PROGRAMMING TIPS, PEEKS AND POKES FOR THE TANDY PORTABLE COMPUTERS

BY TONY B. ANDERSON

Self-published
by
Tony B. Anderson
Post Office Box 60925
Reno, NV 89506

PROGRAMMING TIPS, PEEKS AND POKES FOR THE TANDY PORTABLE COMPUTERS

TABLE OF CONTENTS

CHAPTER 1

Substring Pointers

This chapter describes a method of extracting the position in a string, of a sub-string, where there may be upper or lower-case versions of the sub-string in the main String. The resulting position pointer is reduced to a specific location pointer that can be used by an ON GOTO or ON GOSUB statement without repeating the line numbers.

While writing a program for a client, which displays Race Track Speed Ratings, I developed the following technique to extract an ON GOTO pointer from a string which listed the abreviations of the target Race Tracks. The technique would be useful anytime one wishes to allow for Upper or Lower Case input, and extract the position in the string.

Note the A$ string in line #100, below. You will see it holds the Track Name abbreviations (2 or 3 characters) in both Upper and Lower Case. SA = Santa Anita; HOL = Hollywood, etc. I added a space to any abreviation that was only two characters, to make each one the same length.

The FOR-NEXT loop starting in line 150 only prints the Upper Case abreviations on the screen.

The input routine in lines 200-220 gets the desired track abreviation, in either upper or lower case.

The INSTR routine in the first half of line 230, finds out if the specified track abreviation is in the A$ string; and sets A to = the position in the string. (Note there is no error-trap here) For example, for HOL, A=7; for dmr, A=16 (etc). Then the number of characters necessary to get to the start of the next group is added, in this case, since we are dealing with three-character groups, we add +2. Thus, for HOL, A=9; and for dmr, A=18.

Variable A is then divided by 3 (the number of characters in a sub-group, and 1 is added to the total. This sets variable A to 2 for "SA" and 3 for "sa". (4 for "HOL", and 5 for "hol") The last part of line 230 reduces this figure to the single integer that points to the position in the string for either Upper or Lower Case letters identifying each track. i.e. Both "DMR" and "dmr" are identified as a "3".

In Line 240, the variable A is used to go to a routine which calculates the track speed records.

```
100 A$="SA sa HOLholDMRdmrAC ac BM bm GG gg POMpomLA la BELbelAP ap GP gp CALcal"
110 PR$="Press ENTER to Continue: "
120 N=11:  ' Set # of Tracks
130 CLS:PRINT@10,"Speed Rating Values":PRINT@50,"for Selected Tracks":PRINT
140 PRINT"The following Tracks are Supported:"
150 FORA=1TON
160 PRINTMID$(A$,A*6-5,3);:IFMID$(A$,A*6-3,1)<>" "THENPRINT" ";
170 NEXT:PRINT:PRINT
180 PRINTPR$;:Q$=INPUT$(1)
190 CLS
200 PRINT@200,"Enter Track and Distance, separated by"
205 PRINT"a comma, or ENTER to quit: ";
210 B$="":D=0:INPUTB$,D:CLS
220 IF B$="" THEN MENU
230 A=(INSTR(A$,B$)+2)/3+1:A=INT(A/2)
240 ONAGOTO 300,301,302,303,304,305,306, 307,308,309,310,303
```

Now, I know, you can also simply use a multiple ON-GOTO statement, using the value of A obtained in the first part of line 230. But that would require double entries in the ON-GOTO statement, like this:

```
240 ON A GOTO 300,300,301,301,302,302,303,303,304,304,305,305.....(etc)
```

But in this case I wanted to eliminate all those double GOTO's, because the program will be expanded, and the single GOTO's will even become cumbersome.

Another way of accomplishing the same objective was sent to me by EIJI MIURA, a CompuServe subscriber:

```
      100 A$="SA HOLDMRAC BM GG POMLA BELAP GP CAL"
      110 PR$="Press ENTER to Continue:"
      120 N=11:  ' Set # of Tracks
      130 CLS:PRINT@10,"Speed Rating Values":PRINT@50,"for Selected Tracks":PRINT
      140 PRINT"The following Tracks are Supported:"
      150 FORA=1TON*3STEP3
      160 PRINTMID$(A$,A,3);:IFMID$(A$,A+2,1)<>" "THENPRINT" ";
      170 NEXT:PRINT:PRINT
      180 PRINTPR$;:Q$=INPUT$(1)
      190 CLS
      200 PRINT@200,"Enter Track and Distance, separated by"
      205 PRINT"a comma, or ENTER to quit: ";
      210 B$="":D=0:INPUTB$,D:CLS
      220 IF B$="" THEN MENU
      225 FORG=1TOLEN(B$):F=ASC(MID$(B$,G,1)):IFF>96ANDF<123THENMID$(B$,G,1)=CHR$(F-32)
      230 NEXT
      240 ON(INSTR(A$,B$)+2)/3GOTO 300,301,302,303,304,305,306,307,308,309,310:GOTO130
```

A$ in line 100 doesn't have lower case entries. I added line 225 which converts lower case input into uppercase. This way, you don't have to spend twice as much memory as necessary in A$, and by making the line 255 into subroutine, you can GOSUB from anywhere in the program where you need lower to upper case conversion which saves even more bytes. Short examples like this one don't show much difference in memory size, but it will show a significant difference in longer programs which have several ON GOTO statements. The GOTO 130 at the end of line 240 will return to the original prompt when the program doesn't find a matching string in A$.

CHAPTER 2

Calculating the Date

This chapter describes a programming technique used to determine the date "n" days ago. It might be useful in financial applications, or in other programs which need to supply the date "n" days ago. Method of disassemling the DATE$ string, and reassembling it with another date is also apparent.

I had occasion to write the following routine which calculates the exact date, "n" days ago. Seems this might be useful in financial applications, and some database programs which look for activity within the last "n" days, or similar applications.

```
35 DATA 12,31,11,30,10,31,9,30,8,31,7,31,6,30,5,31,4,30,3,31,2,28,1,31,0,31
36 DATA -1,30,-2,31,-3,30,-4,31,-5,31,-6,30,-7,31,-8,30,-9,31,-10,28,-11,31
 .
 .
 .
100 N=45
 .
 .
 .
300 PRINT@122,"Today's Date is: "DATE$
301 PRINT"  Is this correct? (Y/N): ";: A$=INPUT$(1):IFA$=CHR$(13)THENA$="Y"
302 PRINTA$:IFINSTR("Yy",A$)THEN305
303 PRINT@120,CHR$(27)"J":PRINT@122, "What is today's date";
304 INPUTA$:IFLEN(A$)=8THENDATE$=A$: GOTO300ELSEBEEP:GOTO303
305 M=VAL(LEFT$(DATE$,2)):D=VAL(MID$(DATE$,4,2)):Y=VAL(MID$(DATE$,7,2))
306 READM1,N1:IFM1<MTHEN307ELSE306
307 A=N-D
308 IFA>0THENA=A-N1:READM1,N1:GOTO308
309 A=ABS(A):M1=M1+1:IFM1<1THENY=Y-1
310 M=((M1+11)MOD12)+1
311 M$=STR$(M):M$=MID$(M$,2):IFLEN(M$)=1THENM$="0"+M$
312 A$=STR$(A):A$=MID$(A$,2):IFLEN(A$)=1THENA$="0"+A$
313 Y$=STR$(Y):Y$=MID$(Y$,2)
314 PRINT@122,"Date"N"days ago was: "M$"/"A$"/"Y$;CHR$(27)"J"
315 PRINT@202,"Press ENTER to continue: ";:A$=INPUT$(1)
```

Line 35-36 DATA statements contain the days in the months. Note counting backwards, 12 (December) = 31; 11 (November) = 30, etc.; down to 1 (January) = 31;

then 0 (December, MOD 12) and -1 (November (MOD 12 -12), etc., down through the 12 month calendar again.

Line 100: N = the value of the number of days prior to the current date. In this example, it is a set value, but you could also get this value by asking for input from the keyboard.

Lines 300-304 verify that the internal M100 calendar has not been accidently changed, and that the date is correct. Otherwise, the internal calendar is corrected.

Line 305 breaks the current date into: M=Month, D=Day, Y=Year.

Line 306 sets the DATA pointer to one less than the current month (M) (last month), and gets the # of days in the (last) month.

Lines 307-308 sets the variable A to = N, less the number of days in the current month, and compares if less than 0. If so, count has exceeded the desired number of days, if not, the next month in the data statement is read, getting the number of days in that month, and is subtracted from A again. This continues, and variable A continues to be tested until it is less than 0.

Line 309 sets the date of the target month by setting variable A (a negative value) to a positive value, which will equal the target date. If the data statement has read into the negative values, then the year counter (Y) is reduced by 1 year.

Lines 310-314 reconstruct the date from variables M, A, and Y; and prints it on the screen. The "MOD 12" business in line 310 gets back to a positive month number if the data has been read into the negative numbers, and keeps the same month number if it is already positive.

Note that this routine does not test for "Leap Years" with 29 days in February. You could add that feature by simply dividing the current year by MOD4, and if there is no remainder, add 1 to the value of A before the day is determined in line 312.

CHAPTER 3

A Neat PRINTUSING Technique

This chapter illustrates another method of combining the PRINT@ and PRINTUSING statements with data strings, which is syntax-correct in Microsoft BASIC in the Model 100/102 and Tandy 200.

This is a bright adaptation of the PRINTUSING command, which is not explained in the M100 manual; it is from a CompuServe user, Jim Moore.

Most of us are familiar with the PRINTUSING statement, but not the version that Jim is using. Beginning programmers are often limited in using or combining PRINT@ and PRINTUSING. Generally, If you want to use a text string on the same line as a formatted number, you would structure your statement as follows:

```
10 PRINT"Gallons of Product Left =";: PRINTUSING"###.##";N
```

Or, as we become more experienced, include the first string in the USING statement as follows:

```
10 PRINTUSING"Gallons of Product Left = ###.##";N
```

And, if one wishes to specify on the screen where the statement is to be printed, the PRINT@ specification can be incorporated as follows:

```
10 PRINT@45,USING"Gallons of Product Left = ###.##";N
```

However, Jim has illuminated a fourth approach that is acceptable as correct syntax in Microsoft BASIC for the Model 100 family:

```
10 A$="(Filename)":N=(computed value)
20 PRINT@124,A$;" Checksum = "; USING"#####,###";N
```

Notice that he has included additional string data between the PRINT@ statement, and the USING statement! (I didn't know you could do that!) This is a unique variation, and one which opens additional programming short-cuts; you can tack your formatted numerical data on the end of a PRINT(string) statement, by just adding ";USING" followed by the USING specification and the variable name. You do not have to use the "PRINT(string);:PRINTUSING" form. You can save a couple of programming bytes every time you use this form.

Another trick is to predefine your USING format in a string, and use it as a variable in this way:

```
10 A$="Gallons of fuel left = ###.##"
20 PRINTUSING A$;N
```

or

```
20 PRINT@45,USING A$;N
```

There are many additional forms you can use, and several options in the format specification statement; see the Model 100 manual, pages 170-172 for further examples.

CHAPTER 4

The LINE Statement

This programming tip is a short tutorial on the use of the LINE statement in BASIC, which can be used to draw lines or simple pictures on the LCD screen of the Tandy portables.

Drawing with the LINE statement:

I've written a couple of BASIC programs using the LINE statement (BARCHT.100, DBPADS.100, ISOBOX.100 and KLUGE.DMO, among others, and others which are not yet published), and would like to offer this short tutorial on use of the LINE statement, since the manual documentation is not as clear as it could be, and it took me a lot of experimenting to find out what it all meant, and how it worked.

First, let me describe what we are working with. If you look closely at the LCD (Liquid Crystal Display) screen, you will see it is made up of lots and lots of little dots. (Adjust the contrast knob on the right side of the computer to get a clearer view, or use a magnifying glass.) Every letter, symbol or graphic character on the screen is made by some of those dots being black (PSET), and some of them being "not black" (PRESET).

The reason I chose to say "not black" instead of white, or clear, lies in the way the LCD screen actually works. Just for a brief overview, based on what I've read, it appears that down at the molecular level, there are millions of little molecules in each "dot". In the normal state, they exist in a totally random pattern, and light waves can pass through them easily, and are reflected off the silver grey coating at the back of the screen. But when a small electrical charge is applied, all the molecules line up in the direction of the current flow, and this blocks the passage of light waves, making the "dot" appear black. It's actually not black, but is preventing the "light" from passing through. (OK, you engineer types, so that's an oversimplification, but it WILL serve to explain what's happening.)

Now, each of those "dots" is called a "pixel", an abreviated form of the words PICture ELement, stolen from the television lexicon. (Somehow, the abreviation PICEL just didn't fit! So it got "adjusted" to PIXEL.)

Anyway, the LINE statement works at the pixel level, and for our purposes, we can consider that it "turns on" the pixel (makes it black), which is actually what's happening.

At the pixel level, you have 240 pixels across the screen, which you can consider as "columns"; and 64 pixels down the screen, which you can consider as "rows". (The 200 has 128 rows of pixels down the screen, but the main body of this discussion will relate to use with the 100/102 - you can make mental changes where necessary.) In computerese, these rows are numbered starting at zero instead of one, so the pixel numbers lie in the range of 0-239 horizontal, and 0-63 vertical. By specifying a horizontal and a vertical position, you can deal with a specific pixel. This is why they are referred to as "addressable"; each one has a specific address, consisting of its position, both horizontally, and vertically. For example, there is only one pixel at the address 120,17; and there is no duplication. That is the only pixel with that address. So, if the computer turns on that pixel, it, and no other, will go "black".

Simple multiplication reveals that there are a total of 15,360 pixels in the LCD screen, and somewhere, in some integrated circuit, somewhere in the "internals" of the computer, are 15,360 transistors that turn their respective pixels on and off.

Here are a couple of points of interest that may be useful later, when you are actually using any of the pixel commands, including LINE, PSET, and PRESET; since there are 8 lines on the LCD screen, made up of 64 rows of pixels, each "line" is 8 pixels high. Since there are 40 columns of characters displayed on the screen, and 240 pixels, each character can be 6 pixels wide. In actual practice, however, there needs to be some little space between the characters, so the characters are actually 5 pixels wide, with a 1 pixel "off" space between them. This gives you a character "map" of 8 x 5 pixels.

The first line on the screen, would consist of pixel rows 0-7; the second line would be pixel rows 8-15, etc. The mid-screen position for the 21st character in a line would equate to pixel column 120 (multiply 20x6; the number of character positions skipped, times the number of pixels in a character width).

Remember, pixel rows 1-64 are numbered from 0-63, and pixel columns 1-240 are numbered from 0-239.

Back to the LINE statement: The general form for the statement is:

```
        LINE(X1,Y1)-(X2,Y2)
```

It may be easiest to understand, if you mentally redefine the paramenters as follows:

```
        LINE (From C,R)-(To C,R)
```

where C = column, and R = row.

If you want a line drawn down the middle of the screen, for example, at pixel column 120 (as figured above), then the proper form of the command would be:

```
        LINE (120,0)-(120,63)
```

(Defined: From column 120, row 0, To column 120, row 63.) It's as simple as that, you specify where the line is to start, and where it is to end, by specifying the pixel addresses. And you can use variables, or computed addresses, such as:

```
        LINE(A,B+4*X)-(C-Y,D/17)
```

OK, now let's get into the next parameter, and get a little more exotic. The LINE statement allows two additional specifications; "Switch" and "Box Fill". They are appended to the basic command with commas, in the form:

```
        LINE(X1,Y1)-(X2,Y2),S,BF
```

The "switch" (variable S) can be any number. An odd value, such as "1" tells the BASIC interpreter to turn the pixels on (make them black), and an even value (0,2,4 etc) says to turn the pixels off, as in erasing any pixels previously "set" or "on". "1" is assumed if no switch is specified.

The "Box Fill" command has two forms, "B" and "BF". The "B" command will use the coordinates specified in the LINE statement to draw a square or rectangular box. The "BF" will draw the identical box, but will set, or turn on, all the pixels within the boundaries of that box, turning the whole box black on the screen. Note that if you use either the B or BF, you must use a switch specification, in almost all cases, it would be "1", to draw and/or fill a box at the specified addresses.

```
Samples:  LINE(20,10)-(100,40),1,B
          LINE(20,10)-(100,40),1,BF
```

The first draws a box, and the second draws a filled box at the specified coordinates. Interestingly, you can make a smaller box inside a larger filled box, or vice-versa, resulting in some interesting displays.

```
Sample:  LINE(20,10)-(100,40),1,BF : LINE(25,20)-(95,30),0,B
```

Now we get to PSET and PRESET: which are operators of the LINE statement. PSET is the simple command to turn on a specified pixel. The form is:

```
        PSET (X,Y)
```

where X and Y specify the pixel address. "PSET" from the descriptor, "Point Set".

PRESET is the complimentary command, turning off the pixel at the specified address, and takes the same form:

```
        PRESET (X,Y)
```

PRESET can be considered to mean "set the point to the condition before being set." In other words, "Turn it off".

You could draw a line with the PSET command if you needed to, by putting it in a FOR-NEXT loop, but generally the LINE statement is simpler. However, I did have to use this technique in the program BARCHT.100, where I wanted a line of dots, rather than a solid line, down the center of the chart. The loop took the following form:

```
525 FOR X = 18 TO 52 STEP 2 : PSET(118,X) : NEXT
```

The "STEP" set every other pixel between 18 and 52.

Drawing Pictures:

Drawing simple pictures, or illustrations on the LCD screen is possible, but takes some experimenting to accomplish, usually in the areas of defining the correct starting and ending address for each line, or pixel set. It is not an easy task, but can be accomplished through trial and error. Making a preliminary drawing on a piece of graph paper helps a lot, in determining what pixels need to be set, and where the lines need to start and stop. Just draw a simulated screen on the graph paper, 64 squares high, and 240 squares wide, and draw away. Then translate the graph paper addresses to screen addresses.

By now, you should have a fair understanding of how the LINE statement works, which can be used in your own programs. Here is a simple "Etch-a-Sketch type program you can play with:

```
0 ' SKETCH.100
1 ' COPYRIGHT 1985 TONY B. ANDERSON
2 ' "ETCH-A-SKETCH" type program, starts by putting a dot in the center of the
3 ' screen; by pushing the arrow keys you make a dot next to the previous
4 ' dot, in the direction of the arrow.
5 '
6 ' Drawing a picture is a slow and laborious task. Be careful where you
7 ' put your next dot, as you can't erase a mistake.
8 '
20 CLS
30 Y=32:X=120
40 PSET(X,Y)
50 A$=INKEY$
60 IFA$=CHR$(28)THENX=X+1:GOTO110
70 IFA$=CHR$(29)THENX=X-1:GOTO110
80 IFA$=CHR$(30)THENY=Y-1:GOTO110
90 IFA$=CHR$(31)THENY=Y+1:GOTO110
100 GOTO50
110 PSET(X,Y)
120 GOTO50
```

CHAPTER 5

Opening One of Several Like Files

This file describes a technique which can be used to open whichever file is resident in RAM from some sub-set of files, which may form part of a larger set. If lists are maintained, named LIST1, LIST2, LIST3 etc., the program can open whichever one is actually there.

I was looking for a way to find out, under program control, if a specific file was resident in RAM. Some dialects of BASIC have a command for that function which can be used in an IF statement, returning a "True" if the file is resident in RAM or on disk, depending on the command specification.

In model 100 Microsoft BASIC, if a file is not resident and you open for output or append, the file will be created. However, if you open for input, and the file is not there, the program bombs, and you get an "FF" error (File not Found). Lacking such a command in Model 100 Microsoft BASIC, I worked out the following technique to check for resident files.

Assume your program normally works on lists, such as mailing lists, and your lists are broken down into sub-lists, based on some important criteria such as previous customers, products purchased, zip codes, etc. And your program wants to open whichever file you have loaded... say for example you want to mail a flyer to all the previous customers, and you have one, universal, label printing program. Your previous customers might be saved in "LIST3", one of several such lists, named LIST1, LIST2, LIST3, LIST4, etc.

This technique, added to your label printing program opens whatever file happens to be resident and available.

```
10 ON ERROR GOTO 100: OPEN"LIST1.DO" FOR INPUT AS 1:GOTO 200
20 ON ERROR GOTO 110: OPEN"LIST2.DO" FOR INPUT AS 1:GOTO 200
30 ON ERROR GOTO 120: OPEN"LIST3.DO" FOR INPUT AS 1:GOTO 200
40 ON ERROR GOTO 130: OPEN"LIST4.DO" FOR INPUT AS 1:GOTO 200
50 PRINT"No File Loaded into Memory":STOP
100 RUN20
110 RUN30
120 RUN40
130 RUN50
200 ON ERROR GOTO : REM Cancels previous error trap
210 REM  Body of Label program starts here with correct file opened.
```

Fred Dobbs, a CompuServe subscriber, sent me another way of doing the same thing, which I include here for your information.

