
</tr>
<tr>
<td>ROW 1</td>
<td>600</td>
<td>620</td>
<td>640</td>
<td>660</td>
<td>680</td>
</tr>
<tr>
<td>ROW 2</td>
<td>700</td>
<td>720</td>
<td>740</td>
<td>760</td>
<td>780</td>
</tr>
<tr>
<td>ROW 3</td>
<td>800</td>
<td>820</td>
<td>840</td>
<td>860</td>
<td>880</td>
</tr>
</tbody>
</table>

The timing will not be constant for all the keys because the sooner it identifies the key routine the sooner it will get out of the program. The maximum time is when C=30 and R=3 and is 33.33 milliseconds.
A machine language scan for the keyboard is

A000  LDY #0
A900  LDA #0
9D4F E8  STA $E84F
AA    TAX
AD4F E8 PP LDA $E83F
291F  AND #1F
C91F  CMP #1F
D020  BNE KEYYES
C8    INY
C004  CPY #4
F0EA  BEQ KK
8A    TXA
18    CLC
6920  ADC #$40
90E8  BCC PP

The control is transferred to KEYYES only if a key is pressed. Here we are assuming that no key is pressed. When the control is transferred to KK it means that no key is pressed and the total time can now be calculated.

The time taken for one scan when no key is pressed is 140 microseconds. The time is very short. Decision making in 6502 machine language is also very simple. All we have to do is make an indirect jump. This time is equal to 27 microseconds. If we want a machine language scan and a BASIC decision, then, we can make use of the SYS function. The SYS function transfers the control of the main program to an assembly language program whose starting address is specified by the SYS function. The control can again be transferred to the main program by a return from subroutine instruction in the assembly language subroutine.
SUMMARY: These results show that the interpreter is slow. A simple program like the scanning of the keyboard is very slow when an interpretive language is used. One point that we have observed is that it can never replace machine language when a short execution time is required in the application.
CHAPTER 3

Derivation of Empirical Formulas for Time Calculations.

Timing for PET BASIC

The main purpose of this chapter is to find a convenient way to time the BASIC statements. It is extremely difficult to time BASIC statements which contain constants. This is because the time heavily depends upon the actual values of the constants. The timing will be done on real variables because their timing is more consistent than the constants. Integer variables take slightly more time than real variables. We shall learn about the representation of the various types of variables in PET BASIC.

In PET BASIC (3), the variables are defined by two character alphanumerics. If it is a numerical variable then it has no trailing character. The character A is considered to be a variable, character AA is a different variable, A1 is yet another variable. All these are defined as numeric variables. If the variable contains alpha-numeric data then it is defined as a string. A string variable always ends with $. Thus, A and A$ are numeric and string variables respectively. AA$, likewise is different from AA etc. BASIC distinguishes a variable by the fact that the first
character is always an alphabetic character. The second character may be either numeric or alphabetic.

An integer variable ends with %. A5%, AA% are integer variables. PET can only recognise the first two characters. Hence AA is the same as AAB or AA1. There is a limit on the values of the line numbers. The maximum value of the line number is 63699. The PET can store any integer value from -32767 to 32767. Every integer needs two bytes of memory.

Floating point variables are stored in BASIC in five bytes - one for the exponent and four for the mantissa. The largest legal number the PET BASIC can handle is + or - 1.70141183 E+38 and any number larger than this will result in an overflow error. The smallest magnitude that can be distinguished from zero is 2.92873588 E-38. Any smaller number will result in an underflow error. A string variable is limited to 80 characters of the input buffer.

Arrays are differentiated by parentheses which follow the name of the array variable. Parentheses define a particular value within an array which can be used in an expression. Arrays can be of 3 types: integer, character and real.

The PET has an internal clock function built into the system. This clock can be addressed by TI. It is updated every 1/60 of a second. That is, whenever the screen is refreshed the clock is updated. Making use of this function we can calculate the timing of BASIC statements.
10 A=TI
20 FOR I= 1 TO 500
40 NEXT I
50 B=TI-A
60 PRINT B

In this program we are running through the loop 500 times. The time required to run the loop once is $B*1000/60/500$ milliseconds. The value of $B$ is 44. The clock is accurate only up to $1/60$ of a second. To get better accuracy we put the statement that is timed in the middle of the loop. It is repeated 500 times and the average value is taken. Statement 60 is changed to PRINT $(B-44)*1000/60/500$.

Each statement number 30 below is an independent experiment to determine the time. This statement is put into the timer program and the actual time is computed.

30 LET A=45   Time = 2.93 ms.
30 A=45       Time = 2.9 ms.
30 A=45.45    Time = 11.4 ms.

From the above statement we can see that using LET in a statement is a waste of time. We can also see that the time heavily depends on the actual values of the constants defined. Integer variables take more time than real variables.

30 IF C=45 THEN GOTO 40   Time = 4.9 ms.
30 IF C=45 THEN 40         Time = 4.83 ms.

Eliminating GOTO from an IF THEN GOTO statement saves a little time. PRINT statements, REMARK statements and parentheses must be avoided as much as possible. Throughout
the timing of BASIC we shall avoid time wasting statements.

30 D=0    Time = 1.8 ms.
30 D=5    Time = 1.76 ms.
30 D=9    Time = 1.73 ms.
30 D=10   Time = 2.8 ms.
30 D=99   Time = 2.86 ms.
30 D=100  Time = 3.76 ms.

We can see that the time is increasing at the rate of approximately 1 millisecond for every additional digit. As the expressions become more complex, the timing also gets more complex. If we have real numbers with decimal points then the timing gets still more complex. Since our main interest is in control type of programs, we generally deal with integers. But integers take more time than real variables. Hence we shall use real variables from now on. We shall not use LET before the statements, because, as we have seen previously, they waste time. We shall use two main programs for all time calculations and call them program one and program two.

Program one

3 GH=255: HJ=131
4 AS=121: SD=456541: DP=124765: FG=212
5 Q=2122: W=98745: E=45445: T=4466: R=3245
6 Z=22121: X=54212: C=54634: V=54641
7 B=21212: N=652151: M=52121: P=7878753
8 G=5454: WE=6512: FD=45656: TF=26045
9 JK=121212: KL=234: IO=156: OP=219
These two programs are similar except that the first one has more variables. These programs are chosen because it is found that the timing depends heavily on the number of variables in the program and also on the position of these variables. Even the time for a timing loop depends on the number of variables defined in the program before the loop. For example, we have to subtract 0.9 milliseconds from the time calculations when we are using program one. On an average, the loop takes an extra 0.033 milliseconds per variable defined before the loop. In program one there are 27 variables defined before the start of the loop. Simple constants are used in our programs so it is helpful to know a little about their timing.

```
10 A=TI
20 FOR I=1 TO 500
40 NEXT I
50 B=TI-A
60 PRINT (B-44)/30
```

Programming two

```
3 GH=255: HJ=131
10 A=TI
20 FOR I=1 TO 500
40 NEXT I
50 B=TI-A
60 PRINT (B-44)/30
```

The excess time is 2.733-1.8=0.933 milliseconds. The
number of variables defined are 27. The extra time per variable is \(0.933/27=0.033\) milliseconds. The time function in milliseconds, for a variable=constant type of statement is \(1.8 + \text{number of variables defined in the program} * 0.033\).

\[
D=L \quad \text{Variable} = \text{Variable}
\]

30 \(D=GH\) Time = 2.33 ms. using program one

30 \(D=GH\) Time = 1.56 ms. using program two

In program one, the total number of variables is 27. In program two, the total number of variables defined is 2. GH is the first variable in both the programs. The timing for the first statement is longer because there are 26 variables after the current variable. The current variable here is GH.

The extra time taken per variable defined after the current variable is \((2.33 - 1.56)/26 = 0.033\) milliseconds. It must be noted here that most of the calculations are approximated.

30 \(D=OP\) Time = 3.2 ms. using program one

Variable OP is the last variable defined in program one. This statement takes a considerable time more than the other two. This is because there are 26 variables defined before it. The extra time taken per variable defined before the current variable is \((3.2-1.56)/26 = 0.066\) milliseconds.

The timing function in milliseconds, for variable = variable type of statement is \(1.56 + 0.033 \times \text{NVA} + 0.066 \times \text{NVB}\) where NVA stands for number of variables after the current
variable and NVB stands for number of variables before the current variable. In order to get a clear picture of what we are saying by number of variables before and number of variables after the current variable, we shall take a small example and illustrate the point.

\[
10 \ A=473633; B=4546; C=7843; D=84537
20 \ E=879724; F=7957; G=457; H=758458
30 \ Z=E
\]

In this case, the current variable is \( E \) and the number of variables before it is 4 and the number of variables after it is 3. Hence the time for the statement \( Z=E \) is \( 1.56 + 0.066 \times 4 + 0.033 \times 3 \) milliseconds.

\[
D=Y+W \quad \text{Addition}
\]

\[
30 \ D=GH+HJ \quad \text{Time} = 3.6 \ \text{ms. using program one}
30 \ D=GH+HJ \quad \text{Time} = 2.56\text{ms. using program two}
30 \ D=IO+OP \quad \text{Time} = 5.26\text{ms. using program one}
\]

Here \( GH \) and \( HJ \) are the current variables. In program one there are 26 variables defined after these current variables. In program two there are no variables defined after the current variables. The extra time because of these 26 variables defined after the current variable is \( 3.6 - 2.566 = 1.033 \) milliseconds.

The extra time per variable defined after the current variable is equal to \( (3.6-2.566)/26 \) = .033 milliseconds. \( IO \) and \( OP \) are the last variables defined in program one. There
are 26 variables before them. Here IO and OP are the current variables. In program two, the number of variables defined before the current variables is zero. Hence the extra time per variable defined before the current variable is \((5.266 - 2.566)/26 = 0.1\) milliseconds.

30 \(D = GH + HJ + AS\) Time = 4.8 ms. using program one
30 \(D = OP + IO + KL\) Time = 7.4 ms. using program one

The time for extra addition is \(4.8 - 3.6 = 1.2\) milliseconds.

To find the time for extra addition, add another variable to the already existing statement \((D = GH + HJ + AS)\) and take the time by using program one and program two. In both cases the time for extra addition is the same \(1.2\) milliseconds.). This means the number of variables after the current variable (NVA) does not effect the time. Now we shall check and see whether the time is effected by the number of variables before the current variable.

The time difference between the statements \(D = OP + IO + KL\) and \(D = IO + OP\) is \(7.4 - 5.26 = 2.133\) milliseconds. This time is not equal to \(1.2\) milliseconds. The extra time is \(2.133 - 1.2 = 0.933\) milliseconds. Since there are 26 variables before this additional variable \((KL)\) the extra time per variable is equal to \(0.933/26 = 0.033\) milliseconds. Hence if more than one addition is taking place in one statement then the extra time for the additional addition is \(1.2 + 0.033 \times \text{number of variables defined before the current additional variable.}\)
The time function in milliseconds, for addition is

\[ 2.566 + 0.1 \times NVB + 0.033 \times NVA + 1.2 \]  
per each extra addition +

\[ 0.033 \times NVBAV. \]

Recall NVB stands for number of variables before the current variable and NVA stands for number of variables after the current variable. NVBAV stands for the number of variables before the additional variable.

Now we shall see what we mean by the current variable. To calculate NVB we count the number of variables before the current variable. But in the case of addition and other similar operations (subtraction, multiplication, division and pokes) we have two operands. For example if we have a statement like \( D = K + L \) then both \( K \) and \( L \) are the current variables. In order to calculate NVA and NVB for such cases we take the average values of their individual NVA's and NVB's. If NVA for \( K \) is 10 and if NVA for \( L \) is 6 then the NVA for addition is equal to \( (10+6)/2 = 8 \). We shall now take a small example and illustrate the point.

For example if we have the statement \( 30 \; D = HJ + FG + B \). To calculate the time we have to find NVA, NVB and NVBAV. To find NVA of this statement we take the average NVA of \( HJ \) and \( FB \) which is \( (26+22)/2 = 24 \). To find NVB of the statement then we take the average NVB of \( HJ \) and \( FB \) which is \( (1+5)/2 = 3 \). To calculate NVBAV we have to find the number of variables before \( B \) (B here is the additional variable) Here NVBAV is equal to 15. Hence the time for the statement \( 30 \; D = HJ + FG + B \) is 

\[ 2.6 + 0.1 \times 3 + 0.033 \times 24 + 1.35 \times 1 + 0.033 \times 15 \]
milliseconds. The number of variables after the additional variable does not effect the time.

\[ D = Y \times W \] \hspace{1cm} \text{Multiplication}

30 \( D = GH \times HJ \) \hspace{1cm} \text{Time} = 4.5 \text{ ms. using program one}

30 \( D = GH \times HJ \) \hspace{1cm} \text{Time} = 3.6 \text{ ms. using program two}

30 \( D = IO \times OP \) \hspace{1cm} \text{Time} = 6.2 \text{ ms. using program one}

The extra time per variable after the current variable is \((4.5 - 3.6)/26 = 0.033\) milliseconds. The extra time per variable before the current variable is \((6.2 - 3.6)/26 = 0.1\) milliseconds.

