TRS-80 INFORMATION SERIES - VOLUME IV

Lewis Rosenfelder

BASIC FASTER AND BETTER & OTHER MYSTERIES

A guided tour of **BASIC** programming tricks and techniques

BASIC Faster & Better & Other Mysteries

Written by Lewis Rosenfelder

Edited by Jim Perry

Technical Editor David Moore

Graphics by John Teal

Cover Design by Harvard Pennington

ISBN 0 936200 03 0

First Edition
Fourth Printing
June 1982

Published by
IJG Inc
1953 West
11th Street
Upland,CA
91786 (714)
946-5805

Contents

Acknowledgements

This book was produced with the aid of several *Radio Shack* TRS-80's (Model 1's, 2's, and 3's); an *LNW-80* computer; a *LOBO* expansion interface; a mixture of 35-, 40-, and 77-track disk drives; an *NEC Spinterm* printer; an *Epson MX-80* printer; the *Electric Pencil 2.0*; *Scripsit*; a special type translation program; an *Autologic Micro 5* typesetter (at *Pacesetting Services*, Anaheim, CA.); *LDOS*; *NEWDOS+*; and *NEWDOS-80*.

Most books take a year or more to change from manuscript into final book form. The book you are now reading took less than 3 months. Part of the reason is the technology used (typesetting directly from the original files), but the main reason is the cooperation, and hard work, of several special people. I would like them all to stand, and take a bow:

Lewis Rosenfelder (the author) – for having the skill, perception, and perserverance, needed to research and write this book in the first place.

David Knoch (of Pacesetting Services) – for literally giving me the keys to his business, and letting me 'play' with a hundred-thousand-dollars worth of typesetting equipment.

David Moore (technical editor) – for only making the same mistakes once, he learns fast! Denny Steele – for the main translation software. Mike Wagner – for the machine language interface. Kip Pennington – for making coffee at 7 am, and volunteering for everything.

Harv Pennington – for letting us get on with the job. Bruce – for keeping the ship afloat. And, by no means least, Al Krug – for keeping Lewis afloat!

Thanks to all of you,

Jim Perry,
Editor

Preface

The TRS-80 is a powerful computer . . . I've had mine for more than three years now, and each day I become more convinced of this.

You'd think that with a low-cost, mass-produced, computer you'd soon become frustrated by its limitations. I've found that the opposite is true. Each day I become more and more impressed with its capabilities.

Learning to program a computer is like learning to play the piano. It's easy to play simple melodies from the very first day, but you can spend a lifetime improving your technique and expanding your repertoire.

I started out with the TRS-80, probably much the same way you did, with this simple program . . .

```
10 PRINT"HELLO THERE. I AM YOUR NEW TRS-80 MICROCOMPUTER."
```

From that point to this day, I've spent almost every waking hour in front of my computer, or at least thinking about ways to make it perform better and faster. I even dream about GOSUBS, FOR-NEXT loops, PEEKS and POKES!

I remember the first time I ever saw a TRS-80, back in December of 1978. I walked into a Radio Shack and asked for a demo. I may not have said it, but my original attitude was: *"You call that a computer? Huh!"*.

A few days later I gathered up my credit cards and bought one. I wanted to get into the software business, and I figured that, whether or not the TRS-80 was any good, Radio Shack would sell thousands of them, and there just might be an opportunity. As it turned out, the TRS-80 is a fantastic computer, and Radio Shack has sold hundreds of thousands of them!

My background was as a mini-computer and accounting machine salesman for one of the largest and oldest computer manufacturers. So I knew accounting applications and a little COBOL and assembly language. Having knocked on hundreds of doors trying to sell computers, I had a good understanding of what small business owners need and want. Having been involved in the installation and operator training for dozens of computer systems, I was well aware of the 'real-world' design requirements in making computer systems 'water-tight' and operator-oriented. In summary, I thought I was going to make a fortune selling TRS-80 programs.

Before long, I had developed several Level I programs that did some cash flow planning, inventory, and manufacturing applications, and I took photos of the video display. I realized, that without disk drives and a line printer, the programs wouldn't be practical for use in business, but I showed the pictures to a few business owners, and the Radio Shack manager that sold me my computer. Within a few weeks, I had several orders for programs, which were to be delivered a few weeks after the disk drives and line printer became available.

Little did I know that Level II BASIC and disk programming would be a whole new ball game! By the time I got my disk drives and printer I was buried in orders, and I had grossly underestimated the time it would take to program and deliver the applications. Fortunately, thanks to the patience of my original customers, I was able to develop and deliver the programs.

This book is the result of the efforts I've made to make my BASIC programs run better and faster. Every time I'd have to stop and figure out a routine or technique, I'd put it in my programming notebook. Many times, I've had to throw out a routine and come up with an improvement, because the real test was whether or not it would work successfully on a day-to-day basis at a customer site.

You won't find any trivia here. Each routine and technique solves one or more specific problems that you are likely to encounter when programming the TRS-80. Every thing we'll discuss is pragmatic, with the goal of making the computer do what you want it to do, with the least programming effort.

You won't find any 'pretty-printed' subroutines or programs in this book. Each routine is packed so as to require the smallest amount of memory overhead in your program. Each routine is shown in 64-character lines, as it will appear on your video display, to simplify the entry into your computer. For standard subroutines, performance is the name of the game, and that's the approach this book takes.

The subroutines and techniques in this book don't attempt to be 'all things to all people'. I suppose it would be possible to write a sorting subroutine, or disk file-handling subroutine, that could handle every possible operation you might want to perform. But why sacrifice execution speed? Why waste the memory? Instead, this book gives you relatively flexible routines, with the documentation that will allow you to modify them as your application requires.

I hope you'll find this book as valuable to you as it is to me. I use it daily as a reference in my programming work. Though some of the information can be found elsewhere, this book gives you a handy 'one-source' reference. And, now that these routines and techniques are explained in book format, my documentation efforts for any system I write are greatly simplified. I can now refer anyone who reads one of my program listings back to this book, instead of filling up the program with memory-wasting remarks. If you adopt the same techniques and standards, you too can save a lot of time on documentation. You will be free to concentrate on the logic of the application, rather than the specific techniques required to make the computer perform better and faster!

Lewis Rosenfelder

July 1981

What Is Faster And Better?

If we could define 'faster' and 'better', in a way that would apply to all programming problems, it would be a much simpler matter to design programs. Programming would become less of an art, and more of a science. It would be a simple matter of starting at point 'A' and working to point 'B'.

But a large part of our programming problem is deciding exactly what point 'B' is. In programming and system design we are working in a world of trade-offs. To make a system better in one way we often have to make it not quite as good in another way. We must balance our limited resources to arrive at the best overall solution.

Let's talk about some of the trade-offs we must work with. Each can be maximized only at the expense of one or more other considerations. Every programming technique in your bag-of-tricks has its own advantages and disadvantages. If you can decide on the 'mix' that is best for your application you've cleared away one of the main roadblocks to developing your system.

Efficiency

How economically does the program use limited disk and memory space? We can save disk space through data compression at the expense of memory space, execution time, and compatibility. We can conserve memory space at the expense of execution speed.

Execution Speed

How fast is it overall? How fast is it in those operations that are most critical? How fast and responsive is it for operator-paced operations? We can often make one operation faster by making another operation slower. We can often make a system faster at the expense of reliability or portability.

Programming Time

How long will it take to develop? Can deadlines be met? Given enough time we can improve on many aspects of performance, but nearly every other performance consideration is achieved at the expense of programming time.

Function

Does it do the job intended? By limiting the project to only certain parts of the overall problem we can save on programming time. By doing some things manually we can improve on computer execution speed.

Workability

Does it do the job in a way that is practical and worthwhile to the user? We can maximize the functions performed by the computer, but by doing so, we often sacrifice workability.

Reliability

Is it vulnerable to operator errors or equipment malfunctions? Is it 'crash-worthy'? Is it bug free? We can improve on reliability at the expense of programming time, execution speed and efficiency.

Recoverability

How easily can the results of operator errors or equipment malfunctions be overcome? We can improve on recoverability at the expense of function, workability, design and programming time. Or, we can improve on recoverability with special utility programs that reconstruct data that has been lost. We can live more dangerously in terms of reliability if the system is easily recoverable.

Ease Of Operation

Is it 'operator-oriented'? Are keystrokes minimized? Are operator entries consistent so that it can be run 'instinctively'? We can usually make a system easy to operate at the expense of programming and design time, and memory efficiency.

Ease Of Training

How easy is it to learn for someone who is new to the system? How good are the operator prompting messages? How simple is the overall system? We can make a system easier to learn at the expense of memory usage, programming and documentation time. Too much operator prompting can 'get in the way' of an experienced operator, sacrificing ease of operation.

Capacity

How much data can it handle? Programming a system to handle a small amount of data in memory can be a simple matter. For larger amounts of data we get into the complexities of disk storage. To allow for capacity beyond that of a single disk adds even more complexity.

Portability

How easily can it be transfered for use on a different computer system? We can maximize portability at the expense of efficiency and execution speed. We can make a system easier to transfer by ignoring many of the capabilities and advantages that are unique to the system we are using.

Compatibility

How well does it tie-in with other systems the user might have? We can make the system perform more functions and work faster if we don't have to allow for compatibility with other systems.

Maintainability

If something goes wrong how easy will it be to find the problem and correct it? We can improve on maintainability at the expense of function and efficiency. By conforming to programming standards we make the system more maintainable, but we sometimes sacrifice the ability to use procedures that are best suited to the application.

Ease Of Modification

How easy will it be to modify the system to perform other functions that were not originally considered in the design? We can usually make it easier to modify with more programming and design time.

Understandability

How easily can a programmer other than the one who wrote the program understand the system? We can improve on understandability with extra programming and design time. By sacrificing some techniques that make the system more efficient or faster we can make it more understandable to others.

Documentation

How well are the operating procedures, capabilities, and limitations of the system explained? We can always improve on documentation by spending more time. Internal documentation, by inserting remarks in the body of the program text, can be achieved at the expense of execution speed and memory efficiency.

Attractiveness

How well designed are the video displays and printouts? Does it 'sell' itself to those who must use it? We can make a program look good with more programming time and slower execution speed.

With the 'tools', presented in this book, you can maximize the performance of your system, according to the goals you have defined for the project at hand. Every function and program has been carefully designed to achieve one or more specific purposes. Most of the routines provide exceptional speed. Others operate slower than alternative techniques, but can provide a great savings in programming time. It is up to you to select your programming tools wisely and to test them for your specific application.

How To Use This Book

This book can be valuable to you whether you're a beginner, with only a few weeks experience, or an expert programmer with many years of experience.

If you are new to programming, or the TRS-80 is new to you, you'll need first to get familiar with the capabilities and peculiarities of the TRS-80 and the BASIC programming language. The best way is to work through the examples shown in your operating manuals, and to modify them and experiment with them. Then you can give yourself simple programming challenges, and expand and modify your programs. There is no better teacher for programming than your own computer! It'll tell you when you've made an error and you can try again and again. When you start looking at the examples in this book, you'll get ideas on how to do things differently, (and, hopefully, better).

If you are new to assembly language programming, or if you have not been exposed to it at all, don't let the assembler listings in this book scare you off! Just gloss over them. You don't need to know Z-80 assembly language, and you don't need to own an editor/assembler program to use any of the routines in this book. If you want to learn assembly language for the TRS-80, I recommend *TRS-80 Assembly Language Programming* by Bill Barden. You can pick it up at Radio Shack stores. Then, after you get a feel for assembly language, you can start studying and modifying the assembly language subroutines shown here.

I've made no attempt in this book to duplicate anything that can be found in your instruction manuals, except where some amplification or clarification, or summarization for your convenience is required.

The first 4 chapters of this book cover programming techniques that are important to the implementation of the routines found in the remainder of the book. They discuss subroutines, function calls, USR routines, and techniques for managing the memory of your computer. Again, even if you are an experienced programmer, be sure to go through these chapters first. I guarantee you'll find new ideas and techniques that you've never seen published anywhere else!

Chapters 5 through 15 contain hundreds of ideas, tricks, subroutines, function calls, and USR routines that can be implemented in your programs. It's unavoidable that when you use them, you will need to skip around, because video routines sometimes interact with disk routines, printer routines with disk routines, and so forth. So, before you begin using any of them, be sure to at least 'skim' through the whole book so you'll know what's included.

To get the maximum usefulness from this book, you'll want to create a disk library of the subroutines, functions, test programs, and utilities. That way you can merge what you need into any program that you might be writing.

Subroutines, Handlers, And Shell Programs

The BASIC language, as you'll find it on the TRS-80 computer, has around 150 commands and built-in functions. Have you ever considered which commands and capabilities are the most important to you? My answer to this might suprise you, but to me, MERGE and DELETE are, without a doubt, the most powerful and important commands!

I wouldn't have said that a few years ago, but, now that I've built up a library of programs, subroutines, and functions, I almost never start a program from scratch. You could take away the NEW command, (which clears out memory so you can begin writing a new program), and I wouldn't miss it.

A few years back I was in a computer store having a discussion with a salesman. He thought it was foolish to be in the programming business because "in a couple of years, every program will have been written!" Of course, that statement has turned out to be quite false, but from a programming productivity standpoint, we who program computers would do well to take the attitude that everything has already been written. Our job is to rearrange, modify, combine, insert, and delete so as to come up with programs that can perform any one of an endless range of useful applications.

Subroutines

It doesn't take long to realize that the subroutine capability of BASIC can save you countless hours of work. The GOSUB command lets your program branch to another line, execute some logic, and then RETURN to resume execution with the next command following the GOSUB. Let's consider the advantages of a liberal use of subroutines:

- **Subroutines save memory.** Any significant operation that has to be performed more than once in your program only needs to appear once as a subroutine.
- **Subroutines save programming time.** With subroutines, you are not continually retyping the same logic over and over again.
- **Subroutines provide flexibility.** Simple modifications to a program having a liberal use of subroutines can make it perform new functions that were never considered when the program was originally written.
- **Subroutines simplify testing and debugging.** They let you break your program down to logical modules. Once you've completely tested a subroutine, you can forget about it.

- **Subroutines free you.** They allow you to concentrate on the overall logic and design of the application. You can forget about the details and complexities of those operations you perform again and again.

- **Subroutines increase understanding.** They make programs more readable and understandable. The details and complexities of common operations don't interrupt the 'train-of-thought' in your main program. Even if a routine is used only once in a program, the benefits of readability can sometimes make it worthwhile to design that routine as a subroutine.

- **Subroutines ease conversions.** They can make your program more easily convertible to other computers and operating systems. For example, if a new computer system differs only in its disk handling instructions you simply modify your disk handling subroutines. The rest of your program can remain unchanged.

- **Subroutines can be libraries.** You can create a library of subroutines on disk, and as you need them, merge them into the program you are writing.

This book gives you an extensive library of subroutines that can be used as you need them. Nearly all of them are shown with specific line numbers ranging from 40000 to 59999. You'll find no overlapping of subroutine line numbers shown in this book, except in a few cases where two subroutines perform the same function in a different way, and there would be no reason to have them both in the same program.

If you wish, you can change the line numbers and variables used by any of the standard subroutines in this book. But be aware that by doing so, you'll be missing out on one of the main benefits that this book provides – the pre-written documentation and detailed explanations. The line numbers and variables shown are arbitrary, but I've found that they work well for me. I trust that you'll find similar success with them.

Handlers

A 'handler' is a group of subroutines and procedures that work together to perform a major function within a program.

In this book, for example, we'll be introducing a video display handler for the simplified programming of data entry and video display inquiries.

Handlers provide all the benefits of subroutines, but they go a level above and beyond single subroutines to provide system-wide standards for program organization, disk file organization, and standardized operator-computer dialogs.

A handler gives you specific procedures for using a set of subroutines. To set up a handler within a program, you simply merge the subroutines required, and modify, insert, or delete specific lines according to the instructions provided. A handler provides a starting point for you to begin the modifications required for any particular application. No attempt is made to make any one handler do everything for every possible application. Handlers are designed so that they can be modified for maximum efficiency in a particular application.

You'll find that the time-saving and standardization benefits of handlers are enormous. Once you adopt standard handlers into your programs you'll wonder how you ever got along without them!

Shell Programs

A 'shell program' can be any program that you've designed to be easily modified to perform entirely different applications.

For example, I have used a sophisticated shell program for nearly three years to develop hundreds of different applications. My accounts receivable system has all the handlers for menu selection, video display additions, changes, and inquiries, transaction entry, report printing, and disk file handling. By deleting certain routines, I've got a mailing list system. Other changes have made it into a general ledger system, an inventory control system, an accounts payable system, and many other specialized applications.

When considering a new application, your first question should be, 'What other applications that are already written have the same general structure?' When you think about it, just a few, well-designed, shell programs can be modified to perform almost any application, with upto a 90 percent savings in programming time!

Programming Standards

When I started gathering the subroutines, handlers and function calls for this book I considered changing around the line numbers and variable names to come up with some 'ideal' standards. But, after further consideration, I decided to leave the line numbers and variables unchanged – even though they are quite arbitrary. After all, they've worked well for me, and they can work just as well for you.

I doubt that we'll ever have standard line numbering and variable conventions that everyone can agree upon. The important thing is that you adopt standards that work for you in the types of programs you write. That way you'll always know where to find something in a program and you'll always know how a specific variable is used. I've found that standardization is tremendously valuable to me. Though I've written hundreds of programs, I immediately know by memory where to find any routine in any one of them.

One of the biggest mistakes you can make with a BASIC program is to use a renumber utility and arbitrarily renumber all your lines in increments of 10. That, in my opinion, is like removing all the paragraphs and chapter headings from a book. It no longer makes any sense. You can't see the structure and you can't find anything. Some people may disagree with me on this point, but I believe that line numbers should help to indicate the structure of the program. I think of each group of lines beginning at a multiple of 1000 as a chapter, each group of lines beginning at a multiple of 100 as a major topic within the chapter, and each group of lines beginning at a multiple of 10 as a paragraph.

The following two charts give the general variable naming and line numbering conventions that I have adopted. The specific uses of each variable and line number are explained in the remainder of this book, but for now it will be worthwile for you to get an overview. I invite you to adopt these standards, and to modify them, or add to them, as your needs dictate.

Variable Naming Standards

All variables are pre-defined as integer, except F, which is defined as string for disk file and video display fields. Therefore, at the beginning of a program, "DEFINT A-Z" and "DEFSTR F", can be used. All other variables are explicitly defined within the program text as required, using the "$", "!", and "#" symbols.

```
WORKING VARIABLES:

A$,A%,A!,A#                    - Temporary storage (very transient)
A1$-A5$,A1%-A5%, etc.          - Temporary storage (less transient)
AN$                            - Pointed string, temporary storage
FX$,FX%,FL$,FL%                - Control flags and switches
TC$,TC%,CD$,CD%                - Current transaction code


COUNTERS:

X%,Y%,Z%                      - FOR-NEXT loops, etc.


CONSTANTS:

KD$                            - Current date, 8-byte format
KS$                            - Current date, 2-byte compressed format
KD%, KM%, KY%                  - Current day, month, and year
CN$                            - Company name


GRAPHICS CONSTANTS:

SG$                            - Horizontal bar, STRING$(63,131)
C$                             - Clear to end of display - CHR$(31)
C1$                            - Clear to end of line    - CHR$(30)


VIDEO INPUT AND DISPLAY:

PO%                            - Current print or input position
A1%                            - Current input length limit
PL%                            - Print position - start of current line
LI%                            - First position in scrolling portion
LV%                            - Number of lines in scrolling portion
LT%                            - Horizontal tab position
LZ%                            - Current input line number
LN%                            - Highest input line number entered
LM%                            - Limit, number of entries
F1$()                          - Formatted screen, field storage


SEARCHES AND DISK ACCESS

KY$,FK$                        - Search key
RE$                            - Return string - key found.
```

```
LINE PRINTER

OP$                            - Report Options String
TI$                            - Report title
PN%                            - Page number
H1$                            - Report heading, line 1.
H2$                            - Report heading, line 2.


DISK FILES

FS$,FD$                        - Disk file name
PF%                            - Current file number
PR%(PF%)                       - Current or desired physical record
PP%(PF%)                       - Previous physical record
LR%(PF%)                       - Current or desired logical record
LL%(PF%)                       - Logical record length
LØ%(PF%,Ø) - LØ%(PF%,6)        - Current file statistics
FH$()                          - Field variables


USR ROUTINES

J%                             - Argument passed back to BASIC
US%(), UX%()                   - Magic Array USR routine storage
C%(), P%()                     - Control or parameter arrays
```

Line Numbering Standards

```
Ø  Program name, copyright information, date last modified
1  Memory size modification, CLEAR command
2  DEF commands - DEFUSR's, DEFINT's, DEFSTR's, etc.
3  DIM commands - Array dimensioning
4  Constants and literals to be used in the program
:
3Ø USR routine loading
:
5Ø Function Definitions
:
:
8Ø GOSUB's for opening files and other housekeeping
:
1ØØ Main program menu display
19Ø Operator input of menu selection.  ONGOTO command.
:
2ØØ Secondary menus
:
9ØØ Program close-out and end logic
:
1ØØØ First major routine
2ØØØ Second major routine
:
:
15ØØØ Subroutines peculiar to the application
:
:
4ØØØØ Standard subroutines, keyboard, and video display
41ØØØ Standard subroutines, general
:
57ØØØ Standard subroutines, line printer
58ØØØ Standard subroutines, disk file handling
:
```

Super-Power Function Calls

Did you skip over the section in your BASIC manual that explains how to use functions? If you're like me, and probably thousands of others, the function call capability just didn't seem to be too useful. I completely ignored the function call capability for at least the first year that I had my TRS-80.

Since then, I've discovered that functions provide just about the most useful programming technique. But I'll bet the DEFFN command is one of the most under-used capabilities of BASIC. I guess the unpopularity of the function call is because of the simplistic, and usually useless, examples that are used to illustrate them. The typical BASIC manual gives an example that shows how to use a function to concatenate two strings:

```
10 DEFFNCS$(A$,B$) = A$ + " " + B$
20 INPUT "ENTER FIRST NAME"; F$
30 INPUT "ENTER LAST  NAME"; L$
40 PRINT "FULL NAME IS ";FNCS$(F$,L$)
```

When you run the sample program, the dialog looks something like this . . .

```
ENTER FIRST NAME?JACK
ENTER LAST  NAME?JONES
FULL NAME IS JACK JONES
```

. . . to which your reaction is most likely, "Big deal!".

But, looking at this simplistic and useless example, let's carefully reconsider the advantages:

- The variables used in defining the function are totally unaffected by a use of the function call. In the example, A$ and B$ are not altered. If A$ contains the string "ABCDEF" before using FNCS$(A$,B$), it still contains "ABCDEF" afterwards. Because of this feature, you have total freedom in variable name usage. You can have a whole library of function calls that can be merged into programs when needed – without any concern for variable names.

- The function definition can be done at any line number in the program. Your only requirement is that the program logic must pass through the definition at least once before the function is called. Again, this makes it easy to create a 'merge library' of function calls.

Little-Known Facts About Function Calls

If you experiment with function calls you'll find that they can be very flexible. Here are some of the little-known facts you will discover:

1. You can redefine a function as often as you wish in a program. (In our example, you could later define FNCS$(A$,B$) as B$+“,”+A$.)

2. A function definition can refer to other functions. You can ‘nest’ functions, just as one subroutine can call another.

3. A function definition can call one or more machine language USR subroutines.

4. A function definition can use variables from your program which don't have to be specified as arguments. For example, if, in an inventory control program, LC! contains the quantity when an item was last counted, PR! contains the quantity purchased since the last count, and SO! contains the quantity sold since the last count, you could use FNOH!(0) to get the on-hand quantity. Your function definition would be:

```
DEFFNOH!(A%) = LC! + PR! - SO!
```

In this case, ‘A%’ is a dummy argument. It is not used within the function definition.

5. A function definition must be an expression. It cannot contain any BASIC verbs, such as PRINT or POKE.

Using Function Definitions As Documentation

Function calls can be very documentative. In this book, we'll use A1, A2, A3, etc. as standard variable names to specify the arguments to a function call. So, to document the string concatenation function we used as our example, we would, instead, define it, and document it as follows:

```
DEFFNCS$(A1$,A2$) = A1$+" "+A2$
```

Our documentation, if we were to put this into a library of function calls, might read:

FNCS$(A1$,A2$) adds the string specified by argument 2 onto the string specified by argument 1, inserting a space between them.

A remainder computation function call, FNRE#(A1#,A2#), might be documented as follows:

FNRE#(A1#,A2#) *returns the remainder of argument 1 divided by argument 2.*

Because function calls can be documentative in defining commonly used mathematical computations or other expressions, in certain situations, you may wish to use a function definition as a programming guide. If a computation is used only once within a program, you may wish to program it 'in-line'. For example, the remainder function, as defined in this book is:

```
35 DEFFNRE#(A1#,A2#)=A1#-INT(A1#/A2#)*A2#
```

If you want to print the remainder of X#/Y# within a program, but you don't want to define it as a function, you can use the function definition as a guide. In this way you might come up with a program line such as this:

```
420 PRINT@512,"THE REMAINDER IS ";X#-INT(X#/Y#)*X#
```

As you can see, we substituted X and Y into the pattern shown by FNRE#. You can make the decision on whether to define a function or to program it in-line based on programming convenience and memory availablity in you application.

Packing IF-THEN Logic Into Functions

Suppose you have the following programming problem:

```
If the integer A is between 100 and 300, B is 1.
If the integer A is between 301 and 800, B is 2.
If the integer A is greater than 800, B is 3.
Otherwise, B is 0.
```

You could use IF-THEN expressions to compute B based on A, but you'll need more than one program line. Believe it or not, the following expression takes care of all the logic:

```
B%=-(A%>=100)*-((A%>=100)+(A%>=301)+(A%>=801))
```

To put it into a function, FNCB%(A%), you can use the following definition:

```
10 DEFFNCB%(A%)=-(A%>=100)*-((A%>=100)+(A%>=301)+(A%>=801))
```

Then your main-line program might say:

```
20 INPUTA%
30 B%=FNCB%(A%)
```

The key to this technique is that an expression using any logical operator returns 0 if the expression is false or −1 if the expression is true. For example, if your program contains the expression, "A% =1 > 2", A% will equal 0. If you use the expression, "A=1 < 2", A% will equal −1, indicating that "1 < 2" is a true condition.

In the example above we determined B% by putting each possible condition between parentheses, and manipulated the resulting −1's or 0's with addition and multiplication to return the answer.

With a little creativity and experimentation, you can do unbelievable things with function calls and expressions. And once you've defined and tested the function, it's there for you to plug into any program. This book is full of ready-to-use functions that will save you time in developing programs. The line numbers shown for function definitions in this book are arbitrary, so feel free to change them according to your requirements.

Some functions will provide execution speed improvements over alternate methods. Others will provide capability improvements, sometimes at the expense of speed. Most will save memory, depending on your application. You'll have to judge the trade-offs, but nearly always, the standard function calls will save programming time. Finally, your main-line program logic will be more convenient to write, and easier to follow.

For most of the subroutines, USR routines, and functions in this book, I've provided demonstration or test programs. The best way for you to get familiar with the routines is to try the test programs. That way you can experiment with different modifications and various types of data, and most importantly, you can validate the routines to your satisfaction. Sometimes, in the printed listings for test or demonstration programs, to save space, the subroutines aren't reprinted. You'll need to type-in, or merge from disk, the subroutines and function definitions which are listed separately.

USR Routines – For Speed and Flexibility

Nothing beats the BASIC language for a quick and simple way to program your computer applications. BASIC lets us talk to the computer with commands and mathematical formulas that are quite consistent with the way we think and communicate. But, when super-fast execution speed and truly economical memory usage is required we must speak to the computer in its native tongue, Z-80 machine language. Once we've relieved the TRS-80 of the burden of translating from BASIC to Z-80 commands, its true speed and power can take over.

It is rarely practical to write complete application programs in Z-80 machine language. It's just too time-consuming for most programmers to create, test, and modify programs this way, and the speed and memory-conserving benefits are often not needed. The most useful approach is to have a library of short routines that you can call from BASIC when and where you need them. The USR routine capability lets us jump from BASIC to machine language and back to BASIC again.

In this book, we're going to discuss many special-purpose USR subroutines, and you won't need to know a single Z-80 command to use them. But when you're ready to take the plunge into programming your own Z-80 routines, if you haven't already, the listings provided will give you a good place to start. With an editor-assembler, you can modify or combine the routines shown, or you can create new ones from scratch.

All of the USR routines shown in this book have one very important characteristic – they are relocatable, so you can load and execute them at any location in RAM. In fact, in some cases, we'll be using techniques where a USR routine might be relocated several times during the execution of a BASIC program.

You may have seen or purchased, some of the excellent machine language subroutines for high-speed sorting and other purposes that are available for the TRS-80. Though they often perform well, there are four problems with many of these products:

1. They are designed to load at a specific location in memory. You've got to reserve memory space for them by answering the 'MEMORY SIZE' question properly. If you've got an upper-lower case driver, printer driver, or other 'canned' USR routine that also loads at the same address, you're out of luck.

2. The assembly language documentation is not usually provided with them. You can't easily see how they work, so it is difficult to learn from them, or modify them.

3. They are often provided in packages that contain more than one routine. You must load the routines you don't need along with the one or two routines you do need, wasting valuable memory space.

4. To use them in programs you sell to others you have to pay royalties.

The USR routines we'll be discussing in this book avoid these four problems, giving you the maximum in flexibility and performance. And you don't need to worry about royalties with the routines we'll be discussing, (as long as you don't resell them as a 'library', or copy the documentation.)

Writing USR Routines With An Editor/Assembler

Let's look at the procedures required to create a Z-80 machine language program. We won't get too specific because your editor/assembler manual gives the details, and the exact commands will depend on the version that you use. If you don't have an editor/assembler program, just follow along – you don't need one to use the routines in this book!

For a sample program, we'll write a short subroutine that instantly copies the content from the video display print position 0, to the 1023 other positions on the screen. For example, if we print an 'X' at position 0, a call to this Z-80 subroutine will fill the screen with 'X's'.

With an editor, we can type in the following:

Screen Fill Editor Listing

M 2 Note # 1

```
00010;SFILL - SCREEN-FILL USR ROUTINE
00020;
00030          ORG      0BFF0H        ;ORIGIN
00040          LD       HL,15360      ;HL POINTS TO 0
00050          LD       DE,15361      ;DE POINTS TO 1
00060          LD       BC,1023       ;REPEAT 1023 TIMES
00070          LDIR                   ;HL TO DE. REPEAT.
00080          RET                    ;RETURN TO BASIC
00090          END                    ;
```

1. Line 30 specifies an origin for the USR routine. We have selected **BFF0**, which is 16 bytes below the top of RAM in a 32K TRS-80. For a 48K TRS-80, we might prefer to make our origin **FFF0**. To assemble any Z-80 routine for use on the TRS-80 you will have to specify an origin that is above **3000**, (where ROM ends, and RAM begins.) If you design the routine to be relocatable, (no JP's or CALL's to absolute addresses within the routine), the origin you select need not be the address you'll use when you execute the routine. For assembly and testing purposes, I usually select an origin that is just enough bytes below the top of RAM so that, when assembled, the routine won't wrap back around to the ROM area. I also consider whether any other USR routines are needed in memory at the same time. Sometimes it takes a little trial and error in specifying the ideal origin.

Most assembler listings in this book will show an ORG command specifying **F000** or **FF00** as the origin. To assemble them with a 32K TRS-80 you can change the origin to **B000** or **BF00**. For all routines, the origin is totally up to you.

2. Lines 40 through 80 provide the actual program logic for the routine. We are loading the HL register with the address of the first byte on the TRS-80 video display, and the DE register with the address of the next byte. Then we load the BC register with 1023. The LDIR command in line 70 copies the byte 'pointed-to' by HL to the location pointed-to by DE. Then it adds 1 to HL and DE and subtracts 1 from BC. It repeats this process until BC equals zero. The result of this is that we duplicate the first byte of the video display 1023 times. Line 80 is a RET command, similar to the RETURN command in BASIC. If we call this as a USR routine from BASIC, the RET will bring us back to resume with the next command in our BASIC program.

3. Line 90 satisfies the assembler requirement that there be an END statement.

Now that we've typed it in, we can assemble it into a disk, or tape, machine language 'object code' file. We can also save the 'source code' that we've entered into another file, in case we want to make modifications later – without retyping the whole routine. Here's how our assembled listing for the screen-fill USR routine will look:

Screen Fill Assembly Listing
M 2 Note # 1

```
                 ØØØ1Ø ;SFILL - SCREEN-FILL USR ROUTINE
                 ØØØ2Ø ;
BFFØ             ØØØ3Ø          ORG     ØBFFØH          ;ORIGIN
BFFØ 21ØØ3C      ØØØ4Ø          LD      HL,1536Ø        ;HL POINTS TO Ø
BFF3 11Ø13C      ØØØ5Ø          LD      DE,15361        ;DE POINTS TO 1
BFF6 Ø1FFØ3      ØØØ6Ø          LD      BC,1Ø23         ;REPEAT 1Ø23 TIMES
BFF9 EDBØ        ØØØ7Ø          LDIR                    ;MOVE HL TO DE, REPEAT
BFFB C9          ØØØ8Ø          RET                     ;RETURN TO BASIC
Ø3FF             ØØØ9Ø          END                     ;
ØØØØØ TOTAL ERRORS
```

How To Load And Execute USR Routines From Disk

Let's suppose that we've assembled the screen-fill routine into a disk file named 'SFILL'. Having just assembled it, our executable code is not yet in memory, so our first step is to load it into RAM. From 'DOS READY', we can load the SFILL routine by typing: LOAD SFILL.

Now we want to get into BASIC. But before we do, we'll have set the top of memory so that BASIC will not disturb the area occupied by our USR routine. Looking back at the assembler listing we see that the origin specified was **BFF0**, which corresponds to 49136 decimal. Our answer to the MEMORY SIZE question in this case must not be greater than 49136. (In BASIC we could compute 49136 as our memory size by simply typing, PRINT 65536 + **&HBFF0**.)

Once we're in BASIC, our progam must specify the starting address of our USR routine. The DEFUSR command in disk BASIC lets us define up to 10 addresses as starting points for up to 10 USR routines, 0 through 9. To define our machine language subroutine as USR routine 0, our program line could read:

```
     10 DEFUSRØ=&HBFFØ
or,
     10 DEFUSR=&HBFFØ
or,
     10 DEFUSRØ=49136
or,
     10 DEFUSR=49136
```

If we had more than one USR routine, we could define the second one with DEFUSR1, the third with DEFUSR2, and so forth. Be aware that you may redefine USR addresses as often as you wish in a program. Also, you'll find that a USR routine address remains defined until you redefine it or you reload BASIC. You can RUN or LOAD other programs without altering the USR addresses you've defined.

To execute the screen-fill USR routine that we've assembled and loaded, type-in and RUN the following program:

M 2 Note # 2

```
10 DEFUSRØ=&HBFFØ
20 PRINT@Ø,"X"
30 J=USRØ(Ø)
```

Instantaneously, the screen will be filled with X's. If you modify line 20 to print a different character, the screen will be filled with 1023 copies of that character when you run the program.

Line 30 calls the USR routine. In this case, 'J%' is a dummy variable, as is the '0' between the parentheses. In more sophisticated applications we'll be replacing the '0' with an integer value or expression as a method for passing an argument to a USR routine for use in its computations. We'll be using 'J%' or another integer variable to receive integers passed back to BASIC from USR routines.

Poking USR Routines Into Memory

Each USR routine in this book is shown in 'poke format'. In other words, you'll be given a list of the numbers that you need if you want to POKE the routine into memory. This way, you don't need an editor/assembler program, and you don't need to understand Z-80 machine language. The screen-fill USR routine we've been discussing can be 'loaded' by poking the following 12 numbers into any 12 contiguous bytes in RAM:

M 2 Note # 3

```
33, Ø, 6Ø, 17, 1, 6Ø, 1, 255, 3, 237, 176, 2Ø1
```

Try these steps to see how it works:

1. From DOS READY, load BASIC with a memory size of 49136.
2. Type in the following program:

M 2 Note # 2
M 2 Note # 3

```
10 DEFUSRØ=&HBFFØ
15 DATA 33,Ø,6Ø,17,1,6Ø,1,255,3,237,176,2Ø1
16 FORX=ØTO11 : READ P : POKE &HBFFØ+X,P : NEXT
20 PRINT@Ø,"X"
30 J=USRØ(Ø)
```

3. Run it. Your screen will instantly display 1024 X's. Now, replace line 20 with:

M 2 Note # 4

```
20 PRINT@0,CHR$(191)
```

Run it again. Your screen should instantly go completely white.

Our DATA statement in line 15 specifies a list of numbers which correspond to the 12 bytes in our USR subroutine. Line 16 puts them into 12 bytes of protected memory, starting at **BFF0**, (49136 decimal), so that we can execute the routine.

Since the screen-fill routine is relocatable, we can replace the **&HBFF0** in lines 10 and 16 with any other address in protected memory, and it will run the same. If you have a 48K TRS-80, you might try changing the **BFF0** to **FFF0**. You can also specify a lower number in response to the MEMORY SIZE question, and use an address lower than **BFF0**.