"Here's a slightly neater solution to the problem. This program PEEKs into the file directory (which starts at 63930D) and extracts the name of each active .DO file in turn (the third part, plus of line 100 skips "dead" files and .BA files), loading into S1$. The 4th part of line 110 checks to see if S1$ is "LISTXX" (where "XX" is whatever #s or trailing characters are attached to the resident file. If this check is positive, we return to the main program and open the file. Otherwise, we go through another loop, until either the file has been found, or we've gone through all files in the computer...in which case we get to line 120 and are told, "Ain't no such file!" This demonstration program is, of course, "fancied up" to show off the file-finding feature. Cutting out the bells and whistles, it's a nice, neat little routine. Hope this routine proves useful. - Fred"

```
1 CLS:CLEAR2000:S1$="":MAXFILES=1:GOSUB100:OPENS1$FORINPUTAS1:SOUND1000,64
3 IFNOTEOF(1)THENLINEINPUT#1,A$:PRINTA$:GOTO3ELSECLOSE:END
 .
 .
 .  (additional program lines...)
 .
 .
99 END
100 FORF=0TO18:PTR=F*11+63930:A%=PEEK(PTR):IFA%<>192THENNEXT:GOTO120
110 FORJ=3TO8:S1$=S1$+CHR$(PEEK(PTR+J)):NEXT
115 IFLEFT$(S1$,4)="LIST"THENPRINT@130 ,"Found "S1$".DO":RETURNELSES1$="":NEXT
120 PRINT"Ain't no such file!":END
```

CHAPTER 6

Manipulating DATE Strings

This chapter discusses date input in BASIC programs, error trapping for incorrect input, and building a correct date field from varied input forms. Sorting by date field; ASCII and the Hollerith code; and the "IBM Date Format".

Date Routine:

In many BASIC programs, it is necessary to input a date, in response to a program prompt. Examples would be in bookkeeping programs, dated files, expense or business records, etc. While in many cases, the date could be supplied by the computer itself, in others, manual input is called for.

Generally, date input is not error trapped, and subsequent listing or program operations that use the date string can lead to program crashes or data errors if the date string was not entered in a uniform format which is accepted and understood by the program.

The below routine is one method of error trapping for date input, allowing an almost careless input, and constructing a correctly formed date string from the entry, or from the computer calendar function itself.

Input as varied as 6,6; 7-18-83; 4/5 or 2/5/85 will be correctly formed into a standard 8 character field, 06/06/85; 07/18/83; 04/05/85; and 02/05/85. Incorrect, or unexpected input will be trapped.

```
100 A$="":LINEINPUT"Enter Date: ";A$
110 IFA$=""THENA$=DATE$:GOTO200
120 P=INSTR(A$,","):IFPTHENA$=LEFT$(A$,P-1)+"/"+MID$(A$,P+1):GOTO120
130 P=INSTR(A$,"-"):IFPTHENA$=LEFT$(A$,P-1)+"/"+MID$(A$,P+1):GOTO130
140 IFINSTR(A$,"/")=0THEN190
150 IFMID$(A$,2,1)="/"THENA$="0"+A$
160 IFLEN(A$)<=5THENA$=A$+RIGHT$(DATE$,3)
170 IFMID$(A$,5,1)="/"THENA$=LEFT$(A$,3)+"0"+MID$(A$,4)
180 IFLEN(A$)=8THEN200
190 BEEP:PRINT"Bad date. Try again.":GOTO100
200 PRINTA$
```

Line 100 gets the input date. If there is no input (C/R or ENTER), then line

110 assigns the computer's DATE$ string to A$ and goes on to line 200.

The date can be in almost any form; requiring only two numbers separated by a comma, hyphen or slash, all common date delimiters. (Note that this routine will work equally well with military, Canadian, or European date structures, which generally place the date before the month, in the form: 16-08-85 for 16-AUG-85.)

Lines 120 and 130 check to see if the comma or hyphen has been used in the input, and if so, converts them to slashes.

Line 140 checks to make sure there is a valid delimiter in the string.

Line 150 will place a leading "0" in the date, if needed, changing inputs of 1 through 9 into 01 through 09, conforming with the required 8 character date structure.

In Line 160, if the year has not been input, as in the possible case of month and day only (7/12, for example), the current year date is appended to the string.

Line 170 makes sure day dates between 1 and 9 are converted to 01 through 09.

Line 180 is the final check. If the string does not equal 8 characters, something is wrong, and you go back to try again. This test can be expanded to include tests for correct "/" positions if needed.

```
180 IFLEN(A$)=8ANDMID$(A$,3,1)="/" ANDMID$(A$,6,1)="/"THEN200
```

Line 190 is an error message, and returns for new input.

Line 200, of course, is there so you can see the string that has been built, and normally would lead into the rest of your program.

Sorting by Date String:

ASCII code (American Standard Code for Information Interchange) is a standard based on the Hollerith code, which was developed by Herman Hollerith, for use in the 1890 U.S. census. It was originally used to tally the results by machine, using punched cards, similar to those used in IBM card sorters of the mid 50's and early 60's, commonly called "keypunch" or "IBM cards".

In general, the ASCII code provides an easy method of sorting characters into ascending or descending order, based on the number value assigned to the character, but has some anomalies when sorting multiple character strings, whether right to left, or left to right.

As an result of the popularity of IBM card sorters, a "standard" for sorting cards by date came to be known as the "IBM Date Format". It places the year first, followed by the month and the date, in a six digit field, in the form: "850625" (June 25, 1985). This string form is still used today when it is desirable to sort data by date, or by "date field".

Obviously, if you try to sort the following dates with a normal string sort, working right to left (the common technique),

```
        02/13/84
        06/12/84
        04/02/85
```

the result will be the April date (04), placed between the February (02) and June (06) dates. This is the result of sorting each column of characters from right to left in the field (least significant digit first). If you sort left to right, the April date will be last, but the February date will come after the June date. (13 comes after 12.) The proper way to construct these strings for sorting is to put the year first:

```
        840213    (or 84/02/13)
        840612    (or 84/06/12)
        850402    (or 85/04/02)
```

This always results in a correctly ordered sort where mixed years are involved. The proper sort string can be constructed with the following statement (assume the date is in the A$ string constructed above, in the form "06/12/85"):

```
210 A$=RIGHT$(A$,2)+LEFT$(A$,2)+MID$(A$,4,2)
220 PRINTA$
```

```
Result:  "850612"
```

By the way, "A$" is supposed to be read and pronounced "A String", so the phrase, "the A$ string", as used above, is redundant.

CHAPTER 7

Self-Killing Programs

This chapter shows a technique which allows a program to kill itself from memory after running.

A number of times the question has been raised on various bulletin boards, "How can I kill a program after running it, to remove it from memory?"

The obvious reply has been, 'Type KILL"NAME.BA" and hit the ENTER key.'

But it is often useful to let a running program kill itself, so that when you go back to the menu, the program itself, is gone.

If that is your interest, you can do it by adding the below lines to the bottom of your progam instead of an END or MENU statement. Just renumber the lines to suit your program, and put your program name in the A$ string in line 400, instead of "NAME".

Lines 10 to 60 are just there to demonstrate the program for you. Save the program as "NAME.BA" so it appears on the computer menu. Then run it, and after running, it will be gone.

```
10 ' NAME.BA
20 FORA=1TO25
30 PRINTA;
40 BEEP:NEXT
50 PRINT
60 '
400 A$="NEW"+CHR$(13)+"KILL"+CHR$(34)+"NAME.BA"+CHR$(13)+"MENU"+chr$(13)
401 FORI=1TOLEN(A$):POKE65449+2*I,ASC(MID$(A$,I,1)):POKE65450+2*I,0:NEXT
402 POKE65450,I
```

For those of you who want to know how it works, line 400 creates a string which contains the commands that would be necessary to delete a program from memory. Line 401 pokes the string into the keyboard buffer, which holds it until the computer asks for input from the keyboard. Line 402 pokes the number of characters in the buffer into the buffer counter.

When the program ends, the computer gives you the "Ok" prompt, then checks to see if there are any characters waiting in the keyboard buffer. There will be, so the BASIC interpreter dumps the string into BASIC as a command line, just as if it had been typed in from the keyboard. The commands kill the program, and return to the menu.

Ta Da!

Note that the above addresses, where the string is poked, is the keyboard input buffer for the Model 100 and 102. For the Tandy 200, 65449 in line 401 must be changed to 64797; and 65450 in lines 401 and 402 must be changed to 64798.

CHAPTER 8

Looking at Pixels and MOD

This chapter shows a technique of testing whether a specific pixel on the LCD screen is on or off, and explains use of the MOD statement.

Several people have asked over the years, how to find out if a particular pixel on the LCD screen is "on" or "off". Below is a short routine provided by Neil Wick, as one technique that does it.

Use the routine in lines 100-130, which will determine if the pixel at screen location X,Y is set ("on") or preset ("off"). Program lines 10-60 are there to set the pixel for the test, and have no relation to the actual operation of the subroutine other than establishing the X and Y coordinates for the subroutine.

```
10 CLS
20 X=115:Y=20
30 PSET(X,Y)
40 GOSUB100
50 IFFLTHENBEEP
60 END
70 '
100 PRINT@40*(Y\8)+X\6,CHR$(27)"P";
110 Y=Y:PRINTCHR$(27)"Q"
120 FL=PEEK(XMOD6-20)AND2^(YMOD8)
130 RETURN
```

This program will indicate that the pixel at 115,20 is on by beeping. To make sure it doesn't beep if the pixel is off, simply delete line 30 and run the program again.

Note that testing FL in line 50 with an IF (true) statement, is the actual test. If FL is greater than 0 then the pixel is on. If FL equals, or is less than 0 then the pixel is off.

While the subroutine (lines 100-130) is shown in four lines, you can actually put it all on one line in your program, if you like.

Pgm notes: Those backslashes in line 100 indicate integer division, and are explained on page 107 of the M100 manual. It's the same as INT(Y/8) and

INT(X/6). The backslash is created by holding down the graph key, and pressing the hyphen key simultaneously.

"MOD" is an arithmetic operation, giving the leftover balance after a divide or series of subtractions. For example, 12MOD4 = 0, 13MOD4 = 1, 14MOD4 = 2, etc. 10MOD6 = 4 (4 is the leftover after subtracting 6 from 10 as many times as possible). Or it can be expressed as "6 goes into 10 with a balance of 4 leftover".

The neat thing about the MOD command, is that it also incorporates an automatic "scaling" feature, so that numbers greater than the mod number are brought down to size by repetitive subtraction. Thus, 13MOD6 = 1. (6 is subtracted from 13 as many times as possible, until there is only a "leftover").

Thanks to Neil Wick for the subroutine.

CHAPTER 9

Upper and Lower-case Conversion

This chapter discusses case conversion; the changing of characters from upper to lower case, or from lower to upper case; and a new approach using AND/OR logic.

A number of programs have been written which require case conversion, i.e. converting from upper to lower case, or from lower to upper case. In BASIC, it has generally been done by examining the characters in the string, and changing the value of the character on a character-by-character basis, as in the following example:

```
10 A$ = "The Quick Brown Fox"
20 FOR A = 1 TO LEN(A$)
30 B=ASC(MID$(A$,A,1))
40 IF B>95 AND B<123 THEN MID$(A$,A,1)= CHR$(B-32)  :REM Lower to upper
```

or:

```
40 IF B>64 AND B<91 THEN MID$(A$,A,1)= CHR$(B+32)  :REM Upper to Lower
50 NEXT
```

Well, it turns out there is an easier way to effect case conversion, using AND/OR logic. If, for example, you want to input a character, and make sure it's in upper case, no matter what it was to start with, the following statement (line 20) will do it:

```
10 A$=INPUT$(1)
20 A$=CHR$(ASC(A$)AND223)
30 PRINT A$
```

To convert any input to lower case:

```
10 A$=INPUT$(1)
20 A$=CHR$(ASC(A$)OR32)
30 PRINT A$
```

This routine affects almost all characters above CHR$(63) - the "?" mark, converting them into their "shifted" equivalent. And is quite useful for Yes/No types of inputs. It also allows easier checking for this type of input, requiring only a simple compare, rather than a series of comparisons. An example:

```
10 PRINT" Was that correct? (Y/N) ";
20 A$=INPUT$(1)
30 A$=CHR$(ASC(A$)AND223)
40 IFA$="Y"THEN ...
```

or:

```
40 IFINSTR("YN",A$)THEN 50 ELSE 20
50 .... (program continues)
```

The latter example checks to see if the response was "Y" or "N", and if not, continues to wait for proper input before proceeding. Interestingly, the AND logic will not affect characters that are already upper-case, and the OR logic will not affect characters that are already lower-case.

Unfortunately this technique does little to solve the original problem, that of converting an entire string of characters to upper case, or lower case. Using the following form:

```
 0 ' UPPER CASE
10 A$ = "The Quick Brown Fox"
20 FOR A = 1 TO LEN(A$)
30 IF ASC(MID$(A$,A,1)) >64 THEN MID$(A$,A,1)= CHR$(ASC(MID$(A$,A,1)) AND 223)
40 NEXT
```

```
 0 ' LOWER CASE
10 A$ = "THE QUICK BROWN FOX"
20 FOR A = 1 TO LEN(A$)
30 IF ASC(MID$(A$,A,1)) >64 THEN MID$(A$,A,1)= CHR$(ASC(MID$(A$,A,1)) OR 32)
40 NEXT
```

...the IF statement is just as cumbersome as the original coding was.

But in cases of character input, and case conversion, the new AND/OR logic may prove useful, particularly where the form previously used was:

```
10 A$=INPUT$(1)
20 IF(ASC(A$)>96)AND(ASC(A$)<123)THENA$=CHR$(ASC(A$)-32)
```

Compare:

```
10 A$=INPUT$(1)
20 A$=CHR$(ASC(A$)AND223)
```

A viable solution is to get your keyboard input character by character, convert it and add it to a string, ending the input phase when a carriage return is received; like this:

```
0 ' Converts string input to upper-case
10 PRINT" Input your string: ";
20 A$=INPUT$(1)
30 A$=CHR$(ASC(A$)AND223):B$=B$+A$
40 IFA$=CHR$(13)THEN 50 ELSE 20
50 .... (program continues, string is in B$)
```

CHAPTER 10

Which Computer is the Program Running In?

This chapter describes an interesting technique that can be used by a running program to determine whether it is running in a 100 or 200, and can then make necessary adjustments, allowing a single program to run on either the 100/102 or the 200, even when the program contains address dependent peeks, pokes and calls.

An interesting technique was revealed in the software for the Tandy Disk Drive, allowing a running program to determine whether it was running in a Model 100, 102 or Tandy 200. With the portable disk drive, it allowed the initialization program to load the correct software in either machine.

The technique involves simply reading the ROM byte at address 01 Hex, or 1 decimal. This is easily done with the peek command in BASIC, in the following form:

```
        A=PEEK(1)
```

If the computer is a Tandy 200, A will = 171. If A is anything else, it's a 100 or 102.

This can easily be used in an IF/THEN test as follows:

```
        IF PEEK(1)=171 THEN (it's a T200)
```

OK, what's it good for?

Well, it can be used in programs that use peeks, pokes, or calls, to have the program automatically adjust to whichever computer it is running in...

```
        IF PEEK(1)=171 THEN CALL17600 ELSE CALL21265

        IF PEEK(1)<>171 THEN POKE 62958,128 ELSE POKE 64335,1
```

Obviously, you must know the necessary CALLS, PEEKS and POKES in both of the computers in order to use this technique, and you'll be able to find many of the equivalent PEEKs, POKEs and CALL addresses later on in this book.

In any kind of terminal, or file transfer program, this technique could initialize whatever COM status was required. In a RAM or System Manager program, it could adjust to read the internal RAM directory.... In fact, it could adjust to do anything that is considered machine dependent. It could be used to create "paged" displays on the screen, adjusting for larger displays on the 200's screen, automatically if the program is running in a 200.

It's the easiest way to make a single program usable in either the 100/102 or 200, by simply adjusting the program for the machine. An entire series of variables can be set, depending on the machine, lke this:

```
IF PEEK(1)=171 THEN M=64554:N=640:R=15 ELSE M=62330:N=320:R=8
```

The variables can then be used in the program for various purposes, setting number of screen lines (R), calculating screen display position (N), or looking for a flag, or pointing to a buffer area in the reserved memory area (M).

CHAPTER 11

Using the MAXFILES Statement

This chapter discusses MAXFILES and associated errors caused by the wrong MAXFILES setting. An elegant solution is suggested to prevent such errors.

One area of personal programming that is often overlooked is the correct use of the MAXFILES statement. It is common, for example, for a programmer to set MAXFILES to 0, in order to free up as much RAM as possible, in which a certain program might run. The problem arises if the programmer forgets to reset MAXFILES to some value other than zero when the program ends and/or returns to the Main Menu. When another user, particularly a non-programming user, runs another program that attempts to open a file, or files, if MAXFILES=0 then he encounters a BN error (Bad File Number). "BN" almost always indicates an incorrect maxfiles setting.

On the other extreme, programmers who may use several files, may set MAXFILES to a value greater than zero in order to accomodate the needs of their program, and then forget to reset it to the normal default of 1, before exiting the program. This robs the end user of some of his RAM space, by reserving file buffers which will not be needed.

The most common solution to this problem is for any programmer to specify how many files he will be using at the beginning of his program, by using the MAXFILES=(whatever) statement, and then always resetting MAXFILES to the default value 1, when normally exiting the program, MAXFILES=1:END.

An even more elegant solution, is to have the program read the value of MAXFILES, poke it into a safe place in RAM, then set MAXFILES to whatever is needed by the program. On exiting the program, peeking back the value that was stored, and using it to reset MAXFILES to the previous condition.

There are a number of areas in reserved RAM that can be used as the temporary storage place, including any of the unused bytes in the function key definition table, where you know there will be no conflicts with anything else you may have put there. The advantage of using the unused function key definition space, is that those addresses are well documented for both the Model 100 and Tandy 200.

As an example, 15 bytes are allowed for each function key definition. F1 in BASIC is the command "Files". In the definition string, it is "Files"+ CHR$(13), for a total of 6 bytes, leaving 9 bytes unused. Since the actual addresses allocated for this definition lie between 63369 and 63384 (61684 to 61699 in the T200), it appears that temporary storage of the MAXFILES value could be made at address 63383 (61698 on the T200) with no difficulty.

The MAXFILES command erases any variables previously set, so simply storing the value of MAXFILES in the form:

```
     100 X=MAXFILES:MAXFILES=3
```

would not work... but the following method does work:

```
     100 X=MAXFILES : POKE 63383,X : MAXFILES=3
       .
       . ... program
       .
     900 X=PEEK(63383) : MAXFILES=X : MENU
```

(Of course the peek and poke addresses would be 61698 on the Tandy 200.)

It is even possible to shorten the commands, since there is no syntax problem, as in:

```
     100 POKE 63383,MAXFILES : MAXFILES=3
       .
     900 MAXFILES=PEEK(63383) : MENU
```

If you have changed the definition of function key F1 in BASIC this particular address might not be available to you. Suggest you peek the address in direct mode to see what's there first. If it's zero, then you can probably use the above forms with no problem. These unused bytes are available for your use, and as long as the key definitions are not changed, will present no problems.

If that particular address is not a zero, then simply add 15 to the address number and try again... You'd then be in the definition area for F2. Another 15 would put you in the F3 definition area, etc...

Another couple of interesting features of the MAXFILES statement:

1. To maximize RAM space in a program which does not access any files, you can set MAXFILES=0 to gain an additional 256 bytes of RAM space in which your program can run.

2. You can change MAXFILES in a running program, but doing so will usually clear all existing variables, and may clear the program stack. It should be used carefully, or should be fully tested prior to this type of use.

3. The machine defaults to a value of MAXFILES=1 when new, or after a cold-start. You can set any number of buffers you need at the beginning of your program, and reset the value to 1 as the final statement in your program, just before the END or MENU statement, resetting it to the normal default value.

4. When setting MAXFILES at the beginning of the program, it should appear

before anything else, as it sets aside buffer space, and in the process, clears anything else you may have set.

```
     10 CLEAR 1500: DEFINTA-E: DIM A$(22): MAXFILES=2
```

... would leave you only with the MAXFILES setting, all the rest being over-written when setting the buffer spaces. Proper construction would be:

```
     10 MAXFILES=2: CLEAR 1500: DEFINT A-E: DIM A$(22): A=3
```

This logic does the following:

1. Set two 256 byte buffers for file I/O.
2. Clear 1500 bytes for variable storage
3. Define variables A through E as integers
4. Dimension for 22 string slots in an array called A$
5. Sets the value of A, equal to 3. The rest of the integers, B-E are automatically set to zero when defined.

CHAPTER 12

REM vs. the ' Mark

This quickie describes the difference between tokenized "REM" and "'" remark lines in BASIC programming.