30 \( D = GH \times HJ \times AS \) \hspace{1cm} \text{Time} = 6.93 \text{ ms. using program one}

30 \( D = IO \times OP \times KL \) \hspace{1cm} \text{Time} = 9.766 \text{ ms. using program one}

Time for extra multiplication is \(6.933 - 4.5 = 2.433\) milliseconds. Time for extra multiplication per variable before the current variable is \((9.76 - 6.2 - 2.433)/26 = 0.033\) milliseconds. The time function in milliseconds, for multiplication is \(3.6 + 0.1 \times NVB + 0.033 \times NVA + 2.433 \times \text{number of additional multiplications} + 0.033 \times NVBAV.\)

\[ D = Y - W \] \hspace{1cm} \text{Subtraction}

30 \( D = GH - HJ \) \hspace{1cm} \text{Time} = 3.56 \text{ ms. using program one}

30 \( D = GH - HJ \) \hspace{1cm} \text{Time} = 2.66 \text{ ms. using program two}

30 \( D = IO - OP \) \hspace{1cm} \text{Time} = 5.16 \text{ ms. using program one}

30 \( D = GH - HJ - AS \) \hspace{1cm} \text{Time} = 4.9 \text{ ms. using program one}

30 \( D = KL - IO - OP \) \hspace{1cm} \text{Time} = 7.33 \text{ ms. using program one}

The extra time per variable defined after the current
variable is equal to \((3.56 - 2.66)/26 = 0.033\) milliseconds. The extra time per variable defined before the current variable is equal to \((5.16 - 3.56)/26 = 0.1\) milliseconds. The extra time for additional subtraction is equal to \(4.9 - 3.56 = 1.35\) milliseconds. The time for the extra subtraction per variable defined before the current variable is equal to \((7.33 - 4.9 - 1.35)/26 = 0.033\) milliseconds.

Hence the time function in milliseconds, for subtraction is \(2.6 + 0.1 \times NVB + 0.033 \times NVA + 1.35 \times \text{number of additional subtractions} + 0.033 \times NVBAV\).

\[
D = Y/W \quad \text{Division}
\]

30 \(D = GH/HJ\) \hspace{1cm} \text{Time} = 5.0 \text{ ms. using program two}

30 \(D = GH/HJ\) \hspace{1cm} \text{Time} = 5.93 \text{ ms. using program one}

30 \(D = IO/OP\) \hspace{1cm} \text{Time} = 7.76 \text{ ms. using program one}

30 \(D = GH/HS/AS\) \hspace{1cm} \text{Time} = 9.53 \text{ ms. using program one}

30 \(D = IO/OP/KL\) \hspace{1cm} \text{Time} = 12.16 \text{ ms. using program one}

The extra time per variable defined after the current variable is equal to \((5.933 - 5)/26 = 0.033\) milliseconds. The extra time per variable defined before the current variable is equal to \((7.76 - 5)/26 = 0.1\) milliseconds. The extra time for additional division is equal to \(9.53 - 5.93 = 3.6\) milliseconds. The extra time for additional division per variable defined before the current variable is \((12.16 - 7.76 - 3.6)/26 = 0.033\) milliseconds.

Hence the time function in milliseconds, for division is \(5 + 0.1 \times NVB + 0.033 \times NVA + 3.6 \times \text{number of additional divisions} +\)
The time depends on the number of characters in the remark statement. It does not depend on the number of variables defined. Time for each character is $(1.9 - .3)/77 = .02$ ms. Hence the time in milliseconds, for remark statement is $0.3 + 0.02 \times \text{number of characters in the remark statement.}$

GOTO NNN

It is found that the time for a GOTO statement depends heavily on the number of statements before the statement number NNN, where NNN is the destination statement number. To find this additional time, we shall put in some remark statements at the beginning of the program and observe the increase in time.

<table>
<thead>
<tr>
<th>Remarks</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.40 ms.</td>
</tr>
<tr>
<td>1</td>
<td>1.46 ms.</td>
</tr>
<tr>
<td>2</td>
<td>1.53 ms.</td>
</tr>
<tr>
<td>3</td>
<td>1.60 ms.</td>
</tr>
<tr>
<td>4</td>
<td>1.66 ms.</td>
</tr>
<tr>
<td>5</td>
<td>1.76 ms.</td>
</tr>
</tbody>
</table>

The average extra time for one statement is
(1.76-1.4)/5=0.072 milliseconds. The time for GOTO 40 with 2 remarks is 1.53 milliseconds. There are 12 statements before the statement number 40. Hence the actual time for a GOTO statement is 1.533-12\*0.072=0.7 milliseconds. The time function, in milliseconds for GOTO NNN is 0.7+0.072\*number of statements before the statement NNN.

( ) Parentheses

30 D=G\*H+HJ Time = 3.6 milliseconds.
30 D=(G\*H+HJ) Time = 4 milliseconds.
30 D=(G\*H+HJ)) Time = 4.4 milliseconds.

Hence, the average time per parentheses pair is 0.4 milliseconds.

IF (CONDITION) THEN (DO SOMETHING)

First, we shall find the timing when the condition is not satisfied.
30 IF G\*H=HJ THEN 100 Time = 2.6 ms. (program one)
30 IF G\*H=HJ THEN 100 Time = 1.7 ms. (program two)
30 IF I\*O=O\*P THEN 100 Time = 4.266 ms. (program one)

The extra time per variable after the current variable is (2.6-1.7)/26 = 0.033 milliseconds. The extra time per variable before the current variable is (4.266-1.7)/26 = 0.1 milliseconds. The timing function for IF X=Y THEN GOTO NNN when condition is not satisfied is 1.7+0.1\*N\*V\*B+0.033\*N\*V\*A milliseconds.

It should be noted that the condition is treated as an
ordinary statement when more variables are involved. For example, if instead of \( GH=HJ \), we have \( GH=HJ+AS \), then the time will be increased by an additional addition and so on.

When the condition is satisfied

The timing for condition satisfied is more complex.

30 IF \( GH=HJ \) THEN REM Time = 2.8 ms. (program one)
30 IF \( GH=HJ \) THEN REM Time = 2.8 ms. (program two)
30 IF \( IO=OP \) THEN REM Time = 4.4 ms. (program one)

The extra time per variable after the current variable is 0 milliseconds and the extra time per variable before the current variable is \((4.4-2.8)/26=0.066\) milliseconds.

30 IF \( GH=HJ \) THEN GOTO 40 Time = 3.8 ms. (program one)
30 IF \( IO=OP \) THEN GOTO 40 Time = 5.43 ms. (program one)

The extra time per variable after the current variable is \((5.433-3.8)/26=0.066\) milliseconds. In this statement the time taken by GOTO 40 is \( 0.7+.072*10 =1.4 \) milliseconds.

The time for IF THEN is 3.8-1.4 = 2.4 milliseconds.

30 IF \( GH=HJ \) THEN D=IO+OP Time = 7.7 ms. (program one).

The time taken by the statement D=IO+OP is 5.26 milliseconds. Hence, the total time taken is 2.4+5.26=7.6 milliseconds which agrees with the above result (7.7 milliseconds.)

The statement after THEN is treated as an ordinary statement for time calculations. If the operator is different from \( = \) then add 0.5 milliseconds to the overall
time. If the condition is different from \texttt{variable=variable} then for extra time calculations it is treated as a separate statement. Hence, for IF THEN type of statements the time function in milliseconds, is $2.4+0.5$ if the operator is other than $=$) $+$ the extra time for the condition plus the extra time for the statement that is executed when the condition is satisfied.

\begin{verbatim}
FOR I=K TO NNN:NEXT I
30 FOR J=1 TO 10
35 NEXT J Time = 28.6 ms. (program two)
30 FOR J=1 TO 20
35 NEXT J Time = 52.2 ms. (program two)
30 FOR J=1 TO 30
35 NEXT J Time = 76.1 ms. (program two)
\end{verbatim}

Average time for looping 10 times is 23.6 milliseconds. Hence the average time for looping one time $= 2.36$ milliseconds. The initial time or the base time for a FOR loop is $28.6-23.6=5$ milliseconds.

30 FOR J=1 TO 10 STEP 1

Extra time when \texttt{STEP} is used is $29.36-28.66 = 1.2$ milliseconds.

The \texttt{FOR} statement can have a big expression. Instead of \texttt{J=1 TO 10} we can have \texttt{J=ER*G/L TO JU*3 STEP W+R}. In this case these expressions are treated as ordinary statements and the extra time is calculated. The timing in such cases is not very consistent and if possible, should be
avoided. Hence the time function in milliseconds for the loop is \(5 + 2.36 \times \text{number of times the loop is repeated} + 1.3\) if step is used + extra time for the loop expressions.

**PEEK and POKE**

30 D=PEEK(GH) Time = 3.23 ms. (program one)
30 D=PEEK(GH) Time = 2.36 ms. (program two)
30 D=PEEK(OP) Time = 4.1 ms. (program one)

The extra time per variable defined after the current variable is equal to \((3.23 - 2.36)/26 = 0.033\) milliseconds. The extra time per variable defined before the current variable is equal to \((4.1 - 2.36)/26 = 0.066\) milliseconds. Hence the time function in milliseconds for PEEK function is \(2.36 + 0.066 \times NVB + 0.033 \times NVA + \text{extra time for the expressions in the parentheses.}

30 POKE GH,HJ Time = 2.3 ms. (program one)
30 POKE OP,OP Time = 2.3 ms. (program two)
30 POKE GH,HJ Time = 3.93 ms. (program one)

The extra time per variable defined after the current variable is equal to 0 milliseconds. The extra time per variable defined before the current variable is \((3.933 - 2.3)/26 = 0.066\) milliseconds. Hence the time function in milliseconds for a POKE statement is \(2.3 + 0.066 \times NVB + \text{extra time for the expressions.}

**Logic Function NOT**

Logic function operate only on variables which are
integers or real variables whose value is less than or equal to 255.

30 D=NOT GH  Time = 3.26  ms. (program one)
30 D=NOT GH  Time = 2.4  ms. (program two)
30 D=NOT OP  Time = 4.13  ms. (program one)
30 D=NOT NOT GH Time = 5.03  ms. (program one)
30 D=NOT NOT NOT GH Time = 5.96  ms. (program one)
30 D=NOT NOT OP Time = 6.03  ms. (program one)

The extra time per variable defined after the current variable is equal to (3.266-2.4)/26 = .033 milliseconds.
The extra time per variable defined before the current variable is equal to (4.133-2.433)/26=0.1 milliseconds. The extra time for additional NOT is 5.033-4.133=.9 milliseconds. The extra time for additional NOT per variable defined before the current variable is equal to (6.033-4.133-.9)/26= 0.033 milliseconds. Hence the time function in milliseconds for a NOT statement is 2.4+0.1*NVB +0.033*NVA+0.9 for every additional NOT + 0.033*NVBAV.

Logic Function AND

30 D=GH AND HJ  Time = 4.1  ms. (program one)
30 D=GH AND HJ  Time = 3.23  ms. (program two)
30 D=IO AND OP  Time = 5.73  ms. (program one)
30 D=GH AND HJ AND AS  Time = 6.0  ms. (program one)
30 D=IO AND OP AND KL  Time = 8.8  ms. (program one)

The extra time per variable defined after the current
variable is equal to \((4.1-3.23)/26=0.033\) milliseconds. The extra time per variable defined before the current variable is equal to \((5.73-3.23)/26 = 0.1\) milliseconds. The extra time for an additional AND is \(6.2-4.1=2.1\) milliseconds. The extra time of these additional AND operations per variable defined before the current variable is \((8.8-5.73-2.1)/26=0.033\) milliseconds. Hence the time function in milliseconds, for an AND is 
\[3.23+0.1*NVB+0.033*VAt+2.1*\text{number of additional AND's}+0.033*NVBAV.\]

**Logic Function OR**

30 D=GH OR HJ Time = 4.1 ms. (program one)  
30 D=GH OR HJ Time = 3.3 ms. (program two)  
30 D=IO OR OP Time = 5.73 ms. (program one)  
30 D=GH OR HJ OR AS Time = 5.9 ms. (program one)  
30 D=IO OR OP OR KL Time = 8.66 ms. (program one)  

The extra time per variable defined after the current variable is equal to \((4.1-3.3)/26=0.033\) milliseconds. The extra time per variable defined before the current variable is equal to \((5.73-3.33)/26=0.1\) milliseconds. The extra time for additional OR is \(5.9-4.1=1.8\) milliseconds. The extra time for this additional OR per variable defined before the current variable is \((8.66-5.73)/26=0.033\) milliseconds. Hence, the time function in milliseconds for an OR is 
\[3.3+0.1*NVB+0.033*VNA+1.8*\text{number of additional OR}+0.033*NVBAV.\]
We shall put all our timing information at one place so that it can be easily referenced. We shall, in the following table put all the expressions that are useful in calculating the time. We shall call this Table 1. All the times are in milliseconds. The symbols used in the expressions are NVA, NVB, NVBAV and NAOP, where NVA stands for the number of variables after the current variable, NVB stands for the number of variables before the current variable, NVBAV stands for the number of variables before the additional variable and NAOP stands for the number of additional operations. These operations can be additions or subtractions or multiplications or logical operations etc.