Are you wondering how we got the numbers to be poked? Our assembly listing gave us the hexadecimal codes for the USR routine. The command, 'LD HL,15360', in line 40 generated the machine language instruction, **21003C**. Converting this instruction to decimal:

21 is 33 decimal.
00 is 0 decimal.
3C is 60 decimal.

We then continued the conversion for lines 50 through 80 of the assembly listing to get the 12 numbers to be poked. Or, more easily, we could have gotten the numbers to be poked by loading the assembled program into memory from disk or cassette. Then from BASIC we could have printed the PEEK values from the first byte to the last byte of the routine by issuing the command:

```
FOR X= &HBFF0 TO &HBFFB : PRINT PEEK(X); : NEXT
```

Saving USR Routines To Disk

Each machine language USR routine in this book is shown in 'poke format'. That is, you'll be given a list of numbers that you can POKE, starting at any address in protected memory. Once you've poked the numbers indicated for the USR routine, you can record that routine onto a disk, using any valid disk file name. Suppose you want to save the screen-fill USR routine that we've been using for our example:

1. First you go into BASIC, remembering to specify a memory size low enough so that the planned location of your USR routine will be in protected memory. In our example we specified a memory size of 49136 so that we could locate our 12-byte USR routine at **BFF0**.

2. Then you write or load a program that will poke the required numbers at the desired starting address. Here are the program lines that do the job for the 'SFILL' routine:

M 2 Note # 3

```
15 DATA 33,0,60,17,1,60,1,255,3,237,176,201
16 FORX=0TO11 : READ P : POKE &HBFF0+X,P : NEXT
```

Note that, for this purpose, we just took lines 15 and 16 from our test program.

3. Next you run the program. This reads the data statement and pokes the numbers into memory.

M 2 Note # 5

4. Now, go back to DOS READY. To do so, type, CMD"S".

5. When in DOS READY mode, you can use the DUMP utility. To dump the 12 bytes that are still at location **BFF0** in memory into a disk file named 'SFILL/CIM', enter this command:

M 2 Note # 6

```
DUMP SFILL (START=X'BFFØ',END=X'BFFB')
```

M 2 Note # 7

Note that the dump command automatically adds the file name extension '/CIM' unless you specify an extension. Your disk operating system manual explains this and the other details of the DUMP command.

6. From now on, whenever you know that you'll be calling the SFILL routine in a BASIC program, you can type the command, SFILL, before going into BASIC. The routine will be loaded into RAM at the same address it was when you dumped it. When going into BASIC, you'll again need to protect memory at the address of your USR routine.

If you wish, you can rename 'SFILL/CIM' to any other valid file name. To do this, you'll use the RENAME command. If you do rename it, for example to 'FILLSCRN', and it no longer has the 'CIM' extension, your command to load it from DOS will be, LOAD FILLSCRN.

If you have a Model III, or if you're using the NEWDOS operating system on a Model I, you can load your routine while in BASIC. In NEWDOS, we can have a program line that reads:

```
      1Ø CMD"SFILL"
or,
      1Ø CMD"LOAD SFILL"
```

. . . depending on whether or not the routine on disk has the '/CIM' extension.

If you've got a Model III with TRSDOS 1.3 your DUMP command from TRSDOS READY is:

M 2 Note # 8

```
DUMP SFILL (START=BFFØ,END=BFFB)
```

Then, from TRSDOS READY you can load the routine now stored on disk as SFILL/CMD, by simply typing SFILL. In BASIC you can have a program line that reads:

```
1Ø CMD"L","SFILL/CMD"
```

Magic Strings

Loading USR Routines Into Strings

We can load any relocatable USR routine into a string, as long as it is smaller than 255 bytes. There are some big advantages to this technique. First, when we've got the USR routine in a string, we can avoid the requirement of reserving memory in response to the 'MEMORY SIZE' question. Secondly, we can easily move the routine from one memory location to another by poking the string's VARPTR and LSETing it into another string. Finally, we can store it in an ordinary disk file, which may contain a whole library of routines, for faster and more convenient loading from BASIC.

The screen-fill routine can be loaded into the string S$ with the following command:

M 2 Note # 3

```
S$=CHR$(33)+CHR$(0)+CHR$(60)+CHR$(17)+CHR$(1)+CHR$(60)+CHR$(1)+C
HR$(255)+CHR$(3)+CHR$(237)+CHR$(176)+CHR$(201)
```

Now, to execute the routine, we can define our USR routine address so that it points to the data contained in the string:

```
DEFUSR0=PEEK(VARPTR(S$)+1)+256*PEEK(VARPTR(S$)+2)
```

For safety though, we should define the USR routine address before each call to it. For as we add and work with other strings in the program, BASIC may move S$ to another location in memory.

Here's an easier way to get a longer USR routine into a string, especially after you have already loaded it and tested it in protected memory:

1. Load the routine into protected memory from a file created by the editor/assembler, or poke it into protected memory. We've already discussed how you can do this for the screen-fill routine.

2. Use the DEFUSR command to point USR0 to the routine. For our example, the screen-fill routine starts at BFF0 in memory:

```
DEFUSR0=&HBFF0
```

3. Now define a string using the command:

```
S$=""
```

4. Poke the VARPTR of S$ so that its length equals the length of your USR routine. In our example we would type:

```
POKE VARPTR(S$),12
```

5. Poke the USR routine pointer into the VARPTR of the string. Appendix 2 gives you a list of the USR routine pointer addresses for many of the popular disk operating systems. Here's the command you can use if you are using NEWDOS on a Model I:

```
POKE (VARPTR(S$)+1),PEEK(&H5B14)
POKE (VARPTR(S$)+2),PEEK(&H5B15)
```

Now the string S$ contains the USR routine, and we can put S$ into a random disk file so that we can easily load and execute the routine in future programs without the bothers of protecting memory or using data statements. The random disk file we will create can store dozens of USR routines if we wish. To put the routine stored in S$ into record 1 of a random disk file named, 'USR' we can execute the following commands:

```
OPEN R,1,"USR"
FIELD 1,LEN(S$)AS A$
LSET A$ = S$
PUT 1,1
CLOSE
```

Whenever we want to use the screen-fill routine in a future program, we can, somewhere near the beginning of the program, use the following commands to load the routine into S$:

```
OPEN R,1,"USR"
FIELD 1,12 AS A$
GET 1,1
S$=A$
CLOSE1
```

Then we can call the routine when necessary, using:

```
POKE&H5B14,PEEK(VARPTR(S$)+1)
POKE&H5B15,PEEK(VARPTR(S$)+2)
J=USRØ(Ø)
```

The two pokes perform the function of the DEFUSR command, except that they get the address from the VARPTR of S$. The &H5B14 and &H5B15 shown above will be replaced by the addresses shown in appendix 2 if you are using a different disk operating system.

As an alternative, you can leave the USR routine in the disk buffer during execution. Each disk buffer is, in effect, 256 bytes of protected memory that has been reserved by your response to the 'HOW MANY FILES?' question. The disk buffer addresses are given in Appendix 3.

For example, to use disk file buffer 1 for execution of the screen-fill routine with NEWDOS 2.1 we can use the following command to load the routine:

```
OPEN R,1,"USR"         'OPEN FILE CONTAINING THE ROUTINE
GET1,1                 'GET THE RECORD CONTAINING THE ROUTINE
DEFUSRØ = &H6575       'SPECIFY USR ADDRESS AS DISK BUFFER ADDRESS
```

Then, each time we want to execute it, we can use the command:

```
J=USRØ(Ø)
```

You'll find that the 'magic string' techniques we've discussed in this section provide the one of the fastest, most flexible, and most memory-efficient methods for handling USR routines.

Magic Arrays

How to Load and Execute 'Magic Arrays'

As well as loading a USR routine into a string, and then 'executing' the string, you can also load a USR routine into an integer array, and then execute the 'Magic Array'. I often use this technique because it lets me avoid reserving memory. A 15-element integer array, for example, automatically reserves and protects 30 bytes of memory. An equally important advantage of the technique, as we shall see, is that it provides a convenient and economical method for passing integer arguments to USR routines.

To see how the magic array technique works, enter this short program and run it. It performs the same demonstration that we used for the screen-fill routine. Your screen will be filled instantly with 1024 'X' characters.

Screen Fill Magic Array Demonstration

M 2 Note # 9

M 2 Note # 10

```
5 DEFINTA-Z:J=Ø
1Ø US(Ø)=8448:US(2)=4352:US(4)=256:US(6)=-2Ø243:US(7)=2Ø1
2Ø US(1)=1536Ø:US(3)=15361:US(5)=1Ø23
3Ø PRINT@Ø,"X"
4Ø DEFUSRØ=VARPTR(US(Ø))
5Ø J=USRØ(Ø)
6Ø GOTO6Ø
```

We loaded 7 integers into an integer array. Then, in line 40, we defined our USR routine address to point to the first element of the array. In line 50 we called the USR routine stored in the magic array.

Now look at line 20. We passed the three arguments to the USR routine via array elements 1, 3, and 5. Element 1 specified the address of the byte to be duplicated, in this case, 15360, the memory address of the upper left corner of our display. Element 3, being 1 greater than element 1, specified that just 1 byte was to be duplicated, and element 5 specified that that 1 byte was to be duplicated 1023 times.

Let's try a modification using different parameters. Let's duplicate the word 'TEST' 63 times. Change lines 20 and 30 as follows:

M 2 Note # 11

```
2Ø US(1)=1536Ø:US(3)=15364:US(5)=63*LEN("TEST")
3Ø PRINT@Ø,"TEST"
```

Now run the program. 'TEST' is duplicated 63 times. We changed the arguments for our USR routine by loading array elements. As you can see, it sure beats poking the arguments in!

Before you start playing with this routine, be careful! It's powerful. One wrong move and your computer will go on that strange journey into nowhere. So take these precautions before experimenting:

- Save the program you're working on.
- Remove all diskettes.

Also, we'd better first talk about the rules for using magic arrays:

1. The magic array must be an integer array. In our example we simply used the command 'DEFINT A-Z' to insure that the US% array would be integer.

2. Your program must not use any new variables for the first time between your 'DEFUSR' command and the call to the USR routine. To comply with this rule, note that we pre-initialized the variable, 'J%', in line 5 of our sample program.

This rule is necessary because BASIC moves integer arrays up in memory whenever you use a new variable in a program. If we were using the variable 'J%' for the first time in line 50, the address of our array would have moved up, and our DEFUSR command in line 40 would have been invalidated. It's a good idea to do your DEFUSR immediately before each call to a magic array USR routine. That way, in a complex program you won't accidentally move your USR routine by initializing a new variable.

Each USR routine in this book is shown in 'magic array format'. You are provided with a list of the integers you need to load into an integer array if you want to use the magic array method. For longer routines than the one shown in our example you can use DATA statements to get the integers into the array. The magic array technique works best for short USR routines of about 50 bytes or less. You may have noticed that if your program has several large arrays in it, program execution can begin to get a little sluggish. But for short USR routines with any number of arguments, the magic array technique is indeed 'magic'!.

Writing 'Magic Array' USR Routines

As you've seen, a magic array provides a simple way to load arguments from BASIC into a machine language USR routine. If you know Z-80 assembler language, here's how you can write your own magic array USR routines:

1. Write a Z-80 subroutine and assemble it using the editor/assembler. It must be a relocatable routine!

2. Look at your assembled listing to determine where your arguments will be needed. Then, if necessary, insert 'NOP' commands, or re-organize your routine so that the arguments to be passed start on even numbered bytes within the routine. If the length of the routine is not evenly divisible by 2, add a NOP as the last instruction to make it an even length. Now re-assemble, and check again to verify that the alignment is correct.

Here's the assembler listing that was used in creating the magic array for our screen fill magic array demonstration program. From here on, we'll be calling this subroutine the 'move-data' magic array, because, as you will see, it is useful in many applications where we want to move blocks of data from one memory address to another.

In lines 120, 140, and 160 of the move-data magic array assembler listing we are loading 2-byte integer zeros into the HL,,DE, and BC registers. When loaded into an integer array in BASIC, those zeros line up so that they will be replaced by the contents of elements 1, 3, and 5. So, as we load the parameters to the required

```
BFF0            00100       ORG     0BFF0H      ;ORIGIN - RELOCATABLE
BFF0 00         00110       NOP                 ;NO-OP FOR ALIGNMENT
BFF1 210000     00120       LD      HL,0        ;LOAD "FROM" ADDRESS
BFF4 00         00130       NOP                 ;NO-OP FOR ALIGNMENT
BFF5 110000     00140       LD      DE,0        ;LOAD "TO" ADDRESS
BFF8 00         00150       NOP                 ;NO-OP FOR ALIGNMENT
BFF9 010000     00160       LD      BC,0        ;LOAD # OF BYTES
BFFC EDB0       00170       LDIR                ;MOVE BC BYTES, HL TO DE
BFFE C9         00180       RET                 ;RETURN TO BASIC
BFFF 00         00190       NOP                 ;NO-OP FOR EVEN LENGTH
0000            00200       END                 ;
00000 TOTAL ERRORS
```

Move Data Magic Array Assembly Listing

array elements within a BASIC program, we are actually filling in those instructions.

In lines 110, 130, and 150 we've used NOP's to align the parameters to even bytes. The Z-80 NOP instruction is simply an 8-bit zero, indicating 'no operation'. The computer just ignores it, and continues with the next instruction.

Line 170 is the powerful Z-80 LDIR instruction. It moves the byte from the location pointed to by HL to the location pointed to by DE. Then it increments the HL and DE registers, and decrements the count in the BC register. If BC is non-zero after the decrement, the move, increment, and decrement process is repeated until BC is zero.

In line 200, we used a NOP instruction to make the routine an even number of bytes in length. It is important that magic array routines be of even length.

After you've assembled your routine, load it into memory and go into BASIC, selecting a memory size so that the routine won't be overwritten.

Now, to get the integers that are to be used in the magic array, use the following program:

```
10 S% = &HBFF0  'START ADDRESS
20 E% = &HBFFF  'END ADDRESS
30 FOR X = S% TO E% STEP 2
40 PRINT CVI(CHR$(PEEK(X))+CHR$(PEEK(X+1)));
50 NEXT
```

You will, of course, change lines 10 and 20 to reflect the starting and ending addresses of your program. Usually, you'll want to make line 40 an LPRINT command, to create a printed copy on your line printer.

Putting 'Magic Arrays' Into Random Disk Files

The magic array technique has some nice advantages for getting a USR routine into your computer's memory. When typing the data statements you're working with half as many numbers as you would be with the poke method.

Once you've got a program that reads the required numbers into a magic array, you may wish to record the USR routine that is stored in the array into a random disk file. That way, you will not need to waste the memory required by the data statements in any future programs where you want to use the routine. Here's how to record a magic array into a random disk file, as long as it has 127 or fewer elements:

1. **Open** your disk file in random mode.
2. **Field** it, 255 bytes as A$.
3. **Initialize** a dummy string variable, S$, using S$="".
4. **Poke** the VARPTR of S$ with the length of the routine stored in the magic array. The length will be twice the number of elements because each element takes 2 bytes.
5. **Poke** the VARPTR of S$ + 1 with the LSB (Least Signifigant Byte) of the address of your magic array. If your magic array starts at US%(0) then your command will be:

```
POKE VARPTR(S$)+1, ASC(MKI$(VARPTR(US%(Ø)))
```

6. **Poke** the VARPTR of S$ + 2 with the MSB (Most Signifigant Byte) of the address of your magic array. Continuing our example, your command is :

```
POKE VARPTR(S$)+2, ASC(RIGHT$(MKI$(VARPTR(US%(Ø))),1))
```

Now S$ contains your USR routine. To put it on disk, LSET A$ = S$, and do a disk PUT to the physical record you wish to store it in.

Whenever you want to use the routine in a program, you can OPEN the disk file and GET the physical record in which you stored it. You can then execute it within the disk buffer, move it to another area of protected memory, or move it back into an integer array.

Here's an example. Let's say you've loaded 58 numbers into a magic array, US%, using DATA statements. Your USR routine now starts at US%(0). To record it into physical record 2 of a file named 'ROUTINES' your commands are:

```
OPEN"R",1,"ROUTINES":FIELD1,255ASA$
S$="":POKEVARPTR(S$),116
POKEVARPTR(S$)+1,ASC(MKI$(VARPTR(US%(Ø))))
POKEVARPTR(S$)+2,ASC(RIGHT$(MKI$(VARPTR(US%(Ø))),1))
LSETA$=S$:PUT1,2:CLOSE
```

If you want to load it back into a magic array in a later program, instead of using data statements, you can use the following commands:

```
DIMUS%(58)
OPEN"R",1,"ROUTINES":FIELD1,116ASA$
GET1,2
S$="":POKEVARPTR(S$),116
POKEVARPTR(S$)+1,ASC(MKI$(VARPTR(US%(Ø))))
POKEVARPTR(S$)+2,ASC(RIGHT$(MKI$(VARPTR(US%(Ø))),1))
LSETS$=A$
```

Or, if you don't need to pass arguments via array elements, you can use any of the techniques we discussed for loading and executing magic strings.

Passing USR Routine Arguments With Control Arrays

This is another powerful technique that you won't find in your disk operating system manual. We simply create an integer array that will contain the arguments that we want to pass to a USR routine. This 'control array' may also contain integers computed by the USR routine that are to be passed back to BASIC.

For example, the 'SORT1' USR routine, which sorts a string array into ascending sequence, requires 2 arguments. The BASIC program that calls it must specify the string array to be sorted and the number of elements to be sorted. Those 2 arguments are contained in an integer array. Element 0 contains the VARPTR to the string array, and element 1 contains the highest element number of the string array to be included in the sort.

To sort the first 600 elements of the S$ array, here are the commands that can be used to call the USR routine, with the C% array as our control array:

```
100 C%(0)=VARPTR(S$(0))
101 C%(1)=599
102 J=USR0(VARPTR(C%(0)))
```

Earlier in the program, we would have used the DEFUSR0 command to load the address of the SORT1 USR routine. Also, the dummy integer variable, 'J%', would have to have been defined earlier in the program for this USR call to work properly. The control array method for passing arguments may be used with any USR routine, whether it is stored in protected memory, a magic string, or magic array.

Control arrays are especially useful when many arguments must be passed between a USR routine and BASIC. You'll find a list of the required elements with each of the USR routines that use the control array technique.

There are a few things you should know when using control arrays:

1. A control array must be an integer array, so you should use percent symbols, or DEFINT the variable name you'll be using.

2. Remember that array addresses will change when you define new variables during the execution of a BASIC program. If one of the elements in your control array is the VARPTR to another array, make sure you don't use any new variables between the time you load the control array and the time your program calls the USR routine.

3. You don't need to start from element zero in the control array. You can use other elements of the array for other purposes. For example, we could have used the following commands to call the SORT1 routine:

```
100 C%(14)=VARPTR(S$(0))
101 C%(15)=599
102 J=USR0(VARPTR(C%(14)))
```

If you're writing your own USR routines and you want to use control arrays, take a look at the assembler listing for any of the USR routines in this book that use the technique. You'll see that the first three Z-80 instructions of the routine are:

M 2 Note # 12

```
CALL  0A7FH
PUSH  HL
POP   IX
```

The 'CALL 0A7FH' loads the argument between the parentheses of the USR( ) function in the BASIC program into the HL register pair. The ROM subroutine at 0A7F does this for us. Because the argument passed from BASIC is the VARPTR to a control array, HL points to the first element of that array.

The PUSH and POP instructions copy the contents of HL into the IX register. Then, for example, if we need to load the contents of the second element of the control array into register pair DE, we can use:

```
LD      E,(IX+4)
LD      D,(IX+5)
```

We can put data back into the control array using the opposite procedure. If, for some reason, we want to put the contents of BC into array element 3 for use by BASIC we can say:

```
LD      (IX+6),C
LD      (IX+7),B
```

If we only have one argument to pass back to BASIC, our last command in the USR subroutine is:

M 2 Note # 13

```
JP      ØA9AH
```

This causes a jump to a ROM routine that returns the contents of HL to BASIC. If we used this jump to return to BASIC, and our original call was:

```
J=USRØ(VARPTR(C%(Ø)))
```

. . . the variable, 'J', would receive the last value of HL computed by the USR0 routine. If we simply use a 'RET' instruction to return to BASIC, the contents of J% will be unaffected by the USR call.

Relocatable Multiple-Argument Handler For USR Calls

If you do assembly language programming, here is a standard 'front-end' that you can put on USR routines as an alternate method for handling multiple arguments. The multiple argument handler lets your BASIC program specify all values to be passed to your USR routine in a single expression. For example, our move-data routine requires 3 arguments:

1. **From** address.
2. **To** address.
3. **Number** of bytes to move.

With the multiple argument handler, if we want to move 50 bytes from location 15360 to location 15384, our USR call is:

```
J=USR(1536Ø)ORUSR(15384)ORUSR(5Ø)
```

The handler maintains a count of the arguments passed. When all (3 in this case) arguments have been received, it passes control to the body of the USR routine for the processing of the arguments. The assembly listing for the multiple argument handler is given on the next page.

To write a Z-80 subroutine with the multiple argument handler:

1. Depending on the USR routine number (0-9) you will be using, and depending on the operating system, refer to Appendix 2 to get the USR

Multiple-Argument Handler USR Routine

```
                00000 ;MULTIPLE ARGUMENT HANDLER
                00001 ;
FF00            00100          ORG     0FF00H          ;ORIGIN
FF00 CD7F0A     00110          CALL    0A7FH           ;PUT ARGUMENT IN HL
FF03 DD2A145B   00120          LD      IX,(05B14H)     ;IX = DEFUSR ADDRESS
FF07 DD7535     00130          LD      (IX+53),L       ;
FF0A DD7436     00140          LD      (IX+54),H       ;PUT ARGUMENT IN STORAGE AREA
FF0D DD3409     00150          INC     (IX+9)          ;
FF10 DD3409     00160          INC     (IX+9)          ;ADD 2 TO POINTER
FF13 DD340C     00170          INC     (IX+12)         ;
FF16 DD340C     00180          INC     (IX+12)         ;ADD 2 TO SECOND POINTER
FF19 DD7E09     00190          LD      A,(IX+9)        ;
FF1C 0635       00200          LD      B,53            ;
FF1E 90         00210          SUB     B               ;A = ARGS PASSED * 2
FF1F DD4634     00220          LD      B,(IX+52)       ;B = ARGS REMAINING * 2
FF22 90         00230          SUB     B               ;
FF23 2806       00240          JR      Z,PASS1         ;IF 0, NO MORE ARGS TO PASS
FF25 210000     00250          LD      HL,0000H        ;CLEAR FOR RETURN
FF28 C39A0A     00260          JP      0A9AH           ;RETURN TO GET NEXT ARG
FF2B DD360935   00270 PASS1    LD      (IX+9),53       ;
FF2F DD360C36   00280          LD      (IX+12),54      ;RESTORE COUNT
FF33 1806       00290          JR      START           ;
FF35 0000       00300          DEFW    0               ;ARGUMENT 1 STORAGE
FF37 0000       00310          DEFW    0               ;ARGUMENT 2 STORAGE
FF39 0000       00320          DEFW    0               ;ARGUMENT 3 STORAGE
FF3B 00         00330 START    NOP                     ;BODY OF ROUTINE STARTS HERE
402D            00340          END     402DH           ;
00000 TOTAL ERRORS
```

M 2 Note # 12
M 2 Note # 13

routine pointer address. Modify line 120 accordingly. (The illustration shows 5B14, the address of the USR0 pointer for NEWDOS 2.1.)

2. Insert or delete DEFW commands between lines 290 and 330 so the number of DEFW commands is equal to the number of arguments you want to pass from BASIC to the USR subroutine. It is required that nothing else be between the 'JR START' command and the 'START' label, because the handler uses the difference between these two points to determine the number of arguments to be passed before execution of the main routine.

3. Put the logic for your Z-80 subroutine at, and below, the 'START' label. To access the arguments that have been passed you can use the IX register:

(IX+53) and (IX+54) contain the first argument
(IX+55) and (IX+56) contain the second argument
(IX+57) and (IX+58) contain the third argument, etc.

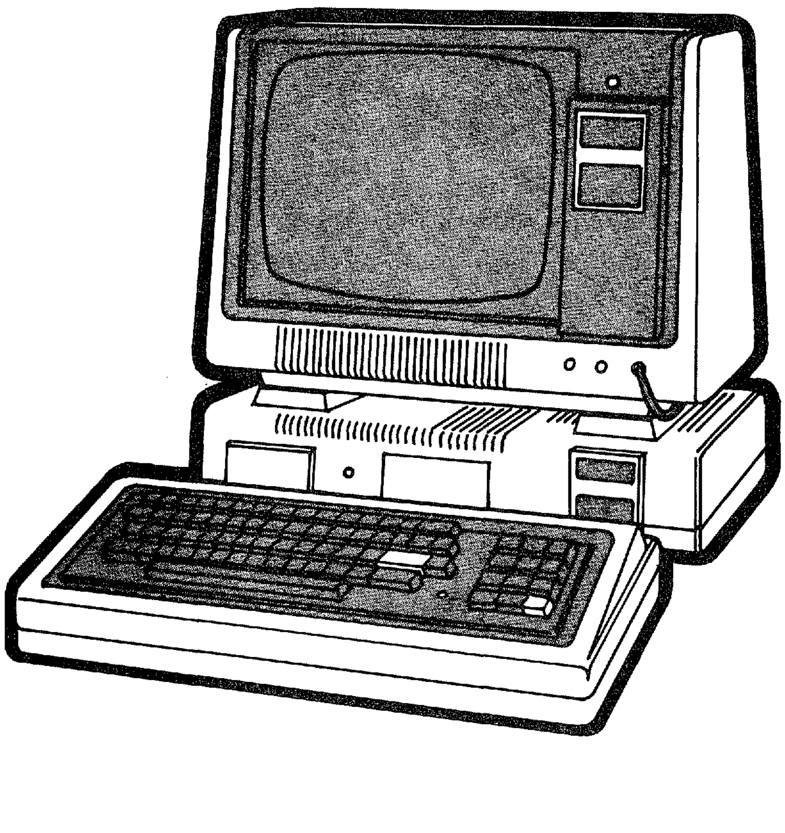
For example, to load the second argument into DE, your command is:

```
LD      E,(IX+55)
LD      D,(IX+56)
```

IX, as you'll see if you analyze the handler, points to the base of the USR routine. IX was loaded in line 70, by an inquiry into the address used when the DEFUSR was done. You're program automatically figures out where it is in memory – no matter where you put it!

The multiple argument handler is probably the most convenient way to call USR routines from BASIC. Keep its limitations in mind when you use it:

1. At most, about 25 arguments can be passed in a single call.
2. You must pre-determine which USR routine you'll be using because that pointer is assembled into the handler. (You can poke in the 6th and 7th bytes if you need more flexibility.)
3. The handler adds about 50 bytes to your routine, so consider the trade-offs when considering whether or not to use it.
4. The logic is self-modifying during run-time. All variables must be passed properly or the handler will not be re-initialized to its original status.
5. You can save memory if your USR routine doesn't need to be relocatable. The main advantage of the multiple argument handler is that it's relocatable, and the working storage memories are imbedded in the routine!



Magic Memory Techniques

'Any given program will expand to fill all available memory'

If you've been programming the TRS-80 computer for any length of time, you'll be able to attest to the truth of that statement. It always seems that, no matter how much memory or disk space you have, you can find a way to use it. This section will give you the techniques you need to make the most of the memory you have.

We've all seen shows where a memory expert entertains the audience by quickly memorizing everyone's name, or the contents of each page in a magazine. These 'super' memory powers are really based on simple techniques that anyone can learn. This section will give you some simple techniques that can, likewise, give your computer's memory some amazing powers. You'll find that when you know how to control your computer's memory, move data quickly, and roll program modules in and out from disk, your programs can enter whole new 'generations' of performance!

How Much Memory Do You Really Have?

If Radio Shack sold you a '48K' TRS-80 computer, you really have 64K of memory. If you bought a '32K' TRS-80, you really have 48K of memory. True, some of the memory is ROM, so it is unmodifiable from a programmer's standpoint, but you might as well start thinking in terms of the upper-limit of your usable memory:

Table of Memory Limits

M 2 Note # 14

Radio Shack Catalog	Top Byte Hexadecimal	Top Byte Decimal	Top Byte Integer Format
"16K"	7FFF	32767	32767
"32K"	BFFF	49151	-16385
"48K"	FFFF	65535	-1

Peek And Poke Above Byte 32767

If you try to POKE 65535,0 you will get an overflow error. This is because the PEEK and POKE commands require an integer argument for the memory address. The secret is that you must convert any address above 32767 by subtracting 65536 from the number. Therefore, the proper command to poke zero into the highest address of a 48K TRS-80 is POKE−1,0. To look at the contents of the top byte in a 48K TRS-80, your program can use, PRINT PEEK(−1).

If your program will be doing a lot of peeking and poking to high memory (above 32767), you may want to include the function calls listed below. They let your program handle memory addresses in single precision format so that you don't have to worry about overflow errors.

To allow peeking or poking any address in the range 0 to 65535, define the following function early in your program:

```
DEFFNSI%(S!)=-((S!>32767)*(S!-65536))-((S!<32768)*S!)
```

Then, if you want to look at the contents of memory location 51400, your program can use the command:

```
PRINTPEEK(FNSI%(51400))
```

Or, to sequentially look at the contents of all addresses in memory, a routine could be written similar to this:

```
FOR A! = 0 TO 65535 : PRINT A!,PEEK(FNSI%(A!)) : NEXT
```

The analogous POKE format is:

```
POKE FNSI%(A!),A%
```

. . . where 'A!' is the address from 0 to 65535, and 'A%' is the number, from 0 to 255, to be poked into that address.

The function call simply converts any unsigned 4-byte single precision number from 0 to 65535, to its 2-byte signed integer equivalent, ranging from −32768 to 32767. To convert back you can use the following function call:

```
DEFFNIS!(I%) = -((I%<0)*(65536+I%)+((I%>=0)*I%))
```

For example:

FNIS!(−1) is 65535
FNIS!(32000) is 32000

Adding And Subtracting Integer Addresses

With many of the subroutines and techniques in this book we'll find it necessary to compute the next address above or below a given address. At other times, we'll need to add or subtract several bytes from an address.

In most cases it's perfectly safe to do the addition or subtraction without any worry as to the validity of the result. But when there's a chance we'll be near 32767 or −32768 we risk getting an overflow error. For example, we know that the next address above 32767 is −32768, but if we add 1 to 32767 or subtract 1 from −32768 we get an overflow.

Most of the subroutines in this book don't consider this danger point unless there's good reason to believe that we'll be encountering it. Usually we will be adding 1 or 2 to an address returned by the VARPTR function. If you get an overflow error when developing a program it's usually a simple matter to reorganize the program or insert a few dummy lines so a VARPTR of 32767 or −32768 won't occur for the variable in question.

FNIA%(A1%,A2%) is a solution to the integer address addition and subtraction problem. It returns the integer address obtained by adding the number specified by the second argument to the address specified by the first argument. If you want it to be safe for any possible integer addition, you can call this function from your subroutines or other function calls:

Integer Address Addition & Subtraction Function

```
10 DEFFNIA%(A1%,A2%)=(65536-(A1%+A2%))*((A1%+A2%)>32767)+((0-A1%
+A2%)*-((A1%+A2%)<-32768))+(A1%+A2%)*-(((A1%+A2%)<32768)AND((A1%
+A2%)>-32769))
```

The logic performed by the FNIA function is:

If the result of the addition is greater than 32767, then subtract the result from 65536.
If the result of the addition is less than −32768, then subtract the result from 0. Otherwise, return the result of the addition.

Here are some examples:

FNIA%(16554,11) is 16565
FNIA%(32767,1) is −32768
FNIA%(−32768,−1) is 32767
FNIA%(−5,1) is −4
FNIA%(−1,10) is 9

Peeking 2 Bytes

As you know, when you PEEK any location in memory, the result will be a number from 0 to 255. And, likewise, the second argument of a POKE command must be from 0 to 255.

Often, it is necessary to work with 2 contiguous memory locations to recall or load an integer ranging from −32768 to 32767. This is because your computer needs 2 bytes to store an integer number. The first byte stores what's called the 'LSB', or 'least significant byte'. The second byte stores the 'MSB' or 'most significant byte'. The MSB is a number from 0 to 255 that tells us how many 256's there are in the number. The LSB is a number from 0 to 255, which when added to the MSB times 256, gives us the integer that's stored in memory.

To look at the 2-byte integer contents of memory, starting at any address except 32767, the expression is:

```
PRINT PEEK(A%) + PEEK(A%+1)*256
```

or,

```
PRINT CVI(CHR$(PEEK(A%))+CHR$(PEEK(A%+1)))
```

If it's possible that your program will be looking at the contents of location

32767, you should use the FNSI% function shown above, and express your address as a single precision number. To look at the 2-byte integer contents of memory, starting at any address expressed as a single precision number, A!, the expression is:

```
        PRINT PEEK(FNSI%(A!)) + PEEK(FNSI%(A!+1))*256
or,
        PRINT CVI(CHR$(PEEK(FNSI%(A!)))+CHR$(PEEK(FNSI%(A!+1))))
```

Poking A 2-Byte Integer Into Memory

From time to time, you may want to change a 2-byte integer located at a given address in memory. We'll be doing it when we start modifying the TRS-80's internal pointers to perform some special tricks. You may also want to do it to poke an integer argument into a USR routine.

To POKE an integer, I%, ranging from −32768 to 32767, into any two contiguous memory addresses, your command is:

```
        POKE A%,I%/256 : POKE A%+1,I%-INT(I%/256)*256
or,
        POKE A%,ASC(MKI$(I%)): POKE A%+1,ASC(MID$(MKI$(I%),2))
```

These simple commands are fine if any of the addresses used will never be 32767. If you will be crossing over from 32767 to −32768, and you need a general routine, you can use the following command to poke any integer into memory, but you will need to define the functions FNSI%(S!) and FNIS!(I%):

```
POKE A%,I%/256 : POKE FNSI%(FNIS!(A%)+1),I%-INT(I%/256)*256
```

How To Change 'Memory Size' From BASIC

M 2 Note # 15
M 2 Note # 7

When your computer goes into BASIC under the TRSDOS disk operating system, you are first asked – MEMORY SIZE?

Under NEWDOS, and other disk operating systems, you specify the memory size as part of your command to load BASIC.

If, for example, you specify a memory size of 61000 using a 48K TRS-80, all memory from 61000 to 65535 is protected. BASIC will not use that area.

From time to time, you might wish to change memory size while in a BASIC program. For example:

- You might want to allocate space for a USR routine which you will be poking in, or loading from a disk file.
- You might want to allocate space in memory to store data, or temporarily save a copy of the video display.
- You might want to establish a common protected area for passing variables between programs.
- You might need to free-up space for program and variable storage when a previously protected area of memory no longer needs to be protected.

First, here's a command that loads the current MEMORY SIZE setting into a single precision variable, MS! :

```
MS! = PEEK(16561)+PEEK(16562)*256+1
```

Here's a command that prints your current MEMORY SIZE setting:

```
PRINT PEEK(16561)+PEEK(16562)*256+1
```

Now, to change the memory size, set MS equal to the desired memory size setting, minus 1, and execute the following command:

```
POKE16562,MS!/256 : POKE16561,MS!-INT(MS!/256)*256
```

You must follow this command with a RUN or CLEAR command to get BASIC to 'read' the new memory size setting. When I change the memory size, I usually do it as the first command in my program. For example, line 1 might read . . .

```
1 MS!=64401:POKE16562,MS!/256:POKE16561,MS!-INT(MS!/256)*256
:CLEAR500
```

. . . to set a memory size of 64401 and clear 500 bytes for string storage. To make it easier (for the computer), you can convert to hexadecimal notation. The number 64400 in hex is FB90. To perform the same memory size setting shown above, (to 64401), we could instead use:

```
1 POKE16562,&HFB:POKE16561,&H90:CLEAR500
```

Reserving Memory Below Program Text

M 2 Note # 16 Here's how to find where your program text begins in memory:

Start of Program Text = PEEK(&H40A4)+PEEK(&H40A5)*256
or
Start of Program Text = PEEK(16548)+PEEK(16549)*256

Below the program text, the disk operating system reserves an area of 290 bytes for each disk file that you specified when answering the question, 'HOW MANY FILES?'. (301 bytes for NEWDOS80, 360 bytes for Model 3 TRSDOS 1.2.) Because of this, your program text will begin at different locations based on the number of files and the disk operating system you are using.

You can poke the program text pointers with a larger value so that the area between the file buffer area and the program text is in effect, reserved. This technique is especially useful when the top of memory is being used by the upper-lower case driver or other machine language program and you want to find another location to load a USR routine.

It's easiest if you move the program text up in even multiples of 256. Simply:

M 2 Note # 16

```
POKE 16549, PEEK(16549)+M
```

. . . where if M% is 1, you are moving the text up by 256, if it is 2, you are moving it up by 512, etc.

M 2 Note # 16 After poking the beginning of text pointers with the desired address, you'll need to poke a zero into the byte preceding the desired address. Then, your next command should be NEW, LOAD or RUN. The next program that you type in, load or run will start at the new address!

Let's suppose you want to load the program, 'PROG1', at address 7000, (28672 decimal.) Your commands are:

```
POKE&H40A4,&H00:POKE&H40A5,&H70:POKE&H6FFF,0:RUN"PROG1"
```

How To Partially Restore DATA Statements

As you know, the DATA command lets you specify a list of information in your program that you can access sequentially with the READ command. The RESTORE command allows you to re-read your data from the first DATA statement. Let's suppose you don't want to restore all the way back to the first data statement. You can restore to any data element by simply saving BASIC's internal pointer the first time you read that element. The data statement pointer is stored in memory locations 40FF and 4100.