In the process of byte-fighting programs, it is prefereable to use the reserved word 'REM' in any remark lines, rather than using the apostrophe mark, the single left quote, "'". (Which, unfortunately, looks a lot neater!)

When "REM" is tokenized, it is saved as a single byte, with a value of 142.

However when the apostrophe is tokenized it is stored as three bytes, in the sequence 58,142,255.

If your program, as many do, contains a lot of instructions or documentation at the head of the program in REM lines, use of 'REM' instead of the apostrophe will save two bytes per line or occurrence. In a large program, with many REM statements, it may be significant.

CHAPTER 13

The Random Number Generator

Results of testing the random number generator in the Model 100 for repeating cycles of numbers; an observation or two comparing results between the Model 100 and Tandy 200, and how to generate numbers within a specific range.

The random number generator in the Model 100 seems to be better than most folks have been led to believe, me included. Some comments by others that it would "recylce" itself, and start repeating the same series of random numbers after about 64,000 numbers, got me curious... So I set up the following short program to see whether that was true.

```
10 CLS:PRINT"Random Generator Test"
20 PRINT"Start Time: "TIME$
30 A=RND(1):PRINT"First Number ="A
40 C=1:PRINT"Count ="
50 B=RND(1):C=C+1:PRINT@128,C
60 IFB=ATHENPRINT:PRINT"Match at "TIME$
70 GOTO50
```

Well, the program ran for 41 hours, generating about 22 numbers per second, racking up a total of 3,259,022 random numbers, without ever matching the first number! I only stopped the program since I needed the computer for other purposes, and couldn't allow it to continue.

Interestingly, the random number generator returns a fourteen-digit number between .00000000000001 and .99999999999999.

But on the basis of that test, I'd say the random number generator is capable of much more than we previously thought. It may be that some particular combination of "adjusting" the generator output to get integers or "real" numbers within a certain range, might show a repeating series after a certain number of operations, but it was not possible for me to test this theory, since it would depend on a specific-case instance, and only be duplicatable by the exact same code.

It was also interesting that both the Model 100 and Tandy 200 generated the same series of numbers, based on a short comparison using the following one-line direct command:

```
FOR A = 1 TO 5:PRINT RND(1):NEXT
```

From this, it would appear that both machines use the exact same code to generate numbers, and start with the same seed. But the Tandy 200 runs a mite slower, as demonstrated with the following one-liner:

```
FOR A = 1 TO 2000:NEXT:BEEP
```

This routine, started simultaneously on both machines (and at the same time, too!), finished on the Model 100 first.

Generating random numbers with the statement RND(1) will use the previously generated number as the "seed" to start the process of generating the next number. Using RND(0) or RND(-1) will generate the same number, over and over. Using RND(1) in a program without reseeding the generator, will always generate the same series of numbers each time the program is run.

You can "reseed" the random number generator yourself at the beginning of the program by asking for the user to choose a number between 1 and 100, or by using the seconds digits from the clock, or by reading one of the two running counters in high memory, then poking the value into the memory seed location at 64634 (100 and 102) or 63277 (200). The running counters are located at 63791 in the 100/102 and 61983 in the 200. The recommended form would be:

For the 100/102:

```
POKE 64634,PEEK(63791)
```

For the 200:

```
POKE 63277,PEEK(61983)
```

Another interesting feature is that if the number in parentheses in the RND statement is negative, as in RND(-45), the value 45 will be used to reseed the number generator. But subsequent use of the generator should be with the RND(1) form, or the same number will be generated over and over. The best way to use this would be to do some number manipulation using the date, time or running counter, then make it a negative value, and use it in the RND statement as a variable, as in RND(-X).

To use the random number generator to generate a specific series of numbers, like numbers between 1 and 12, multiply your top number by the RND(1) value, then add 1 to the result. This will yield a value between 1 and your desired top number. Example, to get random numbers from 1 to 12, do the following:

```
100 X=12
110 Y=INT((RND(1)*X)+1)
120 PRINT Y
```

To generate a series of random numbers, simply put it in a loop that gives you the desired number of numbers.

If you want a series of numbers between two numbers, like between 7 and 15, set X to the difference between the lowest and highest values, plus 1 (15-7+1 = 9), and instead of adding 1 in line 110, add 7 (the lowest value desired), as in:

```
100 X=9
110 Y=INT((RND(1)*X)+7)
120 PRINT Y
```

An interesting test for an ideal distribution of numbers (to see that you're getting a reasonably accurate equal distribution, i.e. different numbers, about the same number of times in any sequence) is the following program. It will generate 10,000 random numbers, and count how many times each number appeared, then provide a report of the distribution. Ideally, every number will appear approximately the same number of times, plus or minus some small percentage. Press any key after you've looked at the results to get back to the menu.

```
10 CLS:PRINT@42,"Working..."
20 DIMZ(12):X=12
30 FOR A=1 TO 10000
40 Y=INT((RND(1)*X)+1)
50 Z(Y)=Z(Y)+1
60 NEXT
70 CLS:PRINT" Distribution:"
80 FORA=1TO6
90 PRINTA;Z(A);TAB(20);A+6;Z(A+6)
100 NEXT
110 A$=INPUT$(1):MENU
```

CHAPTER 14

The Bottom Line

This chapter shows how individual words on the bottom line of the screen can be shown in normal or reversed video, not requiring symbols or graphics characters be used to indicate function key functions. A potential programming change for Xmodem-type programs, TELCOM emulators, and to indicate function key operations.

There a number of programs, notably the Xmodem programs, and TELCOM emulators, where the authors have used words on the bottom line of the screen to simulate the function key label lines normally used in the various program modes in the computer. In cases such as in the Xmodem programs, a prepared line is written to line 8 on the screen (or line 16 on the Tandy 200), and then locked in place with a CALL which disables scrolling of the last line.

In most cases, the author of the program rewrites the bottom line on the screen with different symbols when it is necessary to indicate that different program operations, such as downloading or uploading are taking place. In some cases, the authors have simply replaced the word "Down" or "Up" with a group of @'s, cross-hatches, or graphics characters; none of which really duplicate the normal reverse video display of those words in TELCOM.

The following short program illustrates a better technique of doing that, which maintains the original look of the TELCOM program, writing the word in reverse video. It's not a direct substitution of statements that are in any of the various programs, but an illustration of the technique that could have been used, and which might be used by individulas wanting to make changes in their own displays. The program runs in BASIC to simulate Xmodem TELCOM emulators, and can be adapted for use in any programs with a label line on the bottom of the screen.

```
10 CLS
20 PRINT@280,"Prev Down  Up  Stop Echo Free File Menu";
30 PRINT@42,"Normal  ";:Q$=INPUT$(1)
40 CALL17001:PRINT@285,"Down";:CALL17006
50 PRINT@42,"Reversed ";:Q$=INPUT$(1)
60 PRINT@285,"Down";
70 PRINT@42,"Normal  ";:Q$=INPUT$(1)
```

Line 20 simply writes a TELCOM simulation on line 8 (of a Model 100/102)... the words could be changed to whatever are needed by the program. After the bottom line is written on the screen, "Normal" shows up on the second line of the screen to show you that you are at the normal display. Pressing any key will throw "Down" on the bottom line into reverse video, and give you a "Reversed" indicator on the second line. Pressing any key will again reverse the process, turning off the reversed "Down", returning it to the normal display, and indicating "Normal" on the second line again.

The CALL's in the above example make this version specific to the Model 100 or 102, but similar CALL's exist for the Tandy 200, and are documented. And even changing line 40 so that it uses the ESCape technique works:

```
40 PRINT@285,CHR$(27)"pDown"CHR$(27)"q";
```

In the 100/102, CALL 17001 turns on reverse video, CALL 17006 turns it off.

In the Tandy 200, CALL 20360 turns on reverse video, CALL 20365 turns it off.

This is overly simple, not showing the CALLs necessary to lock and unlock the screen display... but just showing that the words on the bottom line can be reversed, not requiring the use of special symbols or graphic characters to indicate functions. Question is, why didn't the various authors use this technique instead of graphics symbols?

In any case, once you have written a label on the bottom line of the screen, you can lock it there by protecting it from screen scroll, with a simple CALL to the ROM routine which disables screen scrolling.

For the 100/102, CALL 16959 locks the bottom line, CALL 16964 releases it.

For the Tandy 200, CALL 20318 locks the bottom line, CALL 20323 releases it.

CHAPTER 15

Elegant Programming Tips

Some ideas, methods and tips to improve your BASIC programming to conserve memory and increase execution speed of all such programs.

Speed & Conservation of Memory

Every KEYWORD, quoted literal, punctutation mark, and character, occupies at least one byte in memory. Every line number uses two bytes for the number, another two bytes for a pointer to the next line location, and a "Null" byte which terminates the line. Each space takes a byte. Variables use a two-byte pointer to the area in memory that the variable is stored in. In addition, each type of variable uses different amounts of storage space. Integers require two bytes, reals require four bytes and double precision values require eight bytes. Strings use one byte for each character and are allocated dynamically, which means they can change size based on the number of characters assigned to them. This is all background for the following space & time saving tips:

1: All variables should be "typed" at the beginning of each program. That is, you should specify each variable's "type" at the beginning of the program. Use DEFINT, DEFSNG, etc. Even string variables should be typed with the DEFSTR statement, because the "$" used with each such variable in the body of the program takes a byte of storage. Eliminating the $ signs, can save a significant number of bytes!

2: Use integers wherever possible, especially in counting loops. The time savings is dramatic, and can be personally tested with simple programming loops. Integers are 2 1/2 times faster than the default double-precision variables. Reals are only 25% faster than double-precision.

3: Eliminate the variable argument on the NEXT portion of FOR/NEXT loops. Here is a real kicker - besides the saving of one byte of storage for each such occurence, integers are 60% slower, reals are 50% slower, and double-precision is 80% slower when you specify the variable than when you simply specify NEXT.

4: Eliminate final quote marks on all literals that appear at the end of a line. A byte is saved for each quote mark eliminated.

5: Another surprise that violates previous Microsoft BASIC conventions is that semicolons are NOT routinely needed as variable punctuation. They only seem to be required after PRINT USING or if you want to keep the cursor from advancing to a new line or in an INPUT statement. All other uses are gratuitous and can be left out to save one byte per occurence.

6: Use as many multiple statements per line as you can, since for each line number eliminated, you have a net savings of four bytes (the five needed for each separate line, less the one byte used for each colon.) You are allowed to use BASIC lines up to 255 characters in length. Entire routines can often be written in a single line.

7: Eliminate ALL spaces. Your archive copy may have them for intelligibility, but your running copy will save one byte for each space that is removed. Byte-fighting and compacting programs always remove all spaces and REMarks from the program.

8: Use REM in REMark lines instead of the apostrophe, it will save two bytes for each occurrence.

9: Remove all REMark lines from running copies of your programs to save space. Never use program branches (GOTO, GOSUB or whatever) to a REMark line. If the REMark line is removed, your program has nowhere to branch to.

10: Put all initialization code at the end of your program and use a GOTO or a GOSUB to get to it. Little used routines at the end make dramatic improvment in speed since MSBasic must search from the beginning of the file for each line referenced.

11: Put all time critical code and subroutines at the beginning of the program instead of the end, for the same reasons as 10 above. Use a GOTO to jump over them to the main body of the program.

12: Initialize all variables before you start the main body of code. This creates a stable, linear table of variables that are accessed more rapidly.

13: DO NOT use the supposed short-cut of raising a number to the .5 power to save some time over the SQR routine. Inaccuracies in the 11th and 12th decimal place will tend to introduce errors in later calculations which might use this result.

14: Assign literals that are used more than once in a program to a variable. Do this in your initialization code. Almost one byte per character is saved for each such duplication eliminated. Use GOSUBS to get to often repeated prompts or screen displays.

15: Use SPACE$(X) to assign X number of spaces rather than STRING$(X,32) or a literal string of spaces.

One of the serious deficiencies of the built-in Basic interpreter is the lack of user-defined functions, but almost any function can be written as a standard subroutine, and called as a GOSUB. There are many good sourcebooks which contain libraries of such subroutines which can be extracted and rewritten for use in the Tandy portables. For assembly language programmers, many standard 8080 routines will work in the Model 100/102 and Tandy 200, requiring only machine-specific changes.

Thanks to Richard Horowitz for many of these programming suggestions.

CHAPTER 16

Control and Escape Codes

This chapter discusses use of control and escape codes for controlling LCD functions, moving the cursor around the screen, erasing lines and portions of the screen.

Use of the control keys in the Model 100/102 and 200:

In TEXT or EDIT modes:

CTRL-A jumps to the beginning of the last word, same as Shift-Left Arrow

CTRL-B moves the cursor to the bottom of the file, same as Shift-Down Arrow

CTRL-C No action

CTRL-D Moves cursor one space right, same as Left Arrow

CTRL-E Moves cursor up one line, same as Up Arrow

CTRL-F No action

CTRL-G "Save to:" prompt, same as pressing F3

CTRL-H Destructive backspace

CTRL-I Same as Tab key

CTRL-J No action

CTRL-K No action

CTRL-L Same as F7 - Define (Select) for cut and paste

CTRL-M Same as Carriage Return (ENTER)

CTRL-N Same as F1, "String:" prompt

CTRL-O Same as F5, "Copy defined characters into paste buffer, do not delete"

CTRL-P Allows input of control characters as literal characters in text files. The CTRL-P is followed by the CTRL character to be embedded.

Common functions of control characters embedded in text files with a CTRL-P:

CTRL-G Beep (Bell)

CTRL-H Backspace for overstrike

CTRL-I Tab

CTRL-J Line Feed

CTRL-L Form Feed

CTRL-M Carriage Return (same as ENTER)

CTRL-Q Moves cursor to beginning of line, same as Control-Left Arrow

CTRL-R Moves cursor to end of line, same as Control-Right Arrow

CTRL-S Moves cursor one space to left, same as pressing Left Arrow

CTRL-T Moves cursor to top of screen, in same column, same as Shift-Up Arrow

CTRL-U Deletes defined characters, same as pressing F6 after using F7 (CTRL-L)

CTRL-V "Load from:" prompt, same as pressing F2

CTRL-W Moves to top of file, same as Control-Up Arrow

CTRL-X Moves cursor down one line, same as Down Arrow

CTRL-Z Moves cursor to bottom of file, same as Control-Down Arrow

In BASIC:

Some of these can be used in programs, in the form: PRINT CHR$(27)"J"

From the keyboard:

CTRL-H Destructive backspace

CTRL-I Tab

CTRL-M Carriage Return (ENTER)

CTRL-U When typing BASIC statements, erases the current line

CTRL-X Same as CTRL-U

In the program:

CTRL-J Erases all data from the cursor position to the bottom of the screen

CTRL-K Erases data from the cursor position to the end of the line

CTRL-p	Turns on reverse video (note: "p" is in lower case)
CTRL-q	Turns off reverse video (note: "q" is in lower case)
CTRL-P	Turns on the cursor
CTRL-Q	Turns off the cursor
CTRL-T	Turns off screen scrolling (bottom line lock)
CTRL-U	Turns on screen scrolling (bottom line unlock)
CTRL-W	Releases locked screen display
CTRL-Y	Locks screen display

CHAPTER 17

Looking into the RAM Directory

This chapter looks into the RAM directory, describes it's structure and use, and provides two programs to conveniently list the RAM files and their true locations in RAM.

The Model 100 and 102 have 27 "slots" for filenames in the RAM directory in a reserved area of high RAM. There are 55 such "slots" in the Tandy 200. This directory contains the names of the programs and files in RAM that is listed on the computer's main menu when you turn it on. The menu in the 100/102 actually only shows you 24 of the filenames, and on the 200, only 52 filenames are shown. Three of the directory file slots are effectively "invisible" in all three machines. More about that, later.

Along with the names of the programs and files in ROM and RAM, the directory contains an attribute byte which tells the system what type of file it is, and two bytes which indicate the starting location of the file or program. Each directory entry uses eleven bytes; the attribute byte, two bytes for the start or entry address, and eight bytes for the actual name. The eight byte name field contains the name and file extension, without the period, using REPORTDO for the file REPORT.DO, or SORTERBA for SORTER.BA. The built-in ROM programs have no filename extension, so the last two characters of those entries are blank characters. Any name which is less than six characters long, or eight, including the filename extension, is padded with blanks to justify both ends of the entry. PADS.DO, for example, would be stored as "PADS DO". When you go into a file with the TEXT program, and are prompted for the name of the file to edit, you could enter it as PADS.DO or PADS .DO and get to the same file. The period is a kind of delimiter, and tells the computer where the end of the filename is, and is jogged over to match the actual name length, as it is in the directory.

This presents an interesting programming possibility. Suppose you have some sort of file which is updated on a regular basis by a frequently-run program. The program can look through the directory, find the filename, and poke ascii characters into the filename slot to produce a new filename in the directory. Example: suppose you bill customers on an hourly basis, and start each day with a new file called BILL.DO. Each time you perform work for a customer, you run a small BASIC program that adds your billable time to the file, and updates the filename by adding numbers to the name in the directory. You might work for seventeen customers that day, and at the end of the day the file would be named BILL17.DO. Or, on a list of things to do, STUF.DO, your program could poke the date the file was last updated into the filename, so you might have a file called STUF21.DO, indicating the file was last updated on the 21st.

In the 100/102, the directory starts at 63842, and extends to 64139 in eleven-byte steps; in the 200, it starts at 62034, and extends to 62639. Since each "slot", is eleven bytes long, it's a simple matter to step through the entire directory with a program look and look at each name, looking for a matching filename. Here's one example of how it can be done in the 100/102:

```
100 FOR A=63842 TO 64139 STEP 11
110   A$="":FOR B=A+3 TO A+8
120   A$=A$+CHR$(PEEK(B))
130   NEXT
140   IF A$="BUDGET" THEN 200
150 NEXT
160 PRINT"File not found":STOP
200 (File exists, continue with program)
```

If you wanted to poke update characters into the filename, the program might be as follows, assuming you wanted to add "01" to the filename:

```
100 FOR A=63842 TO 64139 STEP 11
110   A$="":FOR B=A+3 TO A+6
120   A$=A$+CHR$(PEEK(B))
130   NEXT
140   IF A$="BDGT" THEN 200
150 NEXT
160 PRINT"File not found":STOP
200 POKE A+7,ASC("0"):POKE A+8,ASC("1")
```

Of course you could use variables, too, as in:

```
200 F$="01"
210 POKE A+7,ASC(F$):F$=MID$(F$,2):POKE A+8,ASC(F$)
```

Now, about those three extra filename slots. There are three floating file buffers in RAM, which you frequently use, but never actually see; the unnamed BASIC program buffer, the paste buffer, and BASIC's editing buffer. The unnamed program buffer is where a program is stored when you type it into BASIC, before you save it with a program name. It is, simply, an unnamed program.

In the Model 100, these three buffers had actual names, and an inspection of the directory would reveal them as "Suzuki", "Hayashi", and "Ricki". There is a story going around that Suzuki and Hayashi are the names of two of the Japanese engineers who worked on the original project, and Ricki is the name of one of the project engineers at MicroSoft. Needing names for the buffers, they used their own names as a sort of inside joke. This story hasn't been firmly verified, but it makes a nice story, anyway. These names were replaced in the 200 with a series of unintelligable characters, since there was no real need to have an actual file name for the buffers. -- You won't see Ricki very often in the directory. It's there in a new machine, or after a cold start; but when you invoke the EDIT command in BASIC, part of the program's assigned name, if one exists, is written into the directory slot, and overwrites "Ricki". So he's hard to find.

The "attribute byte", the first byte in the eleven character slot, tells the system what kind of file it is, and whether it is visible or invisible. It's a single-byte number with the following meanings:

0	A blank (empty) filename slot, or an empty BASIC edit buffer
128	A BASIC program
136	The unnamed program buffer, or an invisible BASIC program
160	A .CO file (assembly language program)
168	An invisible .CO file
176	A ROM program
184	An invisible ROM program name
192	A .DO file
200	The paste buffer, or an invisible .DO file
240	In the Tandy 200 only, 240 is the attribute byte for MSPLAN, which is actually an alternate ROM program
248	MSPLAN's attribute byte when the name is made invisible

With the above information, you've probably just learned that you can make files and programs "invisible" on your main menu. More on that, later.

The file location is stored in hexdecimal notation in the second and third bytes in the directory slot, in the assembly language convention of the lowest part of the address first, followed by the highest part of the address. Unless you are an assembly language programmer, this information is not of much use. But it does tell where, in RAM, each file starts, and by sorting the addresses, and subtracting each from the next highest, you can determine how long a file might actually be. - Seldom needed information, but it is possible.

Interstingly, each time you do something to a file in RAM, like editing it, adding or deleting characters, the other files get moved around to accomodate your changes, and all of the directory pointers are updated to reflect the new location addresses. It's generally done so fast you aren't aware of it.