Quite often in this table we use the expression 'for extra time the expression is treated as an ordinary statement'. In order to understand what we are saying by the above statement we shall take a small example and illustrate the point. For example, the time taken by the statement IF X=Y THEN 100 is, say, 10 milliseconds. The time taken by the statement IF X+Z*R=Y THEN 100 is 10 milliseconds plus the extra time when the expression is treated as an ordinary statement. This extra time is equal to the time for an additional multiplication and an additional addition. Hence the total time is equal to 10+2.43+1.2 milliseconds.
Table 1

1 a) Variable = Variable \( T = 1.56 + 0.066 \times NVB + 0.033 \times NVA. \)

1 b) Variable = Constant \( T = 1.8 + 0.033 \times \) number of variables defined in the program.

2) Addition \( T = 2.566 + 0.1 \times NVB + 0.033 \times NVA + 1.2 \times NAOP + 0.033 \times NVBAV. \)

3) Subtraction \( T = 2.6 + 0.1 \times NVB + 1.35 \times NAOP + 0.033 \times NVBAV. \)

4) Multiplication \( T = 3.6 + 0.1 \times NVB + 0.033 \times NVA + 2.43 \times \)
\( \) NAOP + 0.033 \times NVBAV.

5) Division \( T = 5 + 0.1 \times NVB + 0.033 \times NVA + 3.6 \times NAOP + 0.033 \times NVBAV. \)

6) Remark \( T = 0.3 + 0.02 \times \) number of characters in the remark statement.

7) Goto NNN \( T = 0.7 + 0.072 \times \) number of statements before the label NNN.

8) Parentheses \( T = 0.4 \) milliseconds per parentheses pair.

9) If \( X = Y \) then (do something), when the condition is not satisfied:
\( T = 1.7 + 0.1 \times NVB + 0.033 \times NVA \) (here \( X \) and \( Y \) are the current variables). If the condition is different from \( variable = variable \) type then, for extra time calculations, it is treated as an ordinary statement. (The expression to the right of 'then' does not effect the time.)

If the condition is satisfied:

a) If \( GH = HJ \) then 'statement' \( T = 2.4 + 0.066 \times NVA + 0.5 \) if the operator is different from '=' the time taken by 'statement' + extra time when the condition is different from \( variable = variable \) type. For this extra time the condition
is treated as an ordinary statement.

10) Peek \[ T = 2.36 + .066 \times NVB + .033 \times NVB^+ \] (if the expression in the parentheses is different from a simple variable then for extra time calculations it is treated as an ordinary statement.)

11) Poke \[ T = 2.3 + .066 \times NVB^+ \] (if the expressions in the poke statement are other than a simple variable then for extra time calculations they are treated as ordinary statements.)

12) Not \[ T = 2.4 + .1 \times NVB + .033 \times NVA + .9 \times NAOP + .033 \times NVBAV. \]

13) And \[ T = 3.23 + .1 \times NVB + .033 \times NVA + 2.1 \times NAOP + .033 \times NVBAV. \]

14) Or \[ T = 3.3 + .1 \times NVB + .033 \times NVA + 1.8 \times NAOP + .033 \times NVBAV. \]

Timing for Tiny BASIC

Tiny BASIC has 26 variables (4) that can be used in the user programs. These variables are represented by the 26 alphabetic characters of the English language. Each variable is located in two memory locations. The values are stored in binary in 2's complement form. The memory locations for the variables are fixed.

In order to calculate the timing of the Tiny BASIC statements we need a hardware timer. The details of the timer are given in the appendix. The timer runs on a 1 MHz clock. The timer can count up to 999999 microseconds or nearly equal to 1 second. The minimum time it can count is 10 microseconds. It has three control lines to control the timer - one to clear the counter, one to start the counter and one to stop the counter. The address bus of the
HYATHKIT is decoded to give these control lines. A STAA $D000,$D001,$D002 will clear, start and stop the counter.

The control signals can be generated by the USR function.

```plaintext
10 LET A=USR(7182,53250) :Clear the counter
20 LET A=USR(7188,53249) :Start the counter
40 LET A=USR(7188,53250) :Stop the counter
50 END
```

The comments 'clear the counter', 'start the counter' and 'stop the counter' are put there to make the meaning of the statements clear and they do not appear in the actual BASIC timing program.

The time taken to start and stop the counter is 19.67 milliseconds. Hence in all time calculations this value is subtracted to give the actual time.

Each statement below is a different experiment to determine the time. This statement is put in the timer program and the time is measured on the hardware timer.

```plaintext
30 LET K=999     Time = 6.94 ms.
30 K=999        Time = 10.36 ms.
```

From the above timing information we see that using LET in a statement, saves time. In PET BASIC we wasted time by using LET in the statements. PRINT and parentheses also waste time. REMARK statements should also be avoided from the running program. From now on we shall use LET before the statements.

D=10   Variable=constant.
30 LET C=2       Time = 6.7 ms.
30 LET C=45633   Time = 7.5 ms.

If the constant is simple like 2 or 4 etc. then the time taken is 6.7 milliseconds. But if the constant is a large number then add 0.8 milliseconds to it.

D=L   Variable=Variable
30 LET C=K       Time = 6.7 ms.

D=Y+L   Addition
30 LET C=K+L     Time = 10.16 ms.
30 LET C=K+L+D   Time = 13.71 ms.
30 LET C=K+L+D+F Time = 17.26 ms.
30 LET C=K+L+D+F+G Time = 20.81 ms.

Time for the first addition is 10.16 milliseconds and the time for each extra addition is 3.55 milliseconds.

30 LET C=34567+23644 Time=11.76 ms.
30 LET C=3553+464644+3535 Time=16.11 ms.
30 LET C=25455+53535+3535+45753 Time=20.46 ms.

When constants are used, the time for first addition is 11.76 milliseconds. The number of operands in this expression is 2. The extra time taken by the constants is 1.6 milliseconds. The extra time for using a constant instead of a variable is 1.6/2=0.8 milliseconds. The time for an extra addition is 4.35 milliseconds. Even this time is 0.8 milliseconds more, when compared to variables.
Y=D-L  Subtraction

30 LET C=K-L  Time = 10.52 ms.
30 LET C=K-L-D-F  Time = 18.32 ms.
30 LET C=K-L-D-F-J  Time = 22.22 ms.
30 LET C=24225-56474  Time = 12.12 ms.
30 LET C=346264-3462-6473  Time = 16.82 ms.
30 LET C=24545-6645-5757-56473  Time = 21.52 ms.

Time for the first subtraction is = 10.52 milliseconds and the time for each additional subtraction is 3.9 milliseconds. Time for the first subtraction with constants is 12.12 milliseconds. For each additional subtraction the time is 4.7 milliseconds. The extra time taken by a constant to replace a variable is 0.8 milliseconds.

Y=D*L  Multiplication

30 LET C=K*L  Time = 9.64 ms.
30 LET C=K*L*M  Time = 12.53 ms.
30 LET C=K*L*M*N  Time = 15.52 ms.
30 LET C=52453*4546  Time = 11.24 ms.
30 LET C=3469*87762*5455  Time = 15.04 ms.

Time for the first multiplication using variables is 9.54 milliseconds and the time for each additional multiplication is 3.00 milliseconds. Time for the first multiplication using constants is 11.24 milliseconds and the time for each additional multiplication is 3.8 milliseconds. The extra time taken by a constant to replace a variable is
0.8 milliseconds.

\[
Y = \frac{D}{L} \quad \text{Division}
\]

30 LET \(C = \frac{K}{L}\) \hspace{1cm} \text{Time} = 10.67 \text{ ms.}
30 LET \(C = \frac{K}{L/M}\) \hspace{1cm} \text{Time} = 14.78 \text{ ms.}
30 LET \(C = \frac{K}{L/M/N}\) \hspace{1cm} \text{Time} = 18.89 \text{ ms.}
30 LET \(C = \frac{K}{L/M/N/0}\) \hspace{1cm} \text{Time} = 23.00 \text{ ms.}
30 LET \(C = \frac{34566}{34353}\) \hspace{1cm} \text{Time} = 12.32 \text{ ms.}
30 LET \(C = \frac{5343}{35543/23234}\) \hspace{1cm} \text{Time} = 17.71 \text{ ms.}
30 LET \(C = \frac{54656/5656}{24345/65643}\) \hspace{1cm} \text{Time} = 22.64 \text{ ms.}

The time for the first division is 10.67 milliseconds. The time for each additional division is 4.11 milliseconds. The time for first division using constants is 12.32 milliseconds. The time for each additional division is 4.92 milliseconds. The extra time taken by a constant to replace a variable is 0.8 milliseconds.

Verifications

Now we shall put multiplications, divisions, additions and subtractions in one statement and calculate the timing and compare with the observed timing for 3 examples.

30 LET \(C = \frac{K}{L*M*N/O}\) \hspace{1cm} \text{Time observed} = 20.75 \text{ milliseconds.}

Time calculated is equal to the time for first division + time for additional multiplication + time for additional multiplication + time for additional division = 10.67 + 3 + 3 + 4.1 = 20.77 milliseconds. The difference is only
0.02 milliseconds, which is equal to 0.1\%.

30 LET C=K*L/N*O*P/Q Time observed = 27.83 milliseconds.

Time calculated is 9.64+4.1+3+4.1+3+4.1=27.94 milliseconds.

30 LET C=K*45678+23456 Time observed = 14.78 milliseconds.

Time calculated is 9.64+3.55+.8+.8 = 14.79 milliseconds. We have added 0.8 milliseconds twice because there are two constants. In all these cases we have seen that the calculated time coincided well with the observed time with an error of less than 0.3%.

GOTO NNN

It is observed that the time for a GOTO NNN statement depends heavily on the location of the destination statement number NNN. Consider the following experiment:

30 GOTO 40 Time = 10.64 ms.

1 REM KKKKKKKKKKKKKK
30 GOTO 40 Time = 11.55 ms.

1 REM KKKKKKKKKKKKKK
2 REM LLLLLLLLLLLLLL
30 GOTO 40 Time = 12.47 ms.
In the last example, remark statements 1 and 2 have 20 characters each and remark statement 3 has 40 characters. The time for a remark statement of 20 characters is $11.55 - 10.64 = 0.91$ milliseconds. The time for a remark statement of 40 characters is $12.47 - 10.64 = 1.82$ milliseconds. The time for 20 characters is $1.67 - 0.91 = 0.76$ milliseconds. The time for every new line is $0.91 - 0.76 = 0.15$ milliseconds. The time per character is $0.76 / 20 = 0.038$ milliseconds. It does not matter even if we have some other statement instead of remark statements because the timing is still the same. In our timing program there are 3 statements before the destination statement 40 and the number of characters in these 3 statements is 50. Hence, the actual time for the above GOTO statement is $10.64 - 0.038 * 50 - 0.15 * 3 = 8.24$ milliseconds. Blanks between the statements are also considered as characters.

Hence the time function in milliseconds, for the GOTO NNN is $8.24 + 0.15 * \text{number of lines before the destination statement NNN} + 0.038 * \text{number of characters in the lines mentioned above}.$

In Tiny BASIC we can use variables for line numbers.

30 GOTO M+N Time = 14.26 ms.
30 GOTO M+N+O Time = 14.26 ms.
30 GOTO M+N+O+P Time = 17.83 ms.

For each extra addition the time taken is 3.62
milliseconds.

30 GCTO M/N  Time = 14.74 ms.
30 GCTC M/N/O  Time = 18.84 ms.
30 GCTO M/N/O/P  Time = 22.94 ms.

For each extra division the time taken is 4.1 milliseconds.

30 GGTO M-N  Time = 14.69 ms.
30 GCTC M-N-C  Time = 18.74 ms.
30 GCTO M-N-C-P  Time = 22.79 ms.

For each extra subtraction the time taken is 4.05 milliseconds.

30 GOTO M*N  Time = 13.56 ms.
30 GCTO M*N*O  Time = 16.64 ms.
30 GCTO M*N*O*P  Time = 19.40 ms.

For each extra multiplication the time taken is 2.92 milliseconds.

GOSUB NNN

Of all the Tiny BASIC statements, GOSUB is the most complicated one.