Suppose we have a data statement that contains:

```
DATA A,B,C,D,E,F
```

If we want to restore back to 'D' for re-reading, we just save the pointers the first time we read the 'D'. Here's a program that demonstrates how to do it:

Partial Restore of Data Statements – Demonstration Program

M 2 Note # 17

```
20 DATA A,B,C,D,E,F

100 CLS:PRINT"GROUP 1";TAB(20):FORX=1TO3:READA$:PRINTA$;:NEXT
101 D1%=PEEK(&H40FF):D2%=PEEK(&H4100)

110 PRINT:PRINT"GROUP 2";TAB(20):FORX=1TO3:READA$:PRINTA$;:NEXT
111 POKE&H40FF,D1%:POKE&H4100,D2%

120 POKE&H40FF,D1%:POKE&H4100,D2%
121 PRINT:PRINT"GROUP 2 RESTORED";TAB(20)
122 FORX=1TO3:READA$:PRINTA$;:NEXT
```

Line 20 is our DATA list. In line 100 we read and printed the first 3 data elements. Line 101 saved the data pointer in the integer variables, D1% and D2%, because we knew we'd want to do a RESTORE to this point. Then in line 110 we read the next 3 data elements. In line 120 we poked the data pointers back in so that in line 122 we could re-read the last 3 data elements. Here's what the display looks like when this program is run:

```
GROUP 1                ABC
GROUP 2                DEF
GROUP 2 RESTORED       DEF
```

Data statements can be very memory-efficient for storing strings that are to be used as 'literals', (for headings, file names, standard product descriptions, etc.), because the data only appears once in memory. They can be very wasteful of memory if they are being used to load values into numeric arrays. In the case of numeric arrays, the data appears twice: once in string format within the program text, and once in numeric format within the variable storage area.

The Active Variable Analyzer

Here is one of the most powerful and useful utility programs that you can have in your library. It can be a tremendous aid in debugging programs and in finding ways to improve on memory efficiency. The active variable analyzer is a subroutine that you can temporarily merge into any BASIC program that you might wish to analyze. Then, at any point in the program,

- you can see what integer, single precision, double precision and string variables are currently in use. This includes simple variables as well as single, double or triple dimensioned arrays.
- you can view the current contents of all variables that are currently in use. For strings that are 2, 4 or 8 bytes long, it even shows the CVI, CVS and CVD translations. (In case those strings contain binary compressed numbers.)
- you can analyze the sequence in which the variables were introduced into the program.

The active variable analyzer is particularly useful when you are trying to understand how someone else's undocumented program works. Having the contents of all variables displayed for you can often tell you how each is used, so that you can make corrections, modifications or enhancements.

In many programs you will be able to find ways to save memory. You'll be able to see the 'dead weight' that the program may be carrying. Often you can find arrays that were 'over dimensioned'. You may find simple numeric variables that can be re-used for other purposes. Or, you may find strings that were defined and used in an earlier part of the program, whose contents are not necessary in a later part. To free-up more string storage, you can 'null' those strings or re-use them for other purposes. (To null a string, you change its length to zero. For example, to null XY$, you can say XY$="".)

By minimizing the number of variables in use, you automatically improve on program execution speed because BASIC doesn't have as much searching to do. By nulling strings that are no longer needed, you can cut down on the string reorganization time that BASIC may require.

Analyzing the sequence in which the variables were defined can lead to major performance improvements. If you change your program so that the most frequently used variables are defined first you can cut down on searching time, resulting in much more responsive performance.

The active variable analyzer normally occupies lines 65000 through 65162. It uses its own variables, all of which start with ZZ or ZD. You may want to have several versions of the subroutine that use other variable names or line numbers so that you'll be ready to analyze any program. The version we'll be showing uses PRINT commands. You may also want to have a LPRINT version handy. (You can use the 'CHANGE/BAS' program modification utility, shown in this book, to make your other versions.)

To use the active variable analyzer:

1. Load the program you want to analyze.
2. Merge the active variable analyzer from disk. It must have been previously saved with the 'A' option, in ASCII format.
3. Run your program. When you get to a point where you wish to analyze the variables currently defined, press BREAK and type GOSUB 65000. You can also insert the 'GOSUB 65000' at one or more points in your program before running it. You may need to insert an 'END' or 'STOP' command just before the active variable analyzer subroutine to prevent your program logic from flowing into it. You may also need to adjust your 'CLEAR' command so that you don't get an 'out of string space' error.
4. Be sure to delete the active variable analyzer subroutine before you SAVE your program.

Here's a simple program that initializes some variables so we can see how the active variable analyzer works:

```
1 CLEAR1000
10 TI$="TEST PROGRAM"
20 TI$=" ** "+TI$+" ** ":IFLEN(TI$)<5THENG%=3030
30 DIMA%(3),B%(1,1)
40 B%(0,0)=100:B%(0,1)=B%(0,0)*2:B%(1,1)=LEN(TI$)
50 XY$=MKI$(B%(0,0))
```

Now, if we MERGE the active variable analyzer and insert a 'GOSUB 65000 : END' at line 60, when we type RUN, here's what we get:

```
ACTIVE SIMPLE VARIABLES:
TI$                    " ** TEST PROGRAM ** "
XY$                    "D."
CVI(XY$)                100

ACTIVE ARRAYS:
A%( 0)                  0
A%( 1)                  0
A%( 2)                  0
A%( 3)                  0
B%( 0, 0)               100
B%( 1, 0)               0
B%( 0, 1)               200
B%( 1, 1)               20
```

Notice that only the final content of each variable is shown. The string XY$, which stored the number 100 in 2-byte, MKI$ format, was automatically converted for us. For any strings having undisplayable characters, (less than ASCII 32 or greater than ASCII 191), a period replaces the character. Quotes are shown on both sides of all strings to highlight any leading or trailing blanks. Though the integer, G%, was referenced in line 20, the program logic never got to that point so it is not included in our variable list.

Active Variable Analyzer Subroutine

M 2 Note # 18

```
65000 PRINT"ACTIVE SIMPLE VARIABLES:"
65002 ZD%=0:ZZ%=0:ZZ$="":ZZ$(3)="":ZZ%(0)=PEEK(16633):ZZ%(1)=PEE
K(16634)
65004 GOSUB65110
65006 IFZZ%(0)=PEEK(16635)ANDZZ%(1)=PEEK(16636)THEN65030ELSEGOSU
B65130
65007 GOSUB65140:GOTO65006
65030 PRINT"ACTIVE ARRAYS:"
65032 ZZ%(0)=PEEK(16635):ZZ%(1)=PEEK(16636)
65034 GOSUB65110
65036 IFZZ%(0)=PEEK(16637)ANDZZ%(1)=PEEK(16638)THENRETURNELSEGOS
UB65130:GOSUB65100:GOSUB65100:GOSUB65100:GOSUB65110:ZD%=ZZ%(3):Z
Z%=0
65038 IFZZ%=ZD%THEN65040ELSEGOSUB65100:GOSUB65110:ZZ$(1)=ZZ$(0):
GOSUB65100:GOSUB65110:ZZ%(8+ZZ%)=0:ZZ%(5+ZZ%)=CVI(ZZ$(1)+ZZ$(0))
:ZZ%=ZZ%+1:GOTO65038
65040 ZZ$=LEFT$(ZZ$,2):ZZ$(3)="(":FORZZ%=ZD%TO1STEP-1:ZZ$(3)=ZZ$
(3)+STR$(ZZ%(7+ZZ%))
65041 IFZZ%>1THENZZ$(3)=ZZ$(3)+","ELSEZZ$(3)=ZZ$(3)+")"
65042 NEXT
65050 GOSUB65140
65051 ZZ%(7+ZD%)=ZZ%(7+ZD%)+1:IFZZ%(7+ZD%)<ZZ%(4+ZD%)THEN65040
65052 IFZD%=1THEN65070ELSEZZ%(7+ZD%)=0
65053 ZZ%(6+ZD%)=ZZ%(6+ZD%)+1:IFZZ%(6+ZD%)<ZZ%(3+ZD%)THEN65040
65054 IFZD%=2THEN65070ELSEZZ%(6+ZD%)=0
65055 ZZ%(5+ZD%)=ZZ%(5+ZD%)+1:IFZZ%(5+ZD%)<ZZ%(2+ZD%)THEN65040EL
SE65070
65060 GOTO65040
65070 GOSUB65100:GOSUB65110:GOTO65036
65100 ZZ%(0)=ZZ%(0)+1:IFZZ%(0)=256THENZZ%(0)=0:ZZ%(1)=ZZ%(1)+1
65101 RETURN
65110 ZZ%(4)=CVI(CHR$(ZZ%(0))+CHR$(ZZ%(1))):ZZ%(3)=PEEK(ZZ%(4)):
ZZ$(0)=CHR$(ZZ%(3)):RETURN
65120 FORZZ%=1TOZZ%(2):GOSUB65100:GOSUB65110:ZZ$(1)=ZZ$(1)+ZZ$(0
):NEXT:IFZZ$(3)=""THENGOSUB65100:GOSUB65110
65121 IFINSTR("ZZ$ZZ%ZD%",ZZ$)THENZZ$=""
65122 RETURN
```

M 2 Note # 19

```
65130 ZZ%(2)=ZZ%(3):GOSUB65100:GOSUB65110:ZZ$=ZZ$(0):GOSUB65100:
GOSUB65110:ZZ$=ZZ$(0)+ZZ$:RETURN
65140 ZZ$(1)="":ON(INSTR(" 2 3 4 8",STR$(ZZ%(2)))-1)/2+1GOSUB651
44,65146,65160,65162
65142 RETURN
65144 ZZ$=ZZ$+"%":GOSUB65120:IFZZ$=""THENRETURNELSEPRINTZZ$;ZZ$(
3)TAB(20)CVI(ZZ$(1)):RETURN
65146 ZZ$=ZZ$+"$":GOSUB65120:IFZZ$=""THENRETURNELSEPRINTZZ$;ZZ$(
3)TAB(20)
65148 PRINTCHR$(34);:ZZ$(2)=CHR$(ZZ%(0))+CHR$(ZZ%(1))+CHR$(ZZ%(2
)):ZZ%=ASC(ZZ$(1)):ZZ%(0)=ASC(MID$(ZZ$(1),2)):ZZ%(1)=ASC(MID$(ZZ
$(1),3)):ZZ$(1)="":ZZ%(2)=ZZ%
65150 IFZZ%>0THEN65156ELSEPRINTCHR$(34):ZZ%(0)=ASC(ZZ$(2)):ZZ%(1
)=ASC(MID$(ZZ$(2),2))
65152 IFZZ%(2)=2THENPRINT"CVI(";ZZ$;ZZ$(3);")";TAB(20)CVI(ZZ$(1)
)ELSEIFZZ%(2)=4THENPRINT"CVS(";ZZ$;ZZ$(3);")";TAB(20)CVS(ZZ$(1))
ELSEIFZZ%(2)=8THENPRINT"CVD(";ZZ$;ZZ$(3);")";TAB(20)CVD(ZZ$(1))
65154 ZZ%(2)=ASC(MID$(ZZ$(2),3)):GOSUB65110:RETURN
65156 GOSUB65110:GOSUB65100:ZZ$(1)=ZZ$(1)+ZZ$(0):IFZZ%(3)<32ORZ
Z%(3)>191THENPRINT".";ELSEPRINTZZ$(0);
65158 ZZ%=ZZ%-1:GOTO65150
65160 ZZ$=ZZ$+"!":GOSUB65120:IFZZ$=""THENRETURNELSEPRINTZZ$;ZZ$(
3)TAB(20)CVS(ZZ$(1)):RETURN
65162 ZZ$=ZZ$+"#":GOSUB65120:IFZZ$=""THENRETURNELSEPRINTZZ$;ZZ$(
3)TAB(20)CVD(ZZ$(1)):RETURN
```

Active Variable Analyzer Comments

1. We've sacrificed readability in this subroutine by packing the lines and using only variables starting with 'ZZ' or 'ZD'. This was done to avoid introducing more that a few new entries into the variable list in memory, and to simplify changes to other variable names. In case you want to make modifications, here are the variables we used:

```
ZZ%       Temporary counter and working storage.
ZZ%(Ø)    LSB of the current address.
ZZ%(1)    MSB of the current address.
ZZ%(2)    Type code 2, 3, 4, or 8 for %, $, !, or # variables.
          Also, temporary storage of string length.
ZZ%(3)    Contents of current memory address, Ø - 255.
ZZ%(4)    Current memory address in variable storage area.
ZZ%(5)    Dimension 1, of current array.
ZZ%(6)    Dimension 2, of current array, if any.
ZZ%(7)    Dimension 3 of current array, if any.
ZZ%(8)    Dimension 1 counter.
ZZ%(9)    Dimension 2 counter.
ZZ%(1Ø)   Dimension 3 counter.
ZZ$       Current variable name.
ZZ$(Ø)    Contents of current memory address, CHR$ format.
ZZ$(1)    Current variable or string pointer contents.
ZZ$(2)    Temporary storage of current address during string build.
ZZ$(3)    Current variable subscripts for display with arrays.
ZD%       Dimension of current array.  (Single, double or triple.)
```

2. You may 'GOSUB 65030' if you want arrays only. You may put a 'RETURN' at 65030 if you want simple variables only. Lines 65030 through 65070 are not required if you only want to display simple variables.

M 2 Note # 18

3. In line 65002 we load the beginning address for simple variables in memory. This pointer is found at memory addresses 16633 and 16634. We know we've finished with the simple variables when we reach the address indicated by the contents of 16635 and 16636. This is the beginning the array storage area. Note that we reload the starting address in 65032 in case you GOSUB directly to the array printing routine. We know we've finished with the arrays when we get to the address indicated by the contents of memory locations 16637 and 16638.

4. Subroutine 65100 increments our address for us. This pattern is useful in many applications which require a byte-by-byte 'read' through memory. We add 1 to the LSB of the address. If the LSB reaches 256, we set it back to zero and add 1 to the MSB of the address.

5. Subroutine 65110, for programming convenience and memory efficiency, (at the expense of speed), converts the LSB and MSB back to an integer-format address. Then it gets the 'peek' value of the current address, converts and stores the CHR$ of the peek value.

6. Subroutine 65120 builds a string containing the contents of the current variable at the current address.

7. Line 65121 checks to see if the variable name is part of the active variable analyzer subroutine. If you want to bypass other variable names, you can insert those names in this line, or you can make a modification here so that only those variables you specify are printed. If the variable is in the 'by-pass' list, ZZ$ is set to a null string.

8. Subroutine 65130 builds the variable name.
9. Subroutine 65140 directs the logic to the proper subroutine for integer, string, single precision, or double precision.
10. If you don't want to display the CVI, CVS, and CVD conversions for 2-, 4-, and 8-byte strings, you can delete line 65152.
11. If you make an LPRINT version of this subroutine, you may need to change the '191' in line 65156 to a lower number, such as 128. Many printers use ASCII characters above 128 for special control codes.

The 'Move-Data' Magic Array

Many special effects and high-speed techniques involve nothing more than moving, (or more accurately described, 'copying') a block of data from one location in memory to another. With special Z-80 machine language subroutines, we can perform this function instantaneously. We simply specify the 'from' address, the 'to' address, and the number of bytes to move.

The Z-80 has two instructions that are especially useful for moving data, LDIR and LDDR. To illustrate how they work, lets assume we have a block of 16 bytes in memory. We'll number them starting at zero, but they could start at any location, from 0 to 65535. Let's also assume that the first 4 bytes of this memory block contain the word 'DATA':

```
<00><01><02><03><04><05><06><07><08><09><10><11><12><13><14><15>
 D   A   T   A   ?   ?   ?   ?   ?   ?   ?   ?   ?   ?   ?   ?
```

To move (or copy) the word 'DATA' to location 6, the LDIR command would first move the 'D' to location 6, then the first 'A' to location 7, the 'T' to location 8, and the final 'A' to location 9. After this move, our memory block looks like this:

```
<00><01><02><03><04><05><06><07><08><09><10><11><12><13><14><15>
 D   A   T   A   ?   ?   D   A   T   A   ?   ?   ?   ?   ?   ?
```

We've just done a move of 4 bytes from location 0 to location 6.

The LDDR command can perform the same function, but it starts with the final 'A' in 'DATA' and works down to the 'D'. It first moves the 'A' from location 3 to 9. Then it moves the 'T' from location 2 to 8, the 'A' from location 1 to 7, and finally, the 'D' from location 0 to 6.

These two methods of moving data are interchangeable when our source and destination don't overlap. But let's suppose we want to move 4 bytes from location 0 to 1. Starting with our original memory contents, the Z-80 LDIR command would move the 'D' in position 0 to 1. Then it would move the contents of memory location 1, which is now 'D', to position 2. It would continue this a total of 4 times so our result is:

```
<00><01><02><03><04><05><06><07><08><09><10><11><12><13><14><15>
 D   D   D   D   D   ?   ?   ?   ?   ?   ?   ?   ?   ?   ?   ?
```

On the other hand, the LDDR command 'pulls-up' the memory we want to copy, starting at the last byte. To move the word 'DATA' up 1 position, we can tell the LDDR command to move 4 bytes from position 3 to 4. Working with our original memory contents and the LDDR command, we get:

```
<00><01><02><03><04><05><06><07><08><09><10><11><12><13><14><15>
 D   D   A   T   A   ?   ?   ?   ?   ?   ?   ?   ?   ?   ?   ?
```

We call this an 'overlapping' move because the new data overlaps the old data.

In Z-80 machine language the LDIR and LDDR commands operate based on the contents of 3 registers: HL, DE, and BC. (If you don't speak 'Z-80', you can think of HL, DE, and BC just as you would think of 3 integer variables in BASIC.) The HL register specifies the 'from' address, the DE register specifies the 'to' address, and the BC register specifies the number of times to copy a byte from one address to the other. The LDIR command increments the 'from' and 'to' addresses after each byte is moved. The LDDR command decrements the 'from' and 'to' addresses after each byte is moved. For LDIR and LDDR, the BC register is decremented after each byte is moved. When BC reaches 0, the multi-byte move is complete.

We can take advantage of these high-speed move capabilities in BASIC with the 'move-data magic array.' We simply load the required Z-80 codes into an 8-element integer array, do a DEFUSR to point a USR routine address to the first element of that array, and with the USR function, we execute the move.

Here are the Z-80 codes that go into the move-data magic array:

Element 0: 8448
Element 1: 'From' address.
Element 2: 4352
Element 3: 'To' address.
Element 4: 256
Element 5: Number of bytes to move.
Element 6: −20243 for LDIR, or −18195 for LDDR
Element 7: 201

You'll normally want to pre-load elements 0, 2, 4, and 7 because they are constant for any type of move you might want to make. You might also want to pre-load element 6 with −20243 if you aren't going to be doing any overlapping moves, or if you won't need to do any LDDR moves.

To demonstrate a few moves, let's play with video display memory which occupies addresses 15360 to 16383. Type in the following program:

Move Data Magic Array Demonstration Program

M 2 Note # 20
M 2 Note # 21

```
10 DEFINT A-Z : J=0 : A$=""
20 US(0)=8448:US(2)=4352:US(4)=256:US(7)=201
30 CLS: PRINT"MOVE-DATA DEMO"
40 PRINT@ 64,"FROM           ";:INPUT US(1)
50 PRINT@128,"TO             ";:INPUT US(3)
60 PRINT@192,"# BYTES:       ";:INPUT US(5)
70 PRINT@256,"I=LDIR, D=LDDR ";:INPUTA$
71 IFA$="D"THEN US(6)= -18195 ELSE US(6)= -20243
80 DEFUSR=VARPTR(US(0)):J=USR(0)
90 GOTO 40
```

Now, before you run the move-data demo program, save your program and, as a precaution, remove your disks or make backups. If you accidentally type an incorrect number you could move data to a memory location containing vital BASIC or DOS pointers. This could trigger a command that could 'kill' a disk. (Believe me, I know from experience!) The move-data routine is powerful so it's important to know where the data will go, and how much will be moved. If you follow the examples carefully you shouldn't have any problem.

M 2 Note # 22

Example 1: To copy the top half of the screen to the bottom half, type RUN, and enter '15360' as the from address, '15872' as the 'to' address, and '512' as the number of bytes. When you enter 'I' for LDIR mode, it will be duplicated instantly.

Example 2: To copy the title 'MOVE-DATA DEMO' from position 0 to 32 on your display:

From = 15360,
To = 15392,
# Bytes = 14,
'I' for LDIR

Example 3: To copy the contents of the first 512 bytes of ROM to the bottom half of your video display:

From = 0,
To = 15872,
# Bytes = 512,
'I' for LDIR

Example 4: To give the illusion of shifting the data you just copied from ROM to the bottom of our screen:

From = 1,
To = 15872,
# Bytes = 512,
'I' for LDIR

Example 5: To do an overlapping move-up, so that the 'MOVE-DATA DEMO' title will move over 5 positions, giving us 'MOVE-MOVE-DATA DEMO' in the upper left corner:

From = 15373,
To = 15378,
# Bytes = 14,
'D' for LDDR

Example 6: To fill the screen with M's, (assuming position 0 is still displaying an 'M'):

From = 15360,
To = 15361,
# Bytes= 1023,
'I' for LDIR

Many other examples are possible. Be careful however, not to enter 0 for the number of bytes to move. This is very important if a Z-80 LDIR or LDDR command gets a 0 as the parameter in BC, it will loop through 65536 moves. The result is always disasterous to the current contents of memory.

The following chart gives you a convenient reference for the types of operations you may wish to perform with the move-data magic array, and how to load elements 1, 3, 5 and 6. This chart is also helpful if you are writing assembly language programs:

```
NON-OVERLAPPING MOVE UP OR DOWN
--------------------------------------------------------------
ELEMENT 1 (HL)     =  FROM ADDRESS
ELEMENT 3 (DE)     =  TO ADDRESS
ELEMENT 5 (BC)     =  NUMBER OF BYTES TO MOVE
ELEMENT 6 (LDIR)   =  -20243

OVERLAPPING MOVE UP
--------------------------------------------------------------
ELEMENT 1 (HL)     =  LAST BYTE OF BLOCK TO BE MOVED UP
ELEMENT 3 (DE)     =  LAST BYTE OF DESTINATION
ELEMENT 5 (BC)     =  NUMBER OF BYTES TO MOVE
ELEMENT 6 (LDDR)   =  -18195

OVERLAPPING MOVE DOWN
--------------------------------------------------------------
ELEMENT 1 (HL)     = FROM ADDRESS
ELEMENT 3 (DE)     = TO ADDRESS (LOWER THAN FROM ADDRESS)
ELEMENT 5 (BC)     = NUMBER OF BYTES TO MOVE
ELEMENT 6 (LDIR)   = -20243

UPWARD PROPAGATION OF A BYTE PATTERN
--------------------------------------------------------------
ELEMENT 1 (HL)     = ADDRESS OF FIRST BYTE OF PATTERN
ELEMENT 3 (DE)     = ADDRESS OF FIRST BYTE OF FIRST DUPLICATION
ELEMENT 5 (BC)     = NUMBER OF TIMES THE PATTERN IS TO BE
                     DUPLICATED (NOT INCLUDING ORIGINAL)
                     MULTIPLIED BY THE PATTERN LENGTH
ELEMENT 6 (LDIR)   = -20243

DOWNWARD PROPAGATION OF A BYTE PATTERN
--------------------------------------------------------------
ELEMENT 1 (HL)     = ADDRESS OF LAST BYTE OF PATTERN
ELEMENT 3 (DE)     = ADDRESS OF FIRST BYTE OF PATTERN - 1
ELEMENT 5 (BC)     = NUMBER OF TIMES THE PATTERN IS TO BE
                     DUPLICATED (NOT INCLUDING ORIGINAL)
                     MULTIPLIED BY THE PATTERN LENGTH
ELEMENT 6 (LDDR)   = -18195
```

Here are some examples of applications for the move-data magic array:

1. Insert and delete operations on the video display.
2. Up or down scrolling for complete or partial screens. Scrolling to and from protected memory.
3. Saving the video display in protected memory, and later, restoring it.
4. Moving data to protected memory so that it can be passed from one program to another.
5. Inserting and deleting array elements.

6. Moving data from a random disk buffer directly to video display memory, without fielding. Saving video display screens on disk, 256 bytes at a time by moving data from the video display to the disk buffer, followed by a PUT command.

7. Moving a relocatable USR routine from one address in memory to another.

8. High-speed loading of elements into numeric arrays from disk, and high-speed recording of numeric arrays on disk. For integer arrays, up to 128 elements can be loaded or recorded instantly.

9. Clearing memory, or loading repeating byte patterns into memory. Graphics effects.

10. Instant duplication of array elements.

11. Moving data or USR routines directly from the disk buffer to protected memory.

As you can see, the move-data magic array is quite useful, and it's extremely fast. We'll be getting into the specifics of some of its applications in other sections of this book.

A Deluxe Move-Data USR Routine

Here's a USR subroutine that performs an instant move of a block of memory from an address to any other address. The MOVEX USR subroutine performs the same function as the move-data array, with these differences:

1. You can pass the 'from', 'to', and 'number-of-bytes' arguments to the MOVEX USR routine with a single BASIC expression. This can make it more convenient for you when programming, and your program execution speed will be slightly faster than with the move-data magic array.

2. It handles any move, including overlapping upward and downward moves. You don't have to decide whether to use LDIR or LDDR, as you do with the move-data magic array. You can't 'propagate' a pattern of bytes in memory, as you can with the move-data magic array.

3. Though MOVEX requires 88 bytes, compared to the 16 required by the move-data magic array, in most applications you'll have a net savings in memory with MOVEX. This savings is possible because your BASIC program has to do fewer computations, and you have the single expression argument passing capability.

4. MOVEX employs the 'USR routine multiple-argument handler'. Because of this, you will have to first decide which USR number you'll use (USR0 - USR9), and you may need to modify 2 bytes depending on the DOS you're using.

To illustrate a MOVEX call from BASIC, let's say you want to copy the top half of the video display to the bottom half. Assuming you've loaded and defined MOVEX as USR0, your command is:

M 2 Note # 7

```
J=USR(15360)ORUSR(15872)ORUSR(512)
```

To shift the contents of the top line on the video display right 1 position use:

```
J=USR(15360)ORUSR(15361)ORUSR(63)
```

To shift the top line left 1 position:

```
J=USR(15361)ORUSR(15360)ORUSR(63)
```

To scroll-up any portion of the video display, where LI% is the beginning PRINT@ position of the scrolling portion, and LV% is the number of lines to scroll, you can say:

```
J=USR(15360+LI+64)ORUSR(15360+LI)ORUSR(64*(LV-1))
PRINT@15360+LI+(LV-1)*64,CHR$(30);
```

As you've probably deduced by now, you call MOVEX with an expression in the following format:

```
J%=USR(F%)ORUSR(T%)ORUSR(B%)
```

Where the integer variables are:

J% is a dummy variable. (The new contents are useless to your program after the call).

F% is an integer variable, constant, or expression specifying the 'from' address.

T% is an integer variable, constant, or expression specifying the 'to' address.

B% is the number of bytes to move. **Important**: B must be non-zero!

The 'magic-array format', 'poke-format' and assembly listing for MOVEX are shown below. As shown, it will execute as USR0 with the NEWDOS 2.1 disk operating system. To use it as another USR routine (USR1 - USR9) with NEWDOS 2.1, or to use it on another operating system, refer to Appendix 2 and use the following guidelines:

1. For execution as a magic array, replace the 4th element, 23316, with the the required integer from Appendix 2. For example, if you are using TRSDOS 2.3 and you want to execute MOVEX as USR6, you find **5B83** in Appendix 2. Converting to decimal, **5B83** is 23427, so the 4th element would be 23427.

2. If you are poking the MOVEX routine, replace the 7th and 8th bytes, 20 and 91, with the required bytes from Appendix 2. For example, if you are using NEWDOS 2.1 and you want to execute MOVEX as USR9, you find **5B26** in Appendix 2. The 7th byte should be **26**, (38 decimal), and the 8th byte should be **5B**. (91 decimal.)

3. If you are re-assembling MOVEX, replace the **5B14** in line 160 of the assembly listing with the required hexadecimal number.

MOVEX/DEM is a demonstration program for the MOVEX routine. It lets you input 'from' and 'to' addresses, plus the 'byte count'. The routine is loaded into a magic array from data statements so that you won't have to protect memory when loading BASIC. Remember, though, that you'll need to change the '23316' in line 31 if you are using an operating system other than NEWDOS 2.1 on a TRS-80 Model 1.

You'll find this a useful program to keep in your disk library. I most often use it to move relocatable USR routines from one address to another.

MOVEX
Deluxe Move Data USR Subroutine

```
                 00000 ;MOVEX
                 00001 ;
F000             00100         ORG     0F000H          ;ORIGIN - RELOCATABLE
                 00110 ;
                 00120 ;THE FOLLOWING LOGIC ACCEPTS THE 3 ARGUMENTS
                 00130 ;
F000 CD7F0A      00140         CALL    0A7FH           ;PUT ARGUMENT IN HL
F003 00          00150         NOP                     ;NO-OP FOR ALIGNMENT
F004 DD2A145B    00160         LD      IX,(05B14H)     ;IX HAS DEFUSR ADDRESS
F008 DD7531      00170         LD      (IX+49),L       ;
F00B DD7432      00180         LD      (IX+50),H       ;LD ARG TO STORAGE AREA
F00E DD340A      00190         INC     (IX+10)         ;
F011 DD340A      00200         INC     (IX+10)         ;ADD 2 TO POINTER
F014 DD340D      00210         INC     (IX+13)         ;
F017 DD340D      00220         INC     (IX+13)         ;ADD 2 TO POINTER 2
F01A DD7E0A      00230         LD      A,(IX+10)       ;
F01D 0631        00240         LD      B,49            ;
F01F 90          00250         SUB     B               ;A = ARGS PASSED *2
F020 DD4630      00260         LD      B,(IX+48)       ;B = ARGS REMAIN *2
F023 90          00270         SUB     B               ;
F024 2801        00280         JR      Z,PASS1         ;IF 0 NO MORE ARGS
F026 C9          00290         RET                     ;OTHERWISE, RETURN FOR NEXT
F027 DD360A31    00300 PASS1   LD      (IX+10),49      ;
F02B DD360D32    00310         LD      (IX+13),50      ;RESTORE COUNT
F02F 1806        00320         JR      START           ;
F031 0000        00330         DEFW    0               ;STORAGE FOR "FROM" ADDRESS
F033 0000        00340         DEFW    0               ;STORAGE FOR "TO" ADDRESS
F035 0000        00350         DEFW    0               ;STORAGE BYTES TO MOVE
                 00351 ;
                 00352 ;THE FOLLOWING LOGIC PROCESSES THE MOVE
                 00353 ;
F037 E5          00360 START   PUSH    HL              ;LAST ARGUMENT IS STILL IN HL
F038 C1          00370         POP     BC              ;# OF BYTES TO MOVE NOW IN BC
F039 DD6E31      00380         LD      L,(IX+49)       ;
F03C DD6632      00390         LD      H,(IX+50)       ;"FROM" ADDRESS IN HL
F03F E5          00400         PUSH    HL              ;SAVE "FROM" ADDRESS ON STACK
F040 DD5E33      00410         LD      E,(IX+51)       ;
F043 DD5634      00420         LD      D,(IX+52)       ;"TO" ADDRESS IN DE
F046 B7          00430         OR      A               ;CLEAR CARRY FLAG
F047 ED52        00440         SBC     HL,DE           ;SUBTRACT "TO" FROM "FROM"
F049 E1          00450         POP     HL              ;RESTORE "FROM" ADDRESS FROM STACK
F04A 3803        00460         JR      C,MOVEUP        ;MOVE UP IF "TO" IS GREATER
F04C EDB0        00470         LDIR                    ;OTHERWISE, MOVE THE BLOCK DOWN
F04E C9          00480         RET                     ;RETURN TO BASIC
F04F 09          00490 MOVEUP  ADD     HL,BC           ;HL HAS END OF BLOCK TO MOVE + 1
F050 2B          00500         DEC     HL              ;HL HAS END OF BLOCK TO MOVE
F051 EB          00510         EX      DE,HL           ;HL HAS "TO" ADDRESS
F052 09          00520         ADD     HL,BC           ;HL END OF "TO" BLOCK + 1
F053 2B          00530         DEC     HL              ;HL END OF "TO" BLOCK
F054 EB          00540         EX      DE,HL           ;HL=END OF "FROM", DE=END OF "TO"
F055 EDB8        00550         LDDR                    ;MOVE THE BLOCK UP
F057 C9          00560         RET                     ;RETURN TO BASIC
F04F             00570         END                     ;
00000 TOTAL ERRORS
```

MOVEX
Deluxe Move Data USR Subroutine
M 2 Note # 23

```
Magic Array Format, 44 elements:

 32717     10  10973  23316  30173  -8911  12916  13533  -8950
  2612  13533  -8947   3380  32477   1546 -28623  18141 -28624
   296  -8759   2614  -8911   3382   6194      6      0      0
 -6912  -8767  12654  26333  -6862  24285  -8909  13398  -4681
 -7854    824 -20243   2505  -5333  11017  -4629 -13896

Poke Format, 88 bytes:

205 127  10   0 221  42  20  91 221 117  49 221 116  50 221  52
 10 221  52  10 221  52  13 221  52  13 221 126  10   6  49 144
221  70  48 144  40   1 201 221  54  10  49 221  54  13  50  24
  6   0   0   0   0   0   0 229 193 221 110  49 221 102  50 229
221  94  51 221  86  52 183 237  82 225  56   3 237 176 201   9
 43 235   9  43 235 237 184 201
```

MOVEX/DEM
Move Data Demonstration and Utility
M 2 Note # 21
M 2 Note # 2 3
M 2 Note # 2 4

```
10 DEFINTA-Z :J=0

30 'LOAD MOVEX USR ROUTINE INTO A MAGIC ARRAY
31 DATA 32717, 10, 10973, 23316, 30173,-8911, 12916, 13533,-8950
, 2612, 13533,-8947, 3380, 32477, 1546,-28623
32 DATA 18141,-28624, 296,-8759, 2614,-8911, 3382, 6194, 6, 0, 0
,-6912,-8767, 12654, 26333,-6862
33 DATA 24285,-8909, 13398,-4681,-7854, 824,-20243, 2505,-5333,
11017,-4629,-13896
34 DIM UX(43):FORX=0TO43:READ UX(X):NEXT

100 CLS:PRINT"MOVEX DEMONSTRATION AND UTILITY"
110 PRINT@ 64,"MOVE FROM:          ";:INPUTMF%
120 PRINT@128,"MOVE TO:            ";:INPUTMT%
130 PRINT@192,"NUMBER OF BYTES     ";:INPUTNB%
131 IFNB%=0THEN130
140 DEFUSR=VARPTR(UX(0))
150 J=USR(MF%)ORUSR(MT%)ORUSR(NB%)
160 GOTO110
```

JOURNEY/DEM is a modification to the MOVEX/DEM program. It gives you a quick visual 'journey' through memory. The bottom line of your video display will show the current address, in increments of 64, while the contents of memory scrolls on the top portion of your video display. Besides demonstrating the speed of the MOVEX routine, you can use the journey program to get an idea of what's in memory and where it is.

To run JOURNEY/DEM, delete lines 100 through 160 from the MOVEX/DEM program, and add the following lines:

JOURNEY/DEM
Modifications to MOVEX/DEM
M 2 Note # 25

```
100 CLS:A=0:DEFUSR=VARPTR(UX(0))
110 FORX=-1TO32766STEP64:A=X+1:GOSUB200:NEXT
120 FORX=-32768TO0STEP64:A=X:GOSUB200:NEXT
130 END
200 PRINT@990,A;:J=USR(A)ORUSR(15360)ORUSR(960):RETURN
```

BASIC Overlays

Passing Variables Between Programs

Any time you issue a RUN or LOAD command, all variables that were previously active are cleared so the new program can start with a clean slate. But there are many situations where you don't want those variables cleared as you go from one program to another.

If you can pass variables between programs, you can divide your application into smaller programs. With smaller programs, you have more memory available for storage of variables. One program, for example, might load in data from keyboard entry or disk. The next program might process that data, and a third program might provide a printout.

Before you can use the variable-passing subroutines must know that variables are stored immediately above your BASIC program text in memory. Let's suppose as an example, that you have written this program:

```
10 X%=1
20 A%=2
30 S$=STRING$(5,"X")
```

When you run the program, the contents of X% will be stored in memory just above the address where line 30 is stored. The contents of A% will be stored just above the contents of X%. And just above the location where A% is stored, BASIC will record a pointer that indicates the length and location of the contents of S$. The five X's 'contained in' S$ will be stored just below the top of memory as you defined it with your answer to the 'MEMORY SIZE?' question. Had you defined one or more arrays in the program, they would have been stored just above your simple variables, integers X%, A% and S$.

The area of memory that stores all the active variable names, type codes, dimensions, numeric values and string data pointers is called the variable list. Because the variable list starts just above the program text, the starting location of your variables in memory will depend on the length of the program you have loaded. To pass variables, we override this feature of BASIC, and we decide on a fixed location to begin the variable list. The location we select will be just above the ending address of the longest program we'll be using.