Following are two RAM directory programs, one for the 100/102 and one for the 200, which will give you all the information you'll probably ever need directly from the RAM directory.

```
0 ' RAMDIR.100
1 '
2 ' Copyright 1985 Tony B. Anderson        All Rights Reserved
3 '
4 ' This program prints the internal RAM directory for the Model 100, listing
5 ' the entry address, the type and status byte, the starting address of the
6 ' file in RAM, and the file name.  Buffers are indicated.  Output is 'paged'
7 ' for the M100 screen.
8 '
10 CLS:L=-1
20 PRINT@316,"(c)";:PRINT@48,"Model 100 RAM Directory":PRINT
30 PRINT"   Displays entry location, type and Status byte, Location in RAM
   where file starts, and name."
70 FORA=63842TO64129STEP11:L=L+1
72 IFL/6=INT(L/6)THENPRINT@280,"Press ENTER to continue ";:A$=INPUT$(1):CLS
80 PRINTUSING"##### ";A;
```

```
90 PRINTUSING"#### ";PEEK(A);
100 PRINTUSING"###### ";PEEK(A+1)+256*PEEK(A+2);:PRINTTAB(20);
130 FORB=3TO10
140 PRINTCHR$(PEEK(A+B));
150 NEXT
155 IFA=63897THEN PRINT"(Unnamed Pgm)";
160 IFA=63908THEN PRINT" (Paste Buf)";
165 IFA=63919THEN PRINT" (Edit Buf)" ELSEPRINT
170 NEXT
```

```
0 ' RAMDIR.200
1 '
2 ' Copyright 1985 Tony B. Anderson          All Rights Reserved
3 '
4 ' This program is Model 200 specific!
5 '
6 ' Program prints list of internal RAM directory, giving directory addresses,
7 ' Type and Status Flag, Starting address in RAM, and file name.
8 '
10 CLS:L=7
20 PRINT"ADRSS = The Directory Entry Address"
30 PRINT"T&S   = Type & Status (See manual)"
40 PRINT"LOC   = File Location in RAM"
50 PRINT"NAME  = Directory Name":PRINT
60 PRINT" ADRSS  T&S  LOC    NAME":PRINT
70 FORA=62034TO62628STEP11:L=L+1
75 IFL/14=INT(L/14)THENPRINT@600,"Press ENTER to continue ";:A$=INPUT$(1):CLS
80 PRINTUSING"###### ";A;
90 PRINTUSING"#### ";PEEK(A);:IFFTHENIFPEEK(A)=0THENPRINT:GOTO160
100 PRINTUSING"###### ";PEEK(A+1)+256*PEEK(A+2);:PRINTTAB(20);
110 IFA=62100THENPRINT"NONAMEBA":GOTO150
120 IFA=62111THENPRINT"PASTEBUF":GOTO150
125 IFA=62122THENPRINT"EDIT BUF":F=1:GOTO150
130 FORB=3TO10
140 PRINTCHR$(PEEK(A+B));
150 NEXT:PRINT
160 NEXT
```

Now, on making file and program names invisible... A close inspection of the above list of attribute bytes will show that the difference between a file's name being visible or invisible, is a difference of eight in the number. So all you have to do to make a name invisible, is to run the RAMDIR program in your machine, determine where the attribute byte is located (the address in the first column), and poke a new number into that byte that is 8 higher than the "visible" number. To make the name visible again, just poke a new number into that byte that is 8 lower than the "invisible" value.

Example: To make a file invisible, perhaps a file containing important data that you don't want anybody else to see, run RAMDIR, find out the address to poke the new value, add 8 to the number in the second column, and poke the new number in, using the direct BASIC command:

```
     POKE A,B
```

...where A is the address in the first column, and B is the value in the second column, plus 8.

To make the file visibile again, poke A, with B, minus 8.

Or to make it even easier, once you know the address to poke, use one of the following forms:

```
     POKE A,PEEK(A)+8    or    POKE A,PEEK(A)-8
```

Further example: Suppose you never use the built-in programs ADDRSS and SCHEDL, and would like to remove those names from your menu, just to clear it up a little. Running RAMDIR, you would find that the ADDRSS file slot is at 63875 (in the 100/102), and SCHEDL's file slot is at 63886. Both are of type 176, a ROM-based program. Go into BASIC and type:

```
     POKE 63875,184:POKE63886,184
```

... and press the ENTER key. When you go back to the main menu, you'll find that both ADDRSS and SCHEDL are no longer on the menu.

It's important to note that while this proceedure removes the name from the menu, it doesn't give you any more file slots to store additional programs in. The file name is still in the directory, you've just adjusted the attribute byte so that the ROM routine that lists the directory to the menu will not see those files, and will not list them. The only way you can see them now, is to run the RAMDIR program, or some similar program that accesses the data directly from the RAM directory.

If you want to bring either of the program names back to the menu, all you have to do is poke 176 back into the attribute byte, the same way you poked 184 into the byte to render the names invisible.

You can make any filename visible or invisible. However, it is not recommended that you make BASIC invisible, even if you seldom use it. It may make it hard to enter BASIC, to reenter commands, or run programs, and the only way you can regain control, in case of a system lockup, would be to do a cold-start, with the resulting loss of files. But anything else is fair game if you have a need to hide the file, or just don't use one of the ROM programs.

CHAPTER 18

Disabling BREAK, ^C, and Function Keys

This chapter discusses a poke which will disable the Break key and the Control-C key, so that a running program cannot be aborted in that manner, work-arounds, and rudimentary password protection programs.

It is often useful to disable the Break Key and Control-C to prevent someone from stopping the run of a program. In the Tandy portables, the Break Key is functionally the same as Control-C, sometimes called the "Panic Button" to stop a program running.

The Break Key in the Model 100/102 can be disabled, either in command mode, or in a running program with the command POKE 63056,128.

To re-enable the Break Key, use the command POKE 63056,0.

In the Tandy 200, the address is 61234. Use POKE 61234,128 to disable Break, and POKE 61234,0 to re-enable it.

Actually, this poke disables program interrupts, and disabling the Break Key also disables the Control-C interrupt, and Function Key interrupts. However, if Control-C or a Function Key is pressed while the Break key is disabled, it will be stored in the keyboard buffer, and acted upon when the Break Key is re-enabled, or the program ends normally. This can be demonstrated with the following short program: (for the 100/102)

```
10 POKE 63056,128 : REM Disables Break
20 FOR A= 1 TO 200
30 PRINT A;
40 NEXT
50 POKE 63056,0 : REM Enables Break
```

This program will print the numbers 1 to 200 on the screen. While the numbers are printing, you cannot stop the program by pressing either Break or CTRL-C. However, if you press either, when the program has finished running, the BASIC interpreter will print "OK" on the screen, and on the next line, "^C", which is the Control-C. If you press function key F1 during the program run, after the OK, you will get a list of the FILES, which is the default function of the F1 button.

Generally, you can disable the Break Key at any point in a running program, but once it is disabled, you should not re-enable it until the end of the program.

Otherwise, there is the possibility that the program could still be interrupted at the point where the Break Key is re-enabled.

Recommended: If you disable the Break Key, always be sure to re-enable it at the end of the program. Otherwise, it might not be there when you need it.

Lenny Leedy, a CompuServe subscriber, pointed out a potential problem for this method of disabling the Break Key... If your program has any Input statements, a Control-C for the input will abort the program. This is probably because the BASIC Interpreter has forced an interrupt for the input of data, and a Control-C break can sneak in during the input mode.

There is a way around this problem, which seems cumbersome, but if you want the benefits of disabling the Break Key, you have to put up with the problems, too. Lenny was trying to disable the Break Key during input of a password, and his original program approach was as follows:

```
10 POKE 63056,128
20 CLS:INPUT"Enter Password";P$
30 IF P$="SECRETWORD" THEN CLS:POKE 63056,0:END
40 PRINT"ACCESS DENIED"
50 FOR A = 1 TO 1000:NEXT:'(time delay)
60 GOTO 10
```

The answer to this type of problem is to get input character by character, and construct the password by adding the characters. This approach works:

```
10 CLS
20 T$="":P$=""
30 PRINT"Enter password:";
40 T$ = INKEY$:IF T$="" THEN 40
50 IF T$=CHR$(13) THEN 70
60 P$=P$+T$:GOTO 40
70 IF P$="SECRET WORD" THEN CLS:PRINT"VERIFIED":POKE 63056,0:END
80 CLS:PRINT"ACCESS DENIED":GOTO 20
```

The character-by-character input effectively traps Control-C or Break Key presses. If you don't type SECRET WORD exactly, the only way to get out of the routine is by pressing the reset button.

After "ACCESS DENIED" you could do other things...such as POWEROFF, or changing the power on-time to virtually nothing; POWER1:POWEROFF. Then if the computer is turned off, and back on, hoping to gain control, it will shut itself off almost instantly. The only recourse then, will be a cold-start.

CHAPTER 19

Getting Into Assembly Language

Interested in learning more about assembly language, as it applies in the Tandy portables? I was there, myself, and here are some insights on which way to go, and how to get there.

This chapter describes some of the insights and techniques I've developed to learn more about assembly language programming for the Model 100 family of computers. It includes three programs which provide disassembled listings of decimal or hex data statements in BASIC loader programs or HEX files which are used to create machine language programs.

In an attempt to teach myself assembly language programming for the Model 100 family of computers, with a little help and guidance from others, I've found the following books useful:

"Z-80 and 8080 Assembly Language Programming" by Kathe Spracklen, published by Hayden Book Company; ISBN 0-8104-5167-0

"8080A-8085 Assembly Language Programming" by Lance A. Leventhal, published by Osborne/McGraw-Hill.

"8080/8085 Assembly Language Subroutines" by Lance A. Leventhal and Winthrop Saville, published by Osborne/McGraw-Hill.

"Hidden Powers of the TRS-80 Model 100" by Christopher Morgan, published by the Plume/Waite Group, ISBN 0-452-25578-3.

"Inside the Model 100" by Carl Oppedahl, published by Weber Systems, ISBN 0-938862-31-6

And, of course, all the ROM and RAM maps I could find. In that respect, the most complete single source was a complete map of the Model 100, developed by Robert Covington, and which is available in library 8 of CompuServe's Model 100 Forum. The files are named 100ROM.RC0 through 100ROM.RC6 (seven files), and 100RAM.RDC, along with the various 100 to 200 conversion lists provided by other forum members. Address conversion tables have been published in various support magazines; the most complete was in the May 1986 issue of Portable 100 magazine. Followup articles in subsequent issues detailed many of the 200's addresses. Back issues, or copies of the articles are available from the publisher.

In addition to "book larnin'", I've found it helps to have a real live person, who already knows how, who can answer some questions for you. There's no substitute for specific answers to your
questions.

I've also found it useful to study "Source Code" listings whenever available, and to disassemble various assembly language programs and routines, in order to analyze what they do, and how they do it. There are many such source code listings in the Model 100 forum's database, but not enough, I'm afraid. Many assembly language programmers give you the program, but not the source code. Too bad, so much can be learned from a well-documented source code list. It's one thing to read about the PUSHes and POPs, but quite another to see how they are actually used in a program or routine, and why!

I've noticed that many assembly language programs, written for the Model 100 family, come to us as "Loaders" or HEX files. The actual machine language program is effectively hidden in the form of DATA statements, or HEX code listings. Such programs or listings are notoriousely lacking in documentation about the machine language program or code they develop. Being naturally curious (read "nosey"), and in order to find out more about how these programs operate, I wanted to get a listing of the program in it's assembled form. This involves loading the program to the address where it will execute, then breaking out of the program and doing a disassembly of the execution area.

That seemed like an approach that was overly complicated, since it involved killing other stuff I had resident in my computer, such as the disk operating system, text formatter, utility routines, etc... So I decided to try disass-embling the original Loader's DATA statements, or HEX code file, instead. But, where to start???

Well, I developed the following technique, which works for me, giving me the assembled listing, as if the assembly language program were actually loaded and disassembled. The nice thing, is that it's all done in BASIC, so it doesn't disturb any of those precious machine language programs that I've come to rely on, and which, if removed or corrupted, involve intensive labor and thought on my part, just to get them reloaded and operational again.

First, I would download a HEX file or Loader program from the database, which-ever form the program was in; then, while I still had the file in ASCII form (as downloaded), I'd delete all the program code except the data or HEX state-ments. I would then add my own conversion routine to the remaining data statements, either by loading the routine direct from disk, or using the "cut and paste" technique from another RAM file. These "conversion" routines are common enough, usually reading the data statements, and poking the numbers into RAM. I wrote one myself, and adapted one from another program, modifying it to do the conversion I needed.

The resulting new program is loaded into BASIC, and run. The output of the program is a text file, consisting of the actual numbers (assembly language opcodes) which the original program would have loaded into memory. Thus, I have an ASCII text file of the machine language program, ready for reading by a disassembler... but no such disassembler. Not one that would do exactly what I wanted, anyway...

Well, Jim Irwin came to the rescue... I obtained a copy of his SMALL.100

disassembler and modified it to suit the purpose with Jim's permission. I was attracted to this particular disassembler because it supported both the Model 100 and 200, and used a "small" amount of RAM space to do the work. Other disassemblers, which provide additional information, could also be adapted for this purpose.

So, here's the program that I use to create the data file from decimal DATA statements: (This one's my own)

```
63000 ' CVTDTA  Converts data statements into a data file.
63010 MAXFILES=1
63020 CLS:PRINT@41,"Working..."
63030 OPEN "DATA.DO" FOR OUTPUT AS 1
63040 ON ERROR GOTO 63080
63050 READX:X$=STR$(X):X$=MID$(X$,2)+","
63060 PRINT#1,X$;:PRINT@50,X$
63070 GOTO 63050
63080 CLOSE:PRINT@121,"Done":END
```

This program deals with DATA statements of the following form:

```
100 DATA 205,117,16,0,32,66,125,71,13,10
```

I chose to use very high program numbers for the simple reason very few programmers ever use numbers that high, and it would almost certainly load the program above the data statements from the original Loader program, without accidently deleting any in the conversion to BASIC process. The program simply reads the data statements, converts them to string data, and outputs it to a new file named "DATA.DO".

In order to deal with HEX files, or program loaders which have the data in HEX form in the data statements, I adapted code from a hex program loader. There is one problem with this approach, it reads the HEX data as an entire line, then separates the two-byte hex code from the line, and converts it. This is fine, as long as all the hex lines have an even number of characters in them. This is usually the case, although I have run across lines with an odd number of characters. Usually that can be fixed by moving one character from the end of the line to the beginning of the next line (in the data statements). One other problem, the conversion routine does not deal with "relocatable code", which is often indicated by the presence of question marks in the hex statements. A question mark is not a legal hex character, but is used by the loader program to indicate where a calculated address must be added into the resulting assembly language code. So, if the hex statements contain question marks, this program won't do it.

```
63000 ' CVTHEX Converts HEX data statements to a decimal data file
63010 D$="0123456789ABCDEF":OPEN "DATA.DO" FOR OUTPUT AS 1
63020 CLS:PRINT@42,"Working..."
63030 ON ERROR GOTO 63100
63035 READ LN$:L=LEN(LN$)
63040 IFL/2<>INT(L/2)THEN PRINT@121,"Data String Length Error":GOTO 63110
63045 FOR I=1 TO LEN(LN$) STEP 2
```

```
63050 C1 =(INSTR(1,D$,MID$(LN$,I,1))-1)*16
63055 C2 =INSTR(1,D$,MID$(LN$,I+1,1))-1
63060 C1 = C1+C2
63070 A$=STR$(C1):A$=MID$(A$,2)+","
63080 PRINT#1,A$;:PRINT@55,A$;"   "
63090 NEXTI:GOTO 63035
63100 CLOSE:PRINT@121,"Done"
63110 END
```

This program deals with data statements of the following form:

```
50 DATA "E1CD622CCD3142C36854CD0000CD"
```

Both of the above programs leave a comma at the end of the data written to the new file, which should be removed before running the disassembler.

Speaking of the Disassembler, as mentioned before, it is an adaptation of Jim Irwin's SMALL.DIS. It reads the data file created by one of the above programs, and provides a disassembled listing of the code. The listing goes directly to the screen, and can be scrolled or stopped with a CTRL-S. For those who want a printed copy of the output, just turn on your printer. If it's connected and ready, the program will automatically send a copy of the listing to the printer. (Most of the lines and line numbers are intact, if you'd like to compare this version to the original program.) I named it "DATA-D" for "DATA-DISASSEMBLER".

```
0 ' DATA-D  Data Statement Disassembler
2 ' adapted from Jim Irwin's SMALL.100 disassembler
21 CLEAR1500:DEFSTRN-S:DIMN(255):GOSUB31
22 CLS:INPUT "Filename: ";P
23 OPENPFORINPUTAS1:CLS
24 IFNOTEOF(1)THENINPUT#1,AELSEEND
25 Q=" ":IFA>32ANDA<127THENQ=CHR$(A)
26 B=VAL(N(A)):IFB=0THENS=N(A):GOSUB29:GOTO24
27 IFB=1THENINPUT#1,X:S=MID$(N(A),2)+STR$(X):GOSUB29:A=X:S=" ":Q=" "
   :GOSUB29:GOTO24
28 INPUT#1,X,Z:S=MID$(N(A),2)+STR$(256*Z+X):GOSUB29:S="":Q="":A=X:GOSUB29:A=Z
   :GOSUB29:GOTO24
29 IF(INP(187)AND6)=2THENLPRINTUSING"###   \            \  !";A,S,Q
30 PRINTUSING"###   \            \  !";A,S,Q:RETURN
31 DATANOP,RNZ,2LXI B,POP B,STAX B,2JNZ,INX B,2JMP,INR B,2CNZ,DCR B,PUSH B
32 DATA1MVI B,1ADI,RLC,RST 0,UNDF,RZ,DAD B,RET,LDAX B,2JZ,DCX B
33 DATAUNDF,INR C,2CZ,DCR C,2CALL,1MVI C,1ACI,RRC,RST 8,UNDF,RNC,2LXI D,POP D
34 DATASTAX D,2JNC,INX D,1OUT,INR D,2CNC,DCR D,PUSH D,1MVI D
35 DATA1SUI,RAL,RST 16,UNDF,RC,DAD D,UNDF,LDAX D,2JC,DCX D,1IN,INR E,2CC
36 DATADCR E,UNDF,1MVI E,1SBI,RAR,RST 24,RIM,RPO,2LXI H
37 DATAPOP H,2SHLD,2JPO,INX H,XTHL,INR H,2CPO,DCR H,PUSH H,1MVI H,1ANI,DAA
38 DATARST 32,UNDF,RPE,DAD H,PCHL,2LHLD,2JPE,DCX H,XCHG,INR L
39 DATA2CPE,DCR L,UNDF,1MVI L,1XRI,CMA,RST 40,SIM,RP,2LXI SP,POP PSW
40 DATA2STA,2JP,INX SP,DI,INR M,2CP,DCR M,PUSH PSW,1MVI M,1ORI,STC
41 DATARST 48,UNDF,RM,DAD SP,SPHL,2LDA,2JM,DCX SP,EI,INR A,2CM,DCR A,UNDF
42 DATA1MVI A,1CPI,CMC,1RST 56,ADD,ADC,SUB,SBB,ANA,XRA,ORA,CMP
43 FORI=0TO63:READN(I),N(I+192):NEXT:S="BCDEHLMA":K=64
44 FORI=1TO8:READR:FORJ=1TO8:N(K)="MOV "+MID$(S,I,1)+","+MID$(S,J,1)
45 N(K+64)=R+" "+MID$(S,J,1):K=K+1:NEXT:NEXT:N(118)="HLT":RETURN
```

These three programs have become very useful in learning more about assembly language programming.

A final word...

```
DISPLY  EQU     6DF6H    ; ROM routine address (200)
;                          define message to be displayed
MSG     DM      'Good Luck!'
START   LXI     H,MSG    ; put pointer in HL where MSG is stored
        CALL    DISPLY   ; display the message
        MVI     A,7      ; put BEEP in the accumulator
        RST     4        ; print the character in the accumulator
        RET              ; return to calling program
```

CHAPTER 20

"Paging" in TELCOM for neat printouts

This chapter describes a TELCOM utility which sends a form-feed to the printer, by pressing a function key in TELCOM's TERM mode.

Here's a neat little utility for users of the Tandy 200 who do not normally use the BRK (Break) key in TELCOM, which is also usable in the Model 100/102. In the Tandy 200, we have previously developed methods of using the F6 key, via the vector jump table, but the F7 key has been left pretty much alone, even though the Break key is not often used - or at least is used by a relatively small number of users.

This is a good example of using some of the unused bytes in the function key definition table to hide a small patch that will stay resident and operational without much attention. It will stay in place unless you run a program that redefines the function keys, a poor programming technique, by the way, or you suffer a cold-start which resets everything to the computer's default settings.