30 GOSUB 60 : 50 END : 60 RETURN  Time = 19.13 ms.
5  REM KKKKKKKKKKKKKKKKKK  Time = 20.98 ms.
7  REM LLLLLLLLLLLLLLLLLL  Time = 22.81 ms.
8  REM MMMMMMMMMMMMMMMMMMMMM  Time = 26.17 milliseconds.

The GOSUB statement is located at statement number 30 and all these remark statements appear before the GOSUB
statement. Extra time due to 1 remark statement with 20 characters is \(22.81 - 20.98 = 1.83\) milliseconds. Extra time due to 1 remark statement with 40 characters is \(26.17 - 22.81 = 3.36\) milliseconds.

Time for 20 characters is \(3.36 - 1.83 = 1.53\) milliseconds. Hence the time per character is \(1.53 / 20 = 0.076\) milliseconds. Time per one statement is \(1.83 - 1.53 = 0.3\) milliseconds.

These timings are doubled when compared to the GOTO timings. But if the remark statements (these can be any statements) are located after the GOSUB statement and before the GOSUB routine then the timings are different. If these statements are located after the GOSUB routine then they have no effect on the timing.

42 REM KKKKKKKKKKKKKKKKKK Time = 20.04 ms.
44 REM LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL Time = 21.71 ms.

The time per extra character a after GOSUB statement and before a GOSUB routine is \(((21.71 - 20.04) - (20.04 - 19.13)) / 20 = 0.038\) milliseconds. Time for each extra line is \(0.91 - 0.76 = 0.15\) milliseconds.

In the GOSUB statement we can also use variables.

30 GOSUB M Time = 36.20 ms.
30 GCSUB M+N Time = 39.91 ms.
30 GCSUB M+N+O Time = 43.61 ms.

Extra time for each addition is 3.7 milliseconds.

30 GCSUB M-N Time = 40.26 ms.
30 GCSUB M-N-O Time = 44.32 ms.

Extra time for each subtraction is 4.06 milliseconds.
milliseconds.
30 GCSUB M/N Time = 40.44 ms.
30 GCSUB M/N/O Time = 44.71 ms.
Extra time for each division is 4.27 milliseconds.
30 GCSUB M*N Time = 39.30 ms.
30 GCSUB M*N*O Time = 42.20 ms.
Extra time for each multiplication is 3.1 milliseconds.
30 GCSUB 80 Time = 36.80 ms.
There were 3 statements before the GOSUB statement and
2 statements between the GOSUB statement and the GOSUB
routine. Hence the time for GOSUB is 36.87 - (19.67+3 * 
.3+.48 * .076+2 * .15+23* 0.038) = 11.478 milliseconds, where
19.67 milliseconds is the time taken by the counter.

IF (CONDITION ) THEN (DO SOMETHING)
First we shall time the IF...THEN statement when the
condition is not satisfied.
30 IF M = N THEN GOTO 40 Time = 33.57 ms.
When the condition is not satisfied then the time does
not depend on the number of statements before or after the
IF THEN statement.
30 IF M > N THEN GOTO 40 Time = 34.85 ms.
If the operator is other than ' = ', then the time taken
is 1.3 milliseconds more.
30 IF M=N THEN GOTO 40 Time = 33.58 ms.
30 IF M+N = 0 THEN GOTO 40 Time = 37.15 ms.
The difference of time is \(37.15 - 33.58 = 3.57\) milliseconds, which is nearly equal to the time for an extra addition. Hence, if the condition is not satisfied then the condition operators are treated just like an ordinary statement for time calculations. The destination statement number does not effect the time. Hence, the time when the condition is not satisfied is 13.9 milliseconds.

**Condition satisfied**

30 IF M=N THEN GOTO 40 Time = 23.64 ms.

5 REM

30 IF M=N THEN GOTO 40 Time = 23.91 ms.

5 REM

6 REM

30 IF M=N THEN GOTO 40 Time = 24.18 ms.

In the above cases, the extra time for one remark statement is 0.27 milliseconds. By our previous calculations of a GOTO statement we have seen that the extra time for a new line is equal to 0.15 milliseconds, and the extra time for one character is 0.038 milliseconds. In our case we have an extra line and 3 extra characters. Hence the extra time must be equal to \(3 \times 0.038 + 0.015 = 0.266\) which coincides with the observed value of 0.27 milliseconds. Hence the time for an IF THEN GOTO statement when the condition is satisfied is 23.64 - 3 * 0.15 - 60 * 0.038 = 20.9 milliseconds. If the operator is different from '=' then add 1.3 milliseconds to
the overall time.

30 IF M=N THEN D=K Time = 24.95 ms.

30 IF M=N THEN D=K+L Time = 28.5 ms.

The extra time is 28.5-24.95=3.55 milliseconds. This is equal to the time for an extra addition. For all extra time calculations, an expression to the right of the 'THEN' must be treated as an ordinary statement.

30 IF M>N THEN D=K Time = 26.23 ms.

The extra time for operators other than '=' is 26.23-24.95= 1.3 milliseconds.

30 IF M=N THEN GOSUB 70 Time = 31.68 ms.

5 REM

30 IF M=N THEN GOSUB 70 Time = 32.22 ms.

The excess time for a remark statement of 3 characters is 32.22-31.68=0.54 milliseconds. This again coincides with the timing of GOSUB statement.

30 IF M>N THEN GOSUB 70 Time = 32.98 ms.

Once again when the operator is different from '=', the excess time is 1.3 milliseconds. The time for an IF THEN GOSUB statement is 31.68-.3*3 -.076 * 60-.32 -.038 * 23 = 25 milliseconds.

REM  Remark

30 REM Time = 4.71 ms.

30 REM KKKKKKKKKKKKKKKKKKK..80 characters...KK

The time taken = 7.63 milliseconds. Hence the time for a remark statement is T = 4.71+.038 * number of characters.
Parentheses

30 LET K=C    Time = 6.7 ms.
30 LET K=(C)  Time = 11.9 ms.
30 LET K=((C)) time taken = 17.1 ms.

Hence the time per parentheses pair is T=17.1-11.9 = 5.2 milliseconds. Now we shall put all our timing information for Tiny Basic at one place so that it can easily be accessed. We shall call this as table 2.

Table 2

1) Variable=Constant Time = 6.7 milliseconds.
2) Addition: Time for first addition is 10.16 milliseconds. Time for each extra addition is 3.55 milliseconds.
3) Subtraction: Time for first subtraction = 10.52 milliseconds. Time for each additional subtractions is 3.9 milliseconds.
4) Multiplication: Time for first multiplication = 9.54 milliseconds. Time for each additional multiplications is 3.0 milliseconds.
5) Division: Time for first division = 10.67 milliseconds. Time for each additional division is 4.11 milliseconds.
6) Goto NNN: Time in milliseconds for goto NNN is 8.24 + 0.15 * number of lines before the destination statement(NNN) + 0.038 * number of characters in the lines mentioned above.
7) Gosub NNN: Time in milliseconds for Gosub NNN is 11.48 + .3 * number of lines before the Gosub statement + .076 *
number of characters in these lines + .015 * number of statements after the Gosub statement and before the Gosub routine + .038 * number of characters in these lines.

8) If condition then:
   a) When the condition is not satisfied the time is 13.9 milliseconds.
   b) When the condition is satisfied.
      bb) If M=N then D=R The time is 4.95 milliseconds.
      bc) If M=N then qoto 40 Time in milliseconds is 20.9 + .15 * number of lines before the destination label (40 in this example) + .038 * number of characters in these lines.
      bd) If M=N then qosub 70 Time in milliseconds is 25 + .3 * number of lines before the qosub statement + .076 * number of characters in these lines + .15 * number of lines after the qosub statement and before the statement where the subroutine starts + .038 * number of characters in these lines.

Important: If the operator is other than '=' then add 1.3 milliseconds.

9) Remark: Time in milliseconds is 4.71 + .038 * number of characters.

10) Parentheses: Time = 5.2 milliseconds per parentheses pair.

In all the above cases, when constants are used instead of variables add 0.8 milliseconds for every variable used. If the constants are simple like 1 or 2 or 6, then do not add 0.8 milliseconds.
CHAPTER 4

Timing Verification for both PET BASIC and Tiny BASIC Programs.

The verifications of the programs is done by using the elevator control program. The elevator has 15 input switches and 17 display LED's. They all are mounted on the panel of the elevator model. Out of the 15 switches 5 are UP switches, 5 are DOWN switches and 5 are the CAR switches. The 17 LED's are divided as follows: 5 UP-DOWN displays, 5 CAR displays, 5 FLOOR displays, one for the opening and closing of the doors and one to indicate whether the elevator has satisfied all requests or not.

The elevator is basically designed to run off the Heathkit, but it can run on any system with a little interfacing. The switches information is stored in the latches. Since the interpretive language is slow, the latches are required. If the elevator is running off the assembly language then these latches would not be required.

The elevator has a total of 9 I.C chips on board. They are two 74LS257's, four 74LS279's, and three 74174's. 74LS257 is a tristate multiplexer, 74LS279 is a latch and 74174 is a D flip flop.

The control program of the elevator is written so that when all the requests are satisfied the elevator will come to the first floor. If the elevator is going up then it
will satisfy all the up requests first. Once the elevator starts coming down, will it satisfy all the down requests and only then, it will go up provided, there is such a request. The current position of the car can be seen by the middle row of the LED's. Whenever a key is pressed the corresponding LED is lit to indicate that it has recognized the key closure. The LED will only be switched off when the request (for the corresponding switch) is satisfied by the elevator. The opening and closing of the doors is indicated by the OPEN LED. The diagram is given in figure 3. We shall take each part of the circuit and analyze it. In this circuit we have a common data bus. The switch information must be read by the CPU via the data bus and also the data for the LED display is sent to the LED's via the same data bus. The data bus is only 8 bits but we need 15 bits of input data, that is one bit per each switch. We need eight 2 to 1 data selectors (74LS257). Since 74LS257's have tristate outputs, we can directly connect them to the data bus.

One 74LS279 can store the information of 4 switches. Hence we need 4 of these chips to store the status of the 15 input switches. Each of the 74174 hex D flip flops can store 6 bits of data. All the LED's are connected to a common 5 volts supply. A logic zero in the outputs of the 74174 will light the LED. Six LED's can be connected to each 74174. Since the number of LED's is 17 we need at least 3 74174 chips. They can be directly connected to the
Figure 3. Schematic view of the elevator.
least 6 significant bits of the data bus. These chips need 3 separate control signals to tell them when to take the data off the bus. The OPEN LED is connected to D5 of the first 74174, and the RUN LED is connected to D5 of the second 74174. One of the inputs of the third 74174 is left free (D5). Since the outputs are only effected on the rising edge of the clock, the enable signal can be directly connected to the respective clock inputs and the clear pin is always set to logic one (5 volts.). The clear can be achieved by the software by loading the flip flop with all zeros. Whenever the enable signal (connected to the clock) goes high, the LED's are lit according to the data on the data bus.

In the input side all the R inputs of the latches are connected to the clear control signal. This is done to clear the status of the switches once they are read. Altogether there are 15 latches. One clear control signal cannot supply the required current to clear all the 15 switches. Hence we have two clear signals. One for the first 8 latches and the other for the other 7 latches. Once the clear signal, which is connected to the R, goes low the output is cleared. The switches are connected to the S inputs of the latches. One end of the switch is connected to the ground and the other to the 5 volt supply through pull up resistors. Whenever a switch is closed, the inputs are grounded and the output goes to logic one. There is no need to complement our switch data. The presence of a logic
one on the data bus during a switch read indicates a switch closure. We need another control signal to enable the tristate buffers of the multiplexers. The number of I/O lines to the hardware elevator is 16. They are 8 data lines, 7 control lines and one ground. The seven control lines are divided as follows: 3 for selecting the flip flops, 2 for clearing the latches, 1 for enabling the multiplexer and 1 for selecting the multiplexer. The actual circuit connections are shown in figure 4.

It was found that the complete Tiny BASIC elevator program took less than 1 K of RAM. The PET BASIC elevator program took 1K of RAM. These programs have actually been tested and found to work perfectly. The elevator is mainly designed for the Heathkit assembly language but it can work off any type of I/O port that has a data bus and can generate at least 1 control line.

For the verification of the PET BASIC programs we shall choose the elevator control program. First we shall write the whole program and then we shall take different segments, and verify them.