Here's how to find the first available address, beyond the end of your longest program:

1. Load your program, making sure that you answer the 'HOW MANY FILES?' question the same way you'll be answering it when you'll be

running the program in actual practice.

2. Enter the following commands:

M 2 Note # 26

```
CLEAR
PRINT CVI(CHR$(PEEK(&H40F9))+CHR$(PEEK(&H40FA)))
```

3. Add 17 to the number displayed. The result is the lowest address that you may use for the beginning of your variable list if you wish to pass variables between programs. In actual practice, you may want to add 300 or more to this address so that if you make minor modifications that lengthen your program, you won't have to recompute and reprogram a starting address for your variable list.

Now, here's how we force our variables to be stored starting at the fixed location we've chosen. In the first program we'll be running, we do a 'GOSUB 52000' as one of the first commands. This GOSUB must be executed before we use any variables. Subroutine 52000 modifies BASIC's three pointers that determine the start and end of the active variables:

Variable List Pointer Subroutine

M 2 Note # 27

```
52000 A$="":FORA%=1TO3:A$=A$+MKI$(30000):NEXT:AN$="XXXXXX":POKEV
ARPTR(AN$)+1,&HF9:POKEVARPTR(AN$)+2,&H40:LSETAN$=A$:A$="":RETURN
```

You should change the '30000' in subroutine 52000 to the address you wish to use as the start of your variable list.

Note: The subroutine 52000 uses an interesting method of poking the new pointers into the 6 bytes starting at 40F9. We first create a string, (A$) that contains the 6 bytes to be poked. Then we modify the VARPTR of AN$ so that AN$ points to the address 40F9 for 6 bytes. Finally, we LSET A$ into AN$. The LSET command gives us an instant 6 byte poke. Had we tried to poke the 6 bytes with individual poke commands, BASIC would get confused because the first 2-byte pointer would only be 'half-poked' after the first command.

The final A$="" in subroutine 52000 sets up A$ as the first variable to be initialized. The 'variable-pass' subroutine, and 'variable-receive' subroutine both expect to find A$ as the first variable of our variable list.

Subroutine 52100 is the 'variable-pass' subroutine. When you want to pass variables from one program to another you 'GOSUB 52100', then RUN the new program. Subroutine 52100 loads A$ with all the pointers that BASIC is currently maintaining. Among other things, the 104 bytes loaded into A$ will contain the starting location of our simple variables, the starting and ending location of any arrays that may be active, the current status of our string storage area and the type declarations (DEFSTR, DEFINT, DEFSNG, or DEFDBL) that may be active.

Variable Pass Subroutine

M 2 Note # 28

```
52100 AN$="":POKEVARPTR(AN$),104:POKEVARPTR(AN$)+1,&HB3:POKEVARP
TR(AN$)+2,&H40:A$=STRING$(104,0):LSETA$=AN$:RETURN
```

The final requirement of the variable-passing technique is that for a program to receive the variables, it must 'GOSUB 52200' as its first command. The line that calls subroutine 52200 must contain no other program statements. Subroutine

52200 is the 'variable-receive' subroutine. It must know the fixed address that you've chosen for the start of variable storage. Knowing this, and knowing that A$ was the first variable you defined in the previous program, it reconstructs a temporary A$ to retrieve the 104 bytes of pointers that you saved in the string storage area of memory. Finally, it points AN$ to BASIC's communications region, and instantly 'pokes' the 104 bytes back in with an LSET command.

Variable Receive Subroutine

M 2 Note # 28

```
52200 A$="":FORA%=0TO2:POKEVARPTR(A$)+A%,PEEK(30000+A%+3):NEXT:A
N$="":POKEVARPTR(AN$),104:POKEVARPTR(AN$)+1,&HB3:POKEVARPTR(AN$)
+2,&H40:LSETAN$=A$:RETURN
```

You should change the '30000' in subroutine 52200 to the address you've chosen as the start of your variable list.

To see how the variable passing technique works, you can enter the following two programs. VARPASS/DEM initializes the variable list at memory location 30000. It then creates and displays several variables. Finally it calls the 'variable-pass' subroutine and runs the second program, VARPASS/RCV. The first action taken by VARPASS/RCV is to recover the variables generated by VARPASS/DEM. It does this by calling subroutine 52200. In line 2 of VARPASS/RCV, A$ is set back to a null string because the 104 bytes used for passing BASIC's pointers is no longer needed. Finally VARPASS/RCV displays the variables that it has recovered.

You should be aware that VARPASS/RCV, as it is written, cannot be run directly. The RUN"VARPASS/RCV" command must be executed by VARPASS/DEM.

VARPASS/DEM

Variable Passing Demonstration Program

M 2 Note # 27

M 2 Note # 28

```
0 'VARPASS/DEM
1 CLEAR150
2 GOSUB52000

20 C$="CAT"+"":D$="DOG"+""
30 DATA1,2,3,4,5,6,7,8,9,10
31 FORX=1TO10:READA%(X):NEXT
40 A!=123:A#=456

100 CLS
110 PRINT"PROGRAM 1 - VARIABLES ARE:"
120 PRINT"C$=";C$;TAB(20);"D$=";D$
130 PRINT"A%()=";:FORX=1TO10:PRINTA%(X);:NEXT:PRINT
140 PRINT"A!=";A!;TAB(20);"A#=";A#
200 GOSUB52100:RUN"VARPASS/RCV"

52000 A$="":FORA%=1TO3:A$=A$+MKI$(30000):NEXT:AN$="XXXXXX":POKEV
ARPTR(AN$)+1,&HF9:POKEVARPTR(AN$)+2,&H40:LSETAN$=A$:A$="":RETURN

52100 AN$="":POKEVARPTR(AN$),104:POKEVARPTR(AN$)+1,&HB3:POKEVARP
TR(AN$)+2,&H40:A$=STRING$(104,0):LSETA$=AN$:RETURN
```

VARPASS/RCV
Variable Receiving Demonstration Program

```
0 'VARPASS/RCV
1 GOSUB52200
2 A$=""

100 CLS
110 PRINT"PROGRAM 2 - VARIABLES ARE:"
120 PRINT"C$=";C$;TAB(20);"D$=";D$
130 PRINT"A%()=";:FORX=1TO10:PRINTA%(X);:NEXT:PRINT
140 PRINT"A!=";A!;TAB(20);"A#=";A#
200 END
```

M 2 Note # 28

```
52200 A$="":FORA%=0TO2:POKEVARPTR(A$)+A%,PEEK(30000+A%+3):NEXT:A
N$="":POKEVARPTR(AN$),104:POKEVARPTR(AN$)+1,&HB3:POKEVARPTR(AN$)
+2,&H40:LSETAN$=A$:RETURN
```

The Ultimate Memory Saver

Large computers use sophisticated techniques that automatically load small blocks of program logic from disk as they are needed. This makes it possible to execute programs that are, in effect, larger than the available memory. With the subroutines and procedures we'll discuss in this section, you can do the same thing on your TRS-80! I'm sure you'll find, as I did, that when you implement these techniques, your programs will enter a whole new 'generation' of performance capabilities.

We'll call each group of BASIC program lines loaded with this technique an 'overlay' or 'sub-program' and refer to the lines that remain in memory as our 'master program'. Overlays can be loaded for limited operations or subroutines. They can also be major blocks of program logic which act as sub-programs. Here are some of the advantages of the BASIC program overlay technique:

1. You can, in effect, go from one 'program' to another, retaining all variables that are in use. You can also leave your disk files open as you roll in overlays.

2. Common routines and subroutines can remain in memory as you go from one sub-program to another. Because of this, you don't have to repeat your 'housekeeping' logic in each program, and – you don't need to repeat those subroutines that are 'standard' to the overall application in each program. Because you can look at every application as a group of modules, with little or no logic being repeated, you save disk space. Since you only load what you need, when you need it, your effective 'load' time may be faster.

3. Because your sub-programs share the same standard subroutines and housekeeping logic, you save time when you need to make modifications. Let's say, for example, you want to change a disk file layout. Instead of changing it in several different programs, you only need to change it once if you've got your disk handling subroutine in the master program.

4. Program execution speeds can improve because you have less text in memory at any one time. BASIC doesn't have to search as far when it receives a GOTO or GOSUB command. Since you will be able to reserve more space for string storage, you'll have fewer delays for string reorganization.

5. An overlay program can 'GOTO' or 'GOSUB' to any line in the master program. The master program can execute GOTO's or GOSUB's to any line in the overlay program. One overlay program can even load another.

6. You can make almost any large application run in as little as 1 K of memory! Of course you wouldn't want to run that 'tight' because performance would be seriously degraded by the continual loading of overlays from disk. But in practice, the ability to significantly reduce the memory space required for program text lets you have more space for string and variable storage, and, if you need it, more space for protected memory at the top of RAM.

We'll be discussing two methods for loading overlays. A 'top-loaded' overlay is loaded above the master program in memory. With the top-loaded method, all line numbers in the overlay must be higher than the highest line number in the master program. The top-loaded method also makes it very easy to load in more than one, stacking each above the other in memory.

A 'bottom-loaded' overlay is rolled in from disk below the master program in memory. All line numbers in a bottom-loaded overlay must be lower than the line numbers in the master program. I most often use bottom-loaded overlays because most of my standard subroutines are above line 30000 and I prefer to leave them in memory with my master program. Top-loaded overlays, however, are easier to understand and implement.

Here's an example of how I use bottom-loaded overlays in my general ledger system:

Starting at line 30000 I have the 'master program'. This master program is stored on disk as 'MENU/GL'. It contains all of my function call definitions, the master menu logic, (which lets the operator select the operation to be performed), and my standard subroutines. The standard subroutines used by the system provide the logic for disk file handling, keyboard entry, and video display formatting. Program overlays are loaded with a short routine at line 53000. It loads an overlay program from disk by file name and begins execution at line 1 of the overlay program.

Then, I have an overlay program for each major operation to be performed by the general ledger system. The line numbers in the overlay programs range from 0 to 29999. The overlay programs are:

```
"OPENFILE/GL"   -  To open all files upon startup.
"INQUIRY/GL"    -  To allow account additions, changes, and inquiries.
"INPUT/GL"      -  To allow entry of general ledger transactions.
"POST/GL"       -  To process transactions that have been entered.
"REPORTS1/GL"   -  To print certain standard general ledger reports.
"REPORTS2/GL"   -  To print another group of standard reports.
"BUDGETS/GL"    -  To allow entry of budget amounts.
"FORMAT/GL"     -  To allow custom formatting of financial statements.
"FINSTMTS/GL"   -  To print customized financial statements.
"CHECKINQ/GL"   -  To allow check register inquiries.
"CHECKREG/GL"   -  To print check register reports.
```

Each overlay program takes about 5K of memory or less, and the master program takes about 8K. All together, the system has about 63K of program logic, but no more than 13K is in memory at any one time. Using 'normal' techniques, it would be impossible to store all the programs on one 35-track single density disk, because standard routines would have to be repeated with each program.

What do I do with all the memory I save? I protect the top portion of RAM for my general ledger account nunbers. They are loaded from disk upon startup with the 'OPENFILE/GL' overlay. Because the account numbers are in memory, I can, in under a second, search for any account number, from any sub-program and access the proper disk record. Also, I've got plenty of space for arrays and variables.

As for performance, the operator thinks it's one program. There's just a slight delay of 5 seconds or so when a new function is selected.

To use BASIC program overlay techniques, you'll first need an understanding of the way that your computer stores programs in memory and on disk. Then you'll need to understand the theory behind each overlay technique. Finally, we'll be able to go into the specifics of how to use them. You'll find that once you know the theory, it's very easy to write and use overlay programs.

Top-Loaded Overlay Theory

The top-loaded overlay technique uses many of the same principles that we implemented when we discussed how to pass variables between programs. Here are the key ideas:

1. We decide upon a fixed address in memory to begin the variable list. Since the length of our program text will vary as we load in overlays of different lengths, we force the simple and array variable list to begin at an address that is just above the highest end-of-text we will have when the longest overlay is in memory.

2. Before loading an overlay program we determine the address of the next byte following our master program's text. We poke the beginning of text pointers at 40A4 and 40A5 with this address. Then we do a 'LOAD,R' for the overlay program, causing it to be loaded immediately following our master program text.

3. The 'LOAD,R' option loads and runs a program. It will leave disk files open, but under normal methods, it will clear all variables. To avoid clearing variables, immediately before the load, we store the critical pointers in a 104-byte string, up in the string storage area of memory.

These pointers, which during normal operation are between 40B3 and 411A, specify the current status of the variable list. Upon completion of the load, we move these pointers back into their normal storage area and our variables are restored.

4. The first instruction of each overlay program restores the beginning of text pointer so that it again points to the beginning of the master program. Upon completion of this poke, the master and overlay programs are both active and can operate as one!

Bottom-Loaded Overlay Theory

1. We decide on a fixed address in memory to begin our master program, so that we'll have enough space to load the longest overlay just below it in memory. Before loading the master program, a startup program is required to poke 40A4 and 40A5 with the desired beginning of text address for the master. (I also use this startup program to load any USR routines that I might need, as well as to allow the operator to enter the date.)

2. We load each overlay as required with a 'LOAD,R' command. Just before a load, though, we copy the critical pointers, starting at 40B3, into a 104-byte string up in the string storage area of memory and poke our beginning of text pointer so that it will point to the desired load address of our overlay.

3. The first task of an overlay is to determine its end-of-textand link its last line to the first line of the master program. Then it calls a subroutine in the master program to restore variables. The master and overlay programs are now ready to act as one!

Program Storage - Memory and Disk

Let's first consider the way that programs are normally stored and executed in your computer's memory. A general memory map looks something like this:

TRS-80 Memory Map

```
================Top of Memory================
      Area you protected with "MEMORY SIZE?"
-----------------------------------------------
      STRING STORAGE (allocated by "CLEAR")
-----------------------------------------------
      Working memory used by BASIC (Stack)
-----------------------------------------------
                    ARRAYS
-----------------------------------------------
               SIMPLE VARIABLES
-----------------------------------------------
            Your BASIC program's text
-----------------------------------------------
               DISK FILE BUFFERS
-----------------------------------------------
      Area used by disk operating system.
              LEVEL II BASIC (ROM)
==============Bottom of Memory==============
```

As you can see from the memory map, any program that you type in or load from disk will reside just above the disk file buffer area. When operating with disk BASIC, the beginning of text will vary according to the answer you give for 'HOW MANY FILES?' It will also vary according to which disk operating system you are using. TRSDOS 2.3 and NEWDOS 2.1 reserve 290 bytes per file, while NEWDOS80 reserves 301 and Model 3 TRSDOS 1.2 reserves 360. But under every DOS I've seen, you can get the beginning of text address by typing:

M 2 Note # 16

```
PRINT "BEGINNING OF TEXT IS: ";PEEK(&H40A4)+PEEK(&H40A5)*256
```

It will, for most operating systems, be somewhere between roughly **6400** (25600 decimal) and **7900** (30976 decimal).

You can get a rough idea of how many bytes your program text requires by estimating how long it is compared to the size of your video display. If for example, you typed in a short program and it fills up 1 complete video display (1024 bytes), the program is probably between 750 and 1000 bytes long.

You can also get an idea of the length of your program text by displaying the disk directory. When you look next to your program name in the directory, the number in the 'EOF' column shows how many 256-byte sectors it's using on disk, (that is, if you didn't save it in ASCII format.) If for example, your 'EOF' is 10, your program is about 2560 bytes long. This method for estimating your program text length is based on the fact that, when you SAVE a program, the computer copies an exact image of your program text from memory to disk, (inserting a 1-byte 'FF' as the first byte in the file.)

M 2 Note # 16

Now, we must consider how your program text is stored in memory. If you wish, you can type in a short program, go into 'DEBUG', figure the beginning of text address from the contents of **40A4** and **40A5** and display that address on your screen. In a nutshell, here's what you'll find for each line of your program:

1. The first 2 bytes of each program line is a 2-byte pointer giving the address of the next program line in memory. If this 2-byte pointer is zero, there is no next line – we're at the end of text.
2. The next 2 bytes specify the program line number. The line number is expressed in LSB, MSB format, so if you have a line 10, you'll see '**0A00**' with DEBUG.
3. Next, you'll find your tokenized program line. That is, each of the BASIC commands and functions (CLS, GOSUB, CVS, etc.) will have been changed to a 1-byte code. Any 'literals' though, such as quoted strings, numeric constants, and GOTO or GOSUB line numbers, will be shown in uncompressed ASCII format.
4. Finally, you'll find a 1-byte '**00**' to indicate the end of the line.

As we said before, when you SAVE your program, an exact image will be written to disk. Therefore, the address pointers from one line to the next will be recorded on disk exactly as they were in memory. When you LOAD a program that has been previously saved, BASIC recomputes these address pointers, just in case your beginning of text address has changed. It will have changed only if:

1. You've changed the 'HOW MANY FILES?' specification,
2. or changed from one DOS to another, or
3. poked in a different beginning-of-text address.

Also, during a LOAD or RUN, BASIC will clear any variables that you may have had in memory. It does this because your variable storage area starts just above the end of your program text. When you load a longer program than the one previously in memory, you'll overwrite variables that may have been active previously. When you load a shorter program, you've got additional memory in which to store variables.

How to Use Top-Loaded Overlays

As we discussed in the previous section, the top-loaded overlay technique lets us retain a master program in memory at the lower line numbers, with the ability to load overlay programs to the higher line numbers as we need them. In this section, we'll go over the procedures and the program logic you'll need. We'll also look at a program that demonstrates the techniques.

Required Steps

1. Decide how many files your application will require. From DOS READY, go into BASIC, specifying the number of files that you'll be needing.

2. Make a note of the beginning of text address your master program will use. Since you've just started up from DOS READY, it's currently in memory locations 40A4 and 40A5.

To get the LSB of the address, type:

M 2 Note # 16

```
PRINT PEEK(&H40A4)
```

To get the MSB of the address, type:

```
PRINT PEEK(&H40A5)
```

To get the address in decimal, type:

```
PRINT PEEK(&H40A4)+PEEK(&H40A5)*256
```

3. Decide on where you'll divide your line numbers between master program and overlay program. With the top-loaded overlay technique, I normally use lines 0 through 29999 for my master program and lines 30000 and above for my overlays. (The examples and instructions that follow assume that you are using this line numbering scheme.)

4. Estimate an address to use for the beginning of the variable list. To do so, you can load in a program that will be about the length of your master program and the longest overlay combined. (Leaving the 'HOW MANY FILES?' setting the same.) With the program now in memory, type:

```
CLEAR : A%=0 : PRINTVARPTR(A%)
```

The number displayed will be a good 'working' address for your variable list pointer, but you may want to add 1000 or so, just to be safe. You can 'fine-tune' later.

5. The first line of your master program should be the following:

```
1 CLEAR1000:GOSUB29000:GOSUB29998
```

You may replace the 1000 following the CLEAR command with whatever you'll require for string storage. Remember, though, that the overlay technique requires at least 104 bytes of string storage.

The GOSUB 29000 calls our variable-list pointer subroutine, so that all VARPTR's will be above the desired address. The GOSUB 29998 calls the subroutine in the last line of our master program. Its job is to compute the next byte address following our text and store it in the integer EP%. You will, of course, need to modify these line numbers if you've chosen a different numbering scheme.

You may have lines that precede the one we've shown, but remember that any variables used in preceding lines will be erased.

6. The last line in your master program must be the end of text computation subroutine.

End-of-Text Computation Subroutine

```
29998 A$="":EP%=VARPTR(A$):EP%=CVI(CHR$(PEEK(EP%+1))+CHR$(PEEK(E
P%+2)))+48:RETURN
```

Upon return from the end of text computation subroutine, assuming you have located it as the last line, EP% has the address of the next byte following the master program's text. You must type the line exactly as shown, because it figures the end of text as 48 bytes beyond the contents of A$.

7. You must insert subroutines 29000, 29100 and 29200 in your master program. Note that these are the variable passing subroutines that we discussed in a previous section, but they have been renumbered. Subroutine 29000 is the variable-list pointer subroutine, 29100 is the variable-pass subroutine and 29200 is the variable-receive subroutine.

Variable Passing Subroutines Renumbered

M 2 Note # 27

M 2 Note # 28

```
29000 A$="":FORA%=1TO3:A$=A$+MKI$(30000):NEXT:AN$="XXXXXX":POKEV
ARPTR(AN$)+1,&HF9:POKEVARPTR(AN$)+2,&H40:LSETAN$=A$:A$="":RETURN

29100 AN$="":POKEVARPTR(AN$),104:POKEVARPTR(AN$)+1,&HB3:POKEVARP
TR(AN$)+2,&H40:A$=STRING$(104,0):LSETA$=AN$:RETURN
```

M 2 Note # 28

```
29200 A$="":FORA%=0TO2:POKEVARPTR(A$)+A%,PEEK(30000+A%+3):NEXT:A
N$="":POKEVARPTR(AN$),104:POKEVARPTR(AN$)+1,&HB3:POKEVARPTR(AN$)
+2,&H40:LSETAN$=A$:RETURN
```

You must change the '30000' in line 29000 and the '30000' in line 29200 to the address that you've determined in step 4. This is the fixed address that we'll use for our variable list.

8. You must insert an overlay-loader routine. Lines 29300 and 29301 do the job. First the variables are saved by a call to subroutine 29100. Then a new beginning of text address is poked in. Finally, the overlay program

specified by FD$ is loaded from disk, and execution continues with the first line of that overlay.

Overlay Loader Routine

M 2 Note # 16

```
29300 GOSUB29100:POKE&H40A4,ASC(MKI$(EP%)):POKE&H40A5,ASC(MID$(M
KI$(EP%),2))
29301 LOADFD$,R
```

9. Each place in your master program's logic where you want to load and execute an overlay, you should load the file name into FD$ and GOTO 29300. For example, to load and run the overlay, 'INQUIRY/BAS:1' your command is:

```
FD$="INQUIRY/BAS:1":GOTO29300
```

It's important to note that you can't be in a subroutine when loading an overlay. The load routine reinitializes the 'RETURN' pointers. (Once the overlay is loaded, you can use subroutines whenever you wish.)

10. The first line of each overlay program must poke the beginning of text address to bring back the master program. Then it should call subroutine 29200 to restore all variables. Here's a sample first line for an overlay:

M 2 Note # 16

```
30001 POKE&H40A4,186:POKE&H40A5,104:GOSUB29200
```

The '186' in line 30001 should be replaced with the LSB of your master program text address. The '104' in line 30001 should be replaced with the MSB of your master program text address. You determined both of these values in step 2. I normally put a remark as line 30000 to identify the overlay program name.

11. There are no restrictions for the other lines of the overlay, just so that each line in the overlay is greater than the highest line number in the master program. You may freely use 'GOTO' and 'GOSUB' between master program and overlay.

Top-Loaded Overlay Demo

Here is a program that demonstrates the use of top-loaded overlays. From a master program, by menu selection, you can load in either of two overlays. Each overlay starts at line 30000, and is linked onto the master program. You can prove to yourself that it is working properly by pressing the break key. First, just the master program will be in memory. Then, the master program and overlay 1 will be in memory. Finally, the master program and overlay 2 will be in memory.

You will need to modify line 30001 in both overlays to correspond to the beginning of text pointer for the disk operating system and number of files you are using. (As shown, it is set for NEWDOS 2.1 with 3 files.) To get the numbers to use in place of the '186' and '104', simply type:

M 2 Note # 16

```
PRINT PEEK(&H40A4);PEEK(&H40A5)
```

When you have the programs on disk as OVERLAYT/DEM, OVERLAY1/TOV, and OVERLAY2/TOV, you may run the master program. You won't be able to directly load and run the overlay programs, because they are written to be used with the master.

As a general rule, when you are working with overlay and master programs, you should re-load the program from disk before making modifications. This prevents you from accidently saving a master program with an overlay appended to it, or saving an overlay program with a master program appended to it. Also, be sure that whenever you run the OVERLAYT/DEM program your beginning of text pointers are set properly. If you've pressed break before an overlay program has reset the pointers, the next time you try to run the master, it won't work.

OVERLAYT/DEM

Top-Loaded Overlay Demonstration (Master)

M 2 Note # 29
M 2 Note # 30

```
0 '"OVERLAYT/DEM"

1 CLEAR1000:GOSUB29000:GOSUB29998
10 SG$=STRING$(63,131)
100 CLS:PRINT"

OVERLAY DEMONSTRATION
";SG$
110 PRINT"
<1> LOAD OVERLAY 1
<2> LOAD OVERLAY 2

";SG$
180 PRINT@832,"PRESS THE NUMBER OF YOUR SELECTION...";
190 PRINT@896,CHR$(31);:LINEINPUTA$:A%=VAL(A$):IFA%=0THEN190ELSE
ONA%GOTO1000,2000
191 GOTO190
1000 FD$="OVERLAY1/TOV":GOTO29300
2000 FD$="OVERLAY2/TOV":GOTO29300
```

M 2 Note # 27
M 2 Note # 28

```
29000 A$="":FORA%=1TO3:A$=A$+MKI$(30000):NEXT:AN$="XXXXXX":POKEV
ARPTR(AN$)+1,&HF9:POKEVARPTR(AN$)+2,&H40:LSETAN$=A$:A$="":RETURN
```

M 2 Note # 28

```
29100 AN$="":POKEVARPTR(AN$),104:POKEVARPTR(AN$)+1,&HB3:POKEVARP
TR(AN$)+2,&H40:A$=STRING$(104,0):LSETA$=AN$:RETURN

29200 A$="":FORA%=0TO2:POKEVARPTR(A$)+A%,PEEK(30000+A%+3):NEXT:A
N$="":POKEVARPTR(AN$),104:POKEVARPTR(AN$)+1,&HB3:POKEVARPTR(AN$)
+2,&H40:LSETAN$=A$:RETURN
```

M 2 Note # 16

```
29300 GOSUB29100:POKE&H40A4,ASC(MKI$(EP%)):POKE&H40A5,ASC(MID$(M
KI$(EP%),2))
29301 LOADFD$,R

29998 A$="":EP%=VARPTR(A$):EP%=CVI(CHR$(PEEK(EP%+1))+CHR$(PEEK(E
P%+2)))+48:RETURN
```

OVERLAY1/TOV

Top-Loaded Overlay Demonstration (Overlay 1)

M 2 Note # 16
M 2 Note # 29

```
30000 'OVERLAY1/TOV
30001 POKE&H40A4,186:POKE&H40A5,104:GOSUB29200

30100 CLS:PRINT"
THIS IS OVERLAY PROGRAM 1
";SG$
30110 PRINT"

PRESS <ENTER> TO RETURN TO THE MENU...";:LINEINPUTA$:GOTO100
```

OVERLAY2/TOV

Top-Loaded Overlay Demonstration (Overlay 2)

M 2 Note # 16
M 2 Note # 29

```
30000 'OVERLAY2/TOV
30001 POKE&H40A4,186:POKE&H40A5,104:GOSUB29200

30100 CLS:PRINT"
THIS IS OVERLAY PROGRAM 2
";SG$
30110 PRINT"

PRESS <ENTER> TO RETURN TO THE MENU...";:LINEINPUTA$:GOTO100
```

How to Use Bottom-Loaded Overlays

The bottom-loaded overlay technique lets us retain a master program in memory at the higher line numbers, with the ability to load overlay programs to the lower line numbers as we need them. In this section, we'll go over the procedures and program logic you'll need. We'll also look at a program that demonstrates the techniques. If you haven't tried the top-loaded technique yet, I suggest you get familiar with it first because it's easier to understand and implement.

Steps Required

1. Decide how many files your application will require. From DOS READY, go into BASIC, specifying the number of files that you'll be needing.

2. Make a note of the beginning of text address your overlay programs will use. Since you've just started up from DOS READY, it's currently in memory locations 40A4 and 40A5.

To get the LSB of the address, type:

M 2 Note # 16

```
PRINT PEEK(&H40A4)
```

To get the MSB of the address, type:

```
PRINT PEEK(&H40A5)
```

To get the address in decimal, type:

```
PRINT PEEK(&H40A4)+PEEK(&H40A5)*256
```

The address you get from these peeks will be the minimum address your overlay programs can use, assuming the same number of files and the same disk operating system. You can use a higher address if you wish. Sometimes it's desirable to select a higher address to be compatible with other disk operating systems.

3. Decide on a beginning of text address for your master program. To figure this address, you'll need to estimate the length of your longest overlay program and add it to the address you selected as your overlay beginning of text. It's helpful to take a disk directory and look at the EOF indicator of a program that is about the same length as your longest overlay will be. Multiplying the EOF indicator by 256 and adding 20 will give you a good estimate. During program development you'll want to estimate high. You can 'fine-tune' later.

4. Write a startup program that will be used to load and run your master program. The main purpose of the startup program is to poke in the beginning of text address for the master program, but you may also wish to insert logic for other purposes, such as loading USR routines. Here is an example showing the only startup program logic required to run a master program called 'MENU/GL' at address 28000:

```
10 POKE&H40A4,96:POKE&H40A5,109:POKE27999,0
20 RUN"MENU/GL"
```

You should replace the '96' in line 10 with the LSB of the beginning of text address for your master program. The '109' in line 10 should be replaced with the MSB of the desired master program beginning of text. The 27999 should be replaced with the address 1 byte below your master program beginning of text. Your master program's disk file name should be replaced in line 20.

5. Decide on where you'll divide your line numbers between master program and overlay program. With the bottom-loaded overlay technique, I normally use lines 0 through 29999 for my overlays, and lines 30000 and above for my master program. (The examples and instructions that follow assume that you are using this line numbering scheme.)

6. Estimate an address to use for the beginning of the variable list. To do so, you can poke 40A4 and 40A5 so that your beginning of text is at the location you'll be using for your master program. Then you can load in a program that will be about the length of your master program. With the program in memory, type:

```
CLEAR : A%=0 : PRINTVARPTR(A%)
```

The number displayed will be a good 'working' address for your variable list pointer, but you may want to add 1000 or so, just to be safe. You can 'fine-tune' later.

7. The first line of your master program should be the following:

```
30001 CLEAR1000:GOSUB52000
```

You may replace the 1000 following the CLEAR command with whatever you'll require for string storage. Remember, though, that our overlay technique requires at least 104 bytes of string storage.

The GOSUB 52000 calls our variable-list pointer subroutine, so that all VARPTR's will be above the desired address. You may have lines that precede the one shown, but remember that any variables used in preceding lines will be erased. I usually put a remark in line 30000 that tells the name of the program.

8. You must insert subroutines 52000, 52100, and 52200 in your master program. Note that these are the variable passing subroutines that we discussed in a previous section.

Variable Passing Subroutines

M 2 Note # 27

M 2 Note # 28

M 2 Note # 28

```
52000 A$="":FORA%=1TO3:A$=A$+MKI$(30000):NEXT:AN$="XXXXXX":POKEV
ARPTR(AN$)+1,&HF9:POKEVARPTR(AN$)+2,&H40:LSETAN$=A$:A$="":RETURN

52100 AN$="":POKEVARPTR(AN$),104:POKEVARPTR(AN$)+1,&HB3:POKEVARP
TR(AN$)+2,&H40:A$=STRING$(104,0):LSETA$=AN$:RETURN

52200 A$="":FORA%=0TO2:POKEVARPTR(A$)+A%,PEEK(30000+A%+3):NEXT:A
N$="":POKEVARPTR(AN$),104:POKEVARPTR(AN$)+1,&HB3:POKEVARPTR(AN$)
+2,&H40:LSETAN$=A$:RETURN
```

You must change the '30000' in line 52000 and the '30000' in line 52200 to the address that you've determined in step 6. This is the fixed address that we'll use for our variable list.

9. You must insert an overlay-loader routine. Lines 52300 and 52301 do the job. First the variables are saved by a call to subroutine 52100. Then the beginning of text address for our overlay poked in. Finally, the overlay program specified by FD$ is loaded from disk and execution continues with the first line of that overlay.

Overlay Loader Routine
M 2 Note # 16

```
52300 GOSUB52100:POKE&H40A4,120:POKE&H40A5,105:POKE26999,0
52301 LOADFD$,R
```

You should replace the '120' and '105' in line 52300 with the LSB and MSB of your overlay beginning of text address. (You got these two numbers in step 2.) The '26999' should be replaced with your overlay's beginning of text address minus 1.

10. Each place in your master program's logic where you want to load and execute an overlay, you should load the file name into FD$ and GOTO 52300. For example, to load and run the overlay, 'REPORTS/GL:1', your command is:

```
FD$="REPORTS/GL:1":GOTO52300
```

It is important to note that you can't be in a subroutine when loading an overlay. The load routine reinitializes the 'RETURN' pointers. (Once the overlay is loaded, you can use subroutines whenever you wish.)

11. The first line of each overlay program must call a subroutine to link the last line of the overlay to the first line of to the master. Subroutine 29999, which is the last line of the overlay, does this job. Then the variables must be restored with a call to subroutine 52200. Here's a sample first line for a bottom-loaded overlay:

```
1 GOSUB29999:GOSUB52200
```

I normally use line 0 in each overlay program as a remark, to identify the overlay program name.

12. The last line of each overlay must be the last line linker subroutine. Since, for our examples, 29999 is the highest line number in our overlays, it will contain the linker.

Last Line Linker Subroutine

```
29999 A$="":A%=PEEK(VARPTR(A$)+1):POKEVARPTR(A%)+1,PEEK(VARPTR(A
$)+2):POKEA%-8,96:POKEA%-7,109:RETURN
```

As we discussed earilier, the first 2 bytes of any BASIC program line point to the next program line. The last line linker subroutine computes its own address in memory and pokes the first 2 bytes with the beginning of text address for our master program. Upon return from the last line linker subroutine, our master program has been linked back into the program text.

You'll need to replace the '96' and the '109' in subroutine 29999 with the LSB and MSB of your master program beginning of text address, which you decided upon in step 3. In the example shown, a master program beginning of text address of 28000 is used.

13. You may insert any other program lines you need in the master and overlay programs, and you may freely use GOSUB's and GOTO's between your master program and overlay programs. You'll save a lot of time if you store a master program 'shell' and an overlay program 'shell' on disk in ASCII format. That way, you can simply merge them in when you want to develop a new program that uses overlay techniques.

Bottom-Loaded Overlay Demo

The demonstration programs that follow should run without modification on any of the popular operating systems for the TRS-80, as long as you specify no more than 3 files. The demonstration is started by running 'OVERLAYB/DEM'. It adjusts the beginning of text pointers and chains to 'MASTER/BOV'. The master program displays a menu that allows you to load either of 2 overlays, which are stored on disk as 'OVERLAY1/BOV' and 'OVERLAY2/BOV'. The programs set the following memory addresses:

M 2 Note # 31

```
Overlay program beginning of text:  27000  (LSB=120, MSB=105)
Master program beginning of text:   28000  (LSB= 96, MSB=109)
Variable list address:              30000
```

Remember, it's important to re-load your master or overlay program from disk before making modifications or corrections. This prevents you from accidentally saving any data other than the program itself.