RATIONALE:

I found that I needed the ability to send a form feed to the printer while online, in order to separate program listings and files downloaded, or "echoed" to the printer, from the system. Normally, my printer keeps up pretty well with the incoming data flow, so I just had to wait for the printer to catch up to a normal system pause, and press the printer's form-feed button. This worked pretty well until recently when I added a print buffer to my configuration. That allowed the host computer to send data to me faster than my printer could handle it, and it often happened that I would wait at a system break for some minutes before the printer caught up, to where I could press that form-feed button myself.

Since I use an external modem at 1200 baud, it was a simple matter to make a short BASIC program, and leave it on the menu as FFJUMP.BA (Form-Feed Jump), and simply drop out of TELCOM when I needed the form feed and run this program which would send a form-feed to the printer and jump back into TELCOM. The program form was simple:

```
1 LPRINT CHR$(12) : CALL 25454  '(TERM entry address for the 200)
```

Of course, this was only usable when using the external modem. How much simpler it would be, I thought, if I could simply do it with a function key press, and why not use F7 which I never use for sending a break, anyway?

FOR THE TANDY 200:

The technique I decided on, was to use the F7 vector table to jump to a small machine language program, hidden in one of the unused areas in the function key definition area of BASIC, much like Phil Wheeler's LFUTL (Linefeed Utility) program does. Considering that some users might already have the linefeed utility stuffed in the unused area behind the "Menu" definition of the F8 key, I chose to use the area behind "Run", the F4 key. The exact location can be changed to suit a user's requirements, if he has customized his BASIC function keys differently. The small, eight-byte machine language program can be hidden almost anywhere, even in the LOMEM area, if that's being used.

The machine language code to send a form-feed to the printer in the Tandy 200 is as follows:

```
        MVI     A,12     ;put Formfeed in the A register
        CALL    23060    ;send contents of A to the printer
        JMP     25454    ;jump back into TERM
```

The program was too small to "assemble", so I simply looked up the opcodes, and converted the call and jump addresses into the LO/HI form needed, and came up with a list of numbers which represented exactly what the above program would be if "assembled", and built a short BASIC program using the CHR$() technique to load them into the Key 4 definition as illustrated below in line 1. The first two pokes in line 2, poke the entry address into the F7 vector table address, so pressing F7 would send you to the entry address. The first few times I tried it, it didn't work. It would work perfectly as a CALL from BASIC, but not from TELCOM.

James Yi, a frequent Tandy 200 programmer, suggested that I separate the ML code from the "Run"+CHR(13) description with a CHR$(1), then poke a zero into that position to separate the hidden code from the "normal" function key description, and to call the routine from the "image" table, instead of the "working" table of function key definitions. It seems the working table is redefined and not available when you're in TELCOM. When those suggestions were added, the program worked perfectly, and is listed here in the final form.

```
1 KEY4,"Run"+CHR$(13)+CHR$(1)+CHR$(62)+CHR$(12)+CHR$(205)+CHR$(20)+CHR$(90)+
CHR$(195)+CHR$(110)+CHR$(99)
2 POKE62785,41:POKE62786,241:POKE61737,0
```

If you want to move it from the key 4 definition area to somewhere else, you'll need to change the pokes in line 2 to accomodate the new location. The first two are the entry address of the new location, and the third poke is where that CHR$(1) is stored in the definition image table.

Should you want to remove the program, and "unhook" it from the vector table jump, reenabling whatever function was there originally, simply go into BASIC and type the following:

```
KEY4,"Run"+CHR$(13)  <ENTER>
POKE 62785,168:POKE 62786,156  <ENTER>
```

Or, you could add line numbers and save it as a "cancelling" program.

FOR THE MODEL 100/102:

The routine is adaptable to the Model 100 and 102; the call and jump addresses are simply changed to accomodate the 100's system. The program can be stored in a function key definition the same way, and could be set up to operate from either the F6 or F7 function keys, whichever is most useful. If it is hidden in the unused area of Key 4, as for the 200 version, and called from an F7 key press in TELCOM, the program would be as follows:

```
        MVI     A,12
        CALL    19285
        JMP     21589
```

```
1 KEY4,"Run"+CHR$(13)+CHR$(1)+CHR$(62)+CHR$(12)+CHR$(205)+CHR$(85)+CHR$(75)+
CHR$(195)+CHR$(85)+CHR$(84)
2 POKE64270,62:POKE64271,248:POKE63550,0
```

Should you wish to disable this and restore the system to it's normal defaults, then go into BASIC and type:

```
KEY4,"Run"+CHR$(13)  <ENTER>
POKE 64270,58:POKE 64271,31  <ENTER>
```

Or, you could add line numbers and save this as a "cancelling" program.

CHAPTER 21

How to hide machine language routines in a BASIC program

This chapter describes a seldom-used technique to hide your own machine language patches or routines in a BASIC program, where you can call them at a fixed location whenever you need them.

One of the factors involved in running assembly language programs in the Tandy portables, is that the program must run from a fixed location in RAM; the same location every time. In this design, the area between HIMEM and MAXRAM has been reserved for operation of such programs through the process of clearing one program out of the location, setting the HIMEM pointer, loading the next program into the area, and executing (running) it. There are often cases where this is inconvenient, such as when you have some other assembly language program that needs to be in that area, too, such as a DOS to access disk files, or a text formatting program. In such cases it is often possible to relocate an assembly language program to run in a different area, to eliminate conflicts between the two programs.

In some cases, you don't need an entire assembly language program, just a patch or small subroutine, to speed up the program. In those cases, it is usually inconvenient to load the routine into the reserved area in the normal manner.

It has been found that many assembly language routines can be loaded into the previous screen buffer area normally used by TELCOM. In the 100/102, this is about 320 bytes, and in the 200, about 640 bytes in size.

Another technique used by some commercial software houses, is to hide the assembly language code in the first BASIC program in RAM. Because of an interesting quirk in how programs and files are stored in RAM space, the first BASIC program you load into RAM in an empty machine does not move, does not change it's location. All other programs and files can be moved around in RAM, according to what the machine needs to do to manage the RAM space. But that first program never moves. This has led to the realization that you can create a dummy BASIC program consisting of REMark lines, and stuff assembly language code into those REM lines, which leaves it at a fixed location, which makes it CALLable from any program that needs the routine. Lots of little routines and patches can be handled in the machine in this manner; things such as linefeed patches, case conversion routines, macro sending routines, Xmodem routines that can be called from TELCOM, etc. The secret lies in understanding how BASIC stores program lines, and how you can use that to define storage space for your assembly language routines.

So, the place to start is in understanding how BASIC stores each line of any BASIC program.

A tokenized BASIC line is stored in the following format: The first two bytes in a line are the low order/high order bytes indicating the line number. The second two bytes are a low order/high order pointer to the beginning of the next line (one byte higher than the end of the current line). Up to 255 ascii characters (not tokens), which comprise the statements, followed by a null byte, character zero. The end of the BASIC program is signified by three nulls in a row. Keywords in the statements are reduced, as far as practical, to a single byte "token", which represents the keyword. "REM" for example, is reduced to the single byte character 142.

It is a simple matter to save a single-line BASIC program in an empty machine, a line with a REM in it, and use peeking techniques to find out where the REM byte is located. In a 32K machine, the first byte of available RAM for storage of your BASIC programs is 32769. In a 24K machine, it's 40965. A simple loop will explore the first few bytes in RAM, and tell you where the REM is stored in your single-line program.

First, create a line which contains a line number... any line number, but use a number less than 10 - we're dealing with ascii display bytes, and 1-9 display in less space than 65010. If you must use a line number greater than 10, then stick to 10-99 and reduce you reserved space to 249 bytes. Follow the line number with a space, the letters REM, followed by 250 periods or spaces; the periods are easy to see, and easy to find in RAM. The format of such a line would look like this:

```
1 REM...........................................................................
................................................................................
................................................................................
..................
```

Save that as a BASIC program with some unique name, like MLHOLD or something. Next, type the following following short program into BASIC, don't save it, but just run it to see where that REM is located.

```
1 FOR A=32769 TO 32779
2 IF PEEK(A) = 142 THEN PRINT A
3 NEXT
```

If you are using a 24K machine, including the Tandy 200, change line 1 to read:

```
1 FOR A=40965 TO 40975
```

When you run the program, it will tell you the address where the REM is stored in RAM. Knowing that, you can write, and store any assembly language program or routines in the following 250 bytes, but do not use the byte where the REM is stored. How do you get the program into those 250 bytes? Simple. You write your assembly language program or routines, convert the source code to data statements, and poke the values of the code into the REM, starting at 1 byte higher than the location of the REM. If you don't use all 250 bytes, you can add other routines later on, in the space occupied by the extra periods. If you forget what the next available location is for new code, simply peek through the beginning of RAM space for aseries of periods.

If you don't ever plan on adding more code to this area, you can shorten those 250 periods to the exact number required to hold your code. Conversely, if you have a program or routine which is larger than 250 bytes, just add more lines to your dummy program, exactly as the first line (with a different line number, of course), and at the end of the first 250 bytes of code, jump over the six bytes required for the end of line marker, the line number and pointer to the next line, and the REM token in the second line. You then have a 500-byte capacity for your program.

CAUTIONS: Once you have poked assembly language into the REM statements, never list the dummy program or attempt to edit it with the EDIT command. The interpreter will attempt to restructure your code into readable ascii characters, which will likely destroy your code, and can easily lead to a code start.

Do not attempt to RUN the dummy program. There may be binary zeros in the assembly code in the REM statements which will confuse the interpreter, which could cause a cold start.

It would be a good idea to make the dummy program invisible, so it can't be accidently run from the main menu, too. Refer to Chapter 17 on how to do this.

Next step: Suppose you already have one of those "first programs in RAM" type of programs; like P.G. Design's OMENU program, or a program called SUPERA, or UltraSoft's DOS for the Tandy Portable Disk Drive. In that case, it's a little harder, but not much.

Use RAMDIR for your machine to find out where that program is located in RAM; both starting and ending points. (you'll have to have a second file in RAM to see where the first one ends; 1 byte lower than the start of the second program or file) Then you'll have to look through the first program's code to find it's line numbering scheme; you'll need a monitor program to do this, or do extensive peeking and figuring out what's what. Eventually what you do is to load the first program into BASIC, add a new line number at the end as described above, with a higher line number than any used in the original program, and peek through RAM from the end of the original program, looking for your REM statement. You might want to make it two REM's in a row to make it easier to find. Then write your assembly code to fit the available addresses and poke it into the new line. Your routines will be there as long as the original first program is there.

The technique for use with P.G. Design's Menu program is to add a line 65000 to the MENU.BA program. In this case, your hidden code is limited to 246 bytes, rather than 250. - You're displaying more ascii characters with the larger line number. If you use two REM's to make it easier to find, reduce your periods to 245 bytes.

Then peek around near the end of the program to find your REMs. The following program has been used successfully.

```
0 ' REM finder for MENU.BA
1 FOR A=32325 TO 34570
2 IF PEEK(A)=142 AND PEEK(A+1)=142 THEN PRINT A
3 NEXT
```

Now... here's an interesting technique. Suppose you write a little assembly language program, and store it in a REM statement in a first BASIC program in RAM. You know what the entry point is for that routine, it's usually the first byte of the routine in well-designed code. Just for an example, let's say it's 32774. You can call that routine with a function key press in BASIC by putting a CALL on an unused function key, following the technique given in Chapter 25. Typing:

```
   Key 6,"CALL32774"+CHR$(13)
```

... will define function key 6 to call your routine whenever it's pressed.

As a practical example, suppose your assembly language routine/program is a word and line counter for TEXT files. Further assume that when you wrote it, you wrote it as a stand alone program that asks for a filename to be checked, does the checking and reports the results. When you want to check the word and line count in a file you're writing, just go into BASIC and press the F6 key. It calls the program, prompts you for the filename, and reports the results. Your program can even wait for a key press and return you to the menu. Or, with a little more complex techniques, jump right back into the TEXT file you just checked, for more editing or writing.

CHAPTER 22

Common Pokes for the 100/102

This chapter deals with peeking and poking around in the reserved RAM memory locations, lists some useful addresses and the values that may be poked to them for special applications.

THIS CHAPTER IS MODEL 100/102 SPECIFIC.

The Tandy portable computers have 64K of addressable memory. In the 100/102, the first 32K is read-only memory, in the form of a ROM chip that contains the operating system and programs, and the second 32K is user RAM space. In the Tandy 200, the first 40K is read-only memory, with the last 24K available as user RAM space. You can peek into all the available addresses, both in ROM and RAM, but you can only poke values into the RAM addresses.

Generally there is no purpose in peeking and poking into most of RAM space, in the area where programs and files are stored. Files in RAM are managed dynamically, which means they get moved around in RAM space, depending on what you may be doing at any given time, so specific locations where you might want to peek or poke are likely to move without notice. There are some ways available with advanced programming techniques to locate specific things if you really need to do so, but are not appropriate to this text.

What is valuable, is to be able to peek and poke into the reserved RAM area above MAXRAM, where the operating system stores flags, buffers, pointers, and other important operating information. This chapter lists some useful addresses in the reserved RAM space, and values you can poke into those addresses for some special applications.

TELCOM FUNCTIONS:

POKE 63064,0 will set TELCOM to HALF DUPLEX mode.

POKE 63064,255 will set TELCOM to FULL DUPLEX mode.

POKE 63065,255 will set TELCOM to ECHO to printer. (Turns ECHO on)

POKE 63065,0 will set TELCOM to not ECHO to printer. (Turns ECHO off)

POKE 63066,1 will set TELCOM to send a LF (line feed) after a CR (carriage return) when uploading (sending) files. Some devices need the linefeed.

POKE 63066,0 will reset to No LF after CR (default condition)

To be able to clear the screen when in TELCOM, type the following two pokes in BASIC, before you use TELCOM. POKE 64268,49 and POKE 64269,66. Then when you want to clear the screen, just press the F6 key. -- To reset to normal operation, in BASIC, type POKE 64268,247 and POKE 64269,127. -- These two statements can be entered on one line, with a colon between them; i.e.

```
POKE64268,247:POKE64269,127.
```

It is often useful to poke TELCOM status into memory from a basic program, for example, in a file transfer program. COM "Status" is held in RAM in locations 63067 to 63071, in a five character field. The leading character is the baud rate, and if this is all you want to change, you can use the direct command: POKE 63067,ASC("M") (or whatever you want to poke there).

If, for example, you want to reset TELCOM to modem status before leaving a file transfer program, you could add this short routine to your program:

```
1000 REM  Pokes "M7I1E" into "Status"
1010 FOR A = 63067 TO 63071
1020 READ B : POKE A,B
1030 NEXT
1040 DATA 77,55,73,49,69
1050 REM   M  7  I  1  E
```

The values in the DATA statement are the ASCII value of the character you want to poke into memory. Change the values in the DATA statement to set any Status required.

There are two changes that will enhance TELCOM's basic functions by providing two additional function key actions, using F6 and F7. While in TELCOM's TERM mode, pressing F6 will respond with the current number of bytes free in RAM; and F7 will give you a list of file names in RAM, similar to the F1-FILES command in BASIC. This short routine which will make the necessary pokes:

```
1 FOR I = -1268 TO -1265
2 READ X : POKE I,X
3 NEXT
4 DATA 172,126,58,31
```

While you will probably never need this, the dialing pulse rate is stored at RAM address 63019, and can be set, or changed with a POKE.

```
POKE 63019,1  = 10 pulses/second
POKE 63019,20 = 20 pulses/second
```

BREAK KEY - CONTROL-C - FUNCTION KEYS

The interrupt routines can be turned on and off at RAM location 63056. The command to disable one of these functions will disable them all. There is no way to "selectively" disable one or the other in this manner. It is possible to trap keyboard input and discard unwanted input, but that technique is beyond the scope of this text.

POKE 63056,128 will disable the Break Key, Control-C and Function Keys.

POKE 63056,0 will re-enable the same functions.

SECRET STORAGE SPACES IN RAM

There are 36 bytes of RAM which can effectively be rendered invisible to the operating system, and will be protected from everything short of a cold start, which can be used to store an identification name or number permanently in RAM. You might also find other uses for this 36 bytes. However, if you have installed an alternate ROM in your computer, these 36 bytes are no longer available to you, as they are the bytes that hold the M/L routines for accessing the optional ROM.

If you are not using an optional ROM and want to use this area to store a name or other data, first POKE 62975,201 to completely disable any residual code that might already be in this area. Then, do these two: POKE 62981,201 and POKE 63018,255.

Once these last two pokes are done, addresses 62982 to 63017 are free, and can be used for anything you want. The above pokes remain effective until a cold-start occurs. What's happening here is that the original code from 62981 to 63011 checks for the existence of an optional ROM everytime you power up. If ROM is installed, then the value 255 is stored in 63018, and the name of the ROM (for the Menu) will be placed in 64164-64171. Code from 63012 to 63017 is used when choosing the option ROM from the Menu.

The POKE 62981,201 puts a M/L RETurn there, which effectively makes the M100 think that the optional ROM is installed. You need the POKE 63018,255 to prevent a cold-start on power-up.

Whatever you put in those 36 bytes will stay there, regardless of whatever BASIC or normal machine language programs you are running. Obviously, you use POKE commands to get whatever you want into those locations. You can poke data into those 36 bytes, or a special machine language routine you need. But remember, this precludes your use of ROM-based software in the optional ROM socket.

Interested in more free RAM to poke stuff into? Space used in TELCOM to store the previous screen (64704 to 65023) is generally safe, although it is also used for lots of different things, including some Menu Directory stuff. The lowest addresses might be useful. Also, the optional ROM LUCID makes use of some of this area, if it is installed. This area is more useful for running small machine language programs, rather than any long-term data storage.

The MDM & COM receive buffer (65350 to 65413) can be used as long as MDM/COM ports are not being accessed for input or output. That's 64 bytes to poke away in, at least for temporary use during a running program. If a running program does a CLEAR or LOADM, then all variables are erased. Before using either of these statements, a program can poke important variables, values or characters it wants to save into these unused RAM areas, then PEEK them back later.

There are also a few bytes available in each of the function key definition slots in the function key definition table, 63498-63625, but you'll have to look for the available bytes, since some function keys will have definitions stored in the same table.

BASIC PROGRAM POKES

The following pokes would generally be used from a BASIC program, but could also be used in direct mode, if needed.

As you know, SCREEN,0 will turn off the label line. However, in some programs, it's also desirable to prevent the user from pressing the LABEL key to turn the label line back on. POKE 64173,0 will disable the Label Key, so that pressing it will have no effect. If the label line is on, it will stay on. If it is off, it will stay off. However, the effect is temporary; returning to the menu will automatically re-enable the LABEL key.

POKE 64173,0 will disable the Label key.

POKE 64173,1 will re-enable the Label key.

POKE 63048,175 will turn on Reverse Video display, until cancelled.

POKE 63048,0 will turn off reverse video, and return to normal display.

POKE 65348,175 turns Sound Off.

POKE 65348,0 turns Sound On.

To place a string of characters in the keyboard buffer, just as if they had been typed in from the keyboard, use the following proceedure: (Assume that the string you want to put in the buffer is A$) Use the following line, either in direct mode, or written as a line of code in your program.

```
FOR I = 1 TO LEN(A$):POKE 65449+2*I, ASC(MID$(A$,I,1)):POKE 65450+2*I,0:NEXT
:POKE 65450,I-1
```

This puts the characters from A$ into the odd addresses from 65451+, zeroing out the even addresses which are reserved to indicate Function key entries; then puts the number of characters in the buffer into 65450. This is very useful before a program does a SAVEM, LOAD, or MERGE, which would cause the program to stop running. Using the above code, with A$ = "RUN"+CHR$(13) will make a program restart automatically, which is particularly useful after the SAVEM statement in machine language loaders. But be careful, the length of A$ must be less than 32 characters.

Here's a quickie poke that will initialize the RND seed based on the time:

```
J = 63795:FOR I = 64634 TO 64637:POKE I,16*PEEK(J) + PEEK(J+1):J = J+2:NEXT
```

And, this one will re-seed the random number generator with one of 125 possible random values obtained from a running counter: (thanks to Larry Gensch)

```
POKE 64634,PEEK(63791)
```

The following poke will send the next PRINT statement in a BASIC program to the printer port, instead of to the LCD screen. It works only on the next PRINT statement in a program, so has limited usefullness, but might be useful in a trace or debugging program. Just add it into any BASIC program as a BASIC statement.

```
POKE 63096,1
```

Jim Irwin passed on the information that RAM address 64228 is the place to intercept the print routine, just before it prints a character. Some commercial software uses that location for an intercept to add line feeds. But, if one wanted to, they could intercept the character and re-direct it to another location, to the serial port, for example.

If you wish to disable printer output completely, in order to prevent a program lock-up if the printer is not on and ready, the following two pokes will bypass the printer port, effectively closing the port:

```
POKE 64228,136 : POKE 64229,20
```

To return to normal, use:

```
POKE 64228,243 : POKE 64229,127
```

The "bypass" poke, will apparently also prevent the PRINT key in the function key row from being used.