The program to run off the IEEE-488 port is

```
1 A7=59427: P=1: B5=255: B3=253: B2=52
2 A6=59456: A5=59425: A4=59424: C6=59426
6 B0=60
10 J=1: GOSUB 500
15 GOSUB 800
20 GOSUB 400
23 IF J=16 THEN GOSUB 500
27 GOSUB 400
```
Figure 4. Block diagram for the elevator.
28 GOSUB 900
30 IF J*2 <= (U OR D OR K) THEN 70
40 GOSUB 500
45 IF J*2 <= K THEN 70
60 GOTO 100
70 J=J*2
90 GOTO 20
100 GOSUB 500
101 J=J/2
102 IF J <= 1 THEN 10
105 GOSUB 500:GOSUB 900
115 IF((J-1) AND (D OR K OR U)) <> 0 THEN 100
120 GOTO 20
400 NJ=NOT J
410 IF(J AND (U OR K)) =0 THEN RETURN
415 U=U AND NJ
425 K=K AND NOT J
430 OP=1:GOSUB 900:OP=0:GOSUB 800
440 RETURN
500 IF(J AND (D OR K)) =0 THEN RETURN
510 D=D AND (NOT J):GOTO 425
650 REM INPUT OUTPUT SUBROUTINE
6610 T=A AND P:B1=B
670 A=A/16
680 IF T<>0 THEN K=K OR 16
690 T=B AND P:B1=B
700 B=B/16
705 U=U OR B
710 IF T<>0 THEN U=U OR 16
720 B1=B1 AND 14:A1=A1 AND 6
730 B1=B1/2:A1=A1*4
740 D=D OR A1 OR B1 AND 31:UD=U OR D
750 IF OP=1 THEN UD=UD+32
752 FL=J
755 IF UD <> 0 OR K<>0 THEN FL=FL+32
756 NF=NOT FL AND 255
757 NU=(NOT UD) AND 255
758 NK=NOT K AND 255
760 RETURN
800 POKE A7,B2
802 POKE A6,B3
806 POKE A5,B2
808 POKE A5,B0
810 A=PEEK(A4)
820 POKE A6,B5
830 B=PEEK(A4)
840 POKE A7,B0
850 POKE A6,B3
855 GOSUB 650
860 POKE A5,B2
870 POKE C6,NU
880 POKE A6,B5
884 POKE C6,NF
888 POKE A7,B2
890 POKE C6,NK
892 POKE A6,B3
900 LET A=10
910 GOSUB 800
920 A=A-1
930 IF A<>0 THEN 910
940 RETURN

Now we shall verify the timing of the program segments of the IEEE-488 program.

900 A=2
910 REM
920 C=C-1
930 IF C<>0 THEN 910
940 END

Time for statement 900 is $1.8 + 21 \times 0.033 = 2.49$ milliseconds (equation 1(a) of table 1). The total number of variables defined in the program is 21.

Time for statement 910 is 0.3 milliseconds (equation (6) of table 1).

Time for statement 920 is $2.6 + 1 \times 21 = 4.7$ milliseconds (equation (3) of table 1). 21 is the number of variables defined before the current variable.

The time for statement 930 can be split into 3 parts. (1) The time for a IF THEN statement when the condition is satisfied is 2.4 milliseconds. (equation 9(a) in table 1), (2) the time in milliseconds for A<>0 is 0.5 (because the operator is not =) + 21*0.066 (21 is the NVB) and (3) time in milliseconds for goto 910 is 0.7 (time for a goto statement) + 68*0.072 (68 is the number of lines before the gosub destination statement (910). (equation 7 table 1)

The total time for statement 930 is 9.946 milliseconds. Taken from equation 9(b) of table 1.
Time for 930 when the condition is not satisfied is
1.7+0.1*21+0.5 =4.3 milliseconds (equation 9(a) of table 1).

Statements 910 and 920 are repeated once. Hence the
total time is 2.49+0.3+4.7+9.946+0.3+4.7+4.3=26.73
milliseconds. The time observed is 26.3 milliseconds. the
percentage error is less than 2%. On PET BASIC we can
predict the timing with a high degree of accuracy.

IEEE-488 input/output program.

800 POKE A7,B2  Time =2.3+2*.066=2.432 milliseconds.
802 POKE A6,B3  Time =2.3+4*.066=2.564 milliseconds.
806 POKE A5,B2  Time =2.3+5*.066=2.63 milliseconds.
808 POKE A5,B0  Time =2.3+8*.066=2.828 milliseconds.
810 A=PEEK(A4)  Time =2.36+8*.066+14*.033=3.35
milliseconds.
820 POKE A6,B5  Time =2.3+4*.066=2.564 milliseconds.
830 B=PEEK(A4)  Time =2.36+8*.066+14*.033=3.35
milliseconds.
840 POKE A7,B0  Time =2.3+4*.066=2.564 milliseconds.
850 POKE A6,B3  Time =2.3+4*.066=2.564 milliseconds.
855 REM               Time =0.3  milliseconds.
860 POKE A5,B2  Time =2.3+5*.066=2.63 milliseconds.
870 POKE C6,NU Time =2.3+14*.066=3.22 milliseconds.
880 POKE A6,B5  Time =2.3+4*.066=2.564 milliseconds.
884 POKE C6,OF Time =2.3+14*.066=3.22 milliseconds.
888 POKE A7,B2  Time =2.3+2*.066=2.43 milliseconds.
890 POKE C6,NK Time =2.3+15*.066=3.29 milliseconds.
895 POKE A6,B3  Time =2.3+4*.066=2.564 milliseconds.
The calculated time for this program segment is 47.38 milliseconds. The time observed is 46.5 milliseconds. The error percentage is less than 2%.

Another IEEE program segment.

This time we shall take a bigger program.

650 REM Time = 0.3 milliseconds.
660 T = A AND P Time = 3.23 + 6 * .1 + 16 * .033 = 4.36 milliseconds.
670 A=A/16 Time = 5 + 12 * 0.1 + 10 * .033 = 6.13 milliseconds.
675 K=K OR A Time = 3.3 + 13 * .1 + 9 * .033 = 4.9 milliseconds.
680 IF T<>0 THEN K=K OR 16 Time = 1.7 + 12 * .1 + 10 * .033 = 3.23 milliseconds.
690 T=B AND P:B1=B Time = 3.23 + 7 * .1 + 15 * .033 + 1.56 + 14 * .066 + 8 * .033 = 7.17 milliseconds.
700 B=B/16 Time = 5 + 13 * .1 + 9 * .033 = 6.6 milliseconds.
710 IF T<>0 THEN U=U OR 16 Time = 1.7 + 12 * .1 + 10 * .033 = 3.23 milliseconds.
720 B1=B1 AND 14:A1=A1 AND 16 Time = 3.23 + 15 * .1 + 7 * .033 + 3.23 + 12 * .1 + 14 * .1 + 10 * .033 = 9.9 milliseconds.
730 B1=B1/2:A1=A1 * 4 Time = 5 + 15 * .1 + 7 * .033 + 3.6 + 12 * .1 + 10 * .033 = 11.86 milliseconds.
740 D=D OR A1 OR B1 AND 31:UD=U OR D Time = 3.3 + 16 * .1 + 8 * .033 + 1.8 + 15 * .033 + 1.8 + 22 * .033 + 3.3 + 16 * .1 + 8 * .033 = 15.15 milliseconds.
750 IF OP=1 THEN UD=UD+32  
    Time = 1.7*16 *.1+8 * .033 = 
    3.56 milliseconds.

752 FL=J  
    Time = 2.3+10 * .1+12 * .033=3.69 
    milliseconds.

755 IF UD<>0 OR K<>0 THEN FL=FL+32  
    Time=3.3 + 0.5 + 0.5 + 
    1.7 + 1.7 + 17 * .1 + 5 * .033 + 12 * .1 + 10 * .033 = 9.56 
    milliseconds.

756 NF=NOT FL AND 255  
    Time=2.4 + 18 * .1 + 4 * .033 + 3.2 
    + 22 * .1 = 9.73 milliseconds.

757 NU=(NOT UD) AND 255  
    Time=2.4 + 17 * .1 + 5 * .033 + 0.4 
    + 3.23 + 22 * .1 = 10.1 milliseconds.

758 NK=NOT K AND 255  
    Time=2.4 + 13 * .1 + 9 * .033 + 3.23 
    + 22 * .1 = 9.43 milliseconds.

The calculated time for this program segment is 120.2 
milliseconds. The actual time observed is 123 
milliseconds. The error percentage is less than 3%.

Program to run off the user port.

This program is the same as the IEEE-488 program 
except for 3 initialization statements and the input and 
output routine.

Input output routine.

1 A8=59468:A9=59459:A1=59471
2 D4=204:D6=236:D5=255:D0=00
6 P=1
800 POKE A8,D4:POKE A9,D0
810 POKE AA8,D6:A=PEEK(A1)
820 POKE A8,D4:POKE A8,D6
825 B=PEEK(A1):POKE A8,D6
830 POKE A8,D4:POKE A8,D6
840 POKE A8,D4:POKE A8,D6
850 GOSUB 650
855 POKE A8,D4:POKE A8,D6
860 POKE A1,NU
865 POKE A8,D4:POKE A8,D6
870 POKE A1,NF
875 POKE A8,D4:POKE A8,D6
880 POKE A1,NK
885 POKE A8,D4:POKE A8,D6
890 POKE A9,D0
895 RETURN

The initialization routine is just one statement.

1000 POKE A8,D4:POKE A8,D6

Now we shall take some program segments of user port programs.

400  NJ=NOT J  Time=2.4+10*.1+12*.033 = 3.796 milliseconds.

410 IF(JAND(U OR K))=0 THEN 415 Time=2.4+.4+.4+3.3+15*.1+8*.033+2.1+12*.033+0.7+25*.073 = 13.29 milliseconds.


430 CP=1:REM:OP=0  Time=1.8+18*.033+0.3+1.8+18*.033 =5.0 milliseconds.

The total time calculated is 33 milliseconds. The observed time is 31.5 milliseconds. the percentage error is less than 5%.

Input output program for the user port.

800 POKE A8,D4:POKE A9,D0  Time=2.3+2*.066+2.3+4*.066 =4.996 milliseconds.
810 POKE A8,D6:A=PEEK(A1)  Time=2.3 + 2 * .066 + 2.36+1 * .066+21*.033 =5.51 milliseconds.
820 POKE A8,D4:POKE A8,D6  Time=2.3+2*.066+2.3+2*.066 =4.864 milliseconds.
830 POKE A8,D4:POKE A8,D6  Time 2.3+2*.066+2.3+2*.066 =4.864 milliseconds.
840 POKE A8,D4:POKE A8,D6  Time=2.3+2*.066+2.3+2*.066 =4.864 milliseconds.
850 REM  Time = 0.3 milliseconds.
855 POKE A8,D4:A9,D6  Time=2.3+2*.066+2.3+2*.066 =4.864 milliseconds.
860 POKE A1,NU  Time=2.3+11*.066=3.02 milliseconds.
865 POKE A8,D4:POKE A8,D6  Time=2.3+2*.066+2.3+2*.066 =4.864 milliseconds.
870 POKE A1,NF  Time=2.3+11*.066=3.02 milliseconds.
Timing verification of Tiny BASIC programs

For verification of the Tiny BASIC program we shall choose the elevator control program and try to time different program segments.

Elevator control program in Tiny BASIC.

```
2 LET U=0
4 LET K=0
6 LET D=0
8 LET T=0
10 LET J=1
15 REM
20 GOSUB 400
23 IF J = 16 THEN GOSUB 500
```
25 GOSUB 900
27 GOSUB 400
30 IF U>J THEN GOTO 70
40 IF K>J THEN GOTO 70
50 IF D>J THEN GOTO 70
60 GOTO 100
70 LET J=J*2
90 GOTO 20
100 GOSUB 500
101 LET J=J/2
102 IF J<=1 THEN GOTO 10
105 GOSUB 500
110 GOSUB 900
130 IF ((D*2048)/J)*32 <> 0 THEN GOTO 100
140 IF ((K*2048)/J)*32 <> 0 THEN GOTO 100
150 IF ((U*2048)/J)*32 <> 0 THEN GOTO 100
160 GOTO 20
400 REM
420 IF (U/J-U/(2*J)*2)=0 THEN GOTO 475
430 LET U=U-J
440 IF (K/J-K/(2*J)*2)<0 THEN K=K-J
450 LET T=1
460 GOSUB 900
465 LET T=0
470 RETURN
475 IF(K/J-K/(2*J)*2)<>0 THEN GOTO 440
480 RETURN
500 REM
510 IF (D/J-D/(J*2)*2) = 0 THEN GOTO 475
520 LET D=D-J
530 GOTO 440
900 LET A=18
910 LET A=A-1
920 IF A<>0 THEN GOTO 910
930 RETURN

We shall take a program segment from this program and calculate the time.

900 LET A=2
910 LET A=A-1
920 IF A<>0 THEN GOTO 910

We have to put the start timing function before the statement 900 and the stop timing function after statement 920. The time observed on the timer is equal to the time taken by the 3 statements (900, 910 and 920.)
The time taken by LET A=2 is 6.7 milliseconds. This value is computed from equation (1) of table 2.

The time taken by LET A=A-1 is 10.52 milliseconds. This value is computed from equation (3) of table 2.