OVERLAYB/DEM

Bottom-Loaded Overlay Demonstration (Startup)

```
0 'OVERLAYB/DEM
10 POKE&H40A4,96:POKE&H40A5,109:POKE27999,0
20 RUN"MASTER/BOV
```

OVERLAY1/BOV

Bottom-Loaded Overlay Demonstration (Overlay 1)

M 2 Note # 16

```
0 '"OVERLAY1/BOV"
1 GOSUB29999:GOSUB52200
100 CLS:PRINT"
THIS IS OVERLAY 1
";SG$
110 PRINT"

PRESS <ENTER> TO RETURN TO THE MENU...";:LINEINPUTA$:GOTO30100

29999 A$="":A%=PEEK(VARPTR(A$)+1):POKEVARPTR(A%)+1,PEEK(VARPTR(A
$)+2):POKEA%-8,96:POKEA%-7,109:RETURN
```

OVERLAY2/BOV

Bottom-Loaded Overlay Demonstration (Overlay 2)

M 2 Note # 29

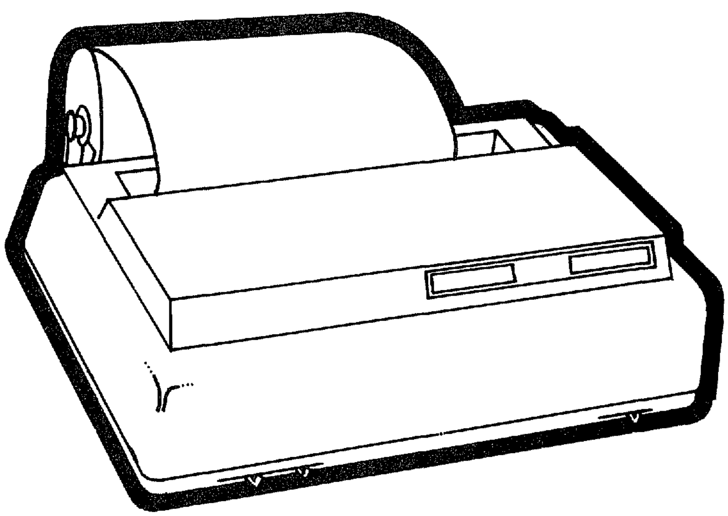
M 2 Note # 31

```
0 '"OVERLAY2/BOV"
1 GOSUB29999:GOSUB52200
100 CLS:PRINT"
THIS IS OVERLAY 2
";SG$
110 PRINT"

PRESS <ENTER> TO RETURN TO THE MENU...";:LINEINPUTA$:GOTO30100

29999 A$="":A%=PEEK(VARPTR(A$)+1):POKEVARPTR(A%)+1,PEEK(VARPTR(A
$)+2):POKEA%-8,96:POKEA%-7,109:RETURN
```

MASTER/BOV
Bottom-Loaded Overlay Demonstration (Master)

M 2 Note # 29
M 2 Note # 30
M 2 Note # 31

```
30000 '"MASTER/BOV"

30001 CLEAR1000:GOSUB52000

30010 SG$=STRING$(63,131)
30100 CLS:PRINT"
BOTTOM-LOADED OVERLAY DEMONSTRATION
";SG$
30110 PRINT"
<1> LOAD OVERLAY 1
<2> LOAD OVERLAY 2

";SG$
30180 PRINT@832,"PRESS THE NUMBER OF YOUR SELECTION...";
30190 PRINT@896,CHR$(31);:LINEINPUTA$:A%=VAL(A$):IFA%=0THEN30190
ELSEONA%GOTO31000,32000
30191 GOTO30190
31000 FD$="OVERLAY1/BOV":GOTO52300
32000 FD$="OVERLAY2/BOV":GOTO52300

52000 A$="":FORA%=1TO3:A$=A$+MKI$(30000):NEXT:AN$="XXXXXX":POKEV
ARPTR(AN$)+1,&HF9:POKEVARPTR(AN$)+2,&H40:LSETAN$=A$:A$="":RETURN

52100 AN$="":POKEVARPTR(AN$),104:POKEVARPTR(AN$)+1,&HB3:POKEVARP
TR(AN$)+2,&H40:A$=STRING$(104,0):LSETA$=AN$:RETURN

52200 A$="":FORA%=0TO2:POKEVARPTR(A$)+A%,PEEK(30000+A%+3):NEXT:A
N$="":POKEVARPTR(AN$),104:POKEVARPTR(AN$)+1,&HB3:POKEVARPTR(AN$)
+2,&H40:LSETAN$=A$:RETURN

52300 GOSUB52100:POKE&H40A4,120:POKE&H40A5,105:POKE26999,0
52301 LOADFD$,R
```



Number Crunchers and Munchers

Regardless of the application, almost every program involves some addition, subtraction, multiplication or division. Whether you are computing an accounting balance, a scientific formula or the number of points accumulated by each player in a computer game, you soon become accustomed to talking to your computer with numbers and formulas. But the problem presented by the application is only the beginning. Just to get the computer to print data where we want it on the video display or to retrieve the desired information from a disk file or array, many numbers and formulas can be involved.

This chapter provides many tricks, function calls and subroutines that can save you hours of programming time. We'll be looking at some mathematical techniques that are often required for everyday programs. In addition, we'll discuss ways to compress numeric data for more efficient disk and memory storage and ways of achieving dramatic speed improvements when adding or printing numbers. Finally, have you ever seen a computer book that didn't cover the subject of hexadecimal and other base conversions? We'll be discussing some efficient subroutines and function calls that can handle this subject once and for all!

Remainder Function Calls

You will find that the remainder obtained when you divide one number by another has many applications in programming. On the video display, for example, when we divide a PRINT@ position by 64, the remainder is the horizontal tab position. In disk applications, when we divide the desired logical record number by the number of logical records per physical record, the remainder shows us the number of preceding logical records within the physical record. In base conversion routines, we are repeatedly dividing by the base to get the remainder.

BASIC provides no automatic way to get remainders. You've got to use a simple formula. The following function, FNRE# (A1#,A2#), computes the remainder of the first argument, A1#, divided by the second argument, A2#:

Remainder Function

```
35 DEFFNRE#(A1#,A2#)=A1#-INT(A1#/A2#)*A2#
```

As an example, if we set A# equal to FNRE# (154,10), A# equals the remainder of 154 divided by 10 or 4. Be careful that your program does not allow 0 as the second argument, because a 'division by zero' error will result.

You can, if you wish, change the FNRE# function call to single precision or integer by changing the # symbol to one of the other symbols. Or, you can eliminate the '#' and DEFINT, DEFSNG or DEFDBL the variable you wish to use before calling the remainder function. Like any other function call, you can also simply use it as a model, including the logic in any program line where needed.

Using 'ANDNOT' to Find Remainders

Here's a convenient trick that lets you find the remainder of any integer divided by a power of 2.

For any integer 'A%',
the remainder of A%/2 is given by the expression A% ANDNOT −2
the remainder of A%/4 is given by the expression A% ANDNOT −4
the remainder of A%/8 is given by the expression A% ANDNOT −8
etc . . .

When you want to find whether a number is even or odd, you can use:

```
IF A% ANDNOT-2 THEN PRINT "ODD" ELSE PRINT "EVEN"
```

When you want to test whether a year is a leap year, you can use:

```
IF(Y% ANDNOT-4)=0 THEN PRINT "LEAP YEAR"
```

If you want to avoid 'illegal function call' errors when using PRINT@ addresses, you can force any print position to be between 0 and 1023 with the command:

```
PRINT@ABS(PO%ANDNOT-1024),A$
```

Rounding Functions

Your 'PRINT USING' command handles rounding for you on formatted and printed output, but it is often useful to insure that the numbers you're handling internally are the same as those printed. We will be discussing two rounding functions. The first of these, FNRW#, rounds any number to an integer whole number. If the decimal portion of the number is greater than or equal to 0.5 the number will be rounded up to the next whole number if positive or down to the next whole number if negative. If the decimal portion is less than 0.5, the decimals will be truncated.

The second function, FNRD#, rounds to 2 decimal places for the proper handling of dollars and cents. The result will be the nearest cent, taking into account positive and negative numbers.

In programming rounding functions, the first challenge is to properly handle positives and negatives. If you're dealing with double precision numbers there is an even bigger challenge – avoiding the 'garbage' that BASIC can sometimes put into the decimal portion of your number. The result of much experimentation and testing, FNRW# and FNRD# handle these two problems.

Rounding Functions

```
Round to nearest whole number:
10 DEFFNRW#(A1#)=FIX((FIX(A1#*10#)+SGN(A1#)*5)/10#)

Round to nearest cent:
11 DEFFNRD#(A1#)=FIX((FIX(A1#*1000#)+SGN(A1#)*5)/10#)/100#
```

To use the rounding functions for single precision numbers, you can change each '#' symbol to a '!'. You'll find that that these functions are more than 2 times faster in single precision.

Rounding Down

This function, FNFL#, requires two arguments. It finds the first multiple of the second argument that is less than or equal to the first argument. Let's say, for example that we want to round a number down to the nearest 100. FNFL# (392, 100) will return 300. FNFL# (3100, 100) will return 3100.

If we want to find the corresponding left position on the video display for any position between 0 and 1023, we can use the function below. FNFL# (514, 64) for example, returns 512. That is, 512 is the PRINT@ position that begins the line containing position 514.

First Multiple Less Than or Equal Function

```
DEFFNFL#(A1#,A2#)=INT(A1#/A2#)*A2#
```

You may change this function for single precision or integer variable types. Just change the # symbols.

Rounding Up

The FNFM# function is similar to the FNFL# function, except that it finds the first multiple of the second argument that is greater than the first argument. To illustrate how the FNFM# function works, FNFM# (3022, 100) will return 3100. FNFM# (3100,100) will return 3200. This function will give the left-most position of the first video display line beyond position defined by the integer, PO%.

First Multiple Greater Function

```
DEFFNFM#(A1#,A2#)=INT(A1#/A2#)*A2#+A2#
```

Again, you may change the symbols if you want to use single precision or integer types.

Saving Space With 1-Byte Numbers

If you know that a numeric field to be stored on disk will always contain an integer in the range 0 to 255, you can use the CHR$ and ASC functions instead of the MKI$ and CVI functions. Rather than using two bytes, you'll be using just one!

If you want to store an array in memory containing integers in the range 0 to 255, you can store up to 255 elements in a string. To initialize the 'array-string', create a string of zeros with a length corresponding to the number of elements you need. Then to put an integer amount, 'A%', into element position, 'E%', of string, 'X$', you can use the command, MID$(X$,E%,1) = CHR$(A%). To recall an amount, A%, from element position E%, you can use the command, A% = ASC(MID$(X$,E%)). You won't be using much more than half the memory and, by avoiding standard arrays, in many cases you can speed up program execution.

Saving Space With 2-Byte Numbers

As you know, an integer-type variable may range from −32768 to 32767. Integers require 2 bytes for both disk storage in random files and memory if we don't count the memory overhead for each variable name. If we need only positive integers, we can convert the negatives so that we can store a range of 0 to 65535 in 2 bytes. Any math we do, however, will have to be done in single precision.

To work with 2-byte unsigned integers, we will need 2 function calls. The function below converts a 4-byte unsigned single precision whole number ranging from 0 to 65535 to a signed integer that can be stored in 2 bytes. FNIS! converts a 2-byte signed integer to a 4-byte, unsigned single precision number.

2-Byte Storage of Unsigned Integers

```
Convert unsigned single to integer:
15 DEFFNSI%(A1!)=-((A1!>32767)*(A1!-65536))-((A1!<32768)*A1!)
Convert integer to unsigned single:
16 DEFFNIS!(A1%)=-((A1%<0)*(65536+A1%)+((A1%>=0)*A1%))
```

Let's suppose you want to store the number 62500 in a 2-byte disk field, FX$. You're command is:

```
LSET FX$ = MKI$(FNSI%(62500))
```

To recall and print it your command is:

```
PRINT FNIS!(CVI(FX$))
```

As another example, let's say you've got an integer array and you want to store unsigned numbers up to 65535 in it. If B! contains 42000, you can store it in element 1 of the array using the command:

```
I%(1)=FNSI%(B!)
```

To put the contents of the array element into variable A! for printing or computing purposes, you can say:

```
A!=FNIS!(I%(1))
```

If you need unsigned decimal numbers, you can also store them in 2 bytes if you use an 'assumed' decimal. You can, for example, store prices ranging from $000.00 to $655.35 by multiplying by 100 before the compression and dividing by 100 after the uncompression.

Saving Space With Unsigned Integers

Here are 4 functions that let you compress and uncompress very large unsigned integers for storage in 3 or 4 bytes on disk. Be sure that the numbers are whole numbers (without any decimal) and that you observe the limits. The functions are:

NAME	CONVERSION PERFORMED	LIMITS
FNU3$(A#)	From A# to a 3-byte string	0 to 16,777,215
FNU3#(A$)	3-byte string to double precision	
FNU4$(A#)	From A# to a 4-byte string	0 to 4,294,967,295
FNU4#(A$)	4-byte string to double precision	

Within your program, you'll work with the numbers in double precision. As an example, let's assume you have a variable, N#, that contains 12345678. To store it on disk in a 3 byte field, FX$, you would LSET FX$ = FNU3$(N#). To get it back later, your command could be, N# = FNU3$(FX$).

These 4 functions call the 2-byte unsigned functions which we discussed earlier, so you will also need to define them in your program.

3 and 4 Byte Unsigned Integer Functions

```
Compress A# to 3-byte string:
21 DEFFNU3$(A#)=CHR$(A#-INT(A#/256)*256)+MKI$(FNSI%(INT(A#/256))
)

Convert 3-byte string, A$ to double precision:
22 DEFFNU3#(A$)=ASC(A$)+FNIS!(CVI(MID$(A$,2)))*256#

Compress A# to 4-byte string:
17 DEFFNU4$(A#)=MKI$(FNSI%(INT(A#/65536)))+MKI$(FNSI%(A#-INT(A#/
65536)*65536))

Convert 4-byte string, A$ to double precision:
18 DEFFNU4#(A$)=FNIS!(CVI(A$))*65536#+FNIS!(CVI(MID$(A$,3)))
```

Saving Space With Signed Integers

You can use the 6 function calls that follow to store large signed integers in 3 or 4 bytes. The procedures for using them in programs are exactly the same as those for the 3 and 4 byte unsigned compressions, except that the absolute limits are lower:

NAME	CONVERSION PERFORMED	LIMITS (+ AND -)
FNS3$(A#)	From A# to a 3-byte string	Ø to 8,ØØØ,ØØØ
FNS3#(A$)	3-byte string to double precision	
FNDI$(A#)	From A# To a 4-byte string	Ø to 1,Ø7Ø,ØØØ,ØØØ
FNDI#(A$)	4-byte string to double precision	
FNS4$(A#)	From A# to a 4-byte string	Ø to 2,1ØØ,ØØØ,ØØØ
FNS4#(A$)	4-byte string to double precision	

Note that FNDI and FNS4 provide two different methods of storing signed integers in 4 bytes. FNDI stores the double precision number as 2 signed integers. Though FNDI has a smaller range, it is faster and it does not require that the other functions be present in your program. You will need to define the 2-byte integer compression functions in your program if you use the FNS4 functions.

These function calls are very useful in accounting applications if you use an assumed decimal place. FNDI, for example, lets you handle positive or negative dollar amounts up to $10,700,000.00 and you need only half the disk or memory space required for normal double precision storage! For printing purposes, you can divide by 100 or you can use some of the special print formatting function calls, such as FNDF$, that are discussed later in this chapter.

Be sure that you use FNDI$ and FNDI# together or FNS4$ and FNS4# together. They are not interchangeable!

3 and 4 Byte Signed Integer Functions

```
Compress A# to 3-byte string:
23 DEFFNS3$(A#)=CHR$(ABS(A#-INT(A#/256)*256))+MKI$(INT(A#/256))

Convert 3-byte string, A$, to double precision:
24 DEFFNS3#(A$)=ASC(A$)+CVI(MID$(A$,2))*256#


Compress A# to 4-byte string (Double integer method):
25 DEFFNDI$(A#)=MKI$(A#/32768)+MKI$(A#-INT(A#/32768)*32768)

Convert 4-byte string, A$ to double precision:
26 DEFFNDI#(A$)=CVI(A$)*32768#+CVI(MID$(A$,3))


Compress A# to 4-byte string:
19 DEFFNS4$(A#)=MKI$(INT(A#/65536#))+MKI$(FNSI%(A#-INT(A#/65536#
)*65536#))

Convert 4-byte string, A$, to double precision:
20 DEFFNS4#(A$)=CVI(A$)*65536#+FNIS!(CVI(MID$(A$,3)))
```

High-Speed 'PRINT USING' Functions

The 'PRINT USING' command is one of the most powerful features of BASIC, but it can also be very slow for the formatted printing of double precision numbers. FNDF$ is a function that formats a double precision number for dollars and cents. I've found that it is up to 3 times faster than 'PRINT USING'.

FNDF$ creates a string which you can PRINT or LPRINT. It requires 4 arguments:

Argument 1 is the double precision number you want formatted. It must be a whole number, with no decimal. The decimal will be assumed to be 2 places from the right.

Argument 2 is an integer that specifies the number of places to be formatted to the left of the decimal.

Argument 3 is a string that specifies a symbol to be appended to the right of the formatted number if it is positive or zero.

Argument 4 is a string that specifies a symbol to be appended to the right of the formatted number if it is negative.

Dollar Format Print-Using Function

```
15 DEFFNDF$(A1#,A2%,A3$,A4$)=RIGHT$(STRING$(A2%," ")+LEFT$(STR$(
ABS(A1#)),LEN(STR$(A1#))-2),A2%)+"."+RIGHT$("0"+MID$(STR$(ABS(A1
#)),2),2)+LEFT$(A3$,-(A1#>=0)*LEN(A3$))+LEFT$(A4$,-(A1#<0)*LEN(A
4$))
```

The chart below gives some examples to help you see how the FNDF$ function works. You should note that this function call does no rounding and if the number overflows the format the leftmost digits will be truncated.

```
If N#=302454, FNDF$(N#,6," DR"," CR") returns "  3024.54 DR"
If N#=-32352, FNDF$(N#,6," DR"," CR") returns "   323.52 CR"
If N#=12345,  FNDF$(N#,4," ","-")     returns " 123.45 "
If G#=-12345, FNDF$(G#,4," ","-")     returns " 123.45-"
If X#=0,      FNDF$(X#,4," ","-")     returns "    .00 "
```

In some applications, accountants like to use brackets to indicate that a dollar amount is negative or that it has a credit balance. The FNBN$ function works like the FNDF$ function, except that brackets enclose the amount when it is negative. Two arguments are required:

Argument 1 provides the double precision integer to be printed.

Argument 2 specifies the number of digit positions to the left of the decimal point.

Brackets-if-Negative Print-Using Function

```
25 DEFFNBN$(A1#,A2%)=RIGHT$(STRING$(A2%," ")+LEFT$("(",ABS(A1#<0
))+LEFT$(" ",ABS(A1#>=0))+MID$(STR$(ABS(A1#)),2,-((LEN(STR$(A1#)
)-3)>0)*(LEN(STR$(A1#))-3)),A2%)+"."+RIGHT$("0"+MID$(STR$(ABS(A1
#)),2),2)+LEFT$(")",ABS(A1#<0))+LEFT$(" ",ABS(A1#>=0))
```

Note that if you type in the 'brackets if negative' function call you will find that it is too long to fit in a BASIC program line unless you use the 'edit' capability. To do it, first type in as much as you can. Then go into edit mode and use the 'X' command to move to the end of the line, where you can continue typing.

The chart below gives you some examples of strings created by the FNBN$ function. The cautions we discussed for the FNDF$ function apply to the FNBN$ function as well.

```
If N#=-8166,  FNBN$(N#,4) returns " (81.66)"
If N#=12500,  FNBN$(N#,4) returns " 125.00 "
If N#=0,      FNBN$(N#,4) returns "    .00 "
If X#=333,    FNBN$(X#,2) returns " 3.33 "
If X#=-333,   FNBN$(X#,2) returns "(3.33)"
```

High-Speed Integer Formatting

This function call, FNNF$, is similar to the dollar format function. It can be used when you want execution speed improvements in the right justified printing of double precision integers where no decimal point is required. When you are using double precision numbers, it can be from 3 to 6 times faster than 'PRINT USING'. FNNF$ creates a string, based on 4 arguments:

Integer Format Print-Using Function

```
35 DEFFNNF$(A1#,A2%,A3$,A4$)=RIGHT$(STRING$(A2%," ")+MID$(STR$(A
1#),2),A2%)+LEFT$(A3$,-(A1#>=0)*LEN(A3$))+LEFT$(A4$,-(A1#<0)*LEN
(A4$))
```

Argument 1 specifies the double precision integer to be formatted.

Argument 2 specifies the maximum number of digits.

Argument 3 provides a string to be appended to the right of the number, if it is positive.

Argument 4 provides a string to be appended to the right of the number, if it is negative.

Here are some examples of numbers formatted into strings with the integer format print function:

```
If N#=-12345, FNNF$(N#,7,"+","-") returns "  12345-"
If N#=-33,    FNNF$(N#,7,"+","-") returns "     33-"
If A#=12345,  FNNF$(A#,7,"+","-") returns "  12345+"
If B#=301,    FNNF$(B#,7," ","-") returns "    301 "
If B#=301,    FNNF$(B#,3," ","-") returns "301 "
```

Special Purpose 'PRINT USING' Functions

It is most economical to store telephone numbers as numeric data. I commonly use 8-byte double precision to store the 10 digits in a telephone number, but with some manipulation you might be able to get it down to 5 bytes.

To let the operator enter a number in telephone format, you can use the formatted inkey routine that is discussed in this book. To display a number in telephone format, you can use the FNTF$(A#) function. It creates a 12-byte string that you can PRINT or LPRINT. Here are some examples:

```
FNTF$(1234567890) = "(123) 456-7890"
FNTF$(1234567)    = "(000) 123-4567"
FNTF$(0)          = "(000) 000-0000"
```

Telephone Format Print-Using Function

```
15 DEFFNTF$(A1#)="("+MID$(RIGHT$("0000000000"+MID$(STR$(A1#),2),
10),1,3)+") "+MID$(RIGHT$("0000000000"+MID$(STR$(A1#),2),10),4,3
)+"-"+MID$(RIGHT$("0000000000"+MID$(STR$(A1#),2),10),7,4)
```

If you study the FNTF$ function you'll see how you can design a print function for just about any special type of number. FNSO$, for example, formats a double precision number into a string in social security format. If SS# contains 123456789, FNSO$(SS#) will return '123-45-6789'.

Social Security Format Print-Using Function

```
25 DEFFNSO$(A1#)=MID$(RIGHT$("000000000"+MID$(STR$(A1#),2),9),1,
3)+"-"+MID$(RIGHT$("000000000"+MID$(STR$(A1#),2),9),4,2)+"-"+MID
$(RIGHT$("000000000"+MID$(STR$(A1#),2),9),6,4)
```

Instantly Sum Arrays

The SUMSNG USR routine lets you instantly find the sum of all elements in a singly dimensioned array of single precision numbers. It can add the contents of a 2000 element array in about 1 second!

This USR routine is 47 bytes long and fully relocatable. You can load it into any protected memory address or execute it as a 'magic array'. The SUMSNG routine calls three ROM subroutines that handle single precision arithmetic. If you want more information about ROM subroutines, I recommend that you get a copy of *Microsoft BASIC Decoded*, by James Farvour.

Before you can use the SUMSNG routine, you must set up a single precision variable in your program that will hold the sum that is computed. For example, if you want your sum to be placed into SM!, initialize the variable with the command 'SM! = 0'. You only need to do this once in your program.

Then, if you are executing SUMSNG as a magic array USR routine, you should load an integer array with the 24 numbers listed below, and you set the 18th element equal to the VARPTR of your single precision sum variable. (In our example, VARPTR(SM!)). Again, you only have to do this once in your program.

Or, if you are executing SUMSNG as a regular USR routine in protected memory, you should poke the VARPTR of your sum variable into the 37th and 38th bytes of the routine.

Now, let's say you want to sum the array, SA!. Your command is,

```
J=USR0(VARPTR(SA!(0)))
```

The sum will be in the single precision variable you specified. (In our example it will be in SM!.) The argument to be passed to the USR routine is always the VARPTR to element 0 of the array to be summed. If you are using the magic array method, be sure that the dummy integer variable, ('J%' in our example) has been previously initialized and that you DEFUSR the first element of your magic array just before you execute it.

Here is a program that demonstrates the mechanics of setting up and using the SUMSNG USR routine within a program. In line 20 we initialize the sum variable, SM!. Line 31 loads the SUMSNG routine into the integer array, UX%. Line 100 generates a 1000 element array containing random numbers. Line 120 calls the USR routine to compute the sum.

SUMSNG/DEM
Array Summing Demonstration Program
M 2 Note # 23
M 2 Note # 32

```
0 'SUMSNG/DEM
10 DEFINTA-Z
20 SM!=0:DIMSA!(999)
30 DATA32717,-6902,17963,20011,-6687,-12859,2481,-7743,30987,104
16,4366,4,-6887,-12859,2498,5837,6151,4587,0,8481,321,4,-20243,2
01
31 DIMUX(23):FORX=0TO23:READUX(X):NEXT:UX(18)=VARPTR(SM!)
100 FORX=0TO999:SA!(X)=RND(9)/RND(9):PRINTX,SA!(X):NEXT
110 LINEINPUT"PRESS ENTER TO SUM THE ARRAY...";A$
120 J=0:DEFUSR1=VARPTR(UX(0)):J=USR1(VARPTR(SA!(0)))
130 PRINTSM!:GOTO110
```

SUMSNG
Single Precision Array Summing USR Subroutine
M 2 Note # 23
M 2 Note # 32

```
Magic Array Format, 24 elements:

 32717  -6902  17963  20011  -6687 -12859   2481  -7743  30987
 10416   4366      4  -6887 -12859   2498   5837   6151   4587
     0   8481    321      4 -20243    201

Poke Format, 47 bytes:

 205 127  10 229  43  70  43  78 225 229 197 205 177   9 193 225
  11 121 176  40  14  17   4   0  25 229 197 205 194   9 205  22
   7  24 235  17   0   0  33  33  65   1   4   0 237 176 201
```

```
                ;
FF00                    ORG     0FF00H       ;ORIGIN - RELOCATABLE
FF00 CD7F0A             CALL    0A7FH        ;GET VARPTR TO ELEMENT 0 OF ARRAY
FF03 E5                 PUSH    HL           ;SAVE IT ON STACK
FF04 2B                 DEC     HL           ;
FF05 46                 LD      B,(HL)       ;
FF06 2B                 DEC     HL           ;
FF07 4E                 LD      C,(HL)       ;BC HAS DIMENSION + 1
FF08 E1                 POP     HL           ;RESTORE VARPTR TO ELEMENT 0
FF09 E5                 PUSH    HL           ;SAVE IT ON STACK AGAIN
FF0A C5                 PUSH    BC           ;SAVE COUNT
FF0B CDB109             CALL    09B1H        ;MOVE FIRST ELEMENT TO WORK AREA
FF0E C1         LOOP    POP     BC           ;RESTORE COUNT
FF0F E1                 POP     HL           ;RESTORE POINTER
FF10 0B                 DEC     BC           ;DECREMENT COUNT
FF11 79                 LD      A,C          ;
FF12 B0                 OR      B            ;TEST IF COUNT IS ZERO
FF13 280E               JR      Z,ENDIT      ;IF SO, GO TO END
FF15 110400             LD      DE,04H       ;
FF18 19                 ADD     HL,DE        ;ADD 4 TO POINTER
FF19 E5                 PUSH    HL           ;SAVE POINTER
FF1A C5                 PUSH    BC           ;SAVE COUNT
FF1B CDC209             CALL    09C2H        ;LOAD NEXT ELEMENT INTO BC/DE
FF1E CD1607             CALL    0716H        ;ADD BC/DE TO WORK AREA
FF21 18EB               JR      LOOP         ;REPEAT
FF23 110000     ENDIT   LD      DE,0000H     ;LOAD VARPTR OF DESTINATION VAR
FF26 212141             LD      HL,04121H    ;LOAD ADDRESS OF WORK AREA
FF29 010400             LD      BC,04H       ;PREPARE TO MOVE 4 BYTES
FF2C EDB0               LDIR                 ;MOVE FROM WORK AREA TO DEST VAR
FF2E C9                 RET                  ;RETURN TO BASIC
0004                    END                  ;
00000 TOTAL ERRORS
```

Instantly Sum Double Precision Arrays

The SUMDBL USR routine is similar to the SUMSNG USR routine. It lets you instantly find the sum of all elements in a single dimensioned array of double precision numbers. It can add the contents of a 1000-element array in about one second!

The SUMDBL routine is 59 bytes long and fully relocatable. It, like the SUMSNG routine, uses calls to some of the ROM subroutines. You can use the same procedures for setting up and using this routine as discussed for the SUMSNG routine, except you will be working with double precision numbers.

If you are using the magic array method, be sure to load element 24 with the VARPTR to your destination variable, a double precision variable that will contain the computed sum of the array. If you are using SUMDBL as a regular USR routine in protected memory, you will need to POKE the VARPTR of your destination variable into the 49th and 50th bytes of the routine.

SUMDBL
Double Precision Array Summing USR Subroutine

M 2 Note # 23

M 2 Note # 32

```
Magic Array Format, 30 elements:

 32717  -6902  17963  20011 -10799  16069  12808  16559   7457
-12991   2515 -11839  30987  10416   8466      8  -6887  -5179
 10017 -12991   2515  30669   6156   4583      0   7457    321
     8 -20243    201

Poke Format, 59 bytes:

 205 127  10 229  43  70  43  78 209 213 197  62   8  50 175  64
  33  29  65 205 211   9 193 209  11 121 176  40  18  33   8   0
  25 229 197 235  33  39  65 205 211   9 205 119  12  24 231  17
   0   0  33  29  65   1   8   0 237 176 201
```

```
FF00          00090         ORG    0FF00H       ;ORIGIN - RELOCATABLE
FF00 CD7F0A   00100         CALL   0A7FH        ;GET VARPTR TO ELEMENT 0 OF ARRAY
FF03 E5       00110         PUSH   HL           ;SAVE IT ON STACK
FF04 2B       00120         DEC    HL           ;
FF05 46       00130         LD     B,(HL)       ;
FF06 2B       00140         DEC    HL           ;
FF07 4E       00150         LD     C,(HL)       ;BC HAS DIMENSION + 1
FF08 D1       00160         POP    DE           ;GET VARPTR TO ELEMENT 0
FF09 D5       00170         PUSH   DE           ;SAVE IT ON STACK AGAIN
FF0A C5       00180         PUSH   BC           ;SAVE COUNT
FF0B 3E08     00190         LD     A,08H        ;DBL PRECISION TYPE CODE TO ACCUM
FF0D 32AF40   00200         LD     (40AFH),A    ;SET THE TYPE
FF10 211D41   00210         LD     HL,411DH     ;LOAD WORK AREA 1 ADDRESS
FF13 CDD309   00220         CALL   09D3H        ;MOVE FIRST ELEMENT TO WORK 1
FF16 C1       00230 LOOP    POP    BC           ;RESTORE COUNT
FF17 D1       00240         POP    DE           ;RESTORE POINTER
FF18 0B       00250         DEC    BC           ;DECREMENT COUNT
FF19 79       00260         LD     A,C          ;
FF1A B0       00270         OR     B            ;TEST IF COUNT IS ZERO
FF1B 2812     00280         JR     Z,ENDIT      ;IF SO, GO TO END
FF1D 210800   00290         LD     HL,08H       ;
FF20 19       00300         ADD    HL,DE        ;ADD 8 TO POINTER
FF21 E5       00310         PUSH   HL           ;SAVE POINTER
FF22 C5       00320         PUSH   BC           ;SAVE COUNT
FF23 EB       00330         EX     DE,HL        ;NEXT ELEMENT POINTER TO DE
FF24 212741   00340         LD     HL,4127H     ;WORK 2 ADDRESS IN HL
FF27 CDD309   00350         CALL   09D3H        ;LOAD NEXT ELEMENT TO WORK 2
FF2A CD770C   00360         CALL   0C77H        ;ADD WORK 2 TO WORK 1
FF2D 18E7     00370         JR     LOOP         ;REPEAT
FF2F 110000   00380 ENDIT   LD     DE,0000H     ;LOAD VARPTR OF DEST VARIABLE
FF32 211D41   00390         LD     HL,411DH     ;LOAD ADDRESS OF WORK AREA 1
FF35 010800   00400         LD     BC,08H       ;PREPARE TO MOVE 8 BYTES
FF38 EDB0     00410         LDIR                ;MOVE WORK AREA 1 TO DESTINATION
FF3A C9       00420         RET                 ;RETURN TO BASIC
0008          00430         END                 ;
00000 TOTAL ERRORS
```

Sum Partial Arrays

SUMSNG and SUMDBL, as they are shown in the previous sections, add entire arrays. They determine the number of elements to be summed by accessing the dimension indicator, which is a 2-byte integer located immediately below array element 0 in memory.

It can often be useful, for example, to sum the first 200 elements of a 1000 element array. A slight modification is possible that works for both the SUMSNG and SUMDBL routines. Simply change the 3rd element of the magic array from '17963' to '256'. Then load the 4th element of the magic array with the number of the element, through which you want a sum. This will be a number ranging from 1 to the dimension of the array plus 1.

To see how this works, replace line 110 in the SUMSNG/DEM program with:

```
110 UX(2)=256:INPUT"FIND CUMULATIVE SUM THROUGH ELEMENT";UX(3)
```

Now run the program. If you enter 3, array elements 0, 1, and 2 will be summed. If you enter 200, array elements 0 through 199 will be summed.

If you are not using the magic array method to execute the USR routine, you can make the modification by poking 0 into the 5th byte of the routine and 1 into the 6th byte. Then, to sum through any element, poke the 2-byte element number into the 7th and 8th bytes of the routine.

Decimal to Hex Conversions

In many cases it's much more efficient to work with hex notation than with decimal. To convert from hex to decimal is easy. Disk basic recognizes and will interpret a hexadecimal number from 00 to FFFF for you. Simply put '**&H**' in front of the hex number. For example, if you enter the command:

```
PRINT &H8000
```

. . . your TRS-80 will respond by displaying −32768.

To convert from decimal to hex, you can use this short program:

DECTOHEX/BAS

Decimal to Hexadecimal Conversion Program

```
0 'DECTOHEX/BAS

15 DEFFNH2$(A1%)=MID$("0123456789ABCDEF",INT(A1%/16)+1,1)+MID$("
0123456789ABCDEF",A1%-INT(A1%/16)*16+1,1)

25 DEFFNH4$(A1%)=FNH2$(ASC(MID$(MKI$(A1%),2)))+FNH2$(ASC(MKI$(A1
%)))

110 CLS:PRINT"DECIMAL TO HEXADECIMAL CONVERSIONS
120 PRINT:INPUT"WHAT IS THE NUMBER FROM -32768 TO 65535";A!
121 IFA!>32767THENA%=A!-65536ELSEA%=A!
130 PRINT"HEXADECIMAL VALUE IS: ";FNH4$(A%)
140 GOTO120
```

Line 15 of the decimal to hex conversion program defines a function, H2$(A1%). It converts an integer from 0 to 255 to the corresponding hex notation from 00 to FF. Line 25 defines function, H4$(A1%). It handles the conversion for integers from -32768 to 32767. Note that within the function, FNH4$(A1%), we are using the function, FNH2$(A1%).

Using the decimal to hexadecimal conversion program, you can enter any decimal number from -32767 to 65535. So, if you enter -1, the program will display FFFF. If you enter 65535, it will also display FFFF. Line 121 provides the logic that converts any entry over 32767.

If you are writing a program in which you want to allow the operator to enter values in hexadecimal, you'll find that INPUT and LINEINPUT do not automatically recognize a hex number. The '**&H**' prefix only works in disk BASIC within a program line or in command mode.

FNDH!(A$) is a function that converts a 4-digit hex number, expressed as a string from 0000 to FFFF, to a single precision number. For example, if H$ is '3C00', FNDH!(H$) returns 15360. If H$ contains 'E411', FHDH!(H$) returns 58385. For valid results you must insure that the length of your string argument is 4 bytes. Any non-hex characters are assumed to be '0'.

Hexadecimal to Decimal Function

```
10 DEFFNDH!(A$)=INSTR("123456789ABCDEF",MID$(A$,1,1))*4096+INSTR
("123456789ABCDEF",MID$(A$,2,1))*256+INSTR("123456789ABCDEF",MID
$(A$,3,1))*16+INSTR("123456789ABCDEF",MID$(A$,4,1))
```

Base Conversion Routine

BASECONV/DEM is a demonstration program that employs a subroutine you can use for converting base 10 numbers to any other base. It asks you for the number to be converted and the base you want to convert it to. Here are some examples:

```
NUMBER,BASE? 3,2
 1 1

NUMBER,BASE? 63022,2
 1 1 1 1 0 1 1 0 0 0 1 0 1 1 1 0

NUMBER,BASE? 39,40
 39

NUMBER,BASE? 43203,16
 10 8 12 3
```

The base conversion subroutine occupies lines 210 and 220. To call the subroutine, 'BS' specifies the base, and 'N' contains the decimal number to be converted. Upon return from the subroutine, 'A$' contains the number in the desired base.

You'll find this program especially useful when you are experimenting with bit manipulations. A conversion to base 2 shows the bits that are set for any number.

BASECONV/DEM
Base Conversion Demonstration Program

```
100 CLEAR1000
110 CLS:PRINT"BASE CONVERSION PROGRAM"
120 INPUT"NUMBER,BASE";N,BS
130 GOSUB210:PRINTA$:GOTO120

200 'BASE CONVERSION SUBROUTINE....
210 A$=""
220 A$=STR$(N-(INT(N/BS)*BS))+A$:N=INT(N/BS):IFN=0THENRETURNELSE
220
```

Using Strings

The string handling capabilities of BASIC provide countless opportunities to design powerful program routines. This chapter will give you some ideas, standard function calls and subroutines that will multiply the power of your programs.

Peeks, Pokes, and Strings

Before we start manipulating strings, it is important to know how BASIC stores them. For each string that has been defined in a program, BASIC maintains a 3-byte pointer. The first byte specifies the current length of the string. The next 2 bytes point to the address where the string data can be found. Thus,

PEEK(VARPTR(A$)) is equal to LEN(A$)

PEEK(VARPTR(A$)+1) gives the LSB of the memory address where the data currently in A$ can be found.

PEEK(VARPTR(A$)+2) gives the MSB of that memory address.

PRINT CVI(CHR$ (PEEK (VARPTR (A$)+1)) +CHR$ (PEEK (VARPTR (A$)+2))) prints the memory address (in decimal) of the data currently in A$.

The CLEAR command defines the space that will be used for string storage. If you 'CLEAR 1000', BASIC will reserve 1000 bytes for string data storage at the top of unprotected memory. If for example, you specify a memory size of 61440 and then CLEAR 1000, memory locations 60439 through 61439 will be used for string storage.

It is important to know that BASIC does not move a string to the string storage area if it is defined as a 'literal' in the program text. For example, if line 10 of your program says,

```
10 A$="XXXXXXXX":B$=STRING$(8,"X"):C$="CAT":D$="DOG"+""
```

. . . the addresses for A$ and C$ will point at the program text. The addresses for B$ and D$ will point to the string storage area. Though four strings were defined, only B$ and D$ used memory in the string storage area. Keeping this in mind, you can judge the ramifications of various methods of programming your application.