Alternately, if you wish to test the printer port to determine if the printer is powered up and ready to accept data, you can use the following statement in a BASIC program:

```
IF (INP(187)AND6) <> 2 THEN BEEP : PRINT "Printer Not Ready" : STOP
```

Or you can devise alternate tests: a 0 means the printer is not ready; a 2 means it is; and a 6 means it is not connected.

```
IF (INP(187)AND6) = 0 THEN (Printer not ready)
IF (INP(187)AND6) = 2 THEN (Printer ready)
IF (INP(187)AND6) = 6 THEN (Printer not connected)
```

Of course you realize, any address which can be "poked" can also be "peeked" to determine what is currently happening at that location, or what will happen depending on the value set at that address.

CHAPTER 23

Common Pokes for the Tandy 200

This chapter deals with peeking and poking around in the reserved RAM memory locations, lists some useful addresses and the values that may be poked to them for special applications.

THIS CHAPTER IS TANDY 200 SPECIFIC.

The Tandy portable computers have 64K of addressable memory. In the Tandy 200, the first 40K is read-only memory, with the last 24K available as user RAM space. You can peek into all the available addresses, both in ROM and RAM, but you can only poke values into the RAM addresses.

Generally there is no purpose in peeking and poking into most of RAM space, in the area where programs and files are stored. Files in RAM are managed dynamically, which means they get moved around in RAM space, depending on what you may be doing at any given time, so specific locations where you might want to peek or poke are likely to move without notice. There are some ways available with advanced programming techniques to locate specific things if you really need to do so, but are not appropriate to this text.

What is valuable, is to be able to peek and poke into the reserved RAM area above MAXRAM, where the operating system stores flags, buffers, pointers, and other important operating information. This chapter lists some useful addresses in the reserved RAM space, and values you can poke into those addresses for some special applications.

TELCOM FUNCTIONS:

POKE 61241,0 will set TELCOM to HALF DUPLEX mode.

POKE 61241,255 will set TELCOM to FULL DUPLEX mode.

POKE 61242,255 will set TELCOM to ECHO to printer. (Turns ECHO on)

POKE 61242,0 will set TELCOM to not ECHO to printer. (Turns ECHO off)

POKE 61243,1 will set TELCOM to send a LF (line feed) after a CR (carriage return) when uploading (sending) files. Some devices need the linefeed.

POKE 61243,0 will reset to No LF after CR (default condition)

To be able to clear the screen when in TELCOM, type the following two pokes in BASIC, before you use TELCOM. POKE 62783,77 and POKE 62784,79. Then when you want to clear the screen, just press the F6 key. -- To reset to normal operation, in BASIC, type POKE 62783,168 and POKE 62784,156. -- These two statements can be entered on one line, with a colon between them; i.e.

```
POKE62783,168:POKE62784,156
```

It is often useful to poke TELCOM status into memory from a basic program, for example, in a file transfer program. COM "Status" is held in RAM in locations 62144 to 662150, in a seven character field. The leading character is the baud rate, and if this is all you want to change, you can use the direct command: POKE 62144,ASC("M") (or whatever you want to poke there; ASC("5"), etc.).

If, for example, you want to reset TELCOM to modem status before leaving a file transfer program, you could add this short routine to your program:

```
1000 REM  Pokes "M7I1ENI" into "Status"
1010 FOR A = 61244 TO 61250
1020 READ B : POKE A,B
1030 NEXT
1040 DATA 77,55,73,49,69,78,73
1050 REM   M  7  I  1  E  N  I
```

The values in the DATA statement are the ASCII value of the character you want to poke into memory. Change the values in the DATA statement to set any Status required.

There are two changes that will enhance TELCOM's basic functions by providing two additional function key actions, using F6 and F7. While in TELCOM's TERM mode, pressing F6 will respond with the current number of bytes free in RAM; and F7 will give you a list of file names in RAM, similar to the F1-FILES command in BASIC. This will superceed the "Break" function of the F7 key if used. This short routine which will make the necessary pokes:

```
1 FOR I = 62783 TO 62786
2 READ X : POKE I,X
3 NEXT
4 DATA 253,154,42,42
```

While you will probably never need this, the dialing type and rate flag is stored at RAM address 61172, and can be set, or changed with a POKE.

```
POKE 61172,0  = Tone Dialing
POKE 61172,1  = 10 pulses/second
POKE 61172,14 = 20 pulses/second
```

Telcom originate or answer mode is stored at location 61252. 0 = Originate mode, and any number greater than 0 = Answer mode.

BREAK KEY - CONTROL-C - FUNCTION KEYS

The interrupt routines can be turned on and off at RAM location 61234. The command to disable one of these functions will disable them all. There is no way to "selectively" disable one or the other in this manner. It is possible to trap keyboard input and discard unwanted input, but that technique is beyond the scope of this text.

POKE 61234,128 will disable the Break Key, Control-C and Function Keys.

POKE 61234,0 will re-enable the same functions.

SECRET STORAGE SPACES IN RAM:

The Tandy 200 has a built-in optional ROM program, the MSPLAN spreadsheet, plus the ability to install and use a second optional ROM in an optional ROM socket. Consequently there is always at least one optional ROM available to the system, and the 36 "secret bytes" available to 100/102 users is not available to the 200 owner. But there is still storage space available in the 200.

The space used by TELCOM to store the previous screen (63408-64047, 640 bytes) is generally safe, although it is also used for lots of different things, including some Menu Directory stuff. The lowest addresses might be useful. Also, the optional ROM LUCID makes use of some of this area, if it is installed. This area is more useful for running small machine language programs, rather than any long-term data storage.

The MDM & COM receive buffer (64758 to 64821) can be used as long as MDM/COM ports are not being accessed for input or output. That's 64 bytes to poke away in, at least for temporary use during a running program. If a running program does a CLEAR or LOADM, then all variables are erased. Before using either of these statements, a program can poke important variables, values or characters it wants to save into these unused RAM areas, then PEEK them back later.

There are also a few bytes available in each of the function key definition slots in the function key definition table, 61685-61811, but you'll have to look for the available bytes, since some function keys will have definitions stored in the same table.

BASIC PROGRAM POKES

The following pokes would generally be used from a BASIC program, but could also be used in direct mode, if needed.

As you know, SCREEN,0 will turn off the label line. However, in some programs, it's also desirable to prevent the user from pressing the LABEL key to turn the label line back on. POKE 61194,0 will disable the Label Key, so that pressing it will have no effect. If the label line is on, it will stay on. If it is off, it will stay off. However, the effect is temporary; returning to the menu will automatically re-enable the LABEL key.

POKE 61194,0 will disable the Label key.

```
POKE 61194,1 will re-enable the Label key.

POKE 64756,175 turns Sound Off.

POKE 64756,0 turns Sound On.
```

To place a string of characters in the keyboard buffer, just as if they had been typed in from the keyboard, use the following proceedure: (Assume that the string you want to put in the buffer is A$) Use the following line, either in direct mode, or written as a line of code in your program.

```
  FOR I = 1 TO LEN(A$):POKE 64798+2*I, ASC(MID$(A$,I,1)):POKE 64799+2*I,0:NEXT
:POKE 64798,I-1
```

This puts the characters from A$ into the even addresses from 64800, zeroing out the odd addresses which are reserved to indicate Function key entries; then puts the number of characters in the buffer into 64798. This is very useful before a program does a SAVEM, LOAD, or MERGE, which would cause the program to stop running. Using the above code, with A$ = "RUN"+CHR$(13) will make a program restart automatically, which is particularly useful after the SAVEM statement in machine language loaders. But be careful, the length of A$ must be less than 32 characters.

Here's a quickie poke that will initialize the RND seed based on the time of day:

```
J = 61987:FOR I = 63277 TO 63280:POKE I,16*PEEK(J) + PEEK(J+1):J = J+2:NEXT
```

And, this one will re-seed the random number generator with one of 150 possible random values obtained from a running counter: (thanks to Larry Gensch)

```
POKE 63277,PEEK(61983)
```

If you wish to test the printer port to determine if the printer is powered up and ready to accept data, you can use the following statement in any BASIC program:

```
IF (INP(187)AND6) <> 2 THEN BEEP : PRINT "Printer Not Ready" : STOP
```

Or you can devise alternate tests: a 0 means the printer is not ready; a 2 means it is; and a 6 means it is not connected.

```
IF (INP(187)AND6) = 0 THEN (Printer not ready)
IF (INP(187)AND6) = 2 THEN (Printer ready)
IF (INP(187)AND6) = 6 THEN (Printer not connected)
```

Of course you realize, any address which can be "poked" can also be "peeked" to determine what is currently happening at that location, or what will happen depending on the value set at that address.

CHAPTER 24

Some Additional Useful Pokes

This chapter describes how to use the unused function keys in TELCOM, in both the 100/102 and Tandy 200, to enable certain functions, calls or jumps; such as clearing the screen, listing files or free RAM space, or jumping to BASIC. Some interesting possibilities become available once you know about the jump table addresses.

Both the 100/102 and Tandy 200 contain a vector table, which is referenced when there are interrupts generated in the computer. Some useful function key interrupts are in this table. The vectors tell the computer to jump to another location, usually a ROM routine, which does something, then returns to the location where the interrupt was originally generated. TELCOM has two such interrupt vectors in the table, for the F6 and F7 function keys in the TERM mode of operation. Pressing either F6 or F7 causes the computer to look at the table, to see where it is supposed to jump to. Normally the vector table holds a RETurn instruction, so pressing the key does nothing. But you can cause these keys to do other things, by redirecting control to other ROM routines by poking new addresses into the vector table. These "vector addresses" are often called "system hooks", because they allow assembly language programmers to hook into the normal operating system routines.

One easy change, a couple of pokes, will allow a user to clear his screen when in the TELCOM program, by pressing either F6 or F7, providing that no conflicting machine language program is being used, such as the Holmes Chipmunk's CDOS, Acroatix' POWR-DISK, or Traveling Software's TS-DOS; or any other program which alters the original function key hooks in the vector table. Users with the above programs cannot use this technique, unless the program conflicts are resolved first.

The pokes effectively reroute the function key interrrupt routine, so as to pass the operating system control through the chosen routine, whenever the function key is pressed, and then return to the normal keyboard scan routines.

Bottom line, the pokes stick a CALL address into the vector table that tells the computer where to go when the key is pressed. Using this same technique, you can cause the computer to execute any documented CALL when the function key is pressed. In TELCOM, in the Model 100/102, F6 and F7 are available. In the 200, F6 is available, F7 being used for a "Break" function - sending a "Break" signal through the modem, but the same general technique applies... you can use the function key to branch to any known CALL function, by converting the known address of the CALL to low-order/high-order form, and poking the resulting numbers into the vector table at the indicated addresses. If you don't use the

F7 key, you can redefine it's function, too, but you won't be able to remove the "Brk" letters above the key. Unfortunately, the label line is defined in the ROM, and there is no convenient way to add letters for your new functions, or to remove letters from the display without additional assembly language programming.

The addresses for some common CALL functions are given in the Model 100 Technical Reference manual, and many more have been located and defined by users. Still more can be can be gleaned from reading available ROM/RAM maps, including the one given later in this book.

For the Model 100: (to enable Clear Screen)

```
    For F6:
      POKE 64268,49 : POKE 64269,66

    For F7:
      POKE 64270,49 : POKE 64271,66
```

Note that the numbers 49 and 66 are the low/high-order address for the CALL function which clears the screen and homes the cursor. 64268/64269 is the F6 vector address, and 64270/64271 is the F7 vector address. Also note that two-part pokes like this should be entered on a single line in BASIC.

Here are some other functions you might like to use, instead of the Clear Screen command.

Clear Screen:	49 and 66	(CALL 16945)
Do Nothing:	243 and 127	(CALL 32755)
Files:	58 and 31	(CALL 7994)
Free RAM:	172 and 126	(CALL 32428)
Reverse Video:	105 and 66	(CALL 17001)
Normal Video:	110 and 66	(CALL 17006)
Jump to BASIC:	73 and 108	(CALL 27721)
Motor On:	168 and 20	(CALL 5288)
Motor Off:	170 and 20	(CALL 5290)

For the Tandy 200: (to enable Clear Screen)

```
    For F6:
      POKE 62783,77 : POKE 62784,79

    For F7:
      POKE 62785,77 : POKE 62786,79
```

62783/62784 is the F6 vector address, and 62785/62786 is the F7 vector address.

Alternate functions:

Clear Screen:	77 and 79	(CALL 20301)
Do Nothing:	168 and 156	(CALL 40104)
Files:	42 and 42	(CALL 10794)

Free RAM:	253 and 154	(CALL 39677)
Jump to BASIC:	143 and 131	(CALL 33679)
Motor On:	192 and 21	(CALL 5568)
Motor Off:	194 and 21	(CALL 5570)

Note that in the above groups, where the numbers are given as "194 and 21", the first number, "194" is poked into the first address given for the vector table address, and the second number, "21", is poked into the second address.

Don't know how to break a CALL address down into the low/high format? Simple; use this short program. You give it the known decimal address of the ROM routine, and it gives you the two numbers to poke into the vector table.

```
0 ' Decimal to Lo/High Hex Conversion
10 CLS:PRINT
20 INPUT" What number";N:PRINT
30 X1=INT(N/256):X2=N-(X1*256)
40 PRINTX2;X1:PRINT
```

CHAPTER 25

High Memory Map

This chapter lists most of the addresses in the reserved area of RAM above the MAXFILES partition. This is the most useful area for peeking and poking.

Addresses followed by a + sign indicate that the address is the first part of a two-byte data field. In cases where hexidecimal numbers are stored, the low order byte is stored first, followed by the high order byte, as in standard assembly language hex notation.

Addresses with a plus sign, followed by a number indicate that the following "n" number of bytes are part of the same field. Example: 61234+7 indicates that address 61234 and the following 7 bytes comprise one 8-byte group of data.

Addresses with a plus sign, followed by a question mark indicate that the address is the start of a field, the size of which is not documented.

100/102	200	Description
62960	61104	Normal MAXRAM address
62960+	61104+	Stores MAXRAM address
62964+	61108+	Holds HIMEM address
62969+2	61113+2	RST 5.5 vector
62972+2	61119+2	RST 6.5 vector
62975+2	61122+2	RST 7.5 vector
62978+2		Low power trap routine
62981+30		Option ROM code
63012+6	61167+5	Option ROM code
63019	61172	Dialing flag

63024	61181	F1 key on/off flag
63025	61182	F2 key on/off flag
63026	61183	F3 key on/off flag
63027	61184	F4 key on/off flag
63028	61185	F5 key on/off flag
63029	61186	F6 key on/off flag
63030	61187	F7 key on/off flag
63031	61188	F8 key on/off flag
63032	61189	Current screen in use
63033	61190	Cursor position - Row, 0-7
63034	61191	Cursor position - Column, 0-39
63035	61192	Number of active lines on screen, 100/102, 0-8; 200, 0-16
63036	61193	Screen width; 0-40
63037	61194	Label line on/off flag; 0=off, 255=on
63038	61195	Screen scroll flag; 0=scroll, 191=scroll locked
63049		Cursor flag; 0=off, 1=on
63040		Cursor line number
63041		Horizontal Print position
63048		Reverse video flag; 0=normal, 1=reverse
63054		X pixel set location
63055		Y pixel set location
63056	61234	Break, ^C, and Function key flag; 0=enabled, 128=disabled
63058	61236	BASIC error trap
63060	61238	Number of interrupts pending
63063	61240	Power down setting
63064	61241	Duplex flag; 0=Half, 255=Full
63065	61242	Printer Echo flag; 0=Off, 255=On
63066	61243	TELCOM's LF after CR switch; 0=off, 1=on

63067	61244	TELCOM device/baud rate; 77=Modem, 53="5" (1200), etc.
63068	61245	Word length (bits); 55="7", 56="8"
63069	61246	Parity; 69="E", 73="I", 78="N", 79="O"
63070	61247	Stop bits; 49="1", 50="2"
63071	61248	Xon/Xoff flag; 69=enabled, 68=disabled
	61249	Character filter flag
	61250	Incoming linefeed flag
	61252	Originate/Answer flag
63073+	61257+	CALL target address
63079	61266	OUT command code
63082	61269	INP command code
63092		Value of LPOS
63093	61280	Printer/screen output flag; 0=lcd, 1=lpt
63096+		Top of available RAM
63098	61285	BASIC line number being executed
63100+	61287+	Starting address of current BASIC program
63104	61291	End of BASIC statement marker
63105	61292	Multi-purpose buffer; a tokenized BASIC line starts here
63109	61296	General-purpose Buffer holds line from INPUT routines
63368	61555	Holds value of POS
63369+15	61556+15	F1 definition, table 1
63385+15	61572+15	F2 definition, table 1
63401+15	61588+15	F3 definition, table 1
63417+15	61604+15	F4 definition, table 1
63433+15	61620+15	F5 definition, table 1
63449+15	61636+15	F6 definition, table 1
63465+15	61652+15	F7 definition, table 1
63481+15	61668+15	F8 definition, table 1

63498+15	61685+15	F1 definition, table 2
63514+15	61701+15	F2 definition, table 2
63530+15	61717+15	F3 definition, table 2
63546+15	61733+15	F4 definition, table 2
63562+15	61749+15	F5 definition, table 2
63578+15	61765+15	F6 definition, table 2
63594+15	61781+15	F7 definition, table 2
63610+15	61797+15	F8 definition, table 2
63785	61977	Day of the month, second digit; integer, 0-9
63786	61978	Day of the month, first digit; integer, 0-3
63787	61979	Day of the week; integer, 1=Mon, 7=Sun
63789	61681	Year, second digit; integer, 0-9
63790	61682	Year, first digit; integer, 0-9
63791	61983	Cursor timer, counts down from 125 to 1 continuously In the 200, counter runs from 150 down to 1
63792	61984	Slower counter, counts down from 12 to 1 continuously
63793	61985	Power down counter
63795		Seconds, second digit, 0-9
63796		Seconds, first digit, 0-5
63797		Minutes, second digit, 0-9
63798		Minutes, first digit, 0-5
63799		Hours, second digit, 0-9
63800		Hours, first digit, 0-2
63805+5		ON TIME$ time, in reversed order, 6 digits
63812	62004	ON COM GOSUB status flag
63813+	62005+	ON COM GOSUB address
63815	62007	ON TIME$ GOSUB status flag
63816+	62008+	ON TIME$ GOSUB address

63818	62010	ON KEY1 GOSUB status flag
63819+	62011+	ON KEY1 GOSUB address
63821	62013	ON KEY2 GOSUB status flag
63822+	62014+	ON KEY2 GOSUB address
63824	62016	ON KEY3 GOSUB status flag
63825+	62017+	ON KEY3 GOSUB address
63827	62019	ON KEY4 GOSUB status flag
63828+	62020+	ON KEY4 GOSUB address
63830	62022	ON KEY5 GOSUB status flag
63831+	62023+	ON KEY5 GOSUB address
63833	62025	ON KEY6 GOSUB status flag
63834+	62026+	ON KEY6 GOSUB address
63836	62028	ON KEY7 GOSUB status flag
63837+	62029+	ON KEY7 GOSUB address
63839	62031	ON KEY8 GOSUB status flag
63840+	62032+	ON KEY8 GOSUB address
63842	62034	* Start of RAM Directory; 11 byte slots The first byte in the slot is the file attribute byte; the second and third byte is the file's starting location in RAM, bytes 3-11 are the filename and type
63842	62034	BASIC
63853	62045	TEXT
63864	62056	TELCOM
63875	62067	ADDRSS
63886	62078	SCHEDL
63897	62100	Unsaved BASIC program buffer
63908	62111	Paste Buffer
63919	62122	BASIC's Edit buffer
63930	62133	User's file #1
63941	62144	User's file #2

63952	62155	User's file #3
63963	62166	User's file #4
63974	62177	User's file #5
63985	62188	User's file #6
63996	62199	User's file #7
64007	62210	User's file #8
64018	62221	User's file #9
64029	62232	User's file #10
64040	62243	User's file #11
64051	62254	User's file #12
64062	62265	User's file #13
64073	62276	User's file #14
64084	62287	User's file #15
64095	62298	User's file #16
64106	62309	User's file #17
64117	62320	User's file #18
64128	62331	User's file #19
	62342	User's file #20
	62353	User's file #21
	62364	User's file #22
	62375	User's file #23
	62386	User's file #24
	62397	User's file #25
	62408	User's file #26
	62419	User's file #27
	62430	User's file #28
	62441	User's file #29
	62452	User's file #30
	62463	User's file #31
	62474	User's file #32
	62485	User's file #33
	62496	User's file #34
	62507	User's file #35
	62518	User's file #36
	62529	User's file #37
	62540	User's file #38
	62551	User's file #39
	62562	User's file #40
	62573	User's file #41
	62584	User's file #42
	62595	User's file #43
	62606	User's file #44
	62617	User's file #45
	62628	User's file #46
64139	62639	End of Directory flag
63898	62651	Directory address of current BASIC program
64173	61194	Label line flag; 0=disable, 1=enable
64175+7	62685+7	IPL Program name String
64190	62700	Temporary storage of Stack pointer address
64192+	62702+	Lowest address available in RAM