The time taken by if A <> 0 THEN GOTO 910 is T = 20.9 + 1.3 + 580 * 0.038 + 42 * .15 = 50.54 milliseconds. The time is calculated by equation (8 (bc)) of table 2. The additional 1.3 milliseconds is added because the operator of IF THEN statement is different from '=' ('<>').

The number of characters before label 910 is 580. The number of lines before label 910 is 42. These include the two statements LET A=USR(7188,53248) and LET A=USR(7188,53249) to stop and to clear and start the counter.

Time taken by statement 920 when the condition is not satisfied is 15.18 milliseconds. The time taken to start and stop the counter is 19.67 milliseconds. Hence the calculated time must be = 6.7 + 10.52 + 50.54 + 10.52 + 15.18 + 19.67=113.13 milliseconds. The time observed on the hardware timer is 113.01 milliseconds. The difference of time is only 0.12 milli secs which is nearly equal to 0.1% error.

When statement 900 LET A=2 is changed to 900 LET A=4 then the time calculated is 113.13 + (50.54 + 10.51)*2=235.23 milliseconds. and the observed time on the timer is 235.08 milliseconds. The error percentage is less than 0.1%.
When the statement 900 is changed to LET A=6 then the time calculated is \(235.23 + (50.54 + 10.51) \times 2 = 357.33\) milliseconds. Again the error percentage is less than 0.1%.

Before the experiment was performed the percentage error was expected to be somewhere around 10%. But surprisingly the experimental results have shown that time can be predicted with a very high degree of accuracy.

Timing of another segment

```
20 GOSUB 400
400 REM
420 IF(U/J-U/(2*J)*2)=0 THEN GOTO 475
475 IF(K/J-K/(2*J)*2)<0 THEN GOTO 440
480 RETURN
```

The conditions in 420 and 475 are not satisfied.

```
20 GOSUB 400 \hspace{1cm} \text{Time} = 11.478 + 9 \times 0.03 + 57 \times 0.076 \hspace{1cm} \text{(there are 9 lines and 97 characters in them and they all are before statement 20 (this includes the statement 20)} + \hspace{1cm} \text{20} \times 0.15 + 234 \times 0.038 \hspace{1cm} \text{(there are 20 lines and 234 characters in them and they are after the GOSUB 20 statement and before the destination statement 400 REM.)} = 33.132 \text{ milliseconds.}
```

```
400 REM \hspace{1cm} \text{Time taken T=4.71 milliseconds.}
```

```
420 IF(U/J-U/(J*2)*2)=0 THEN GOTO 475 \hspace{1cm} \text{Time} = 20.9 + \hspace{1cm} 5.2 + 5.2 \text{(parentheses)} + \hspace{1cm} 4.11 + 4.11 \text{(division)} + \hspace{1cm} 3 + \hspace{1cm} 3 \text{(multiplication)} + 3.98 \text{(subtraction)} + \hspace{1cm} 35 \times 0.15 \text{(number of lines before destination address)} + \hspace{1cm} 450 \times 0.038 \text{(number of characters in these lines)} = 71.76 \text{ milliseconds.}
```
475 IF (K/J-K/(2*J)*2) <> 0 THEN GOTO 440 Time=15.2 + 5.2 + 5.2 + 4.11 + 4.11 + 3 + 3 + 3.9=43.72 milliseconds.

Time to start and to stop the counter is 19.67 milliseconds.

The calculated time is 31.13 + 71.76 + 43.72 + 19.67=172.99 milliseconds. The observed time on the timer is 169.87 milliseconds. The percentage error is less than 2%.

Another program segment

16 LET A=USR(7188,53248)
23 IF J=16 THEN GOSUB 500
500 LET A=USR(7188,53249)
505 REM
510 IF (D/J-D/(2*J)*2) = 0 THEN GOTO 475
475 IF (K/J-K/(2*J)*2) <> 0 THEN GOTO 440
476 LET A=USR(7188,53250)

Time for 505 REM T=4.7 milliseconds.

Time for 510 IF (D/J-D/(2*J)*2) = 0 THEN GOTO 475 T=20.9 + 5.2 + 5.2 + 4.11 + 4.11 + 3 + 3 + 3.9 + 34*0.15 + 410*0.038 = 70.08 milliseconds.

Time for 475 IF (K/J-K/(2*J)*2) <> 0 THEN GOTO 440 T=15.2 + 5.2 + 5.2 + 4.11 + 4.11 + 3 + 3 + 3.9=43.72 milliseconds.

Time to start and stop the counter is T=19.67 milliseconds.

The calculated time is 70.84 + 43.72 + 4.71 + 19.67=138.94 milliseconds. The observed time is 139.08 milliseconds. The percentage error is well below 1%.

Another program segment

Now we shall take a big segment and compute the time.

16 LET A=USR(7188,53248)
17 LET A=USR(7188,53249)
20 GOSUB 400
23 IF J=16 THEN GOSUB 500
25 GOSUB 900
29 GOSUB 400
30 IF U>J THEN GOTO 70
40 IF K>J THEN GOTO 70
50 IF D>J THEN GOTO 70
55 LET A=USR(71988,53250)
57 END

Time for statement 20 which we have already calculated is 151.27 milliseconds.

Time for statement 23 is 13.9 milliseconds.

Time for statement 25 is 11.47 + 125*0.76 + 440*0.038 + 11*.30 + 32 *1.15 + the time for the timing loop which we have calculated (113.09)= 158.88 milliseconds.

Time for statement 29 is about the same as for statement 20 but a little extra time because the number of statements before the statement 29 GOSUB 400 is more than before statement 20 GOSUB 400. The extra time is 3*0.15 + 40*0.038=1.97 milliseconds. Hence the time for 29 GOSUB 400 is 151.27 + 1.97=153.24 milliseconds.

The time for statements 30,40,50 is the same because the conditions are not satisfied. Their total time is 15.2*3=45.6 milliseconds.

The time to start and stop the counter is 19.67 milliseconds.

Hence the total time is 151.27 + 13.9 + 158.88 + 153.24 + 45.6 + 19.67 = 542.56 milliseconds. The observed time is 540.39 milliseconds. The percentage error is less than 0.5%.

From these timing calculations and comparisons we see
that the Tiny BASIC timing can be predicted with a very high degree of accuracy.
CHAPTER 5

The Hardware and the Software Programs for the Model Elevator

Hardware: Before we go into the details of the software programs which control the model elevator, we shall study about the individual chips in greater detail.

74LS257: They are a tristate (5) quad 2 data selectors / multiplexers. They have a tristate outputs that can interface directly with data lines of bus organized systems with all but one of the common outputs disabled. The low impedance of the single enabled output will drive the bus line to a high or low logic level. To minimize the possibility that two outputs will attempt to take a common bus to opposite logic levels, the output enable circuitry is designed such that the output disable times are shorter than the output enable times. Depending on the select pin low or high, the input data lines are selected. Whenever the output control pin goes low the data is available on the data bus.

74LS279: They are quad S-R latches. These latches are ideally suited for use as temporary storage of binary information between processing units and I/O units. When either one of the data inputs is at a low logic level the output will follow the level of the R input. When both the inputs are high the output is latched in its previous state.
Two of the 4 latches have an enable signals on them. In our circuit they are tied to logic one (5 volts).

74174: These are clocked hex D-Flip flops with a common clear. These positive edge triggered flip flops utilize TTL circuitry to implement D-type flip flop logic. All have a direct clear input. Information at the D inputs meeting the setup time requirements is transferred to the Q outputs on the rising edge of the clock pulse. Clock triggering occurs at a particular voltage level and is not directly related to the transition time of the positive going pulse. When the clock inputs is either the high or low level, the D input signal has no effect at the outputs.

The model elevator, which was mentioned before, consists of 5 floors. The total number of switches on this elevator is 15, out of which five are for UP switches, five for DOWN switches and five for CAR switches. The total number of LED's (which indicate the status of the elevator) are 17, out of which five are for UP and DOWN switches, five for the CAR switches, five for the current floor indicators, one to indicate the opening and closing of the doors, and one to indicate whether the elevator has completed all the requests or not. We shall call the last two LED's OPEN and RUN LED's respectively.

The control program of the elevator is so written that when all the requests are satisfied the elevator will come to the first floor. If the elevator is going up then it will satisfy all the up requests first. Once the elevator
starts coming down will it satisfy all the down requests and only then, it will go up provided there is such a request. The current position of the car can be seen by the middle row of the LED's.

Programs to control the elevator.

On the Heathkit we control the elevator in Tiny BASIC. In order to do this we have to generate 7 control signals and make many logic decisions. Tiny basic is relatively weak in I/O type of operations. We shall use a small machine language subroutine. All the logic decisions are done by Tiny BASIC. First we shall write the whole program and then, explain its logic by taking small program segments. Couple of remark statements are used in the control program. This is done for the verification of the experimental work. They must be avoided while running the control program. The circuit diagram is given in figure 4.

2 LET U=0
4 LET D=0
6 LET K=0
8 LET T=0
10 LET J=1
15 LET B=USR(1794)
20 GOSUB 400
23 IF J=16 THEN GOSUB 500
25 GOSUB 900
27 GOSUB 400
30 IF U>J THEN GOTO 70
40 IF K>J THEN GOTO 70
50 IF D>J THEN GOTO 70
60 GOTO 100
70 LET J=J*2
90 GOTO 20
100 GOSUB 500
101 LET J=J/2
102 IF J<=1 THEN GOTO 10
105 GOSUB 500
110 GOSUB 900
130 IF D-D/J*J <> 0 THEN GOTO 100
140 IF K-K/J*J <> 0 THEN GOTO 100
150 IF U-U/J*J <> 0 THEN GOTO 100
160 GOTO 20
400 REM
420 IF(U/J-J/(2*J) * 2) = 0 THEN GOTO 475
430 LET U=U-J
440 IF(K/J-K/(J*2) * 2) <> 0 THEN K=K-J
450 LET T=1
460 GOSUB 900
465 LET T=0
470 RETURN
475 IF (K/J-K/(J*2) * 2) <> 0 THEN 400
480 RETURN
500 REM
510 IF(D/J-D/(2*J) * 2) <> 0 THEN 475
520 LET D=D-J
530 GOTO 440
900 LET A=50
910 LET A=USR(1794)
920 LET A=A-1
930 IF A <> 0 THEN GOTO 910
940 RETURN

This is the machine language subroutine for I/O operation.

ORG $700
J EQU $95
K EQU $97
O EQU $9F
R EQU $A5
T EQU $AB
U EQU $AB
D EQU $89
TEMP RMB 2
LDAA $D000
LDAB $D010
STAA $D001
STAB $D002
STAA TEMP
STAB TEMP+1
LSRA
BCC B1
ORAA #$80
B1 LSRA
LSRA
LSRA
ORAA U
STAA U
LSRB
BCC B2
The switch data read from the bus is in the following form:

Accumulator A = U4:U3:U2:U1:D3:D2:D1:U5 where U5 is the least significant bit and U4 is the most significant bit. U stands for the UP switches and D stands for the DOWN switches.

Accumulator B = K4:K3:K2:K1:XX:D5:D4:K5 where K5 is the
least significant bit, XX is a don't care and k4 is the most significant bit. K stands for the CAR switches. In the machine language subroutine we identify these switches. A logic one (a binary 1) implies a switch closure. Once the switches are identified, the information is OR'ed with the old switch information. A logic 1 in the memory location \( J \) will give the current position of the elevator. If a switch request is satisfied then we erase that particular bit from the memory in which the status of that switch is stored (U for UP, D for DOWN, and K for CAR). This is done by Tiny BASIC logic.

In the Tiny BASIC program we initialize all memory locations which store switch information to zero and we set \( J \) (current floor) to 1, indicating, that the elevator is on the first floor. Then we enter the machine language subroutine to read the switches. This subroutine also sends the data to the LED's.

In the main program there are two main subroutines which start at 400 and 500. These subroutines are used to test for requests on the current floor for opening and closing of the doors of the elevator. If such a request exists then the subroutines 400 and 500 open and close the doors and erase the switch status bit from the memory location, in which the switch status is stored.

One subroutine is used by the main program when the elevator is going up and another when the elevator is coming down. Statement 23 tests whether we have reached the
highest floor (fifth). Subroutine 900 is a time delay loop which just scans the input switches and displays the corresponding LED's. First the elevator tests for requests on the current floor for UP and CAR. If so, then the doors open and close. Then the elevator checks for UP and CAR switches above the floor and if such a request exists then the elevator goes up.

Otherwise the elevator comes down. Now it checks for DOWN and CAR switches. If there are any requests for the current floor then the doors open and close. If there are any DOWN or CAR switches below the current floor then the elevator will go down. Multiplying J by 2 will move the 1 bit in J to left, indicating that the elevator has reached the next upper floor. Dividing the J by 2 will move the 1 bit in J to right, indicating that the elevator has reached the next lower floor.