If we use a command that lengthens 'A$' string during a BASIC program, the new contents of the string will be put in the next available location of the string storage area. If another string has been defined since 'A$' was first defined, then BASIC will put the new 'A$' below the data for the last string defined. Then the

VARPTR for the string is adjusted to point to its new address in memory. If there isn't any contiguous space in the string storage area that is long enough for the new 'A$' string, BASIC pauses to reorganize the data in string storage. This reorganization is often called 'garbage collection'. If, after reorganizing, there still isn't enough space, you get an 'out of string space' error.

If we use a command that shortens a string or leaves it the same length, BASIC simply records the new data in the same area and puts the new length into the string's VARPTR. The address of string data doesn't change as long as it is stored in the string storage area and isn't made longer than the original string length.

The LSET and RSET commands leave the length and address of a string unaltered. They simply replace the data at its current location, filling in spaces to the left or right of the string. Though LSET and RSET are most often used for loading data into random disk buffers, they can be very useful in many other ways also.

'Pointing' a String

We can 'load' the contents of any contiguous 255 or fewer bytes of memory into a string. To do it, we simply poke the string's VARPTR with the length and memory address we want. If for example, we want A$ to contain the first 25 bytes of memory, we can use the following sequence of commands:

```
POKE VARPTR(A$),25
POKE VARPTR(A$)+1,0
POKE VARPTR(A$)+2,0
```

Here's a general subroutine you can use to point a string at any memory address for any length. Simply load A% with the desired address, from -32768 to 32767 and A1% with the desired length, from 1 to 255 bytes and GOSUB 41000. Upon return, AN$ will be pointing where your parameters specified.

Note that your address must be expressed as an integer. For memory addresses 0 through 32767, no conversion is necessary. For memory addresses 32768 through 65535, subtract 65536 to get the integer address, A%.

String Pointer Subroutine

```
41000 AN$=" ":POKEVARPTR(AN$),A1%:POKEVARPTR(AN$)+1,ASC(MKI$(A%)
):POKEVARPTR(AN$)+2,ASC(RIGHT$(MKI$(A%),1)):RETURN
```

To load AN$ with the top 16 bytes of memory in a 48K TRS-80, your command would be:

```
A%=-16:A1%=16:GOSUB41000
```

To load AN$ with the contents of memory locations 16001 to 16049 the command is:

```
A%=16001:A1%=49:GOSUB41000
```

To load 8 X's into the 8 bytes starting at memory location 15360, you can use the command:

```
A%=15360:A1%=8:GOSUB41000:LSETAN$="XXXXXXXX"
```

M 2 Note # 7

Note that the video display string pointer subroutine, which is also discussed in this book, is just a special version of the string pointer subroutine. Instead of requiring an address, A%, it uses PO% to specify a position on the video display. You can use the string pointer subroutine to point to any PRINT@ position on the video display by adding 15360 to the desired position to get your address, A%.

The ability to point strings to any location in memory gives us a fast and convenient way to move data from one memory location to another. We simply point one string to the source address, and point a second string to the destination address. Then we LSET the second string equal to the first. For example, let's suppose we want to instantly write the first 127 elements of the I% integer array to the first disk record in file 1. We can say:

```
FIELD 1, 254 AS B$
A1%=254:A%=VARPTR(I%(0)):GOSUB41000
LSET B$ = AN$ : PUT 1,1
```

To load the array from disk we can reverse the procedure:

```
FIELD 1,255 AS B$ : GET 1,1
A1%=254:A%=VARPTR(I%(0)):GOSUB41000
LSET AN$ = B$
```

To move 64 bytes from memory location 15360 to memory location 32000 we can use the following sequence of commands:

```
A1%=64:A%=15360:GOSUB41000
A$=AN$
A1%=64:A%=32000:GOSUB41000
LSET AN$ = A$
```

Stripping Trailing Blanks from a String

Here's a function call that you can use when you want to insure that there are no trailing blanks on a string. For a string argument, A$, function FNSS$(A$) returns the contents of A$ with trailing blanks removed. The only restrictions are that A$ must be shorter than 253 bytes, and there must not be 2 contiguous blanks within A$, other than at the end of the string.

Strip Trailing Blanks Function

```
21 DEFFNSS$(A$)=LEFT$(A$+"  ",INSTR(A$+"  ","  ")-1)
```

FNSS$ strips the trailing blanks by adding 2 blanks to the end of the string. It then looks for the first 2 contiguous blanks and returns all characters to the left of those 2 blanks. If you are likely to have contiguous non-trailing blanks within a string, you may want to use the RSTRIP USR routine that is explained in this chapter. It does a 'true' strip of trailing blanks, and it's faster.

There are several common situations in which you might want to strip trailing blanks. If you are 'pulling' strings from video display memory using the string pointer subroutine, you may want to strip blanks before outputting the string with a PRINT or LPRINT command. If you are using random disk files, and a string has been LSET into a field, you may want to strip the right spaces so that you can print it in a sentence. If you are loading a large amount of string data into an array, you may wish to strip the right spaces from each string to conserve memory.

Padding and Centering Strings

The FNPL$, FNPR$, and FNCN$ functions are very useful when you are working with variable length strings and you want to print them in special formats on the video display or line printer.

FNPL$(A$,A%) pads enough spaces to the left of any string, A$, so that it will be right justified within a string, whose length is specified by A%. For example, if ST$ is 'JOE', FNPL$(ST$,5) will be ' JOE', with 2 spaces added to the left of the string to make it 5 characters long. FNPL$(ST$,2) will return the string 'OE'. In essence, FNPL$ is analogous to the RSET command, except you can use it in many situations where you can't use RSET.

FNPR$(A$,A%) pads enough spaces to the right of a string, A$, so that its length will be A%. In effect, it forces the length to be A% by stripping characters or adding blanks. It is analogous to the LSET command. FNPR$ is handy when you want to print variable length strings in columns on the line printer, especially past tab position 64. FNPR$ makes the lengths what you want them to be so that your columns will line up. FNPR$('JOE',5) pads 2 blanks onto the right side of the string, 'JOE', so that it is 5 bytes long. FNPR$("WALTER",5) generates the 5-byte string, 'WALTE'.

FNCN$(A$,A%) pads just enough blanks to the left of a string, A$, to center it in a field of width, A%. If, for example, you want to center the title, 'Inventory-Status' on the first line of a printout whose width is 128 characters, you could use the command,

```
LPRINT FNCN$("Inventory-Status",128)
```

If you want to center the same title on the video display, you can say,

```
PRINT FNCN$("Inventory-Status",64)
```

For the FNCN$ function call, the length of the string you wish to center must not be greater than the width specified by A%. If it is, you'll get an 'illegal function call' error.

String Padding and Centering Functions

```
Pad right, enforcing a length of A%:
22 DEFFNPR$(A$,A%)=LEFT$(A$+STRING$(A%," "),A%)

Pad left, enforcing a length of A%:
23 DEFFNPL$(A$,A%)=RIGHT$(STRING$(A%," ")+A$,A%)

Center by padding left, for a width of A%:
24 DEFFNCN$(A$,A%)=STRING$(A%/2-LEN(A$)/2-.5," ")+A$
```

Last Name First Function

In mailing lists, payroll and many other applications, it is useful to store names on disk with the last name preceding the first. This makes it possible to sort the data in alphabetical order. The FNFL$ function call lets us convert a string stored in 'last, first' format to a string in 'first last' format. It looks for a comma followed by a blank within the string. If one is found, the string is reversed and the comma removed. If a comma-blank isn't found, the string is not modified.

Here are some examples:

```
NM$="JONES, SALLY"
FNFL$(NM$) returns "SALLY JONES"

NM$="JOHNSON, MR. & MRS. BILL"
FNFL$(NM$) returns "MR. & MRS. BILL JOHNSON"

NM$="ABC SUPPLY"
FNFL$(NM$) returns "ABC SUPPLY"

TI$="Strings, How to Sort"
FNFL$(TI$) returns "How to Sort Strings"
```

The only major restriction with the FNFL$(A$) function is the string you wish to reverse, A$, must not have any trailing blanks. You can use the FNSS$(A$) function to remove them before calling the FNFL$ function. Then, if you want to restore the string to its original length, you can use the FNPR$(A$,A%) function.

Last Name First Function

```
25 DEFFNFL$(A$)=LEFT$(MID$(A$+" , ",INSTR(A$+" , ",", ")+2),INST
R(MID$(A$+", ",INSTR(A$+", ",", ")+3)+"  ","  "))+LEFT$(A$+", ",
INSTR(A$+", ",", ")-1)
```

You may modify the FNFL$ function call so that it uses a delimiter other than a comma to separate the first and last names. To do so, replace those commas in the function definition that are logically between quote marks with the character you want to use.

Stripping Blanks with USR Calls

LSTRIP and RSTRIP are two relocatable USR routines that let you strip leading or trailing blanks from any string. LSTRIP removes any blanks that may precede the first character in a string. RSTRIP removes any blanks that are on the right end of a string.

After one or both routines have been loaded into protected memory or a magic array and you have done a DEFUSR command, you can call LSTRIP or RSTRIP using the VARPTR to the string you want to alter as your calling argument. For instance, if you want to strip leading spaces from the string A$ and you have loaded and defined LSTRIP as USR1, your command is:

```
J=USR1(VARPTR(A$))
```

If you want to strip trailing spaces from the string A$ and you have loaded and defined RSTRIP as USR2, your command is:

```
J=USR2(VARPTR(A$))
```

If both routines have been loaded and defined, you can strip leading and trailing spaces with one call:

```
J=USR1(VARPTR(A$))ORUSR2(VARPTR(A$))
```

The integer variable 'J' in the examples above is a dummy variable. LSTRIP and RSTRIP do not return an argument to BASIC. The string that is stripped remains at the same location in memory. The USR routines simply search for the first non-blank character and modify the length and address pointers for the string accordingly.

M 2 Note # 23

LSTRIP
Strip Left Blanks
USR Subroutine

```
Magic Array Format - 16 elements

 32717  -6902   9038   9054  -5290 -18567   2344   8254   8382
  3332   6179  -5133  29153  29475  29219    201
Poke Format - 31 bytes

 205 127  10 229  78  35  94  35  86 235 121 183  40   9  62  32
 190  32   4  13  35  24 243 235 225 113  35 115  35 114 201
```

```
             00000 ;LSTRIP
             00001 ;
FF00         00020         ORG    0FF00H        ;ORIGIN - RELOCATABLE
FF00 CD7F0A  00030         CALL   0A7FH         ;HL HAS STRING VARPTR
FF03 E5      00040         PUSH   HL            ;SAVE HL
FF04 4E      00050         LD     C,(HL)        ;BC HAS STRING LENGTH
FF05 23      00060         INC    HL            ;HL POINTS TO POINTERS
FF06 5E      00070         LD     E,(HL)        ;
FF07 23      00080         INC    HL            ;
FF08 56      00090         LD     D,(HL)        ;DE NOW POINTS TO STRING
FF09 EB      00100         EX     DE,HL         ;HL NOW POINTS TO STRING
FF0A 79      00110 REDO    LD     A,C           ;PREPARE FOR PRE-TEST
FF0B B7      00120         OR     A             ;PRE-TEST FOR ZERO LENGTH
FF0C 2809    00130         JR     Z,RBAS        ;IFLENGTH=0 THEN RETURN
FF0E 3E20    00140         LD     A,020H        ;SPACE CODE TO ACCUM
FF10 BE      00150         CP     (HL)          ;COMPARE & INCREMENT
FF11 2004    00160         JR     NZ,RBAS       ;RETURN IF NON SPACE
FF13 0D      00170         DEC    C             ;SUBTR 1 FROM LENGTH
FF14 23      00180         INC    HL            ;ADD 1 TO ADDRESS
FF15 18F3    00190         JR     REDO
FF17 EB      00200 RBAS    EX     DE,HL         ;HOLD NEW ADDR IN DE
FF18 E1      00210         POP    HL            ;GET VARPTR TO STRING
FF19 71      00220         LD     (HL),C        ;NEW LENGTH RECORDED
FF1A 23      00230         INC    HL            ;POINT TO POINTERS
FF1B 73      00240         LD     (HL),E        ;
FF1C 23      00250         INC    HL            ;
FF1D 72      00260         LD     (HL),D        ;POINTERS NOW MODIFIED
FF1E C9      00270         RET                  ;RETURN TO BASIC
FF0A         00280         END                  ;
00000 TOTAL ERRORS
```

RSTRIP
Strip Right Blanks
USR Subroutine
M 2 Note # 23

```
Magic Array Format - 15 elements

 32717  -6902      6   9038   9054  -5290  11017 -18567   2344
  8254   8382   3332   6187  -7693 -13967

Poke Format - 30 bytes

 205 127  10 229   6   0  78  35  94  35  86 235   9  43 121 183
  40   9  62  32 190  32   4  13  43  24 243 225 113 201
```

RSTRIP

Strip Right Blanks USR Subroutine

```
            ;RSTRIP
            ;
FE00                ORG     0FE00H      ;ORIGIN - RELOCATABLE
FE00 CD7F0A         CALL    0A7FH       ;HL HAS STRING VARPTR
FE03 E5             PUSH    HL          ;SAVE HL
FE04 0600           LD      B,0         ;
FE06 4E             LD      C,(HL)      ;BC HAS STRING LENGTH
FE07 23             INC     HL          ;HL POINTS TO POINTERS
FE08 5E             LD      E,(HL)      ;
FE09 23             INC     HL          ;
FE0A 56             LD      D,(HL)      ;DE NOW POINTS TO STRING
FE0B EB             EX      DE,HL       ;HL NOW POINTS TO STRING
FE0C 09             ADD     HL,BC       ;HL POINTS TO END OF STRING +1
FE0D 2B             DEC     HL          ;HL POINTS TO LAST BYTE OF STRING
FE0E 79       REDO  LD      A,C         ;PREPARE FOR PRE-TEST
FE0F B7             OR      A           ;PRE-TEST FOR ZERO LENGTH
FE10 2809           JR      Z,RBAS      ;IFLENGTH=0 THEN RETURN
FE12 3E20           LD      A,020H      ;SPACE CODE TO ACCUM
FE14 BE             CP      (HL)        ;COMPARE
FE15 2004           JR      NZ,RBAS     ;RETURN IF NON SPACE
FE17 0D             DEC     C           ;SUBTR 1 FROM LENGTH
FE18 2B             DEC     HL          ;POINT TO NEXT TO LAST CHARACTER
FE19 18F3           JR      REDO
FE1B E1       RBAS  POP     HL          ;GET VARPTR TO STRING
FE1C 71             LD      (HL),C      ;NEW LENGTH RECORDED
FE1D C9             RET                 ;RETURN TO BASIC
FE0E                END                 ;
00000 TOTAL ERRORS
```

Using Strings to Store Data

When you have a small amount of string data to use in a program, such as a list of file names or a list of the months of the year, it can be very convenient and efficient to store the list in a string. Supose your program will use 3 disk files, 'MASTER:1', 'TRANS:1' and 'INDEX:1'. You can store those file names in a single string,

```
FL$="MASTER:1TRANS:1 INDEX:1 "
```

. . . and extract them by number as needed. To open the 3 files, your command could be:

```
FOR PF% = 1 TO 3
FD$=MID$(FL$,(PF%-1)*8+1,8)
OPEN"R",PF%,FD$
NEXT
```

The programming pattern of the string extraction is defined by the FNRR$(A1%,A2%,A3$) function, where:

Argument 1 is a 'field' number within a string, (the first field is 1),

Argument 2 is the length of each field, and

Argument 3 is the string containing the data.

Substring Extraction Function

```
15 DEFFNRR$(A1%,A2%,A3$)=MID$(A3$,(A1%-1)*A2%+1,A2%)
```

Here's an example. To extract the 3-letter month abbreviation from a string, based on the month number, your program can use the following logic:

```
INPUT"MONTH NUMBER";M%
PRINTFNRR$(M%,3,"JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC")
```

Whether you define the substring extraction function or you program the extraction 'in-line', you'll find that strings can be very good substitutes for data statements and arrays.

Code Lookup With Strings

The FNRC% function seaches a string for a code entered by the operator and returns a code number based on the position in the validation string. It is very useful in validating transaction codes and in converting them to a number usable by your program.

Code Lookup and Validation Function

```
DEFFNRC%(A1$,A2$,A3%)=(INSTR(A1$,LEFT$(A2$+STRING$(A3%," "),A3%)
)-1)/A3%+1
```

The code lookup and validation function, FNRC%(A1$,A2$,A3%), returns a code number where:

Argument 1 is a string list of valid codes separated by spaces,
Argument 2 is a string containing the code to be tested,and
Argument 3 is the uniform length of the codes in the valid code string.

An accounts receivable posting program might use 'PD', 'CR', 'CM', 'IN', 'DR' and 'LC' as valid transaction codes. To validate an entry by the operator and to branch to the proper line number, our program logic could be:

```
VC$="PD CR CM IN DR LC "
PRINT"ENTER THE TRANSACTION CODE"
PRINT"VALID CODES ARE ";VC$
LINEINPUT"CODE: ";A$
TC%=FNRC%(VC$,A$,3)
IF TC%=0 THEN PRINT"INVALID CODE":GOTO100
ON TC% GOTO 1000,2000,3000,4000,5000,6000
```

Notice how we designed the program so that the validation string also serves as an operator prompt. The space after each code insures that a partial code won't be accepted as valid.

Easy Input With Strings

Here's a subroutine that you can use to process a list of commands entered by the operator. The 'peel-off' subroutine gets, one by one, each word in a string of commands separated by one or more spaces. Upon each call to subroutine 41100, CS$ contains a list of commands. Upon return, A$ contains the next command, unless all commands have been exausted. Then A$ will have a length of zero.

Command String Peel-off Subroutine

```
41100 A$="":IFMID$(CS$,1,1)=""THENRETURNELSEIFMID$(CS$,1,1)=" "T
HENCS$=MID$(CS$,2):GOTO41100
41101 A$=A$+MID$(CS$,1,1):CS$=MID$(CS$,2):IFMID$(CS$,1,1)=""ORMI
D$(CS$,1,1)=" "THENRETURNELSE41101
```

The KILLFILE/BAS program demonstrates the peel-off subroutine. The operator is instructed to type a list of disk files to be killed, using a space between each file name. After the last file name, the operator presses ENTER. Then the program repeatedly calls 'peel-off'. After each call, A$ contains the next file name to be killed. When A$ is null, the program ends. The dialog looks something like this:

```
TYPE A LIST OF THE FILES YOU WANT TO KILL
SEPARATE EACH WITH A SPACE.  PRESS <ENTER> AFTER THE LAST ONE.

INVEN:1 AR:1 DATA:2 SORT:0

INVEN:1    KILLED.
AR:1       KILLED.
DATA:2     ERROR.  NOT KILLED.
SORT:0     KILLED.
```

KILLFILE/BAS
Multifile Purge Utility Program

```
1 CLEAR1000

100 CLS:PRINT
110 PRINT"TYPE A LIST OF THE FILES YOU WANT TO KILL"
120 PRINT"SEPARATE EACH WITH A SPACE.  PRESS <ENTER> AFTER THE L
AST ONE.":PRINT
130 LINEINPUT CS$                    'ENTER THE COMMAND STRING
140 GOSUB41100:IFA$=""THEN END       'GET NEXT COMMAND FROM STRING
150 PRINTA$;                         'PRINT IT
160 ONERRORGOTO300
161 KILL A$                          'EXECUTE THE COMMAND
162 PRINT" KILLED."
170 ONERRORGOTO0
180 GOTO140                          'REPEAT

300 PRINT" ERROR.  NOT KILLED.":RESUME170

41100 A$="":IFMID$(CS$,1,1)=""THENRETURNELSEIFMID$(CS$,1,1)=" "T
HENCS$=MID$(CS$,2):GOTO41100
41101 A$=A$+MID$(CS$,1,1):CS$=MID$(CS$,2):IFMID$(CS$,1,1)=""ORMI
D$(CS$,1,1)=" "THENRETURNELSE41101
```

Substring Replacement Subroutine

The substring replacement subroutine, 41200, replaces each occurrence of one string within another. Three calling variables are required:

A$ is the string to be searched.
A1$ is the substring to search for.
A2$ is the replacement for A1$ when found.

A% and A1% are used temporarily within the subroutine. Upon return, A$ contains the modified string.

Example:

```
If    A$  = "JOE IS A GOOD GUY.  JOE IS RICH."
and,  A1$ = "JOE"
and,  A2$ = "BILL"
```

. . . a GOSUB 41200 command will modify A$ so that:

```
A$  = "BILL IS A GOOD GUY.  BILL IS RICH."
```

The substring replacement subroutine can be very useful in word processing applications. You can also use it to modify programs that have been saved on disk in ASCII format. CHANGE/BAS is a short utility program that implements the substring replacement subroutine to let you change variable names, line numbers or other information in an ASCII program or text file.

Substring Replacement Subroutine

```
41200 A1%=1
41201 IFLEN(A$)-LEN(A1$)+LEN(A2$)>255THENRETURNELSEA%=INSTR(A1%,
A$,A1$):IFA%=0THENRETURNELSEA$=LEFT$(A$,A%-1)+A2$+MID$(A$,A%+LEN
(A1$)):A1%=A%+LEN(A2$):GOTO41201
```

CHANGE/BAS
Program File Modification Utility

```
1 CLEAR1000
100 CLS:PRINT"
PROGRAM MODIFICATION UTILITY
"
110 LINEINPUT"SOURCE FILE NAME:      ";SF$
120 LINEINPUT"DESTINATION FILE NAME: ";DF$
130 PRINT
140 LINEINPUT"STRING TO BE REPLACED:  ";A1$
150 LINEINPUT"REPLACE IT WITH:        ";A2$
200 OPEN"I",1,SF$
210 OPEN"O",2,DF$
220 IFEOF(1)THEN290
230 LINE INPUT#1,A$
240 GOSUB41200
250 PRINT#2,A$
260 GOTO220
290 CLOSE:GOTO100

41200 A1%=1
41201 IFLEN(A$)-LEN(A1$)+LEN(A2$)>255THENRETURNELSEA%=INSTR(A1%,
A$,A1$):IFA%=0THENRETURNELSEA$=LEFT$(A$,A%-1)+A2$+MID$(A$,A%+LEN
(A1$)):A1%=A%+LEN(A2$):GOTO41201
```

Storing 3 Bytes in 2

Suppose you could compress an alphanumeric string down to two-thirds of its original length for disk or memory storage. In effect, you'd be increasing your storage capacity by 50 percent!

The COMUNCOM USR subroutine lets you do just that. You can store a 24-byte name or address field in 16 bytes, a 60-byte field in 40 bytes or a 3-byte field in 2 bytes. The compression or uncompression is faster than a blink of the eye. The only restriction is the string to be compressed must consist of characters from a 40-character set. The 40 characters of the set you define may consist of any ASCII or non-ASCII character codes from 0 to 255. I've found the following 40 character set to be generally useful:

The letters, A through Z.
The digits, 0 through 9.
The space, period, comma and dash.

Within your character set, one character can be a default. The most common default character is the space. When you try to compress a character that is not in the character set, COMUNCOM changes it to the default character. For example, if we tried to compress the string 'A&B SUPPLY', COMUNCOM would replace the '&' character with a space, making the string, 'A B SUPPLY' before compressing.

Before going into the specifics of using the COMUNCOM USR routine, let's look at the theory behind it.

As you know, we can store a number ranging from 0 to 65535 in 2 bytes or 16 bits, because 2 to the 16th power is 65536. Now, consider a character set consisting of 40 characters. Any combination of 3 characters from that set can be stored in 2 bytes, because 40 times 40 times 40 equals 64000! To compress, COMUNCOM looks at the string, 3 characters at a time, converting each 'triplet' to a 2-byte 'token'. The resulting string of 2-byte tokens is the compressed string. To uncompress, a string is built by converting each 2-byte token back to 3 bytes.

In effect, each compressed character, instead of taking 8 bits, takes only 5 and a third bits. Since we can't work with a third of a bit, every compressed string is a multiple of 16 bits (or 2 bytes) in length. Every string that is uncompressed from a previously compressed string will be a multiple of 24 bits (or 3 bytes) in length. If you try to compress a 2-byte or 1-byte string with COMUNCOM, the resulting compressed string will be 2 bytes. In designing your applications with COMUNCOM you should plan your uncompressed length as a multiple of 3 whenever possible.

The COMUNCOM USR routine requires 4 arguments:

Argument 1 is the VARPTR to the source string, (the string that is to be compressed or uncompressed).

Argument 2 is the VARPTR to the destination string, (the string that will result from the compression or uncompression).

Argument 3 is the VARPTR to the character set string. This string must be exactly 40 characters in length and if you wish the compressed strings to be sortable, the characters must be in ascending sequence. The first character of the character set string is the default character, to be substituted when compression of an invalid character is attempted.

Argument 4 is an integer '1' to compress or '2' to uncompress.

The COMUNCOM USR routine implements the 'relocatable multiple argument handler' as its method for getting the 4 arguments from BASIC. Therefore, to call the USR routine from BASIC, assuming it has been loaded and defined as USR7, your command is in the format of . . .

```
J=USR7(ARG 1)ORUSR7(ARG 2)ORUSR7(ARG 3)ORUSR7(ARG 4)
```

Assume that we have specified our valid character set as CS$:

```
CS$=" ,-.ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

The following command would compress the 9-byte string 'MYSTERIES', currently stored in U$, down to a 6-byte compressed string, C$, using CS$ as the character set:

```
J=USR7(VARPTR(U$))ORUSR7(VARPTR(C$))ORUSR7(VARPTR(CS$))ORUSR7(1)
```

Now, assuming we have a compressed string in C$, we can uncompress it into the string U$ with the following command:

```
J=USR7(VARPTR(C$))ORUSR7(VARPTR(U$))ORUSR7(VARPTR(CS$))ORUSR7(2)
```

To make the compression and uncompression especially convenient, I use a function call to handle the USR arguments.

FNKM$(A$,1) returns a compressed string when the argument is an uncompressed string. FNKM$(A$,2) returns an uncompressed string when the argument is a compressed string. As you can see, the first argument to FNKM$ is the string to be compressed or uncompressed. The second argument is '1' to compress or '2' to uncompress.

The program statement . . .

```
S$="COMPUTER":QS$=FNKM$(S$,1)
```

. . . loads a 6-byte compressed string into QS$. To uncompress and print QS$ later we say,

```
PRINT FNKM$(QS$,2)
```

. . . and we'll get the 9-byte string, 'COMPUTER '.

String Compress and Uncompress Function

```
25 DEFFNKM$(A$,A%)=LEFT$(A$,(USR7(VARPTR(A$))ORUSR7(VARPTR(W$))O
RUSR7(VARPTR(CS$))ORUSR7(A%))*0)+W$
```

Notice that the string compress and uncompress function does all the work for us. To use it though, you will need to load and DEFUSR the COMUNCOM USR routine. CS$ must have been loaded with your character set and W$, a work string, must have been initialized. (You can use different variable names for W$ and CS$).

The 'magic array format', 'poke format' and assembly listing for COMUNCOM are shown below. As shown, it will execute as USR7 with the NEWDOS 2.1 disk operating system. To use it as another USR routine (USR0 – USR9) with Note: This technique cannot be used with sequential files.

NEWDOS 2.1 or to use it on another operating system, refer to appendix 2 and use the following guidelines:

1. For execution as a magic array, replace the 4th element, '23330', with the required integer from appendix 2.
2. If you are poking the COMUNCOM USR routine into memory, replace the 7th and 8th bytes, '34' and '91', with the required bytes from appendix 2.
3. If you are re-assembling COMUNCOM, replace the 5B22 in line 160 of the assembly listing with the required hexadecimal number from appendix 2.

In line 1080 of the assembler listing, we are calling the ROM subroutine at 2857. It allocates space in the string storage area for a new string, the length being specified by the A register. Upon return, the pointers to the new string address are contained in 40D4 and 40D5. If there isn't enough space, we get an 'out of string space' error when we return to BASIC.

M 2 Note # 23
M 2 Note # 34

COMUNCOM
String Compress & Uncompress USR Subroutine

```
               00100          ORG     0F000H          ;ORIGIN - RELOCATABLE
               00110 ;
               00120 ;THE FOLLOWING LOGIC ACCEPTS THE 4 ARGUMENTS
               00130 ;
F000 CD7F0A    00140          CALL    0A7FH           ;PUT ARGUMENT FROM BASIC IN HL
F003 00        00150          NOP                     ;NO-OP FOR ALIGNMENT
F004 DD2A225B  00160          LD      IX,(05B22H)     ;IX HAS USR7 ADDRESS
F008 DD7531    00170          LD      (IX+49),L       ;
F00B DD7432    00180          LD      (IX+50),H       ;PUT ARGUMENT IN STORAGE AREA
F00E DD340A    00190          INC     (IX+10)         ;
F011 DD340A    00200          INC     (IX+10)         ;ADD 2 TO POINTER
F014 DD340D    00210          INC     (IX+13)         ;
F017 DD340D    00220          INC     (IX+13)         ;ADD 2 TO SECOND POINTER
F01A DD7E0A    00230          LD      A,(IX+10)       ;
F01D 0631      00240          LD      B,49            ;
F01F 90        00250          SUB     B               ;A HAS NUMBER OF VARIABLES * 2
F020 DD4630    00260          LD      B,(IX+48)       ;B HAS NUMBER OF VARIABLES * 2
F023 90        00270          SUB     B               ;
F024 2801      00280          JR      Z,PASS1         ;IF ZERO, NO MORE VARIABLES
F026 C9        00290          RET                     ;OTHERWISE, RETURN FOR NEXT
F027 DD360A31  00300 PASS1    LD      (IX+10),49      ;
F02B DD360D32  00310          LD      (IX+13),50      ;RESTORE COUNT
F02F 1808      00320          JR      START           ;
F031 0000      00330          DEFW    0               ;STORAGE FOR UNCOMPRESS VARPTR
F033 0000      00340          DEFW    0               ;STORAGE FOR COMPRESS VARPTR
F035 0000      00350          DEFW    0               ;STORAGE FOR CHARACTER SET VARPTR
F037 0000      00360          DEFW    0               ;STORAGE FOR COMMAND CODE
               00370 ;
               00380 ;NOTE:   THE PRECEDING STORAGE AREA MUST NOT BE MODIFIED
               00390 ;        AS THE USR ROUTINE CALCULATES THE NUMBER OF
               00400 ;        ARGUMENTS TO PASS FROM THE "JR START" COMMAND
               00410 ;
               00420 ;        (IX+49) AND (IX+50)  =  SOURCE VARPTR
               00430 ;        (IX+51) AND (IX+52)  =  DESTINATION VARPTR
               00440 ;        (IX+53) AND (IX+54)  =  CHARACTER SET VARPTR
               00450 ;        (IX+55) AND (IX+56)  =  COMMAND CODE, 1 OR 2
               00460 ;
               00470 ;THE FOLLOWING LOGIC POINTS IX+53&54 TO CHARACTER SET DATA
F039 DD6E35    00480 START    LD      L,(IX+53)       ;
F03C DD6636    00490          LD      H,(IX+54)       ;HL POINTS TO VARPTR
F03F 23        00500          INC     HL              ;
```

```
F040 5E       00510          LD     E,(HL)          ;
F041 23       00520          INC    HL              ;
F042 56       00530          LD     D,(HL)          ;DE POINTS TO CHARACTER SET DATA
F043 DD7335   00540          LD     (IX+53),E       ;
F046 DD7236   00550          LD     (IX+54),D       ;IX+53&54 POINTS TO CHARACTER SET
F049 DD4637   00560          LD     B,(IX+55)       ;LOAD COMMAND CODE TO B
              00570 ;
              00580 ;THE FOLLOWING LOGIC COMPUTES LENGTH OF STRING TO BE CREATED
F04C DDE5     00590 SKP1     PUSH   IX              ;
F04E FDE1     00600          POP    IY              ;COPY IX TO IY FOR USE IN LOOP2C
F050 DD6E31   00610          LD     L,(IX+49)       ;
F053 DD6632   00620          LD     H,(IX+50)       ;HL HAS SOURCE VARPTR
F056 4E       00630          LD     C,(HL)          ;C HAS LENGTH OF SOURCE STRING
F057 3E00     00640          LD     A,0             ;INITIALIZE COMPRESS COUNT
F059 0C       00650          INC    C               ;
F05A 0D       00660          DEC    C               ;INC AND DEC C TO TEST FOR ZERO
F05B 2818     00670          JR     Z,LOOP1B        ;
F05D 3C       00680 LOOP1A   INC    A               ;
F05E 3C       00690          INC    A               ;ADD 2 TO COMPRESS COUNT
F05F CB48     00700          BIT    1,B             ;TEST IF COMPRESS OR UNCOMPRESS
F061 2801     00710          JR     Z,SKP2          ;SKIP THIS IF COMPRESS
F063 3C       00720          INC    A               ;
F064 0D       00730 SKP2     DEC    C               ;SUBTRACT 1 FROM LNTH OF UNCOMPR
F065 280E     00740          JR     Z,LOOP1B        ;END IF ZERO
F067 0D       00750          DEC    C               ;SUBTRACT 1 FROM LNTH OF UNCOMPR
F068 280B     00760          JR     Z,LOOP1B        ;END IF ZERO
F06A CB48     00770          BIT    1,B             ;TEST IF COMPRESS OR UNCOMPRESS
F06C 2802     00780          JR     Z,SKP3          ;SKIP THIS IF COMPRESS
F06E 18ED     00790          JR     LOOP1A          ;
F070 0D       00800 SKP3     DEC    C               ;SUBTRACT 1 FROM LNTH OF UNCOMPR
F071 2802     00810          JR     Z,LOOP1B        ;END IF ZERO
F073 18E8     00820          JR     LOOP1A          ;OTHERWISE, ADD 2
              00830 ;
              00840 ;THE FOLLOWING LOGIC ALLOCATES A NEW ADDRESS WITHIN
              00850 ;STRING STORAGE IF THE LENGTH OF THE COMPRESSED STRING
              00860 ;IS GREATER THAN THE PREVIOUS LENGTH OF THAT STRING
              00870 ;OTHERWISE, IT ADJUSTS THE LENGTH OF THE COMPRESSED STRING
              00880 ;IF IT IS LESS THAN THE PREVIOUS LENGTH OF THAT STRING
              00890 ;
F075 DD6E33   00900 LOOP1B   LD     L,(IX+51)       ;
F078 DD6634   00910          LD     H,(IX+52)       ;DEST VARPTR TO HL
F07B E5       00920          PUSH   HL              ;SAVE DEST VARPTR
F07C 4E       00930          LD     C,(HL)          ;C HAS CURRENT LNTH OF COMPR STR
F07D 23       00940          INC    HL              ;
F07E 5E       00950          LD     E,(HL)          ;
F07F 23       00960          INC    HL              ;
F080 56       00970          LD     D,(HL)          ;DE POINTS TO COMPRESS STRING DATA
              00980 ;NOTE:   A HAS LENGTH OF DESTINATION STRING TO BE CREATED
F081 B9       00990          CP     C               ;COMPARE NEW LNTH IN A TO CURRENT
F082 2821     01000          JR     Z,LOOP2A        ;NO CHANGE IF LENGTHS ARE EQUAL
F084 381B     01010          JR     C,LOOP2B        ;CURRENT LENGTH IS LONGER
              01020 ;NOTE:   AT THIS POINT, LENGTH OF CURRENT STRING IS TOO SHORT
              01030 ;        WE WILL HAVE TO CREATE A NEW ONE
F086 F5       01040          PUSH   AF              ;SAVE THE LENGTH
F087 DDE5     01050          PUSH   IX              ;SAVE IX
F089 C5       01060          PUSH   BC              ;SAVE BC
F08A FDE5     01070          PUSH   IY              ;SAVE IY
F08C CD5728   01080          CALL   02857H          ;CALL ROM RTNE TO ALLOCATE SPACE
F08F FDE1     01090          POP    IY              ;RESTORE IY
F091 C1       01100          POP    BC              ;RESTORE BC
F092 DDE1     01110          POP    IX              ;RESTORE IX
F094 ED5BD440 01120          LD     DE,(040D4H)     ;DE HAS THE ADDRESS
F098 F1       01130          POP    AF              ;LENGTH IS BACK IN A
F099 E1       01140          POP    HL              ;HL HAS COMPRESS VARPTR
```