64201	62711	offset into jump table
64206+	62715+	"Top" address is stored here after a LOADM
64208	62717	Length of CLOAD/CSAVEd program
64218	62727+95	Start of interrupt vector table; 2 byte steps
64357		Basic variable type
64359+	63002+	Pointer to file buffers
64377+25		26 bytes, variable type for each letter of the alphabet
64404+		Line number of active data statement
64409+		Location of BASIC variable for assignment (LET=)
64411+		Pointer to start of current BASIC statement
64415+	63058+	BASIC line number where last error occurred
64417+	63060+	Most recently listedor entered line number
64419+	63062+	Address where error occurred
64421+	63064+	Address of ON ERROR GOTO line
64423		Error status flag
64424		End of BASIC expression
64426+	63069+	Line number where a break occurred
64428+		Location after error
64430+	63073+	Pointer to start of .DO file area
64432+	63075+	Pointer to start of .CO files
64434+	63077+	Pointer to start of variable storage
64436+	63079+	Pointer to start of array table
64438+	63081+	Pointer to first free byte in RAM
64536		Floating point arithmetic accumulators; to 64543
64617		Floating point arithmetic accumulators; to 64624
64642	63285	MAXFILES value
64659+8	63302+8	Name of current BASIC program; null at end
64668+8	63311+8	Name of last file found or loaded from tape; null at end

64704	63408	Start of Previous Screen buffer
64904+?		Start of DATE$ and TIME$ strings used by Menu program temporarily stored in PREV screen buffer
65023	64047	End of Previous screen buffer
65024	64048	Start of LCD image buffer
65343	64687	End of LCD image buffer
65346	64754	Xon/Xoff flag; 0=off, 1=on
65348	64756	Sound on/off flag; 0=on, 175=off
65349	64757	Cassette on/off flag
65350+63		RS-232 buffer, 64 bytes
65414	64757	Number of unread bytes in buffer
65416+		Pointer to buffer position
65431		Binary indicator of certain keys being pressed
65432		Pressing a function key sets the corresponding bit here
65442		Binary indicator of certain keys being pressed
65450	64798	Number of characters waiting in the keyboard buffer
65451+63	64799+63	Keyboard buffer holds 32 characters and 32 function key markers in alternate bytes

Undocumented, but equivalent addresses:

100/102	200	100/102	200	100/102	200
63062	61239	64634	63277	64658	63301
63090	61275	64652	63295	64679	63322
63099	61286	64264	62775	64929	63433
63794	61986	64266	62777	64981	63701
64194	62704	64300	62813	64983	63537
64195	62705	64306	62819	65006	63560
64198	62708	64308	62821	65007	63561
64210	62719	64310	62823	65423	64771
64216	62725	64312	62825	65433	64781
64220	62729	64396	63039	65448	64796
64240	62751	64413	63056	65523	65185
64244	62755	64440	63083		
64256	62767	64488	63131		

CHAPTER 26

Removing ROM names from the Menu

This chapter provides the necessary POKEs to remove unwanted ROM Program names from the computer's main menu.

It is often useful to remove unused ROM Program names from the computer's main menu. An example might be if you don't use specific programs, and just want to unclutter the menu. This subject has been touched on before in this book, but this chapter gives the specifics.

It is not necessary to list the RAM directory in order to remove specific ROM program names from the menu, but it would be necessary if you want to hide a BASIC program or file by making it invisible. In that case, use one of the RAMDIR programs from chapter 17, whichever is appropriate for your machine, to find the necessary address to poke.

To make program names, such as TEXT, TELCOM, ADDRSS, SCHEDL or MSPLAN invisible (if you don't use them), use the following pokes in direct mode in BASIC. It is not recommended that you remove the BASIC program name from the menu; if you have a lockup problem, it may not be possible to enter BASIC to recover control if it is not available on the menu, particularly in the 200, which does not allow direct typed input at the main menu to access files or programs.

For the Model 100/102:

```
POKE 63853,184   (deletes TEXT name)
POKE 63864,184   (deletes TELCOM name)
POKE 63875,184   (deletes ADDRSS name)
POKE 63886,184   (deletes SCHEDL name)
```

To recover use of any of the above programs, previously made invisible, just use the appropriate poke listed below in BASIC's direct mode:

```
POKE 63853,176   (recovers TEXT name)
POKE 63864,176   (recovers TELCOM name)
POKE 63875,176   (recovers ADDRSS name)
POKE 63886,176   (recovers SCHEDL name)
```

For the Tandy 200:

```
POKE 62045,184   (deletes TEXT name)
POKE 62056,184   (deletes TELCOM name)
POKE 62067,184   (deletes ADDRSS name)
POKE 62078,184   (deletes SCHEDL name)
POKE 62089,248   (deletes MSPLAN name)
```

To recover use of any of the above programs, previously made invisible, use the appropriate command listed below in BASIC's direct mode:

```
POKE 62045,176   (recovers TEXT name)
POKE 62056,176   (recovers TELCOM name)
POKE 62067,176   (recovers ADDRSS name)
POKE 62078,176   (recovers SCHEDL name)
POKE 62089,240   (recovers MSPLAN name)
```

Making individual files or programs invisible: Refer to Chapter 17.

CHAPTER 27

Lots of Useful CALL's

List of documented calls for Model 100 and Tandy 200, which may be used in BASIC, or in function key techniques.

The CALL statement is a very powerful tool, which can be used in either BASIC or Assembly Language programs, and works basically the same way in either. Essentially, it gives the BASIC programmer access to many of the built-in ROM routines, many of the same routines as used by machine-language programmers. The benefit of using a CALL to a ROM routine is the increase in speed caused by bypassing going through the BASIC interpreter, which has to interpret your statements before calling the same routines. The way a CALL works, is that it transfers control to the address CALLed, and somewhere at the end of the CALLed routine, presumably there is a RETurn, which will return control to the program or the keyboard, at the next executable command. If you were waiting for a key press before you issued the CALL, you will be waiting for a keypress again, after returning. If your CALL was an executable statement in a BASIC program, when you return, the program goes on to the next executable statement in the program.

The following function calls have been verified, and there are many others which are useful primarily in specific applications programs. These are some of the common ones that you can use right away in your own BASIC programs. Many of these calls can be combined with the technique shown in Chapter 22 to perform the function from a function key press in TELCOM. They can also be put on a function key in BASIC, to perform specific functions desired by the user.

In BASIC programs, use the form: CALL nnnn, where "nnnn" is the number listed in the chart.

To put a CALL statement on a function key in BASIC, select the key you want to use, and type the key defining statement directly as follows (assume you want to put CALL 36 on function key 5 to turn off the computer with a single key-press). Type:

```
Key 5,"CALL36"+CHR$(13)
```
and press the ENTER key.

The word "CALL" will appear above the [5] marks at the bottom of the screen when you press the ENTER key, indicating the key has been defined. You can use both upper and lower case letters in your definition.

In some cases, you might want the key to do the CALL, but call it something else on the screen. An example might be if you want a keypress to turn on the cassette recorder with a MOTORON command. In that case, "MOTORON" is too long to appear on the label line (only four letters can be displayed, and "MOTO" doesn't look too neat). In this case, you can put one thing on the display line, and hide the command out of sight. The technique to put "Cas" on the label line, for example, and turn on the cassette motor, would be:

```
Key 5,"Cas"+CHR$(24)+"MOTORON"+CHR$(13)
```

Note the CHR$(24) in there... when you press the function key, the definition is dumped into the keyboard buffer. As the macro starts to be placed in the keyboard buffer, CHR$(24), a CTRL-X, erases the characters back to the beginning of the line, so that MOTORON and a carriage return is all that gets into the buffer. You have to be careful with this technique, because the function key definition can only hold 15 actual characters. "Cas" is 3, CHR$(24) is 1, "MOTORON" is 7, and CHR$(13) is 1; a total of 12 characters.

The "*n" mark after some functions in the chart below, refers to the Notes at the bottom of the chart. Some functions are not really useful in TELCOM, but might be usfull in other situations.

Function	100/102	200
Turn Power Off:	36	5449
Cold Start:	32231	33820
Warm Start (Reset)	0	0
Reverse Video: *1	17001	20360
Normal Video: *1	17006	20365
Scroll Off: *1	16959	20318
Scroll On: *1	16964	20323
BEEP	16937	20293
List Files:	7994	10794
List Free RAM:	32428	39677
LLIST:	4411	
LCOPY:	7774	10566
Clear Screen:	16945	20301
Jump to Menu:	22423	26532
Jump to BASIC:	27721	33679
Jump to TELCOM:	20806	24573
Jump to TERM:	21589	25464
Jump to ADDRSS:	23400	28391
Jump to SCHEDL:	23407	28397
Jump to TEXT:	24046	29085
Connect phone line:	21200	25040
Disconnect phone line:	21179	25018
Dial a digit: *2	21514,n	25386,n
Send Xoff (^S)	28190	34327
Send Xon (^Q)	28171	34312
Motor On: *1	5288	5568
Motor Off: *1	5290	5570
Do Nothing (Return):	32755	40104
Send Char to Printer:*3	27967,n	23060,n

* Notes:

1. Recommended for Model 100 only because of the need to have one key to turn the function on (F6), and the second (F7) to turn it off.

2. The digit to be dialed is passed to the dialing routine by the statement.

3. The ascii value of the character to be printed is passed by the statement.

Miscellany:

In the 100/102, CALL 24367 causes the computer to wait for the space bar to be pressed before continuing, with no indication of why it's waiting. In the Tandy 200, it's CALL 29413.

For additional functions, refer to the ROM map which follows.

CHAPTER 28

Useful ROM Addresses

This chapter lists many of the ROM routines that are documented. Many are useful only to machine language programmers. Many others have been left out of this list because of their relative obscurity, and there was simply no point in including them, because it is easier to do what you want to do in BASIC rather than trying to do it with CALLs to ROM routines.

Notes: "message text" indicates a group of actual characters that are used in screen messages. Some equivalent addresses are indicated, which are useful for conversions of programs, but not identified as to function because they are not documented.

100/102	200	Function
0	0	Warm reset, RST 0
1	1	ID byte; 51 = 100, 167 = 102, 171 = 200
3	3	'MENU' message text
7	7	Null character, CHR$(0)
8		RST 1
30	30	Print a space
32	32	Print the character in the A register
128	128	Table of keywords used in BASIC
428	428	
796	796	BASIC error code table
1002	1045	'Error' message text
1009	1053	'in' message text
1014	1057	'Ok' message text
1019	1062	'Break' message text
1094	1137	Syntax error routines
1097	1140	divide by 0 error
1109	1152	ov error
1112	1155	mo error
1117	1160	error message based on content of E register
1147	1190	Return error
1245		Display error message
1281	1324	
1282	1325	
1520	1563	Builds BASIC line pointers
1523	1566	
1576	1619	Search for a BASIC line number
1606	1649	Tokenize BASIC text
2267	2310	Function Call error
2283	2326	Get Value of line to jump to on GOTO, GOSUB, RUN
2381	2424	Undefined Line error

3188	3231	`?Redo from start' message text
3441	3529	`Extra ignored' message text
4072	4115	Upper case conversion of character pointed to by HL
4073	4116	Upper case conversion of character in A register
4165	4207	
4335	4377	
4398	4440	Convert ascii value to integer
4514	4556	Prints from buffer terminated with 0, pointed to by HL
4811	4855	Get character from the keyboard routine
5029		Toggles the label line
5045	5086	
5083	5154	
5169	5449	Power off and resume
5232	5520	Send character to printer; expand tabs to spaces
5268	5548	
5288	5568	Turn on cassette motor relay
5290	5570	Turn off cassette motor relay
5296	5576	Get character from cassette and checksum
5313	5593	Send character to cassette and checksum
5342	5621	
5354	5633	
6118	6429	
6148	6152	
6415	6782	Read system time into buffer
6447	6814	Read system date into buffer
6498	6853	Read system day into buffer
6558	6913	
6965	7353	
6980	7368	
7089	9879	Reset power down counter
7774	10566	Dump screen content to printer (except graphics)
7963	10763	
7994	10794	Print Files routine
8126	10932	Kill a .DO file
8153	10958	
8215	11020	
8329	11174	
8335	11180	
8352	11197	
8367	11212	
8405	11250	Search directory for next valid file entry
8420	11273	Search directory for an empty file slot
8447	11292	
8500	11344	
8518	11362	
8698	11587	Get Length of a filename
8710	11635	Open a TEXT file
8719	11644	Make a TEXT file
8761	11692	Insert entry into directory
8780	11711	
9030	12467	
9205	12645	
9498	12938	
9533	12962	
9538	12967	Moves B bytes from HL to DE, increasing

9581	13010	Beep and jump to main menu
9636	13065	
9643	13072	
9685	13114	`Top:' message text
9691	13120	`End:' message text
9697	13126	`Exe:' message text
9984	13412	`Found:' message text
9989	13419	`Skip:' message text
10161	13591	Prints message pointed to by HL
11997	15445	
12006		Moves C bytes from HL to DE, decreasing
13392	16801	
13399	16808	
13417	16826	Move B bytes from DE to HL, increasing
13426		Move B bytes from DE to HL, decreasing
14072	17484	
14117	17529	
14206	17618	
14804	18187	Prints the value in HL
16162	19517	
16175	19530	
16625	19980	Check for upper-case character in A
16930	20286	Prints Carriage Return and Line Feed.
16937	20293	BEEP
16941	20297	Home Cursor
16945	20301	Clear screen and Home Cursor
16945	20301	Lock bottom screen line
16954	20313	Unlock bottom line
16959	20318	Scroll lock
16964	20323	Scroll unlock
16969	20328	Cursor on
16974	20333	Cursor off
16989	20348	Erase from cursor to end of current line
17001	20360	Reverse video on
17006	20365	Reverse video off
17008	20367	Print CHR$(27) plus the character in A
17015	20374	Move cursor to bottom left corner of screen
17020	20379	Position cursor
17034	20393	Clear label line
17061	20420	Set and display function keys
17064	20423	Display function keys (label line)
17082	20393	Erase function key display (label off)
17875	21499	
17982	21744	Display ? mark and get a line from keyboard
17988	21750	No prompt; get a line from keyboard
18130	21895	
18678	22442	
19268	23045	Print character in A
19285	23060	Send character in A to printer, expand tabs to spaces
19360		Send carriage return to printer
19384	23159	
19467	23242	Format filename and check validity
19585	23360	
19647	23422	
19730	23505	

19768	23543	
19801	23576	
20002	23777	
20007	23782	
20090	23862	
20106	23878	
20234	24001	Fill block of memory with binary zeros
20567	24334	File not found error
20806	24573	TELCOM main entry point
20928	24695	Stat routine
21060	24898	'Calling' message text
21179	25018	Disconnect modem from phone line
21220	25063	Connect modem to phone line
21264	25102	2 second time delay
21274	25112	1/2 second time delay
21293	25131	Dial a phone number
21514	25386	Dial a single digit
21589	25464	TERM entry point
22213	26210	
22361	26470	
22376	26485	
22385	26494	'aborted' message text
22396	26505	'No file' message text
22406	26515	'Disconnect' message text
22417	26526	prints message
22423	26532	Main Menu entry
22710	27215	
22723	27228	
22747	27244	
22754	27251	
22763	27260	
22896	27825	
22985	27985	
23013	28013	
23058	28055	Prints time and date at top of screen
23128	28150	Prints contents of buffer at HL, until a null is found
23138	28166	Move a block of memory
23161	28189	Clear function key definition table
23164	28192	Set function key definitions
23198	28226	Display function key table
23206	28234	
23209	28239	Search directory for a specified file
23211	28239	
23254	28278	
23267	28300	find storage location address of specified file
23268	28301	
23273	28306	'JanFebMarAprMayJunJulAugSepOctNovDec' message text
23322	26693	
23400	28391	ADDRSS main entry point
23407	28397	SCHEDL main entry point
23450		Prints 'Not found press space bar for menu'
23615	28738	
23661	28790	
23668	28797	
23726	28855	

23851	28979	
23891	29003	
23901	29013	
23920		Prints date and time at top of screen, updates it until any key is pressed
23985	29020	
23996	29035	
24005	29044	
24046	29085	TEXT main entry point
24356	29402	Prompt for press space bar
24367	29413	'Text ill-formed' message text
24395	29441	'Press space bar for TEXT' message text
24753	29842	'Memory full' message text
25446	30535	
25456	30545	
25475	30564	
27437	33384	
27489	33436	
27501	33448	
27551	33498	
27611	33558	Moves BC bytes from HL to DE, increasing
27622	33569	Moves BC bytes from HL to DE, decreasing
27721	33679	BASIC main entry point
27795	33747	Copy function key table 1 to table 2
27804	33756	Copy function key table 2 to table 1; Restores function keys
27815	33773	
27862	33820	Cold start reset
27967	33993	Send character to line printer
28013	34056	Check RS232 buffer for characters
28030	34073	Get next character from RS232 buffer
28171	34312	Send Xon character to COM port
28190	34327	Send Xoff character to COM port
28210	34371	Send character to RS232 port, honor Xon/Xoff protocol
28218	34383	
28363	34741	
28399	34634	Detect carrier signal
28869	35195	
28962	35288	
29124	35461	
29171	35508	
29172	35509	
29181	35518	
29250	35587	Puts ascii value of keypress in A, similar to INKEY$
29296	35633	
29315	35661	Check for break characters
29343	35689	
29361	35707	
29592	36068	
29756	36198	Turn on RST 7.5 Interrupt
30300	36768	Turn off and rearm RST 7.5 Interrupt
30326	36809	clicks the speaker
32428	39677	Prints number of free bytes in RAM
32555	39905	
32664	40014	'bytes free' message text
32755	40104	

Chapter 29

Resources

This chapter lists books and magazines that are dedicated to the care, feeding, and operation of the Model 100 family of computers, plus other information resources. Many books are no longer in print, and are included here for reference and research purposes only.

MAGAZINES:

Portable 100
P.O.Box 428
Peterborough NH 03458

100% dedicated to coverage of the Tandy portable and laptop computers. Heavy on Model 100/102 and 200. Occasional coverage on 600 and 1400. Available only by subscription. Eleven issues per year, $24.97 for 12 issues. Primary source of Model 100 support; highly recommended Back issues are available.

PCM
P.O.Box 209
Prospect KY 40059

During the period from 1983 to 1988, PCM had several articles, games, or programs in each issue. Beginning in 1988 this dwindled to one a month, then occasional articles. Back issues are a good source of material.

80 micro
P.O. Box 981
Farmingdale NY 11737

TRS-80 oriented magazine includes one or more articles for the M100 each month during the 1983 period until they discontinued publishing.

BOOKS:

TRS-80 Model 100 Portable Computer; aka THE TRS-80 Model 100 Owners Manual is still available from Radio Shack on special order, or from Tandy's National Parts Center hotline, (800) 342-2425. No catalog number, ask for it by name. $18.95. Similar manuals are available for the Tandy 102 and 200 computers.

TRS-80 Model 100 Technical Reference Manual; Radio Shack #26-3810. Available from Tandy/Radio Shack as above. $9.95. Similar manuals are available for the Tandy 102 and 200 computers.

The TRS-80 Model 100 Service Manual, available from Tandy/Radio Shack as above. Similar manuals are available for the Tandy 102 and 200 computers, the Disk/Video Interface (D/VI), the Tandy Portable Disk Drives, etc.

The Model 100 Book, A Guide to Portable Computing; from Osborne/McGraw- Hill, by Jonathan Erickson & Robert J. Sayre; Walden Books; 310 pgs; soft cover; $17.95

The Radio Shack Notebook Computer; from Sybex by Orson Kellogg; 118 pgs, soft cover; $9.95

The TRS-80 Model 100 Portable Computer; from Tandy/Radio Shack, by David A. Lien; 555 pages, soft cover; very good beginner's book, $14.95

The Tandy 200 Portable Computer; from Tandy/Radio Shack, by David A. Lien; 595 pages, soft cover; very good beginner's book, $19.95

Five book set: Introducing the TRS-80 Model 100; from The Waite Group, by Diane Burns and S. Venit.; 180 pgs, soft cover, $15.95

Mastering BASIC on the TRS-80 Model 100; from The Waite Group, by Bernd Enders; 340 pgs, soft cover; $19.95

Practical Finance on the TRS-80 Model 100; from The Waite Group, by S. Venit and Diane Burns; 166 pgs, soft cover; $15.95

Games and Utilities for the TRS-80 Model 100; from The Waite Group, by Ron Karr, Steven Olsen and Robert Lafore; 198 pgs, soft cover; $16.95

Hidden Powers of the TRS-80 Model 100; from The Waite Group, by Christopher L. Morgan; 244 pgs, soft cover; one of the best technical books. $19.95

The above five books are available seperately, although they constitute a "set". Although they plainly state "The Waite Group" on the cover, they are listed in the Books in Print catalog as "A Plume/Waite Book", and as such, are catalogged under "Plume" instead of "Waite".