It is very easy to test for requests above the current floor. If $J*2 \leq U$ then there is such a up request. If there is a request on the current floor then the corresponding subroutine (400 or 500) will take care of it. If there is an up request above the current floor then the value of $U$ will be at least equal to two times the value of $J$. Hence if $J*2 \leq U$, then there is an up request above the current floor. It is also very easy to test for floors below the current floor. If $D-D/J*J<0$ then there is a floor request below the current floor. We should remember that variables are nothing but two bytes of memory.
Divisions are first executed and then the multiplications. When D/J is executed, all the bits to the right of the current floor are shifted out. When D/J*J takes place, zeros are shifted into the variable. If there were only zeros (that is no requests below the current floor) then D-D/J*J will be equal to zero, otherwise there is a request below the current floor.

If U/J-U/(2*J)*2 <> 0 then there is a request on the current floor. To erase the corresponding bit we use the statement U=U-J. U/J shifts out all the bits to the right of the current floor. U/(J*2)*2 will shift all the bits to the right of the current floor including the current floor and it shifts zero in the position of the current floor. If these two expressions are not equal then there is a request for the current floor. If we are in the UP mode and there is a request for the CAR on the current floor then we must also check also the UP request on the current floor. If any such request exists then the doors open and close and the corresponding bit or bits are erased.

The Generation of the Control Signals: The control signals are obtained by decoding the address bus. Heathkit does not have any memory from hexadecimal address DO00 to DFFF. Hence the address lines are decoded to give the control signals. They are D000, D001, D010, D002, D003, D004 and D005. The circuit diagram is given in figure 5. We use a couple of 4 input nand gates and a 3 to 8 decoder. D000 line is used to enable the tristate multiplexer to get the
Figure 5. Heathkit interface with the elevator.
first 8 inputs. This signal also enables the tristate buffers. D010 enables the other 8 inputs. Address line A3 is connected to the data select pin of the tristate buffer. Lines D001, D002 are connected to the R (clear) of the latches. Control lines D003, D004 and D005 are connected to the clocks of the 74174 hex D-flip flops. For example, if a STAA D005 instruction is executed and then the data in the accumulator is latched to the last hex D-flip flop while the other hex D-flip flops are not effected.

The Elevator Off the IEEE-488 port: Since PET BASIC is faster and also more flexible than Tiny BASIC, we shall write the I/O instructions in BASIC. First we shall write the whole program and understand its logic.

```
1 A7=59427; P=1; B5=255; B3=253; B2=52
2 A6=59456; A5=59425; A4=59424; C6=59426
6 B9=60
10 J=1; GOSUB 500
15 GOSUB 800
20 GOSUB 400
23 IF J=16 THEN GOSUB 500
27 GOSUB 400
28 GOSUB 900
30 IF J*2 <= (U OR D OR K) THEN 70
40 GOSUB 500
45 IF J*2 <= K THEN 70
60 GOTO 100
70 J=J*2
90 GOTO 20
100 GOSUB 500
101 J=J/2
102 IF J <= 1 THEN 10
105 GOSUB 500; GOSUB 900
115 IF ((J-1) AND (D OR K OR U)) <> 0 THEN 100
120 GOTO 20
400 NJ=NOT J
410 IF (J AND (U OR K)) =0 THEN RETURN
415 U=U AND NJ
425 K=K AND NOT J
430 OP=1; GOSUB 900; OP=0; GOSUB 800
```
Since PET BASIC has more instructions than Tiny BASIC, the program is comparatively smaller. First we equate the constants to variables, because we have seen before that the variables save a lot of time. Then we initialize the current floor (J) to 1. Variables U, D and K contain the
switch information for UP, DOWN and CAR requests respectively.

As in the previous case, subroutine 400 checks for a current floor request in the up mode and subroutine 500 checks for a current floor request in the down mode.

If \( J*2 \leq U \text{ OR } D \text{ OR } K \) then there is a request from floors above the current floor. If there is a request above the current floor then \( U \text{ OR } D \text{ OR } K \) will be at least equal to twice the value of \( J \). If a person pushes a down request, say on the fourth floor and changes his mind and when he gets in to the elevator and wants to go up, then he can do so without any problem. This is taken care by statement number 45. The variable \( J \) has a 1 in its current floor bit position. If the elevator is going up then the 1 bit is shifted to the left and if it is going down the 1 bit is shifted to the right.

To check for floor requests below the current floor only one instruction is required. If \( ((J-1) \text{ AND } D \text{ OR } U \text{ OR } K) \neq 0 \) then there is such a request. Since \( J \) has only one bit equal to 1 all the time, \( J-1 \), will have all 1's to the right of the current floor bit and the rest will be zeros. For example if \( J=16 \) then \( J-1=15 \) \( (00001111) \). If we AND this value with \( D \text{ OR } K \text{ OR } U \) then we are checking for floor requests below the current floor. If the result of this expression is not zero then there must be floor requests below the current floor. To check for the current floor requests subroutines 400 and 500 have simple statements.
If \((J \text{ AND } (U \text{ OR } K)) \neq 0\) then there is request on the current floor. Variables \(A \) and \(B \) are read off the IEEE-488 bus. In PET BASIC we decode this information and update \(U, D \) and \(K \).

The switches information is read into two variables \(A \) and \(B \) where 
\[
A = U_4:U_3:U_2:U_1:D_3:D_2:D_1:U_5;
\]
\[
B = K_4:K_3:K_2:K_1:X:D_5:D_4:K_5 \quad (X \text{ is a don't care})
\]
The statement
\[
T = A \text{ AND } P \quad (P = 1)
\]
tests for \(5\). \(A\) is now shifted 4 times to the right \((A = A/16)\) and OR'ed with \(D_5\) if \(D_5\) was one. Similarly \(D\) is calculated using variable \(B\). To calculate \(D\) we have to first AND \(A\) with \(6(00000110)\) and \(B\) with \(14(00001110)\). This is done in statement 720. Now we shift \(A\) to the left by 2 bits and \(B\) to the right by one bit and then OR them together. This result is agained OR'ed with \(D\).

OPEN and RUN LED's are connected with the UPDOWN and CAR LEDs respectively. We check for these two conditions and change the corresponding \(D_5\) bit to logic one depending on the status of the OPEN and RUN conditions. Before the switch status is latched to the LEDS they have to be negated. Once these values are negated it is advisable to AND them with 255 to get rid of any non zero bits that might come in during the negation process.

Interface Between the IEEE-488 bus and the Elevator: Since the elevator is not designed for the IEEE-488 bus, we need an interface between the elevator and the IEEE-488 bus. We shall make use of 3 control lines of the IEEE-488 (6) bus. They are \(DAV\) (data valid), \(NRFD\) (not ready for data),
NDAC (data not accepted). As we have seen previously that these lines can be controlled by software. Data can be received and sent to the IEEE-488 bus by the PET through PEEKS and POKEs. There are certain memory locations reserved for addressing the IEEE-488 bus. Since we need a total of 7 control lines to control the elevator we can get them by decoding these 3 control lines by using a 3 to 8 lines decoder. Only at one instant we need two control lines to be active low. This problem can be easily overcome by using an additional AND gate. The '0' output line of the decoder is left floating. The decoder is enabled all the time.

Another important point to be noted here is that we cannot get the control signals in a sequence (from '0' to '7'). This is because we can only control one input (DAV or NFRD or NDAC) at a time. As an example consider input to be 3(011), and the next input is 4(100). In order to achieve this we have to change all the 3 control lines at the same time which is not possible. So the solution is, to change one line at a time and still get all the seven different control lines for seven changes of the 3 control lines, changing one control line at a time. We shall put a small table to illustrate the point.

<table>
<thead>
<tr>
<th>DAV</th>
<th>NFRD</th>
<th>NDAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Poke A7,B0  Set DAV=1  
Poke A7,B2  Set DAV=0  
Poke A6,B5  Set NFRD=1  
Poke A6,B3  Set NFRD=0  
Poke A5,B0  Set NDAC=1  
Poke A5,B2  Set NDAC=0

The IEEE-488 bus has internal latches built into it. Hence the logic levels on the bus remain unchanged until they are changed by the user. Initially we bring all the 3 control signals to zero. Now we make NDAC equal to 1. The first eight bits of the switch information is read. Next we make NFRD = 1 and the second input is read. Now we make DAV=1 and clear the first 8 latches. Next we make NFRD=0 and the other 7 latches are cleared. Next we bring the NDAC=0 and put the data for the first set of LED's on the bus. Next we bring NFRD=1. The previous data is latched to the first hex D flip flops. The data for the second set of LED's is put on the bus. Now we make DAV=0 and the data is latched to the second set of hex D flip flops. The data for
The third set of LED's is put on the bus. Now we make NFRD=0 and the data is latched to the third set of hex D flip flops. Before we latch the LED information to the D flip flops we go to the subroutine where the input switch information is put in the proper format so that it can be properly displayed on the LED'S. The hardware diagram for the I/O interface with the elevator is given in figure 6.

74155: This is a TTL circuit which feature dual 1 line to 4 line demultiplexers with individual strobes and a common binary address with individual strobes and a common binary address inputs in a single 16 pin package. The inverter following the C1 data input permits us to use it as a 3 to 8 line decoder. Data strobes C1 and C2 are tied together and are connected to the DAV line. Data strobe is tied to logic zero so as to enable the chip all the time. In the decoder only one input is low at a time. But when we are reading the second input of the switch information, both the control lines 1 and 6 must be low (mux select and mux enable). Control line 1 and 6 are given to a two input AND gate and the output, to the control line 1 of the I/O cable. When the control line 1 is low then the I/O control line 1 is also low. When control line 6 is low then also the I/O control line 1 is low.

User port: Interfacing the user port is more complicated than the IEEE-488 bus. Here we have only one control line, CB2. The total number of control signals to be
Figure 6. IEEE-488 interface with the elevator.
generated are the same (7). The IEEE-488 bus has separate memory locations for reading and writing on the data bus. The user port has only address. The data can be programmed for input or output by the user.

POKE A8,D4  Set the CB2=0
POKE A8,D6  Set the CB2=1

The program to control the elevator off the user port is:

```
1 A8=59468:A9=59459:A1=59471
2 D4=204:D6=236:D5=255:D0=00
6 P=1
10 J=1;GOSUB 500
15 GOSUB 800
20 GOSUB 400
23 IF J=16 THEN GOSUB 500
27 GOSUB 400
28 GOSUB 900
30 IF J*2 <= (U OR D OR K) THEN 70
40 GOSUB 500
45 IF J*2 <= K THEN 70
60 GOTO 100
70 J=J*2
90 GOTO 20
100 GOSUB 500
101 J=J/2
102 IF J <= 1 THEN 10
105 GOSUB 500:GOSUB 900
115 IF ((J-1)AND(D OR K OR U))<>0 THEN 100
120 GOTO 20
400 NJ=NOT J
410 IF (J AND (U OR K)) =0 THEN RETURN
415 U=U AND NJ
425 K=K AND NOT J
430 OP=1:GOSUB 900:OP=0:GOSUB 800
440 RETURN
500 IF (J AND (D OR K))=0 THEN RETURN
510 D=D AND (NOT J):GOTO 425
650 REM INPUT OUTPUT SUBROUTINE
6610 T=A AND P:A1=A
670 A=A/16
680 IF T<>0 THEN K=K OR 16
690 T=B AND P:B1=B
700 B=B/16
705 U=U OR B
```
710 IF T<>0 THEN U=U OR 16
720 B1=B1 AND 14:A1=A1 AND 6
730 B1=B1/2:A1=A1*4
740 D=D OR A1 OR B1 AND 31:UD=U OR D
750 IF OP=1 THEN UD=UD+32
752 PL=J
755 IF UD <> 0 OR K<>0 THEN FL=FL+32
756 NF=NOT FL AND 255
757 NU=(NOT UD) AND 255
758 NK=NOT K AND 255
760 RETURN
800 POKE A8,D4:POKE A9,D0
810 POKE A8,D6:A=PEEK(A1)
820 POKE A8,D4:POKE A8,D6
825 B=PEEK(A1):POKE A8,D6
830 POKE A8,D4:POKE A8,D6
840 POKE A8,D4:POKE A8,D6
850 GOSUB 650
855 POKE A8,D4:POKE A8,D6
860 POKE A1,Nu
865 POKE A8,D4:POKE A8,D6
870 POKE A1,NF
875 POKE A8,D4:POKE A8,D6
880 POKE A1,NK
885 POKE A8,D4:POKE A8,D6
890 POKE A9,D0
895 RETURN
900 LET A=10
910 GOSUB 800
920 A=A-1
930 IF A<>0 THEN 910
940 RETURN
1000 POKE A8,D4:POKE A8,D6

The above statement (1000) is used for the initialization of the counter. The main control logic for the user port is same as that of the IEEE-488 bus. The I/O for the user port is completely different. This is mainly because we have only one control signal to generate the seven signals. The generation of these 7 control signals from one signal (C2B2) can be done by using a binary counter.