```
F09A 77       01150          LD      (HL),A         ;RECORD NEW LENGTH
F09B 23       01160          INC     HL             ;
F09C 73       01170          LD      (HL),E         ;RECORD LSB OF ADDRESS
F09D 23       01180          INC     HL             ;
F09E 72       01190          LD      (HL),D         ;RECORD MSB OF ADDRESS
F09F 1805     01200          JR      LOOP2C         ;
              01210 ;NOTE:   AT THIS POINT, LENGTH OF CURRENT STRING IS TOO LONG
F0A1 E1       01220 LOOP2B   POP     HL             ;HL HAS DEST VARPTR
F0A2 77       01230          LD      (HL),A         ;RECORD NEW LENGTH
F0A3 1801     01240          JR      LOOP2C         ;
F0A5 E1       01250 LOOP2A   POP     HL             ;RELIEVE STACK
              01260 ;
              01270 ;THE FOLLOWING LOGIC INITIALIZES COUNTERS AND POINTERS
F0A6 D5       01280 LOOP2C   PUSH    DE             ;SAVE POINTER TO DEST DATA
F0A7 D9       01290          EXX                    ;
F0A8 FD6E31   01300          LD      L,(IY+49)      ;
F0AB FD6632   01310          LD      H,(IY+50)      ;VARPTR TO SOURCE STRING IN HL
F0AE 46       01320          LD      B,(HL)         ;SOURCE LENGTH IN B
F0AF 23       01330          INC     HL             ;
F0B0 5E       01340          LD      E,(HL)         ;
F0B1 23       01350          INC     HL             ;
F0B2 56       01360          LD      D,(HL)         ;
F0B3 D5       01370          PUSH    DE             ;
F0B4 FDE1     01380          POP     IY             ;IY POINTS TO SOURCE DATA
F0B6 D1       01390          POP     DE             ;DE POINTS TO DEST DATA
F0B7 04       01400          INC     B              ;
F0B8 05       01410          DEC     B              ;SET Z FLAG IF NO DATA TO PROCESS
F0B9 D9       01420          EXX                    ;
F0BA C8       01430          RET     Z              ;END IF NO BYTES TO PROCESS
F0BB DD6E35   01440          LD      L,(IX+53)      ;
F0BE DD6636   01450          LD      H,(IX+54)      ;HL POINTS TO CHARACTER SET
F0C1 CB48     01460          BIT     1,B            ;TEST IF COMPRESS OR UNCOMPRESS
F0C3 2073     01470          JR      NZ,UNCOM       ;JUMP IF UNCOMPRESS
F0C5 112700   01480          LD      DE,027H        ;LOAD DE WITH 39
F0C8 19       01490          ADD     HL,DE          ;HL POINTS TO LAST IN CHAR SET
F0C9 E5       01500          PUSH    HL             ;SAVE IT ON STACK
              01510 ;
              01520 ;THE FOLLOWING LOGIC IS REPEATED FOR EACH GROUP OF 3 CHARACTERS
F0CA E1       01530 COM1A    POP     HL             ;GET POINTER TO LAST IN SET
F0CB E5       01540          PUSH    HL             ;RESTORE IT ON STACK
F0CC FD7E00   01550          LD      A,(IY)         ;A HAS NEXT IN UNCOMPRESSED STRING
F0CF 012800   01560          LD      BC,028H        ;LOAD BYTE COUNTER WITH 40
F0D2 EDB9     01570          CPDR                   ;SEARCH CHARACTER STRING
F0D4 114006   01580          LD      DE,0640H       ;PREP TO MULTIPLY BY 1600
F0D7 0600     01590          LD      B,0            ;JUMP INDICATOR FOR AFTER MULTIPLY
F0D9 210000   01600 MUL0     LD      HL,0           ;MULTIPLY DE BY C GIVING HL
F0DC CB39     01610 MUL1     SRL     C              ;CONTINUE......
F0DE 3001     01620          JR      NC,MUL2        ;CONTINUE......
F0E0 19       01630          ADD     HL,DE          ;CONTINUE......
F0E1 2805     01640 MUL2     JR      Z,MUL9         ;CONTINUE......
F0E3 EB       01650          EX      DE,HL          ;CONTINUE......
F0E4 29       01660          ADD     HL,HL          ;CONTINUE......
F0E5 EB       01670          EX      DE,HL          ;CONTINUE......
F0E6 18F4     01680          JR      MUL1           ;CONTINUE......
F0E8 CB40     01690 MUL9     BIT     0,B            ;TEST ON WHERE TO GO AFTER MULTIPLY
F0EA 201A     01700          JR      NZ,COM1B       ;
F0EC EB       01710          EX      DE,HL          ;PUT PRODUCT IN DE
F0ED D9       01720          EXX                    ;
F0EE 05       01730          DEC     B              ;SUBTRACT FROM COUNT OF CHARACTERS
F0EF D9       01740          EXX                    ;
F0F0 283D     01750          JR      Z,END2         ;IF ZERO NO MORE TO COMPRESS
F0F2 E1       01760          POP     HL             ;GET POINTER TO LAST IN CHAR SET
F0F3 E5       01770          PUSH    HL             ;RESTORE IT ON STACK
```

```
F0F4 FD23     01780          INC     IY           ;POINT TO NEXT IN UNCOMPRESSED
F0F6 FD7E00   01790          LD      A,(IY)       ;A HAS NEXT IN UNCOMPRESSED STRING
F0F9 012800   01800          LD      BC,028H      ;LOAD BYTE COUNTER WITH 40
F0FC EDB9     01810          CPDR                 ;SEARCH CHARACTER STRING
F0FE D5       01820          PUSH    DE           ;SAVE CURRENT TOKEN
F0FF 112800   01830          LD      DE,028H      ;PREPARE TO MULTIPLY RESULT BY 40
F102 0601     01840          LD      B,01H        ;SET RETURN INDICATOR
F104 18D3     01850          JR      MUL0         ;GO MULTIPLY IT
F106 D1       01860 COM1B    POP     DE           ;RESTORE CURRENT TOKEN
F107 19       01870          ADD     HL,DE        ;UPDATE CURRENT TOKEN
F108 EB       01880          EX      DE,HL        ;PUT CURRENT TOKEN IN DE
F109 D9       01890          EXX                  ;
F10A 05       01900          DEC     B            ;SUBTRACT FROM COUNT OF CHARACTERS
F10B D9       01910          EXX                  ;
F10C 2821     01920          JR      Z,END2       ;IF ZERO NO MORE TO COMPRESS
F10E E1       01930          POP     HL           ;GET POINTER TO LAST IN CHAR SET
F10F E5       01940          PUSH    HL           ;RESTORE IT ON STACK
F110 FD23     01950          INC     IY           ;POINT TO NEXT IN UNCOMPRESSED
F112 FD7E00   01960          LD      A,(IY)       ;A HAS NEXT IN UNCOMPRESSED STRING
F115 012800   01970          LD      BC,028H      ;LOAD BYTE COUNTER WITH 40
F118 EDB9     01980          CPDR                 ;SEARCH CHARACTER STRING
F11A EB       01990          EX      DE,HL        ;PUT TOKEN IN HL
F11B 09       02000          ADD     HL,BC        ;ADD RELATIVE CHARACTER NUMBER
F11C EB       02010          EX      DE,HL        ;PUT TOKEN BACK IN DE
F11D D9       02020          EXX                  ;
F11E 05       02030          DEC     B            ;SUBTRACT FROM COUNT OF CHAR
F11F D9       02040          EXX                  ;
F120 280D     02050          JR      Z,END2       ;IF ZERO, NO MORE TO COMPRESS
F122 D9       02060          EXX                  ;
F123 D5       02070          PUSH    DE           ;PUT PTR TO COMPRESS STR ON STACK
F124 13       02080          INC     DE           ;
F125 13       02090          INC     DE           ;DE POINTS TO NEXT IN COMPRESS STR
F126 D9       02100          EXX                  ;
F127 E1       02110          POP     HL           ;HL POINTS TO COMPRESS STRING
F128 72       02120          LD      (HL),D       ;STORE FIRST BYTE OF TOKEN
F129 23       02130          INC     HL           ;POINT TO NEXT
F12A 73       02140          LD      (HL),E       ;STORE SECOND BYTE OF TOKEN
F12B FD23     02150          INC     IY           ;POINT TO NEXT IN UNCOMPRESSED STR
F12D 189B     02160          JR      COM1A        ;COMPRESS NEXT SET OF UP TO 3 CHAR
              02170 ;
              02180 ;THE FOLLOWING LOGIC RELIEVES THE STACK, AND RECORDS A PARTIALLY
              02190 ;COMPLETED TOKEN INTO THE COMPRESS STRING IF WE'VE RUN OUT OF
              02200 ;CHARACTERS TO COMPRESS.
F12F E1       02210 END2     POP     HL           ;RESTORE STACK
F130 D9       02220          EXX                  ;
F131 D5       02230          PUSH    DE           ;PUT PTR TO COMPRESS DATA ON STACK
F132 D9       02240          EXX                  ;
F133 E1       02250          POP     HL           ;GET POINTER TO COMPRESS DATA
F134 72       02260          LD      (HL),D       ;
F135 23       02270          INC     HL           ;
F136 73       02280          LD      (HL),E       ;TOKEN RECORDED IN COMPRESS STRING
F137 C9       02290 END1     RET                  ;RETURN TO BASIC
              02300 ;
              02310 ;UNCOMPRESS ROUTINE
              02320 ;AT ENTRY, NOTHING IS ON STACK
              02330 ;        IX POINTS TO BASE OF USR ROUTINE
              02340 ;        B' HAS NUMBER OF BYTES LEFT TO UNCOMPRESS
              02350 ;        DE' POINTS TO UNCOMPRESSED DATA
              02360 ;        IY  POINTS TO COMPRESSED DATA
              02370 ;        HL POINTS TO CHARACTER SET
              02380 ;
F138 E5       02390 UNCOM    PUSH    HL           ;SAVE HL FOR LOOKUPS
F139 D9       02400          EXX                  ;
F13A CB80     02410          RES     0,B          ;FORCE EVEN LNTH COMPRESS STRING
```

```
F13C D5         Ø242Ø          PUSH    DE            ;
F13D D9         Ø243Ø          EXX                   ;
F13E DDE1       Ø244Ø          POP     IX            ;IX POINTS TO UNCOMPRESSED STRING
F14Ø DD2B       Ø245Ø          DEC     IX            ;IX POINTS TO 1 BYTE BEFORE
F142 FD66ØØ     Ø246Ø UNCOM1   LD      H,(IY)        ;
F145 FD23       Ø247Ø          INC     IY            ;
F147 FD6EØØ     Ø248Ø          LD      L,(IY)        ;2 BYTES FROM COMPRESS STR IN HL
F14A FD23       Ø249Ø          INC     IY            ;POINT TO NEXT IN COMPRESS STRING
F14C FDE5       Ø25ØØ          PUSH    IY            ;SAVE IY DURING DIVISION
F14E ØEØ3       Ø251Ø          LD      C,3           ;SET UP 3 BYTE COUNTER
F15Ø 1628       Ø252Ø DIVØ     LD      D,Ø28H        ;DIVIDE 2 BYTE TOKEN IN HL BY 4Ø
F152 7D         Ø253Ø          LD      A,L           ;CONTINUE DIVISION
F153 6C         Ø254Ø          LD      L,H           ;CONTINUE DIVISION
F154 26ØØ       Ø255Ø          LD      H,Ø           ;CONTINUE DIVISION
F156 1EØØ       Ø256Ø          LD      E,Ø           ;CONTINUE DIVISION
F158 Ø61Ø       Ø257Ø          LD      B,16          ;CONTINUE DIVISION
F15A FD21ØØØØ   Ø258Ø          LD      IY,Ø          ;CONTINUE DIVISION
F15E 29         Ø259Ø DIV1     ADD     HL,HL         ;CONTINUE DIVISION
F15F 17         Ø26ØØ          RLA                   ;CONTINUE DIVISION
F16Ø 3ØØ1       Ø261Ø          JR      NC,DIV2       ;CONTINUE DIVISION
F162 2C         Ø262Ø          INC     L             ;CONTINUE DIVISION
F163 FD29       Ø263Ø DIV2     ADD     IY,IY         ;CONTINUE DIVISION
F165 FD23       Ø264Ø          INC     IY            ;CONTINUE DIVISION
F167 B7         Ø265Ø          OR      A             ;CONTINUE DIVISION
F168 ED52       Ø266Ø          SBC     HL,DE         ;CONTINUE DIVISION
F16A 3ØØ3       Ø267Ø          JR      NC,DIV3       ;CONTINUE DIVISION
F16C 19         Ø268Ø          ADD     HL,DE         ;CONTINUE DIVISION
F16D FD2B       Ø269Ø          DEC     IY            ;CONTINUE DIVISION
F16F 1ØED       Ø27ØØ DIV3     DJNZ    DIV1          ;CONTINUE DIVISION
F171 7C         Ø271Ø          LD      A,H           ;REMAINDER TO ACCUM
F172 D1         Ø272Ø          POP     DE            ;DE POINTS TO COMPRESS STRING
F173 E1         Ø273Ø          POP     HL            ;HL POINTS TO CHARACTER SET
F174 E5         Ø274Ø          PUSH    HL            ;RESTORE PTR TO CHAR SET ON STACK
F175 D5         Ø275Ø          PUSH    DE            ;RESTORE PTR TO CMPRSS STR ON STK
F176 5F         Ø276Ø          LD      E,A           ;REMAINDER IN E
F177 16ØØ       Ø277Ø          LD      D,Ø           ;REMAINDER IN DE
F179 19         Ø278Ø          ADD     HL,DE         ;HL POINTS TO CHARACTER
F17A 7E         Ø279Ø          LD      A,(HL)        ;CHARACTER TO A
F17B DDE5       Ø28ØØ          PUSH    IX            ;SAVE IX TEMPORARILY
F17D Ø6ØØ       Ø281Ø          LD      B,Ø           ;
F17F DDØ9       Ø282Ø          ADD     IX,BC         ;POINT TO POS IN STR FOR NEW CHAR
F181 DD77ØØ     Ø283Ø          LD      (IX),A        ;RECORD NEW CHARACTER
F184 DDE1       Ø284Ø          POP     IX            ;RESTORE IX
F186 ØD         Ø285Ø          DEC     C             ;SUBTRACT FROM COUNTER
F187 28Ø5       Ø286Ø          JR      Z,UNCOM2      ;SKIP IF ALL 3 CHAR PROCESSED
F189 FDE5       Ø287Ø          PUSH    IY            ;PREP FOR TRANSFER OF QUOTIENT
F18B E1         Ø288Ø          POP     HL            ;QUOTIENT IN HL FOR RE-DEVIDE
F18C 18C2       Ø289Ø          JR      DIVØ          ;GO DIVIDE AGAIN
F18E FDE1       Ø29ØØ UNCOM2   POP     IY            ;RESTORE PTR TO COMPRESSED STRING
F19Ø DD23       Ø291Ø          INC     IX            ;
F192 DD23       Ø292Ø          INC     IX            ;
F194 DD23       Ø293Ø          INC     IX            ;POINT TO NEXT 3 IN UNCMPSS STRING
F196 D9         Ø294Ø          EXX                   ;
F197 Ø5         Ø295Ø          DEC     B             ;
F198 Ø5         Ø296Ø          DEC     B             ;SUB 2 FROM COUNT
F199 D9         Ø297Ø          EXX                   ;
F19A 28Ø2       Ø298Ø          JR      Z,END3        ;
F19C 18A4       Ø299Ø          JR      UNCOM1        ;GO UNCOMPRESS MORE
F19E E1         Ø3ØØØ END3     POP     HL            ;RESTORE STACK
F19F C9         Ø3Ø1Ø          RET                   ;RETURN TO BASIC
F142            Ø3Ø2Ø          END                   ;
ØØØØØ TOTAL ERRORS
```

M 2 Note # 23
M 2 Note # 34

COMUNCOM
String Compress & Uncompress USR Subroutine

```
Magic Array Format - 208 elements

  32717     10  10973  23330  30173  -8911  12916  13533  -8950
   2612  13533  -8947   3380  32477   1546 -28623  18141 -28624
    296  -8759   2614  -8911   3382   6194      8      0      0
      0  -8960  13678  26333   9014   9054  -8874  13683  29405
  -8906  14150  -6691  -7683  28381  -8911  12902  15950   3072
  10253  15384 -13508  10312  15361  10253   3342   2856  18635
    552  -4840  10253   6146  -8728  13166  26333  -6860   9038
   9054 -18090   8488   6968  -8715 -14875  -6659  22477   -728
 -15903  -7715  23533  16596  -7695   9079   9075   6258  -7931
   6263  -7935  -9771  28413   -719  12902   9030   9054 -10922
  -7683   1233  -9979  -8760  13678  26333 -13514   8264   4467
     39  -6887  -6687  32509    256     40 -17939  16401   1542
   8448      0  14795    304  10265  -5371  -5335  -3048  16587
   6688  -9749  -9979  15656  -6687   9213  32509    256     40
 -17939   4565     40    262 -11496   6609  -9749  -9979   8488
  -6687   9213  32509    256     40 -17939   2539  -9749  -9979
   3368 -10791   4883  -7719   9074   -653   6179  -7781 -10791
  -7719   9074 -13965  -9755 -32565  -9771  -7715  11229  26365
   -768   -733    110   9213  -6659    782  10262  27773     38
     30   4102   8701      0   5929    304   -724   -727 -18653
  21229    816   -743   4139  31981  -7727 -10779   5727   6400
  -8834   1765  -8960  -8951    119  -7715  10253   -763  -7707
 -15848  -7683   9181   9181   9181   1497  -9979    552 -23528
 -13855

Poke Format - 416 bytes

 205 127  10   0 221  42  34  91 221 117  49 221 116  50 221  52
  10 221  52  10 221  52  13 221  52  13 221 126  10   6  49 144
 221  70  48 144  40   1 201 221  54  10  49 221  54  13  50  24
   8   0   0   0   0   0   0   0   0 221 110  53 221 102  54  35
  94  35  86 221 115  53 221 114  54 221  70  55 221 229 253 225
 221 110  49 221 102  50  78  62   0  12  13  40  24  60  60 203
  72  40   1  60  13  40  14  13  40  11 203  72  40   2  24 237
  13  40   2  24 232 221 110  51 221 102  52 229  78  35  94  35
  86 185  40  33  56  27 245 221 229 197 253 229 205  87  40 253
 225 193 221 225 237  91 212  64 241 225 119  35 115  35 114  24
   5 225 119  24   1 225 213 217 253 110  49 253 102  50  70  35
  94  35  86 213 253 225 209   4   5 217 200 221 110  53 221 102
  54 203  72  32 115  17  39   0  25 229 225 229 253 126   0   1
  40   0 237 185  17  64   6   6   0  33   0   0 203  57  48   1
  25  40   5 235  41 235  24 244 203  64  32  26 235 217   5 217
  40  61 225 229 253  35 253 126   0   1  40   0 237 185 213  17
  40   0   6   1  24 211 209  25 235 217   5 217  40  33 225 229
 253  35 253 126   0   1  40   0 237 185 235   9 235 217   5 217
  40  13 217 213  19  19 217 225 114  35 115 253  35  24 155 225
 217 213 217 225 114  35 115 201 229 217 203 128 213 217 221 225
 221  43 253 102   0 253  35 253 110   0 253  35 253 229  14   3
  22  40 125 108  38   0  30   0   6  16 253  33   0   0  41  23
  48   1  44 253  41 253  35 183 237  82  48   3  25 253  43  16
 237 124 209 225 229 213  95  22   0  25 126 221 229   6   0 221
   9 221 119   0 221 225  13  40   5 253 229 225  24 194 253 225
 221  35 221  35 221  35 217   5   5 217  40   2  24 164 225 201
```

Here is a program that demonstrates the COMUNCOM USR routine. It lets you enter a string for compression. Then it instantly compresses the string, uncompresses it, and displays it for you. COMUNCOM/DEM uses the magic array method for loading the USR subroutine so that you won't have to enter a special memory size. Because of its length, though, you should put the COMUNCOM routine in protected memory for actual applications.

Remember that if you are using a disk operating system other than NEWDOS 2.1, you'll need to change the '23330' in line 31 according to the instructions we discussed.

COMUNCOM/DEM

String Compress & Uncompress Demonstration

M 2 Note # 23

M 2 Note # 34

```
0 'COMUNCOM/DEM
10 CLEAR1000:DEFINTA-Z

20 W$=CHR$(0):U$=CHR$(0):C$=CHR$(0)
21 CS$=" ,-.0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"
25 DEFFNKM$(A$,A%)=LEFT$(A$,(USR7(VARPTR(A$))ORUSR7(VARPTR(W$))O
RUSR7(VARPTR(CS$))ORUSR7(A%))*0)+W$

30 'LOAD COMUNCOM USR ROUTINE INTO A MAGIC ARRAY

31 DATA 32717, 10, 10973, 23330, 30173,-8911, 12916, 13533,-8950
, 2612, 13533,-8947, 3380, 32477, 1546,-28623
32 DATA 18141,-28624, 296,-8759, 2614,-8911, 3382, 6194, 8, 0, 0
, 0,-8960, 13678, 26333, 9014
33 DATA 9054,-8874, 13683, 29405,-8906, 14150,-6691,-7683, 28381
,-8911, 12902, 15950, 3072, 10253, 15384,-13508
34 DATA 10312, 15361, 10253, 3342, 2856, 18635, 552,-4840, 10253
, 6146,-8728, 13166, 26333,-6860, 9038, 9054
35 DATA-18090, 8488, 6968,-8715,-14875,-6659, 22477,-728,-15903,
-7715, 23533, 16596,-7695, 9079, 9075, 6258
36 DATA-7931, 6263,-7935,-9771, 28413,-719, 12902, 9030, 9054,-1
0922,-7683, 1233,-9979,-8760, 13678, 26333
37 DATA-13514, 8264, 4467, 39,-6887,-6687, 32509, 256, 40,-17939
, 16401, 1542, 8448, 0, 14795, 304
38 DATA 10265,-5371,-5335,-3048, 16587, 6688,-9749,-9979, 15656,
-6687, 9213, 32509, 256, 40,-17939, 4565
39 DATA 40, 262,-11496, 6609,-9749,-9979, 8488,-6687, 9213, 3250
9, 256, 40,-17939, 2539,-9749,-9979
40 DATA 3368,-10791, 4883,-7719, 9074,-653, 6179,-7781,-10791,-7
719, 9074,-13965,-9755,-32565,-9771,-7715
41 DATA 11229, 26365,-768,-733, 110, 9213,-6659, 782, 10262, 277
73, 38, 30, 4102, 8701, 0, 5929
42 DATA 304,-724,-727,-18653, 21229, 816,-743, 4139, 31981,-7727
,-10779, 5727, 6400,-8834, 1765,-8960
43 DATA-8951, 119,-7715, 10253,-763,-7707,-15848,-7683, 9181, 91
81, 9181, 1497,-9979, 552,-23528,-13855
44 DIMUS(207):FORX=0TO207:READUS(X):NEXT

100 DEFUSR7=VARPTR(US(0))
110 CLS
120 LINEINPUT"UNCOMPRESSED STRING: ";U$
130 C$=FNKM$(U$,1)
140 U$=FNKM$(C$,2)
150 PRINT"COMPRESSED AND RESTORED: ";U$
160 GOTO120
```

Upper Case Conversions

The UPPERCON USR routine scans a string for lower case characters and converts them to upper case. This can be important to you when you are doing string compression and when you are doing alphabetical sorts of string data.

To use UPPERCON, simply load it and define it as a USR subroutine. Then call the routine, using the VARPTR of the string you want converted as your argument.

Let's assume, for example, that you've poked the 28 required bytes into protected memory, starting at F000. You can convert any string entered by the operator with the following logic:

```
10 DEFUSR=&HF000
20 LINEINPUT "ENTER A STRING:    ";A$
30 J=USR(VARPTR(A$))
40 PRINT "CONVERTED STRING IS:  ";A$
50 GOTO 20
```

UPPERCON
String Upper-Case Conversion USR Subroutine

M 2 Note # 23

```
Magic Array Format, 14 elements:

  32717  17930  24099  22051   1259 -14331   -386  14433   -505
  12411  -6653  30559   4131 -13839

Poke Format, 28 bytes:

 205 127  10  70  35  94  35  86 235   4   5 200 126 254  97  56
   7 254 123  48   3 230  95 119  35  16 241 201
```

```
              00000 ;UPPERCON
              00001 ;
F000          00060        ORG    0F000H        ;ORIGIN - RELOCATABLE
F000 CD7F0A   00070        CALL   0A7FH         ;HL HAS STRING VARPTR
F003 46       00080        LD     B,(HL)        ;B HAS STRING LENGTH
F004 23       00090        INC    HL            ;
F005 5E       00100        LD     E,(HL)        ;
F006 23       00110        INC    HL            ;
F007 56       00120        LD     D,(HL)        ;DE POINTS TO STRING
F008 EB       00130        EX     DE,HL         ;HL POINTS TO STRING
F009 04       00140        INC    B             ;
F00A 05       00150        DEC    B             ;INC & DEC B TO TEST IF ZERO
F00B C8       00160        RET    Z             ;RETURN IF ZERO LENGTH
F00C 7E       00170 LOOP   LD     A,(HL)        ;PUT BYTE IN ACCUM
F00D FE61     00180        CP     61H           ;COMPARE TO LOWER CASE A
F00F 3807     00190        JR     C,OK          ;JUMP IF LOWER
F011 FE7B     00200        CP     7BH           ;IS IT ABOVE LOWER CASE Z?
F013 3003     00210        JR     NC,OK         ;JUMP IF IT IS
F015 E65F     00220        AND    5FH           ;CONVERT TO UPPER CASE
F017 77       00230        LD     (HL),A        ;PUT IT BACK
F018 23       00240 OK     INC    HL            ;POINT TO NEXT BYTE
F019 10F1     00250        DJNZ   LOOP          ;DECREMENT COUNT & REPEAT
F01B C9       00260        RET                  ;RETURN TO BASIC
F00C          00270        END                  ;
00000 TOTAL ERRORS
```

Date & Time Manipulation

Sooner or later in your programming efforts, you're likely to work with date or time computations. Why be the millionth programmer to spend hours and hours re-inventing this old wheel? Here are some 'plug-in' function calls and subroutines that can save programming time while conserving valuable computer memory and disk space.

The 8-Byte Date

The '8-byte date' is simply a string that expresses the month, day and year in the format, 'MM/DD/YY', where:

MM is a 2-digit month number in the range of 01 to 12,
DD is a 2-digit day number, ranging from 01 to 31, and
YY is a 2-digit year number, ranging from 00, to 99.

The string, '02/14/82' is an example of an 8-byte date that stands for 'February 14, 1982'.

If the operator has set the date at startup, your program can get it back in 8-byte date format by taking the left 8 bytes of the TIME$ function. That is,

M 2 Note # 35

8-byte date = LEFT$(TIME$,8)

Or you can load the 8-byte date into your program using the formatted inkey routine, (which is discussed in the chapter about keyboard and video routines). To have it handy, you can POKE the month, day and year into the memory locations given in your disk system owner's manual, so that you can get it back with the TIME$ function. This is especially useful when your application 'chains' between 2 or more programs. When you've got the date in TIME$ you don't have to reload it each time you run a new program.

A Simple Date Validity Check

Here is a function call that checks the validity of a date entered by the operator. FNDV%(A1$,A2%) checks that, for the string, A1$:

The month (in positions 1 and 2) is between 01 and 12.
The day (in positions 4 and 5) is between 01 and 31.
The year (in positions 7 and 8) is greater than or equal to A2%.
The string is 8 characters long.

To use the valid date function, you must first define it in your program:

Date Validity Function

```
15 DEFFNDV%(A1$,A2%)=(VAL(A1$)>0)AND(VAL(A1$)<13)AND(VAL(MID$(A1
$,4))>0)AND(VAL(MID$(A1$,4))<32)AND(VAL(MID$(A1$,7))>=A2%)AND(LE
N(A1$)=8)ORA1$="00/00/00"
```

Here is an example that shows how FNDV% might be used within a program:

```
130 INPUT"DATE";A$

140 'CHECK IF DATE IS VALID, AND THE YEAR IS 1980 OR GREATER
141 IFFNDV%(A$,80)THEN150ELSEPRINT"INVALID":GOTO130

150 'PROGRAM FALLS-THROUGH HERE IF DATE IS VALID
```

A big advantage of the valid date function call is that you can handle the validity test in one line of program logic. The function equals 0 if the date is invalid or −1 if it's valid. If you don't want to check on a minimum year, you can simply use 0 as the second argument.

Note that we are accepting '00/00/00' as a valid date. If you don't want to accept a zero date, modify the function call by deleting the last 16 bytes, which read:

```
ORA1$="00/00/00"
```

With a slight modification, you can add a third argument that specifies whether a zero date should be accepted as valid.

The 3-Byte Date

For disk and memory array storage, it is quite convenient to store dates in 3-byte format. If MO% is the month, DY% is the day and YR% is the year, the 3 byte format is created using the expression:

```
CHR$(YR%)+CHR$(MO%)+CHR$(DY%)
```

We use a year-month-day sequence so that the 3-byte date can be sorted and we can use 'greater than' and 'less than' tests if necessary.

You'll find that the 3-byte approach is much more convenient than storing a date as a single precision number. Besides the advantage of using 3 bytes instead of 4, the execution speed for any conversions will normally be much faster with string manipulation than with multiplication and division.

Here are 2 function calls that you can use when working with 3-byte dates. FNCD$(A1$) converts an 8-byte date string, A1$, to a 3-byte date string. FNUD$(A1$) uncompresses a 3-byte date string back to an 8-byte date string:

8-Byte to 3-Byte Date Compression Functions

```
Compress 8-byte date to 3-byte date:
15 DEFFNCD$(A1$)=CHR$(VAL(MID$(A1$,7,2)))+CHR$(VAL(MID$(A1$,1,2)
))+CHR$(VAL(MID$(A1$,4,2)))

Uncompress 3-byte date to 8-byte date:
25 DEFFNUD$(A1$)=RIGHT$(STR$(ASC(MID$(A1$,2))),2)+"/"+RIGHT$(STR
$(ASC(MID$(A1$,3))),2)+"/"+RIGHT$(STR$(ASC(A1$)),2)
```

Don't try to store a 3-byte date in a sequential disk file! It will appear to work fine . . . until you get to the 13th of the month. Remember that BASIC uses CHR$(13) as an 'end of field marker' in sequential files. You'll have no problems in random files though. Simply create a 3-byte field and LSET or RSET the 3-byte date into it.

Storing a Date in 2 Bytes

Using bit manipulations, we can store a year, month and day in 16 bits or 2 bytes. Since the year will range from 0 to 99, we can store the year in the first 7 bits. (2 to the 7th power = 128). The month will range from 1 to 12. We can store it in the next 4 bits. (2 to the 4th power = 16). And, because the day will range from 1 to 31, we can store it in 5 bits. (2 to the 5th power = 32). When we add 7 bits for the year, 4 bits for the month and 5 bits for the day, we get a total of 16 bits or 2 bytes!

The following two function calls handle the conversions. FNC2$(A1$) compresses a date in 3-byte format, A1$, to a 2-byte string containing the date in 2-byte format. FNU2$(A1$) uncompresses a date in 2-byte format, A1$, back to 3-byte format.

3-Byte to 2-Byte Date Compression Functions

```
Compress 3-byte date to 2-byte date:
35 DEFFNC2$(A1$)=CHR$((ASC(A1$)*2)OR-((ASC(MID$(A1$,2,1))AND8)<>
0))+CHR$((ASC(MID$(A1$,2,1))ANDNOT8)*32+ASC(MID$(A1$,3,1)))

Uncompress 2-byte date to 3-byte date:
45 DEFFNU2$(A1$)=CHR$((ASC(A1$)ANDNOT1)/2)+CHR$((ASC(MID$(A1$,2)
)/32)OR((ASC(A1$)AND1)*8))+CHR$(ASC(MID$(A1$,2))ANDNOT224)
```

Using the 8-byte to 3-byte conversion, and the 3-byte to 2-byte conversion we can compress the current date specified by TIME$ to a 2-byte string, D2$:

```
D2$ = FNC2$(FNCD$(LEFT$(TIME$,8)))
```

We can get it back and print it later using the uncompress function calls:

```
PRINT FNUD$(FNU2$(D2$))
```

If we want to store an 8-byte date, DT$, in a 2-byte integer variable, A%, we can use the command:

```
A% = CVI(FNC2$(FNCD$(DT$)))
```

To print A% in 8-byte date format, we can use the command:

```
PRINT FNUD$(FNU2$(MKI$(A%)))
```

Here is a test program that you can use to test the date compression function calls to your satisfaction. To use it, type in or merge the function definitions shown above for FNCD$, FNUD$, FNC2$ and FNU2$.

Date Compression Test Program

```
15 'DEFINE FNCD$(A1$) HERE
25 'DEFINE FNUD$(A1$) HERE
35 'DEFINE FNC2$(A1$) HERE
45 'DEFINE FNU2$(A1$) HERE

110 CLS:PRINT"DATE COMPRESS-UNCOMPRESS TEST PROGRAM"
120 PRINT
130 INPUT"WHAT IS THE DATE IN MM/DD/YY FORMAT";D8$
140 'COMPRESS TO 3-BYTES
141 D3$=FNCD$(D8$)
150 'COMPRESS TO 2-BYTES
151 D2$=FNC2$(D3$)
160 'UNCOMPRESS TO 3-BYTES
161 V3$=FNU2$(D2$)
170 'UNCOMPRESS TO 8-BYTES
171 V8$=FNUD$(V3$)
180 PRINT"DATE HAS BEEN COMPRESSED TO 2 BYTES"
181 PRINT"AND THEN UNCOMPRESSED BACK TO: ";V8$
190 GOTO120
```

As a final note on 2-byte dates, be sure that your month and day are both valid before doing the compression to avoid 'illegal function call' errors. Also, avoid using 2-byte dates in sequential disk files.

Find a Day of a Year

Here is a function call that lets you compute the day within any year from 1901 to 2099. You simply provide the 4-digit year as the first argument, the month as the second argument and the day as the third argument. FNJD% takes into account whether or not the year is a leap year.

Day Number Function

```
70 DEFFNJD%(Y%,M%,D%)=(M%-1)*28+VAL(MID$("0003030608111316192124
26",(M%-1)*2+1,2))-((M%>2)AND((Y%ANDNOT-4)=0))+D%
```

If you look carefully at this function definition, you'll see that the day number is computed first by figuring the number of preceding months multiplied by 28 days. Then a table is accessed based on month number for an adjusting amount. This is added to the number of days beyond 28 for all preceding months. Then, if the year is evenly divisible by 4, (leap year), and the month is greater than 2, 1 day is added to account for 29 days in February. Finally the day within the month is added.

After defining this function in a program, we could, for instance, issue the command,

```
PRINT FNJD%(1981,5,14)
```

. . . to find that May 14, 1981 is the 134th day of the year.

Simplified Date Computing

To find the number of days between dates, the day of the week, or the date that it will be any number of days into the future, I've found that the best way is to convert each date to a number. Then, for example, the number of days between dates is simple subtraction.

The FNDN! function returns a single precision number which I call a 'computational date.' The computational day number, as provided by FNDN!, is useful for any date between the years 1901 and 2099. (If you're curious about the reasons for limiting the valid range from 1901 to 2099 you can consult any good almanac. In brief, even numbered centuries, unless divisible by 400, are exceptions to the rule that leap years are divisible by 4. Thus, 2000 is a leap year, while 1900 and 2100 are not.)

Note that the 'computational dates' we are discussing here are only useful for certain date computations. Because of changes in the calendar in past centuries, and leap year variations every century, they do not represent a number that is useful for any other purpose, such as astronomical calculations.

Here's the computational date function call. The arguments are 4-digit year, 1 or 2 digit month, and 1 or 2 digit day:

Computational Date Function

```
51 DEFFNDN!(Y%,M%,D%)=Y%*365+INT((Y%-1)/4)+(M%-1)*28+VAL(MID$("0
0030306081113161921242 6",(M%-1)*2+1,2))-((M%>2)AND((Y%ANDNOT-4)=
0))+D%
```

Days Between Dates

To find the number of days between 2 dates, define the computational date function call, FNDN!, shown above, in your program. Then subtract the computational day number of the first date from the computational day number of the second date. For example, the number of days between January 15, 1980 and January 15, 1981 is 366, computed using the expression:

```
FNDN!(1981,1,15)-FNDN!(1980,1,15)
```

Within a program you would normally use integer variables for the 3 arguments to the FNDN! function call.

Day of the Week

This function returns a 9-byte string that contains the day of the week for any date between 1901 and 2099. The argument that you must supply to FNDY$ is the computational day number that was obtained using the FNDN! function call.

Day of the Week Function

```
60 DEFFNDY$(N!)=MID$("FRIDAY   SATURDAY SUNDAY   MONDAY   TUESDA
Y  WEDNESDAYTHURSDAY ",(N!-INT(N!/7)*7)*9+1,9)
```

To find the day of the week for May 15, 1981, you can use the following 2 commands:

```
A!=FNDN!(1981,5,15)
PRINT FNDY$(A!)
```

Or you can combine them into one command:

```
PRINT FNDY$(FNDN!(1981,5,15))
```

Back to 8 Byte Dates

The computations to convert from a computational day number back to an 8-byte date are rather complex, but you'll need them if you want to find out something like, what will the date will be 200 days from today. To do it, we will use 4 functions.

FNRY%(N!) recalls the year from a computational date. FNRJ%(N!) recalls the day number within the year for any computational date. FNRM%(J%,Y%) recalls the month based on the day number within the year, J%, and the year, Y%. FNRD%(Y%,M%,J%) recalls the day of the month based on the year, Y%, the month, M%, and the day number within the year, J%.

Reverse Date Computation Functions

```
Recall year from computational date:
52 DEFFNRY%(N!)=INT((N!-N!/1461)/365)

Recall day number within year from computational date:
53 DEFFNRJ%(N!)=N!-(FNRY%(N!)*365+INT((FNRY%(N!)-1)/4))

Recall month for day number within year, and year:
54 DEFFNRM%(J%,Y%)=-((Y%ANDNOT-4)<>0)*(1-(J%>31)-(J%>59)-(J%>90)
-(J%>120)-(J%>151)-(J%>181)-(J%>212)-(J%>243)-(J%>273)-(J%>304)-
(J%>334))-((Y%ANDNOT-4)=0)*(1-(J%>31)-(J%>60)-(J%>91)-(J%>121)-(
J%>152)-(J%>182)-(J%>213)-(J%>244)-(J%>274)-(J%>305)-(J%>335))