Portable Computing with the Model 100; Radio Shack #26-3820; from Radio Shack by George Stewart; 171 pgs, soft cover book with program cassette; $14.95.

The Model 100 Companion, Business and Entertainment Programs for Portable Computing; The Editors of Osborne McGraw/Hill; 166 pgs, soft cover; $15.95.

The TRS-80 Model 100 and NEC PC-8201 Idea Book; Creative Computing Press by David H. Ahl; 141 pgs, soft cover; $8.95.

TRS-80 Model 100 - A User's Guide; TAB Books, by Joseph Coleman; 152 pgs, soft cover; $15.50.

The Simon & Schuster Guide to the TRS-80 Model 100; from Simon and Schuster by Danny Goodman; 221 pgs, soft cover; $9.95.

Financial Decision Making with your TRS-80 Model 100, including 18 Programs; from TAB Books by Leslie Sparks; 144 pgs, paperback; $9.95

Getting What You Want from the TRS-80 Model 100: BASIC Programming for Business; from Harper-Rowe by E. Paul Cone; 200 pages, paperback; $14.95

The TRS-80 Model 100 User's Guide; from Wiley Press; 160 pgs, paperback; $16.95

Sixty Business Applications Programs for the TRS-80 Model 100 Computer; from Scott, by Terry Kepner & Mark Robinson; paperback; $17.95 *

TRS-80 Model 100: The Micro Executive Workstation; from Reston, by Kenniston Lord; paperback; $16.95

The TRS-80 Model 100 Computer, The Micro Executive Workstation; from Reston Publishing Co, by Kenniston W. Lord; 258 pgs, soft cover; $15.95

Guide to TRS-80 Model 100 Utility Programs; from Wiley Press by Ronald Konkel; paperback; $15.95

How to Use the TRS-80 Model 100 Portable Computer; from Alfred Publishing by Robert K. Louden; 64 pgs, paperback; $2.95

Twenty-Five Games for your TRS-80 Model 100; from TAB Books by David Busch; 160 pgs, paperback; $9.95

Things to do with your TRS-80 Model 100 Computer; by Jerry Willis, Merl Miller and Cleborne Maddux; paperback; $3.95

Forty-Four Programs for the TRS-80 Model 100 Portable Computer; from ARCsoft, by Jim Cole; 96 pags; $8.95

TRS-80 Model 100 Subroutine Cookbook; from Brady Communications Co., by David Busch; 166 pgs, soft cover; $12.95

Computer of the Century: Radio Shack Model 100; from DATAMOST by William Sanders; paperback; $4.95

Executive Recreations for the TRS-80 Model 100; from Wiley Press by Ed Yasaki, Bob Albrecht and Karl Albrecht; 160 pgs, paperback; $14.95

User Guide & Applications for the TRS-80 Model 100 Portable Computer; from Scott, Foresman & Co., by Steven A. Schwartz; 124 pgs, paperback; $17.95; with Cassette, $25.00. *

Inside the Model 100; from Weber Systems, by Carl Oppedahl; 334 pgs, soft cover, one of the best, includes info on assembly language and advanced BASIC programming techniques. $21.95 *

Portable Programs: Games & Utilities for the TRS-80 Model 100; by R. Carr, S. Olson and R. Lafore; $9.95

Portable BASIC: BASIC Programming Guide for the TRS-80 Model 100; by Bernd Enders --- announced by publisher, never seen.

Portable Tutor: User's Guide to the TRS-80 Model 100; by Diane Burns --- announced by publisher, never seen.

Computing; from Osborne, McGraw/Hill, by the Editors of...; 150 pgs; $15.95

The Best of the Model 100 Forum; published by Compuserve Information Service, Special Interest Group/Forum: from CIS,
may be ordered on-line; 61 pgs, $9.95

Book & Cassette: TRS-80 Model 100 Basic Language Lab: Radio Shack Catalog # 26-3821; $29.95

Two Radio Shack Service Manuals for the M100: Catalog numbers: 26-3801 and 26-3802. Order from National Parts, price not known.

How to Do It on the TRS-80 by William J. Barden; additional details unknown, $29.95

Moonlighting with your Personal Computer from World Almanac Publications, by Robert Waxman; 159 pgs, soft cover; $7.95

Programming Tips, Peeks and Pokes For the Tandy Portable Computers, self-published by Tony B. Anderson, $14.95

The Model 100 Program Book, by Terry Kepner, publisher of Portable 100 magazine, $14.95 *

The Secrets of ROM Revealed; King Computer Services, by Mo Budlong. About 60 pages, soft cover, $39.95. An IBM/PC software support disk is available for an additonal $10.00.

Not directly related to the Model 100 family, but directly applicable, the beginners book on assembly language, Introduction to 8080/8085 Assembly Language Programming, A Self-teaching Guide, published by John Wiley and Sons, by Judi Fernandez and Ruth Ashley. 303 pgs, soft bound, $12.95.

* The books so marked are currently still available (late 1989) from Granite Street Portables, Box 651, Peterborough NH 03458.

PROGRAM COLLECTIONS:

P100-To-Go; A monthly collections of programs published in Portable 100 Magazine, on disk. Formats supported include the Tandy portable disk drive and the Holmes Chipmunk disk drive. P100-To-Go, Box 428,)Peterborough NH 03458. $9.95 each, or a six-month subscription for $47.70.

Public Domain Programs on disk, a different selection each month. Available on disk for the Tandy Portable Disk Drive, $7.00 per disk. Thomas Quindry, 6237 Windward Drive, Burke VA 22015. Mr. Quindry also writes a monthly article in Portable 100 magazine.

Programs for the Tandy 200 from Paul Globman, 9406 NW 48th Street, Sunrise FL 33351. Paul writes many innovative programs for the Tandy 200, and a monthly column in Portable 100 magazine.

Over 200 Programs and Information files for the 100/102/200 and 600. Complete catalog on disk, $5.00; formats available include Tandy portable disk drive, Holmes Chipmunk disk drive, MS-DOS 5.25 inch disk, and cassette. Additional services include custom ROM burning, hardware modifications, dual disk drive interfaces, modem cables, and other unique items, including EPROM-based software. Available from Tony Anderson, Post Office Box 60925, Reno NV 89506.

ONLINE RESOURCES:

CompuServe, The Model 100 Forum, available to CompuServe Subscribers. Almost 4,000 programs and files for the Tandy Portable Computers. An active message board where you can get technical help, advice and information 24 hours a day, often within hours, from a widely experienced group of users who provide all levels of support. Some of the best, currently active programmers frequent this forum. There is a signup and free time kit in each modem cable package sold by Tandy/Radio Shack, and you can also signup through an 800 phone number provided by CompuServe.

GEnie Information Service, The Laptop roundtable. This forum supports all laptops, and as such, Model 100/102/200 users might find it less interesting, and less useful. You hear very little about activity on the GEnie service.

Club 100, a private membership BBS, which features one of the largest collection of downloadable programs and information files in the country. A privately run BBS, it has restricted access due to the number of available phone lines. It's a long-distance call from everywhere except its local dialing area. They have recently (late 1989) been advertising program collections on disk through Portable 100 magazine. Based in the San Francisco area, call (415) 932-8856 for information and logon instructions. The club holds frequent membership meetings which are open to the public. Club 100, Box 23438, Pleasant Hill CA 94523.

Chapter 30

Loading and Running Machine-Language Programs

This chapter provides basic information about HIMEM and MAXRAM, and how they relate to loading and running machine (assembly) language programs.

New users of the Tandy portables are often confused about how to load machine language programs, what HIMEM and MAXRAM mean, and what relationship they have to loading and running the .CO program. This chapter started out as a response to those questions from a new user on the CompuServe Model 100 Forum's message board.

HIMEM - MAXRAM - What they are, and how they relate to use of .CO programs in the Tandy Portable Computers:

HIMEM is a movable partition which can be set to allow you to load and protect a machine language program in a reserved area of RAM just below MAXRAM, in an area between the HIMEM partition and MAXRAM. Normally, HIMEM = MAXRAM, but using the CLEAR,N,X command, you can move the HIMEM pointer to some address below MAXRAM, "X", then load a machine language program between HIMEM and MAXRAM, and the operating system will not overwrite the program by storing variables, the file buffers, or the stack pointer in that area. Such things are normally stored just under the HIMEM address. The "N" figure in the above CLEAR statement specifies a number of bytes which should be reserved for variable storage; usually it's 256, but it could be more, or less, depending on what you want.

MAXRAM is a partition that separates RAM space available to the user from the reserved area of RAM that is used by the computer's operating system - an area that stores flags, buffers, pointers, the file directory, and other information needed by the operating system. Machine language programs - .CO programs - are loaded and execute just under MAXRAM.

You can determine what your current settings are by going into BASIC and typing

```
PRINT HIMEM;MAXRAM
```

In order to load a machine language program, designated with the filename extension ".CO", first you go into BASIC and test to see if HIMEM = MAXRAM. Use the above command, PRINT HIMEM;MAXRAM and press the ENTER key.

If the two addresses are different, then you clear out any existing machine language program with the command CLEAR 256,MAXRAM. That resets the HIMEM partition back to it's default value, in effect unloading any machine language program that was already in that area. Then you attempt to load the .CO program you want to run with the command LOADM"(filename)". I say "attempt", because the program will not load because the HIMEM partition has not yet been set.

BASIC will return three addresses, "Top", "End", and "Exe". The display might look like this:

```
        Top: 60905
        End: 61103
        Exe: 60920
        ? OM error
```

The "error" indicates the program didn't load. It might be a different type of error - no matter, it means the program didn't load. If it did, you'd get "Ok" on the next line below the Exe address.

"Top" is the loading address of the program. "End" is where the program ends. "Exe" is usually the "executing" address, the place you enter when you run the program. Exe is often the same as Top. Note what the "Top" address is and ignore the others, they aren't of interest to you at this point.

Now type CLEAR 256,"top", putting the actual top address where I indicated "top". A typical clear might be CLEAR 256,58000, if 58000 was the Top address indicated above.

You can now LOADM the .CO program again, and it will load properly. You can return to the menu, move the cursor over the .CO filename and press ENTER to run the program, or you can type RUNM"(filename)", which will load and run the program automatically.

HIMEM will remain where you put it, and the machine language program should remain loaded for further use until you set HIMEM to something else, load another program, or unless it gets corrupted by some other program. Depends on how the .CO program was written, and what other programs may do to the HIMEM pointer, or what may be loaded into the reserved area, or what might be poked into the reserved area. It is possible, although not suggested, that several different machine language programs might run in the same area, each one overwriting the previous one. This could lead to problems, so is not advised unless you really know what you're doing.

Machine language programs which are designed to run in the computers alternate screen buffer do not need to have an area reserved for them, and you do not have to set HIMEM. Once such programs are on your menu as a .CO program, all you have to do is move the cursor over the filename and press the ENTER key. They can also be LOADMed from BASIC, and run with a call to the "Exe" address, or they can be RUNMed from BASIC. Some of these programs can be used as subroutines from other programs, check the documentation or check with the author for information on those applications.

For additional information, refer to the Tandy BASIC Reference Manual which came with your computer. It's in the form of a dictionary, more or less, and you can look up the words you want more information on, such as HIMEM, MAXRAM, LOADM, RUNM, CLEAR, etc.

Chapter 31

CASSETTE USE

This chapter deals with use of cassette recorders for data storage, explores some of the problems, provides some insights and tips, and recommends upgrading to a disk drive.

Use of a Cassette for data storage, or for distribution of programs has always presented problems, from the earliest days of personal computers. They are unreliable, temperamental, and often can not be exchanged from one user to another due to differences in the machines used to record, and those used to playback the data tapes.

Beginners often think that data storage on standard cassettes is a routine, money-saving method of saving programs and files for personal use, and attempt to save money by using an existing cassette recorder that they may currently have available. Frequently, after trying to save and load files unsuccessfully from the cassettes, they give up in frustration, and either sell the computer or put it in storage, moving on to another computer that "works right" with cassettes. -- It's not the computer, guys, it's the cassette recorder!

Several manufacturers have attempted to optimize cassette machines for use as data storage devices, but the basic problem remains, data is basically a square wave signal, digital, and the cassette medium is basically an analog signal device, used to store sine waves - audio signals, which are seldom optimized for digital use. The major problem lies in setting useful recording and playback levels. Cassette recorders often feature automatic level control, which makes recording of computer data signals very difficult, as the volume keeps changing, going up and down, depending on input level. Then too, the computer puts out line-level signals, and most cassette recorders are designed to accept microphone-level signals on their input jacks. The input signal simply overloads the recorder, recording a distorted signal that cannot be recovered.

For the Tandy portable computers, Tandy/Radio Shack sells two cassette machines that are "optimized" for computer use, the CCR-81 and CCR-82. They are priced at, or around $59.95, and are often available on sale at even lower prices. Each machine has a fixed reference point for setting recording and playback levels, and are signal-matched to what the computer is sending. They also do not have that automatic recording level or automatic volume control problem.

If you are not using one of the Radio Shack cassette machines, the 81 or 82, you may have "temperamental" problems recovering the data from cassettes; sometimes it will work, sometimes it won't. If you use another recorder, it is very important that it have a remote start/stop feature, since the computer needs to stop the dataflow from the tape every so often when loading files, and also start and stop the tape when saving files. If you use one of the Radio Shack models, there is a fixed playback volume level that is used to play the data back into the computer. If you use some other machine, you may have to experiment with volume level, tone, or dolby settings, as they are all inter-related to some extent, in order to make it work. In any event, powering the cassette recorder from AC may help, and going into BASIC and typing SOUND OFF and pressing the ENTER key may also help by eliminating the annoying screeching sound as the data or program is loaded.

Distributors of programs on cassette generally assume you know how to load programs and files from cassette into your machine. If you're new to cassette use, here are some short instructions of how to do it.

Files on cassette may be in three forms, just like they can be in the computer. They can be .BA files (BASIC programs), .DO files (Documents), or .CO files (machine language programs). Information about the programs on a cassette will generally be noted on the cassette label.

.BA files, programs, are loaded into BASIC. Connect the cassette player to the computer using the standard cassette cable. Insert the cassette into the player and press the PLAY button. The cassette should not start yet. Enter BASIC, press the F2 button, and at the Load " prompt, type 'CAS:' and press the ENTER key. The tape should start now. BASIC will report 'Found:' and give the name of the file it found. If it was not a program, instead of Found, BASIC will report 'Skip:' and tell you the name of the file it is skipping. When a program is found, it is loaded into BASIC. It may take a few minutes, cassettes are very slow, and that's another reason they aren't looked on with any great favor. The cassette may start and stop several times during the loading process. Once the file is loaded, the cassette will stop, and you will get the 'Ok' prompt. You can now press F3 to Save the file under the name you specify. Once saved, it will appear on your computer Menu, and you can Run it by moving the wide cursor over the program name and pressing the ENTER key.

BASIC programs can also be loaded with the direct command CLOAD"filename". If no filename is specified, the first program on the tape will be loaded. You can add ",R" to the end of the command to automatically run the loaded program after it is loaded.

Machine Language programs, with a .CO extension, can be loaded with the CLOADM command, which works much like the CLOAD command for BASIC programs.

.DO files (documents) are loaded into TEXT. To load files, setup the cassette machine as before, insert the cassette and press the PLAY button. Then go into TEXT and press F2. At the prompt 'Load from:' type 'CAS:' and press the ENTER key. The cassette will start, and the first file that is found that is a .DO file will be loaded into TEXT. It may take a few minutes to load, depending on size. A 10K file may take up to four minutes. Be patient. While the file is loading, nothing happens on your screen. When the file is completely loaded it will suddenly appear and the 'Load from:' message at the bottom of the screen will disappear.

If you are loading several .DO files, such as might be stored on tape as a collection of data files, you can load each file into TEXT, and once the file is displayed, you can press the Shift key and push the PRINT button at the same time (called SHIFT-PRINT) to send a copy of the file to your printer. When you press Shift-PRINT, you get a prompt on the bottom line of the screen for width. You can type 80 for an 80 column printer, but unless you have an odd printer, you can just press the ENTER key and the file will be printed in 80-column format anyway, since that is the way most files are already formatted on the tape. On the Tandy 200, you have more printing options with Shift-PRINT, including some basic formatting capabilities.

Once you have a paper copy of the file, you can go into BASIC and kill the .DO file with the command `KILL"filename.DO', return to the menu by pressing F8, go back into TEXT and load the next file from the cassette the same way. (In the Tandy 200, you can kill the file directly at the main menu by pressing F5) In TEXT, press F2, and type CAS: and press the ENTER key. The cassette will load the next file from the tape. It is not necessary to know the file names prior to loading them, using this method.

It sometimes happens that a cassette simply isn't readable. It may be possible to recover some, most, or all of the data on the tape by using a BASIC program that opens a file on the cassette, and reads data from the tape line by line, or character by character. There are also some data recovery utility programs available, mainly on various BBS's and services.

As a final word of advice, allow me to strongly urge you to upgrade to a disk drive, and get away from cassette use. As handy as they are, cassettes just aren't up to reliable data storage. The Tandy/Radio Shack disk drive is a viable upgrade, is often available at less than catalog price, either on sale or by mail order; and is often half-price or less for used drives. Upgrading to disk will change the way you use your computer, and will make it much more productive. For one thing, you can keep programs and files on disk, and only load those programs or files you will be using at any given time, keeping your RAM space mostly empty, allowing more space for data files which may be used by your programs. There are also programs available that will backup the entire contents of RAM to disk, and DOS's that allow you to read and write directly to disk files from your own programs. You can save an entire bank of programs and files to disk, and load another set, expanding the capabilities of your machine as if you had a multi-bank machine. One disk will hold between three and six entire banks worth of data and programs, so you can switch various work efforts in and out of RAM easily. You could have one disk for business related files, one for personal stuff, one for games, notes, research, or whatever. Working with disk is so much easier, and you won't have to fight with the machine to make it work properly, like you do with cassettes. Additionally files can be saved and loaded by name - you don't have to wait for the cassette recorder to find the file and load or save it.

A shirt-pocket full of disks can hold over a megabyte of programs and data... anywhere up to 400 programs and files on five to ten disks. It's well worth making the change for only a couple hundred bucks.

ADDENDA

Production Notes:

This book was written, edited and produced entirely on a Tandy 200 computer, with the exception of text formatting, and page numbering, which was done in a PC. The manuscript copy was printed on a Qume daisy-wheel printer with an Ascii-96 character typewheel.

Program disk:

A program disk, containing a copy of each program listed in the book is available for $5.00, postpaid. Each program is referenced by page number.

Disks are available in the following formats: 3.5 inch for the Tandy Portable Disk drive (100K or 200K models), 3.5 inch Chipmunk disk, and MS-DOS 5.25 inch disk. Order the Programming Tip Book disk, and specify which format you want. Each disk contains the following programs or routines:

Page	Disk Name	Program Description
2	P2-1.BA	Substring Pointers #1
2	P2-2.BA	Substring Pointers #2
4	P4.BA	Date calculator
11	P11.BA	Sketch.100
12	P12.BA	Which file is loaded program
13	P13.BA	Which file, version #2
14	P14.BA	Date input routine
17	P17.BA	Kill the program called NAME.BA routine
19	P19.BA	Looking at Pixels program
21	P21.DO	Eleven routines in a text file; cut and paste to separate
29	P29.BA	Random Generator test
31	P31.BA	Program to text for ideal distribution of numbers
32	P32.BA	Bottom line demo program
40	P40-1.BA	Directory search, program #1
40	P40-2.BA	Directory poke, program #2
41	P41.BA	RAMDIR.100 Ram directory listing program for 100/102
42	P42.BA	RAMDIR.200 Ram directory listing program for the 200
44	P44.BA	Break key disable test
45	P45-1.BA	Lenny's password program
45	P45-2.BA	Improved password program
48	P48-1.BA	CVTDTA.BA Converts data statements in decimal form
48	P48-2.BA	CVTHEX.BA Converts data statements in hex form
49	P49.BA	DATA-D.BA Data disassembler
52	P52.BA	TELFF2.BA TELCOM formfeed program for the 200
53	P53.BA	TELFF1.BA TELCOM formfeed program for the 100/102
59	P59.DO	Two programs for the 100/102
64	P64.DO	Two programs for the 200
69	P69.BA	Decimal to Lo/High Hex conversion
70	P70.DO	High memory map for searching and reference
83	P83.DO	ROM address map for searching and reference