Except for one case, we only need one control signal active at a time. Hence if we have a counter that counts the CE2 pulses (which can be controlled by the user) we can
get a binary output from the counter. The least 3 significant bits of this output (counter) can be fed into a 3 to 8 line decoder. For this one case when we need two control signals active at the same time we can use another AND gate and overcome the problem. Like the IEEE-488 bus, the user port also has internal latches built into it.

74LS163: This is a synchronous presettable counter and it also features an internal carry look ahead. It has a 4 bit binary counter. The clear function of the 74LS163 is synchronous and a low level at the clear inputs sets all the four flip flop outputs low regardless of the levels of the enable inputs. Setting the counter to zero at the beginning of the program is a little difficult. One way to solve this problem is to ground the clear pin and set the CB2=0 and then set CB2=1. This clears the counter. This is done only once in the program. This can be done in the main program by using the command RUN 1000. Once this is done, the clear pin is tied to 5 volts or left free.

I/O program: First we make CB2=0 and then make CB2=1. This increments the counter from 0 to 1. The bus is programmed for input and the status of the first switches is read into the variable A. Now we make CB2=0 and then CB2=1. This increments the counter to 2. The decoder enables the second set of switch inputs. The status of the second set of the switches is read into the variable B. The CB2 is made low and then high again. The counter reads 3. The first 8 latches are cleared now. The same is repeated and the second
set of 7 latches are cleared. The user port is now programmed for output. We now jump to a subroutine which reads variables A and B and puts them in a proper format. CB2 is pulled low and then high again. The counter output is now 5. The data for the first set of LED's is put on the bus. The CB2 is pulled low and then high again. The data is latched to the first hex D flip flops. The data for the second set of LED's is put on the data bus. The CB2 is pulled low and then high again. The data is latched to the second set of hex D flip flops. The data for the third set of LED's is put on the data bus. CB2 is pulled low and then high again. This latches the data to the third set of hex D flip flops. The user port is now programmed for the input. As we have seen the interface turned out to be quite simple using a counter. In this case we mostly needed one control line active at one time. The circuit diagram for the user port interface with the elevator is given in figure 7.

But if more control lines are active at the same time then this might not be a good way of overcoming the problem. A better way would be to use a 8 bit latch so that the control data can be stored in to the latch first and the next pulse reads or writes the data into the data bus. We need a toggle flip flop which loads the latch with the control word in one cycle of CB2 and on the next cycle the data is stored or read from the data bus.

IEEE-488 Bus

The standard IEEE-488 connector is not used on the PET.
Figure 7. User port interface with the elevator.
Instead, a standard 12 position, 24 contact edge connector with 0.156 inch spacing between contact centers is provided. The electrical drive capability and line impedance matching is in accordance with IEEE-488 specifications. The pin identification table is given below.

<table>
<thead>
<tr>
<th>PIN</th>
<th>SIGNAL</th>
<th>SIGNAL DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DIO 1</td>
<td>Data input/output line #1</td>
</tr>
<tr>
<td>2</td>
<td>DIO 2</td>
<td>Data input/output line #2</td>
</tr>
<tr>
<td>3</td>
<td>DIO 3</td>
<td>Data input/output line #3</td>
</tr>
<tr>
<td>4</td>
<td>DIO 4</td>
<td>Data input/output line #4</td>
</tr>
<tr>
<td>5</td>
<td>EOI</td>
<td>End or identify</td>
</tr>
<tr>
<td>6</td>
<td>DAV</td>
<td>Data valid</td>
</tr>
<tr>
<td>7</td>
<td>NFRD</td>
<td>Not ready for data</td>
</tr>
<tr>
<td>8</td>
<td>NDAC</td>
<td>Data not accepted</td>
</tr>
<tr>
<td>9</td>
<td>IFC</td>
<td>Interface clear</td>
</tr>
<tr>
<td>10</td>
<td>SRQ</td>
<td>Service request</td>
</tr>
<tr>
<td>11</td>
<td>ATN</td>
<td>Attention</td>
</tr>
<tr>
<td>12</td>
<td>GND</td>
<td>Chassis ground</td>
</tr>
<tr>
<td>A</td>
<td>DIO 5</td>
<td>Data input/output line #5</td>
</tr>
<tr>
<td>B</td>
<td>DIO 6</td>
<td>Data input/output line #6</td>
</tr>
<tr>
<td>C</td>
<td>DIO 7</td>
<td>Data input/output line #7</td>
</tr>
<tr>
<td>D</td>
<td>DIO 8</td>
<td>Data input/output line #8</td>
</tr>
<tr>
<td>E</td>
<td>BNR</td>
<td>Remote enable</td>
</tr>
<tr>
<td>F</td>
<td>GND</td>
<td>DAV ground</td>
</tr>
<tr>
<td>H</td>
<td>GND</td>
<td>NFRD ground</td>
</tr>
<tr>
<td>J</td>
<td>GND</td>
<td>NDAC ground</td>
</tr>
<tr>
<td>K</td>
<td>GND</td>
<td>IFC ground</td>
</tr>
</tbody>
</table>
L    GND   SRQ ground
M    GND   ATN ground
N    GND   Data ground (DIO1-8)

All the control lines on the IEEE-488 port can be controlled by the PET software.

User Port

The lines for this interface are brought out from the PET main logic board to a 12 position, 24 contact edge connector with a 0.156 inch spacing between contact centers. Pins 1 to 12, which are the top line of contacts are primarily intended for the use by the PET service department. The lower line of contacts can be used by the user. 8 of these lower pins are programmed input/output lines. Pin M(CB2) is a special I/O pin of VIA. This pin can be controlled by the user by PET software.

The lines on the bottom side of the user port originate from a Versatile Interface Adapter. The parallel port consists of eight programmable bi-directional I/O lines PA0-7, an input handshake line for eight lines. CA1, which can also be used for other edge-sensitive inputs and a very powerful connection, CB2. This has most of the abilities of CA1, but can also act as the input or output of the VIA shift register. A detailed specification of the VIA is given below.
ADDRESS ADDRESS LOCATION
59456 Output register for I/O port B
59457 Output register for I/O port A with handshaking
59458 I/O port B Data Direction Register
59459 I/O port A Data Direction Register
59460 Read Timer 1 Counter low order byte write to Timer 1 Latch low order byte.
59461 Read Timer 1 Counter high order byte write to Timer 1
59461 Latch high order byte and initiate count.
59462 Access Timer 1 Latch low order byte.
59463 Access Timer 1 Latch high order byte.
59464 Read low order byte of Timer 2 and reset Counter interrupt.
59464 Write to low order byte of Timer 2 but do not reset interrupt.
59465 Access high order byte of Timer 2; Reset counter
59465 Interrupt on write.
59466 Serial I/O Shift register.
59467 Auxiliary Control register.
59468 Peripheral Control register.
59469 Interrupt Flag register (IFR)
59470 Interrupt Enable register.
59471 Output register for I/O port A, without handshaking

Data lines are individually programmed to function for input/output as required. For example POKE 59459,255 will
program the lines for output. POKE 59459,0 will program the lines for input. POKE 59459,240 will program D0 to D3 as input and lines D4 to D7 as output. The programming need only be carried out at the beginning. From then on, POKE 59471 can be used to drive the pins programmed as outputs, and PEEK (59471) will read the inputs.
CHAPTER 6

Summary and Conclusions

In the introductory chapter we have described briefly the functions of the interpretive systems. From experimental results we have determined that the interpretive systems are very slow. This disadvantage is more than compensated by the programing ease associated with the interpreter highlevel languages. The interpretive language is inefficient (in terms of time) for a keyboard scan. We have also seen that the same interpretive language is efficient (in terms of time) in controlling a five floor elevator model. Since the elevator is a modest size control program we can conclude that the interpretive systems can be used for control programs.

To know whether the interpreter works on a particular control program, we have to know the execution times of the statements of the interpreter language. If we can measure the execution times of the interpreter language we can find whether or not it would be efficient for a particular application. The main purpose of the third chapter is to find the empirical formulas to calculate the time. These formulas are derived from experimental results. Once these formulas are derived, their accuracy is determined. The error was around 3%.

We have taken the elevator control program to check the
accuracy of the formulas. Even the I/O programs (PET BASIC) of the elevator were written by the highlevel language. The model elevator was constructed and the programs were tested and were found to run correctly. The elevator is tested by 3 different I/O ports. With the information of the empirical formulas we can take any control program and analyze it for time, and tell whether or not the program will run efficiently (in terms of time) on the interpreter.

We have also seen that time can be saved by using variables instead of constants. Time on PET BASIC can be saved by not using LET in the statements and by defining all the variables at the beginning of the program. Remark statements and parentheses must be avoided, as much as possible from the running program. Avoiding blanks from the program will save time and memory.

As the cost of software increases the interpretive systems become more attractive. With the increase in speed of the microcomputers the interpreter execution times decrease. This makes the interpreter systems more powerful, and control systems by interpreters more feasible.
BIBLIOGRAPHY


The Timer for Heathkit.

This is an hardware timer that can count accurately up to the 999.99 milliseconds, with an accuracy of 10 microseconds. With a little change we can use the timer to count accurately up to the last microsecond. The clock of the Motorola 6800 is used in this timer. This timer is essentially a counter, which needs three control signals. One to clear the counter, one to start the counter, and one to stop the counter. These control signals are decoded by the address bus of the Motorola 6800. The counter counts the number of clock pulses between the start and the stop signals. Its count is displayed on the LEDs mounted on the board. The maximum time the counter can count is 999999 microseconds or nearly one second. Since none of the instructions take that much time, we are safe as far as the maximum time is concerned.

The details of the counter.

The timer can be divided into four parts. (1) the decoding circuit, (2) the control circuit, (3) the counter and (4) the display.

Decoding: In order to generate the control signals we have to do some address decoding. We can do this by using a 74154, 4 line to 16 line decoder. This decoder has two
enabling signals, G1 and G2 both of which are active low. The memory locations from D000 to DFFF are not used by the Heathkit. Hence we can decode DXXX for control signals (where X is a don't care). A four input NAND gate (7420) is used to enable one of the input enables GI. The four inputs of the NAND gate are, one valid memory address, two A15, three A14, and four A12. A12,A14,A15 are the address lines. This means that if and only if the four inputs of the NAND gate are high then only G1 is pulled low. A13 is connected to G2. Hence the decoder will be enabled whenever A15,A14,A12,VMA are high and A13 is low. We can now get a decoder output when addressed within the range D000 to DFFF. The four inputs of the decoder can be A0,A1,A2,A3. In this way if the decoder is addressed form D000 to D00F we can get 16 active low outputs.

The control circuit: The purpose of the control circuit is to enable the clear, the enable, and the disable signals at the correct time. Output D000 can be used to clear the counters. It is connected to the first pin of all the six counters. Since this is an active low clear we can connect them directly. The load on the D000 line is 1.6*6=9.6 which is less than 16 milliamperes, which the decoder can handle. The output D001 is connected to the preset of the JK flipflop. Both clear and preset of the 7476 JK flipflop are asynchronous and active low. The output Q of the flipflop is connected to the enables P of all the 6 counters and enable T of the first counter. The total current drawn is
$7 \times 1.6 = 11.2$ milliamperes which is well within the limit of the decoder (16 milliamperes). The $1 \text{ MHz}$ clock of the system is connected to all the six counters.

The counter: The counter consists of six 74162 decade counters. They have two enabling inputs $P$ and $T$ which are both active high. They have a ripple carryout pin which goes high when the count reaches 9 and goes low after the next clock pulse. The ripple carryout of counter 1 is connected to the $T$ enable of counter 2 and the ripple carryout of counter 2 is connected to the $T$ enable of counter 3 and so on up to the sixth counter. For the first counter both $P$ and $T$ are shorted together. When the first counter counts up to 9, the ripple carryout pin goes high enabling the next counter for one clock period. All the six counters are connected in the same way. That means the sixth counter will change by 1 for every 100,000 clock pulses. The maximum count that we can see at one time is 999999 microseconds. The load inputs of all the counters must always be high to avoid paralleled loading. The output of the counters is connected to the LED's.

The display circuitry: 7447 are designed for active low output to drive common anode LED's directly. The outputs of these drivers can be connected directly to the LED's. But the LED's will soon get heated up and may even get burnt out. To avoid this 140 ohm resistors are connected between the output of the driver and the input of the LED's. When the timer is stopped, that is, when the
counter is disabled the counter will show a steady output to the drivers which in turn produce a stable reading in the LEDs.

The minimum time we can count on this counter is 10 microseconds. But we can count upto 1 microsecond, if we eliminate the first counter. But the maximum count will be reduced by 10 times. This timer can be used on any system which can produce 3 active low control signals.
Figure 8. Block diagram for the timer.
Figure 9. Block diagram for the timer (cont.)