Recall day of month from year, month, and day within year:
55 DEFFNRD%(Y%,M%,J%)=(J%-((M%-1)*28+VAL(MID$("000303060811131619212426",(M%-1)*2+1,2))))+((M%>2)AND((Y%ANDNOT-4)=0))
```

To find the date, 200 days into the future, we can use the following program logic, assuming that the required function calls were defined earlier in the program:

```
100 INPUT"DAY";D%
101 INPUT"MONTH";M%
102 INPUT"4-DIGIT YEAR";Y%
110 N!=FNDN!(Y%,M%,D%)+200
120 Y%=FNRY%(N!):J%=FNRJ%(N!):M%=FNRM%(J%,Y%):D%=FNRD%(Y%,M%,J%)
130 PRINTUSING"DATE 200 DAYS HENCE IS: ##/##/####";M%;D%;Y%
```

Going Fiscal

It is often necessary in accounting application programs to provide for a fiscal month and year that differs from the calendar month and year. The following subroutine computes the 2-digit fiscal year, FY%, and the fiscal month, FM%, based on the calendar year, Y%, and the calendar month, M%. The variable, S%, specifies the first month of the fiscal year. S% is positive if the fiscal date precedes the calendar date, and negative if the fiscal date trails the calendar date. S% is 1 if calendar date and fiscal date are the same.

Suppose that the fiscal year begins in October, preceding the calendar date. The current calendar month is 12 and the current calendar year is 1981. You would load S% with 10, M% with 12, and Y% with 81, and GOSUB 5010. Upon return from the subroutine, FY% would equal 82, and FM% would equal 3.

Calendar Date to Fiscal Date Subroutine

```
5010 IFABS(S%)=1THENFM%=M%:FY%=Y%:GOTO5020ELSEIFS%<0THEN5013
5011 IFS%>0THENIFM%>=S%THENFM%=M%+1-S%:FY%=Y%+1ELSEFM%=M%+13-S%:
FY%=Y%
5012 IFFY%=100THENFY%=0:GOTO5020ELSE5020
5013 IFM%<ABS(S%)THENFM%=M%+13-ABS(S%):FY%=Y%-1ELSEFM%=M%+1-ABS(
S%):FY%=Y%
5014 IFFY%=-1THENFY%=99
5020 RETURN
```

DATECOMP/BAS

1901 - 2099 Perpetual Calendar Program

```
0 'DATECOMP/BAS
1 CLEAR100:SG$=STRING$(63,131)

50 'MERGE FNDN!, FNRY%, FNRJ%, FNRM%, FNRD%, FNDY$, FNJD% HERE

100 CLS:PRINT:PRINT"DATE COMPUTATION TEST PROGRAM":PRINTSG$
110 PRINT"
<1> COMPUTE DAYS BETWEEN DATES
<2> COMPUTE DAY OF THE WEEK
<3> COMPUTE DAY WITHIN THE YEAR
<4> COMPUTE DATE, X DAYS HENCE"
120 PRINT:PRINTSG$:PRINT"PRESS THE NUMBER OF YOUR SELECTION..."
200 GOSUB40500:A%=INSTR("1234",A$):IFA%=0THEN200ELSEONA%GOTO1000
,2000,3000,4000

300 PRINT:INPUT"MONTH          ";MO%
310 INPUT"DAY           ";DY%
320 INPUT"4-DIGIT YEAR  ";YR%
330 RETURN

400 PRINT:PRINT"PRESS <ENTER>...";:GOSUB40500:GOTO100

1000 CLS:PRINT"FIRST DATE:":GOSUB300
1020 A!=FNDN!(YR%,MO%,DY%)
1030 PRINT:PRINT"SECOND DATE:":GOSUB300
1050 PRINT:PRINT"DAYS BETWEEN DATES =";ABS(A!-FNDN!(YR%,MO%,DY%))
1060 GOTO400

2000 CLS:PRINT:GOSUB300
2030 PRINT:PRINT"DAY OF THE WEEK = ";FNDY$(FNDN!(YR%,MO%,DY%))
2040 GOTO400

3000 CLS:GOSUB300
3020 PRINT:PRINT"DAY WITHIN THE YEAR IS";FNJD%(YR%,MO%,DY%)
3030 GOTO400

4000 CLS:GOSUB300
4020 PRINT:INPUT"DAYS HENCE";DH!
4040 A!=FNDN!(YR%,MO%,DY%)+DH!
4050 YR%=FNRY%(A!):J%=FNRJ%(A!):MO%=FNRM%(J%,YR%):DY%=FNRD%(YR%,
MO%,J%)
4060 PRINT:PRINTUSING"##/##/####";MO%;DY%;YR%
4070 GOTO400

40500 A$=INKEY$:IFA$=""THEN40500ELSERETURN
```

1901 – 2099 Perpetual Calendar

The date computation test program, DATECOMP/BAS, will let you test the function calls we've discussed. In addition, it will come in handy whenever you need to perform a date computation. To use it, type the program as shown, and merge or add the function definitions required anywhere between lines 2 and 99.

Timing Benchmark Tests

A 'benchmark' is simply a timed test of a program or routine. You can use the TIME$ function to compare the speed of alternative programming methods. When you tell the computer to PRINT TIME$, the date and time will be printed in the format, 'MM/DD/YY HH:MM:SS'. To do a benchmark test on any routine, design your program so that TIME$ is printed, followed by a FOR-NEXT loop giving multiple repetitions of the routine you want to test, followed by another command to print TIME$.

Here are two function calls that you can use when working with TIME$ to compute elapsed time. FNSE!(A1$) computes total seconds for any string, A1$, whose 8 rightmost characters are in the format 'HH:MM:SS', (where 'HH' is hours, 'MM' is minutes, and 'SS' is seconds.) FNHM$(A1!) performs the opposite computation. It creates a string in the format 'HH:MM:SS' from the number of seconds specified by A1!.

Hours, Minutes, Seconds Conversion Functions

```
"HH:MM:SS" string to seconds:
25 DEFFNSE!(A1$)=VAL(RIGHT$(A1$,2))+VAL(RIGHT$(A1$,5))*60+VAL(RI
GHT$(A1$,8))*3600

Seconds to "HH:MM:SS" string:
15 DEFFNHM$(A1!)=RIGHT$("0"+MID$(STR$(INT(A1!/3600)),2),2)+":"+R
IGHT$("0"+MID$(STR$(INT((A1!-INT(A1!/3600)*3600)/60)),2),2)+":"+
RIGHT$("0"+MID$(STR$(INT(A1!-INT(A1!/60)*60)),2),2)
```

Once you have converted hours, minutes, and seconds to seconds, you can compute elapsed times by simple subtraction. If you wish to express those elapsed times in hours, minutes, and seconds, you can use the FNHM$ function call to convert them back.

Time Clock Math

You'll want to use this function call the next time you design a program to accumulate times from employee time cards. FNTD! accepts two arguments. The first argument is a string indicating the start time. The second is a string indicating the stop time. Both arguments are in the format 'HH:MM' where 'HH' ranges from 1 to 12 and 'MM' ranges from 0 to 59. The start and stop times must be less than 12 hours apart. The single precision number returned by the function call is in decimal format, ready for you to multiply it by an hourly rate if necessary.

Time Clock Subtraction Function

```
15 DEFFNTD!(A1$,A2$)=ABS(-12*((VAL(A2$)+VAL(MID$(A2$,INSTR(A2$+"
:",":")+1))/60)<(VAL(A1$)+VAL(MID$(A1$,INSTR(A1$+":",":")+1))/60
))+(VAL(A2$)+VAL(MID$(A2$,INSTR(A2$+":",":")+1))/60)-(VAL(A1$)+V
AL(MID$(A1$,INSTR(A1$+":",":")+1))/60))
```

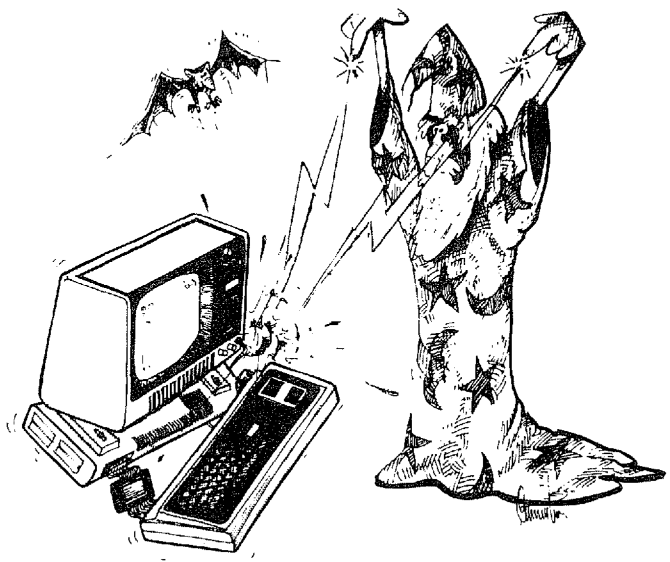
Here's a program that illustrates the use of the time clock math function call:

Time Clock Subtraction Demonstration Program

```
15 'MERGE TIME CLOCK SUBTRACTION FUNCTION DEFINITION HERE

110 CLS:PRINT"TIME CLOCK SUBTRACTION TEST PROGRAM
120 PRINT
130 LINEINPUT"1ST TIME:  ";A1$
140 LINEINPUT"2ND TIME   ";A2$
150 PRINT"DIFFERENCE=";FNTD!(A1$,A2$);" HOURS"
160 GOTO120
```

Remember that you can use the formatted inkey routine that is discussed in this book to simplify operator input, while enforcing valid entries. To use it for entry of hours and minutes, your command is:

```
AF$=STRING$(2,95)+":"+STRING$(2,95)
GOSUB40150
```



Bit Manipulation

There are 8 bits in each byte, 524,288 bits in the memory of a 48K TRS-80 and 686,080 useable bits on a formatted 35-track diskette. Are you getting your money's worth?

In this chapter we'll look at ways to take advantage of each of the 8 bits in a byte in real-world applications.

Setting a Bit of a Byte

The 'byte' is the most common unit of measure in computer applications. A byte is usually described as one character of information, such as a letter, ('A', 'B', 'C'), a single digit ('1', '2', '3') or a special character, ('$', '?', '%'). In reality, a byte is any of 256 possible codes interpreted from the 'on/off' status of 8 bits. A bit is the smallest unit of information storage on a computer. It represents the on or off status of a specific electronic or magnetic location in memory or on a diskette. In a byte we can store a number from 0 to 255 or we can store the 'yes-no' status of 8 different conditions.

We number the 8 bits in a byte from 0 to 7. BASIC lets us create a 1-byte string with the CHR$ function. CHR$(1), for example, generates a byte with the zero bit is set. CHR$(2) generates a byte in which bit 1 is set. CHR$(3) generates a byte in which bit 1 and 0 are set. CHR$(65) generates a byte, which by ASCII standards, represents the letter 'A'. For the letter 'A', bit 0 and bit 6 are set.

To convert the bits in a byte to a number, we look at each bit as a power of 2 and add. For example, we said that to represent 3, bits 1 and 0 are set. The 3 was obtained by adding 2 to the 0 power, which is 1 and 2 to the 1st power, which is 2. The 65 was obtained by adding 2 to the 0 power, which is 1 and 2 to the 6th power, which is 64. You'll find it very useful know the powers of 2. They are:

$2^0=1$	$2^1=2$	$2^2=4$	$2^3=8$
$2^4=16$	$2^5=32$	$2^6=64$	$2^7=128$
$2^8=256$	$2^9=512$	$2^{10}=1024$	$2^{11}=2048$
$2^{12}=4096$	$2^{13}=8192$	$2^{14}=16384$	$2^{15}=32768$

M 2 Note # 36 To set any bit, B%, in a 1-byte string, A$, our command is...

```
A$=CHR$(ASC(A$)OR2↑B%)
```

To set bit 5 in string S$, our command would be,

```
        S$=CHR$(ASC(S$)OR2↑5)

or,

        S$=CHR$(ASC(S$)OR32)
```

In these expressions we used the ASC function to convert the character stored in a string to an integer. Then we used the OR operator with a power of 2 to set the desired bit. Finally, we used the CHR$ function to convert back to a 1-byte string.

An integer number in BASIC is stored as two contiguous bytes in memory. We can set any bit, B%, in an integer, I%, with the following expression:

```
I%=I%OR2↑B%
```

To set bit 12 in integer I% we can say:

```
        I%=I%OR2↑12
or,
        I%=I%OR4Ø96
```

Be careful not to try to set bit 15 in an integer with this method. Since 32768 is beyond the valid range for integers, you'll get an overflow error.

A Bit on Bit Testing

When we 'test' on a bit we are checking to see whether it has been set or not. We can test on any bit by using the AND operator and a power of 2. A 'true' test, meaning that the bit is set, will return a non-zero integer. A 'false' test, meaning that the bit is not set, will return a zero. Using the result of a bit test, we can perform an 'IF/THEN' operation.

To test on bit, B%, in a 1-byte string, A$, with the result of the test as R%, our command is:

```
R%=ASC(A$)AND2↑B%
```

More commonly though, we will want to put this test into an IF/THEN expression:

```
IF ASC(A$)AND2↑B% THEN...
```

Then we could have an expression that reads:

```
IF ASC(A$)AND8 THEN PRINT "BIT 3 IS SET"
```

To test all 8 bits of a 1-byte string, A$, we can use:

```
FOR X = Ø TO 7
PRINT "BIT";X,
IF ASC(A$)AND2↑X THEN PRINT "YES" ELSE PRINT "NO"
NEXT
```

To test on a bit, B%, in an integer, I%, returning the result in R%, we can use the same logic:

```
 R%=I%AND2↑B%
or,
 IF I%AND2↑B% THEN PRINT "BIT";B%;" IS SET"
```

We use the term 'reset' to mean 'turn off' or 'zero' a bit. When we reset a bit, we are returning it to a 'no' condition.

To reset a bit, we can use the 'ANDNOT' operator with a power of 2. To reset a bit, B%, in a 1-byte string, A$, our command is:

```
A$=CHR$(ASC(A$)ANDNOT2↑B%)
```

To reset bit 4 of the 1-byte string, S$, we could say:

```
 S$=CHR$(ASC(S$)ANDNOT2↑4)
or,
 S$=CHR$(ASC(S$)ANDNOT16)
```

When working with integers, we can reset bit B% in integer I% with the expression:

```
I%=I%ANDNOT2↑B%
```

Useful Bit Uses

The ability to set, reset and test any bit lets us store 8 'yes-no' status indicators or 'flags', in a single byte. Efficient use of this fact can provide a great savings in memory and disk storage. We want to store as many names and addresses as possible and we often want to store coded information about each name. If you can spare 1 byte per name, you can store 8 additional information codes for each name, each code being a yes-no indicator.

In a mailing system I once developed, we wanted to keep track of which letters had been sent to each prospective customer and which other actions had been taken. The program was designed so that, for example, bit 0 could indicate that the original letter was sent, bit 1 could indicate that a follow-up letter was sent, bit 5 might indicate 'telephone call', bit 6 could indicate 'in-person sales call'. The user was able to use the 8 bits for any 8 yes-no indicators.

In an invoicing application, 1 byte for each product on file may be used to indicate any combination of 8 pricing, stocking and invoice printing codes. If a bit is set, the condition applies to the product. For example,

Bit 0 indicates a non-taxable product.
Bit 1 indicates a non-discountable product.
Bit 2 indicates variable price - operator entry.
Bit 3 indicates variable description - operator entry.

Here's another idea I've used. When you have several operations to perform on each record of a disk file, you can set a bit within each disk record as each operation is completed. That way, if the process is interrupted, your computer will know exactly which operations have been completed and recovery is possible without a complete restart.

I'm sure you'll find many other ways to take advantage of bit manipulations.

Combination Bit Tests

To test for a combination of bits you simply create a 'template' byte composed of the bit combination you want to test for. For an exact match, the byte you are checking will be exactly equal to the template byte. If you want to accept a partial match, (one or more bits, but not necessarily all, match the template), you can 'AND' the template byte with the byte you are checking. A non-zero result will indicate either a partial or exact match.

Let's say you are searching a 199 element array of 1-byte strings, each consisting of 8 indicator bits. You want to find all those that have bits 3 and 5 set. Your commands, to find the exact and partial matches could be:

```
T$=CHR$((2↑3)OR(2↑5))
FOR X=ØTO199
PRINT X,
IF S$(X)=T$ THEN PRINT "MATCH"
   ELSE IF ASC(S$(X))ANDASC(T$) THEN PRINT "PARTIAL MATCH"
        ELSE PRINT "NO MATCH"
NEXT
```

We've been looking at ways to set, reset and test bits within a single byte. Since a string can hold 255 bytes, we can store up to 2040 bits in a string. A 'bit-map string' is simply a string of any length, which we are using to store bit indicators. Each bit represents a yes or no condition. If the bit is set, 'yes' is indicated.

The length of your bit-map string will depend on the number of conditions you want to allow for. A 5-byte bit-map string can, for example, store the status of 40 conditions. The required length, L%, of a bit-map string to handle a specific number of conditions, N%, is given by the expression:

```
L% = INT(N%/8)+1
```

To initialize bit-map string, BM$, of length, L%, so that each bit is preset to a 'no' condition, your command is:

```
BM$ = STRING$(L%,Ø)
```

To initialize a bit-map string, BM$, of length, L%, so that each bit is preset to a 'yes' condition, your command is:

```
BM$ = STRING$(L%,255)
```

The FNSB$, FNRB$ and FNTB% functions let you set, reset or test any bit within a string. The desired bit is specified based on its position relative to the first bit of the string. Bit 0 is considered to be the first bit.

FNSB$(A1$,A2%) returns the string specified by argument 1, modified so that the bit specified by argument 2 is set. Argument 2 can be any bit ranging from 0 to 2031, provided that the bit is not beyond the length of the string.

The expression,

```
Z$=FNSB$(Z$,1234)
```

. . . sets relative bit 1234 in the string, Z$. The expression,

```
X$=FNSB$(Z$,334)
```

. . . loads X$ with the contents of Z$, with relative bit 334 set. Z$, in this case, is unaltered.

Set Any Bit Function

```
21 DEFFNSB$(A1$,A2%)=LEFT$(A1$,INT(A2%/8))+CHR$(ASC(MID$(A1$,INT
(A2%/8)+1,1))OR2↑(A2%-INT(A2%/8)*8))+MID$(A1$,INT(A2%/8)+2)
```

FNRB$(A1$,A2%) returns the string specified by argument 1, modified so that the bit specified by argument 2 is reset. Argument 2 can be any bit in the range 0 through 2031, provided that the bit is not beyond the length of the string.

You can use FNRB$ exactly the same way that you use FNSB$, except the specified bit is reset. The expression:

```
Z$=FNRB$(Z$,2011)
```

. . . resets relative bit 2011 in the string Z$.

Reset Any Bit Function

```
22 DEFFNRB$(A1$,A2%)=LEFT$(A1$,INT(A2%/8))+CHR$(ASC(MID$(A1$,INT
(A2%/8)+1,1))ANDNOT2↑(A2%-INT(A2%/8)*8))+MID$(A1$,INT(A2%/8)+2)
```

FNTB%(A1$,A2%) tests the bit specified by argument 2 within the string specified by argument 1. If the bit is set, −1 will be returned by the function, indicating a 'true' condition. If it is not set, 0 will be returned, indicating a 'false' condition.

FNTB%(Z$,35) will equal −1 if relative bit 35 is set in the string, Z$. It will equal 0 if relative bit 35 is not set.

You can easily use FNTB% in IF-THEN statements. For instance, to allow the operator to inquire into the status of a bit in the string, S$, your program can use the following logic:

```
INPUT "TEST WHICH BIT";B%
IF FNTB%(S$,B%) THEN PRINT "YES" ELSE PRINT "NO"
```

Test Any Bit Function

```
23 DEFFNTB%(A1$,A2%)=(ASC(MID$(A1$,INT(A2%/8)+1))AND2↑(A2%-INT(A
2%/8)*8))<>0
```

The BITMAPFN/DEM program lets you test the bit-map function calls. It first initializes a 255 byte string, BM$, to zeros. Then it lets you enter 'S', 'R' or 'T' to set, reset or test any bit in the string. You will need to merge in the FNSB$, FNRB$ and FNTB% function definitions at any available line numbers before line 100.

You'll notice that the CLEAR command in line 1 sets aside a large amount of string space for this simple program. This is necessary, because during the processing of the FNSB$ and FNRB$ functions, BASIC needs to temporarily store up to 4 copies of the string we are modifying. That space is automatically freed when the function returns, but it can be a consideration to keep in mind for programs that you write.

BITMAPFN/DEM

Bit-Map String Function Demonstration

```
0 'BITMAPFN/DEM
1 CLEAR1030

20 'MERGE FNSB$, FNRB$, AND FNTB% IN THIS AREA

90 BM$=STRING$(255,0)    'INITIALIZE BITMAP STRING FOR 2040 BITS

100 CLS:PRINT"BIT-MAP STRING FUNCTION DEMONSTRATION"
105 PRINT
110 INPUT"<S>SET  <R>RESET  <T>TEST  ";A$
111 A%=INSTR("SRT",A$):IFA%=0THEN110ELSEONA%GOTO200,300,400

200 INPUT"SET WHICH BIT                ";A%
201 IFA%<0ORA%>2031THENPRINT"ERROR...":GOTO200
210 BM$=FNSB$(BM$,A%)
220 PRINT"BIT";A%;" HAS BEEN SET.":GOTO105

300 INPUT"RESET WHICH BIT              ";A%
301 IFA%<0ORA%>2031THENPRINT"ERROR...":GOTO300
310 BM$=FNRB$(BM$,A%)
320 PRINT"BIT";A%;" HAS BEEN RESET.":GOTO105

400 INPUT"TEST WHICH BIT               ";A%
401 IFA%<0ORA%>2039THENPRINT"ERROR...":GOTO400
410 IFFNTB%(BM$,A%)THENPRINT"IT'S SET"ELSEPRINT"IT'S NOT SET"
411 GOTO105
```

Brisk Bit Finding

BITSRCH is a relocatable USR subroutine that lets you, quick as a crash, find the next bit that is set within a string, starting from any bit position in that string. When combined with the capabilities of the bit-map functions, BITSRCH can provide many powerful high-speed capabilities.

Here are some examples:

1. You can set up a bit-map string that indicates which disk records are active or which have been deleted. Each bit in the string corresponds to a disk file logical record. Each call to the bit-map search USR routine can return the next record number to access. The same idea can be used with arrays.

2. You can set bits in a string corresponding to random disk file logical records that meet specific criteria. Then, rather than reading the entire disk file for printing or processing, you can search the bit-map string, getting only those disk file records corresponding to the bits that are set. Tremendous performance improvements are possible with this technique.

3. You can set up a bit-map string in which each bit corresponds to a check or invoice number. If the bit has been set, that check or invoice number has been used. With the BITSRCH USR routine, you can quickly print a list of the missing checks or invoices or alternatively, the checks or invoices that have been used.

The calling argument to the BITSRCH USR routine is the starting relative bit number in the string to be searched. The integer returned is the number corresponding to the next bit that is set. The routine returns -1 if no subsequent bits are set in the string. The VARPTR of the string to be searched must be loaded into the 5th and 6th bytes of the BITSRCH USR routine. This can be done by loading the 3rd element with the VARPTR if you are using the magic array method or with poke commands to the 5th and 6th bytes if you've got the routine in protected memory.

Let's say for example, you've got a bit-map in the string, S$. Let's also assume you've loaded the BITSRCH routine into the US% integer array. To search for the first bit that is set, your commands are:

```
US%(2)=VARPTR(S$)           'LOAD STRING VARPTR
J=0                         'MAKE SURE J IS INITIALIZED
DEFUSR0=VARPTR(US%(0))      'DEFINE AS USR0
J=USR0(0)                   'CALL ROUTINE, RESULTS IN J
IF J=-1 THEN ....           'HANDLE NOT-FOUND CONDITION
PRINT J                     'PRINT BIT NUMBER
```

To sequentially search the entire string, returning the relative number of each bit that is set, you can use the following logic:

```
10  X=0                     'STARTING BIT IS ZERO
20  J=USR0(X)               'CALL ROUTINE, STARTING FROM BIT X
30  IF J=-1 THEN 50
      ELSE PRINT J          'END IF BIT IS SET, OTHERWISE PRINT
40  X=X+1:GOTO20            'REPEAT SEARCH FROM NEXT BIT
50  PRINT"NO MORE BITS"     'END SEARCH
```

As shown below, the BITSRCH routine searches for the next bit that is set. You can modify it to search for the next bit that is not set with the following guidlines:

1. If you are using the magic array method, replace the 24th element, '8263' with 10311.

2. If you are using the poke method, replace the 48th byte, '32' with 40.

3. If you are assembling the BITSRCH USR routine, replace the 'JR NZ,FOUND' in line 350 with 'JR Z,FOUND'.

BITSRCH
Bit-Map String Search USR Subroutine
M 2 Note # 23
M 2 Note # 37

Magic Array Format, 36 elements:

32717	4362	0	-5147	9038	9054	-10922	-7715	4577
0	3340	10792	32477	1536	-6904	-4681	-7854	2344
-13549	4159	-8716	6179	-13336	8263	-13548	9023	-2288
9181	10253	-8953	126	2054	-5352	-223	-15361	2714

Poke Format, 72 bytes

205	127	10	17	0	0	229	235	78	35	94	35	86	213	221	225
225	17	0	0	12	13	40	42	221	126	0	6	8	229	183	237
82	225	40	9	19	203	63	16	244	221	35	24	232	203	71	32
20	203	63	35	16	247	221	35	13	40	7	221	126	0	6	8
24	235	33	255	255	195	154	10								

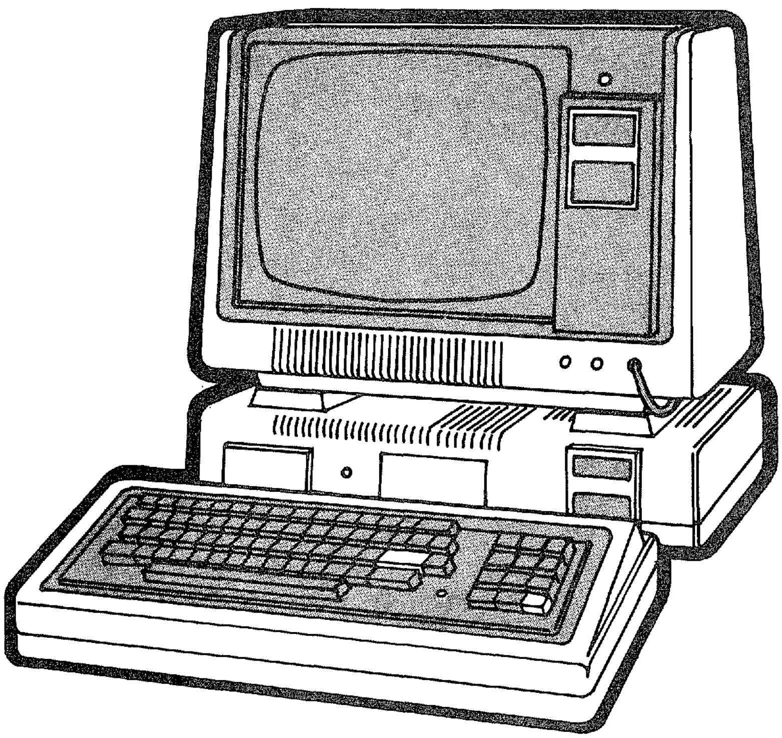
```
FF00          00020          ORG     0FF00H         ;ORIGIN - RELOCATABLE
FF00 CD7F0A   00040          CALL    0A7FH          ;HL=STARTING RELATIVE BIT
FF03 110000   00050          LD      DE,0000        ;DE=STRING VARPTR
FF06 E5       00060          PUSH    HL             ;SAVE STARTING REL BIT
FF07 EB       00070          EX      DE,HL          ;
FF08 4E       00080          LD      C,(HL)         ;C=STRING LENGTH
FF09 23       00090          INC     HL             ;HL POINTS TO POINTERS
FF0A 5E       00100          LD      E,(HL)         ;
FF0B 23       00110          INC     HL             ;
FF0C 56       00120          LD      D,(HL)         ;DE POINTS TO STRING
FF0D D5       00130          PUSH    DE             ;
FF0E DDE1     00140          POP     IX             ;IX POINTS TO STRING
FF10 E1       00150          POP     HL             ;HL NOW POINTS TO START
FF11 110000   00160          LD      DE,0           ;INITIALIZE COUNT
FF14 0C       00170          INC     C              ;
              00180 ;THE FOLLOWING LOGIC INCREMENTS TO STARTING BIT
FF15 0D       00190 LOOP1    DEC     C              ;SUBTRACT FROM BYTE COUNT
FF16 282A     00200          JR      Z,ENDSTR       ;END OF STRING IF ZERO
FF18 DD7E00   00210          LD      A,(IX)         ;GET CURRENT BYTE FROM STRING
FF1B 0608     00220          LD      B,08H          ;LOAD BIT COUNTER
FF1D E5       00230 LOOP2    PUSH    HL             ;SAVE DESIRED START
FF1E B7       00240          OR      A              ;CLEAR CARY FLAG
FF1F ED52     00250          SBC     HL,DE          ;ARE WE THERE YET?
FF21 E1       00260          POP     HL             ;RESTORE DESIRED START
FF22 2809     00270          JR      Z,ATSTRT       ;WE'RE AT THE START
FF24 13       00280          INC     DE             ;ADD TO COUNT
FF25 CB3F     00290          SRL     A              ;SHIFT NEXT BIT INTO POSITION
FF27 10F4     00300          DJNZ    LOOP2          ;LOOK AT NEXT BIT IF NECESS
FF29 DD23     00310          INC     IX             ;POINT TO NEXT BYTE
FF2B 18E8     00320          JR      LOOP1          ;GO REPEAT FOR NEXT BYTE
              00330 ;THE FOLLOWING LOGIC LOOKS FOR NEXT BIT THAT IS SET
FF2D CB47     00340 ATSTRT   BIT     0,A            ;IS THE BIT SET
FF2F 2014     00350          JR      NZ,FOUND       ;FOUND NEXT BIT THAT'S SET
FF31 CB3F     00360          SRL     A              ;SHIFT NEXT BIT INTO POSITION
FF33 23       00370          INC     HL             ;ADD TO COUNT
FF34 10F7     00380          DJNZ    ATSTRT         ;REPEAT IF MORE BITS THIS BYTE
FF36 DD23     00390          INC     IX             ;POINT TO NEXT BYTE
FF38 0D       00400          DEC     C              ;DEC STRING BYTE COUNT
FF39 2807     00410          JR      Z,ENDSTR       ;END OF STRING IF ZERO
FF3B DD7E00   00420          LD      A,(IX)         ;LOAD NEXT BYTE TO ACCUM
FF3E 0608     00430          LD      B,08H          ;INITIALIZE BIT COUNT
FF40 18EB     00440          JR      ATSTRT         ;REPEAT FOR NEXT BYTE
FF42 21FFFF   00450 ENDSTR   LD      HL,0FFFFH      ;PASS BACK -1 IF END OF STR
FF45 C39A0A   00460 FOUND    JP      0A9AH          ;RETURN RESULT IN HL TO BASIC
0A9A          00470          END                    ;
```

You can demonstrate and test the BITSRCH USR routine by modifying the bit-map function demonstration program. Simply merge in the following lines:

BITSRCH/DEM
Modifications to BITMAPFNDEM for Bit-Map Searches

M 2 Note # 23
M 2 Note # 37

```
0 'BITSRCH/DEM

30 'LOAD BIT SEARCH ROUTINE INTO A MAGIC ARRAY
31 DATA 32717, 4362, 0,-5147, 9038, 9054,-10922,-7715, 4577, 0,
3340, 10792, 32477, 1536,-6904,-4681
32 DATA-7854, 2344,-13549, 4159,-8716, 6179,-13336, 8263,-13548,
 9023,-2288, 9181, 10253,-8953, 126, 2054
33 DATA-5352,-223,-15361, 2714
34 DIMUS%(35):FORX=0TO35:READUS%(X):NEXT

100 CLS:PRINT"BIT-MAP STRING SEARCH DEMONSTRATION"
110 INPUT"<S>SET  <R>RESET  <T>TEST  <L>LIST";A$
111 A%=INSTR("SRTL",A$):IFA%=0THEN110ELSEONA%GOTO200,300,400,500

500 US%(2)=VARPTR(BM$):J=0:DEFUSR=VARPTR(US%(0))

510 X=0
520 J=USR(X):IFJ=-1THENPRINT:GOTO105ELSEPRINTJ;
530 X=J+1:GOTO520
```



Arrays, Searches & Sorts

When programming the TRS-80 or any other computer, you'll often find a need to work with lists of data. When you think about it, a major percentage of computer programming involves the storage and retrieval of information in one way or another.

In this section, we'll reveal some techniques that can give you dramatic increases in memory storage capacity and fantastic improvements in program execution speed. We'll be dealing with the array handling capabilities of BASIC and we'll go beyond BASIC for some special-purpose high-performance array storage techniques.

Peeks and Pokes for BASIC Arrays

When you dimension an array, you are setting aside a block in memory for the storage of data. The command, DIM A% (40), reserves space for 41 integers, which you can load or retrieve using the subscripted variables A% (0) through A% (40). In total, 82 bytes are reserved for the storage of the data in the A% array, because each integer requires 2 bytes. In addition, several bytes are used by BASIC to store information about the variable name, the dimension and the type of array it is.

The command,

```
PRINTVARPTR(A%(Ø))
```

. . . will display the memory address of the first element in the array. The second element of the array, A% (1) will be stored 2 bytes above the base of the array.

The dimension of an array is stored in the first 2 bytes preceding the first element. If we type,

```
PRINT PEEK(VARPTR(A%(Ø))-2) + PEEK(VARPTR(A%(Ø))-1)*256
```

. . . we get 41, the number of elements in the array. If we tell the computer,

```
PRINT PEEK(VARPTR(A%(Ø))-8)
```

. . . we get the type code, 2, indicating that this is an integer array, each element being 2 bytes long.

Single and double precision arrays are stored the same way. For a single precision array, the type code is 4, indicating that each element takes 4 bytes. For a double precision array, the type code is 8. Each element occupies 8 bytes.

In a string array, BASIC sets aside 3 bytes for each element. Therefore, if we dimension the array, S$, using DIM S$(99), 300 bytes will be used, plus several bytes for the variable name, array type and dimension indicators. If we issue the command,

```
PRINT PEEK(VARPTR(S$(Ø))-8)
```

... we get 3, the type code for a string variable. Those 3 bytes for each element in the array indicate the length and a pointer to the address of the data contained in the string. If we say,

```
PRINT PEEK(VARPTR(S$(5)))
```

... we get the length of S$(5). If we use the command,

```
PRINT PEEK(VARPTR(S$(5))+1)+PEEK(VARPTR(S$(5))+2)*256
```

... we get the address of the data stored in S$(5).

How to Instantly Clear an Array

We can use the memory block duplication capabilities of our move-data magic array USR routine to load zeros into all elements of an array or to load any desired value into each element of an array. We simply load the first element with the value to be duplicated, (zero) and duplicate that value as many times as we want. The array element duplication demonstration program shows how to quickly clear a large array and instantly load each element with the same value.

In BASIC, you'll find that it takes 8 to 9 seconds to clear or load a value into 1000 elements of an array. The technique shown below does it in a small fraction of a second. Before trying it, be sure to read the section on magic arrays.

ELEMDUP/DEM
Array Element Duplication Demonstration Program

```
10 N=1000:DIM A!(N):J%=0
20 US%(0)=8448:US%(2)=4352:US%(4)=256:US%(6)=-20243:US%(7)=201
30 PRINT"LOADING 1234 INTO EACH ELEMENT OF THE A! ARRAY..."
35 A!(0)=1234: GOSUB100
40 PRINT"LOADING 0    INTO EACH ELEMENT OF THE A! ARRAY..."
45 A!(0)=0: GOSUB100
50 END

100 US%(1)=VARPTR(A!(0)):US%(3)=VARPTR(A!(1)):US%(5)=N*4
101 DEFUSR=VARPTR(US%(0)):J%=USR(0):RETURN
```

You can modify the array element duplication demo to do the same thing with an integer or double precision array. Just change the A!'s to A%'s or A#'s. For integer arrays, US%(5), in line 100 should be set to N*2. For double precision arrays, US%(5) in line 100 should equal N*8. To see how this works for a string array, change the A!'s to A$'s. Then change line 35 to read:

```
35 A$(0)="1234":GOSUB100
```

... and change line 45 to read:

```
45 A$(0)="":GOSUB100
```

Finally, change line 100 so US%(5)=N*3.

When we duplicate elements in a string array, we are really just duplicating the pointers. In our example, the '1234' string is in memory at only one location and each of the 1000 elements in the A$ array point to that location.

Insert & Delete Array Elements – Instantly

Suppose you have dimensioned a string array for a capacity of 1000 elements. Currently you are storing 900 names in that array in elements 1 through 900. You want to delete the 5th name and then move the names in positions 6 through 900 down 1 position, leaving 899 names. Or perhaps you want to make space to insert a new name at the 40th position by moving every name above position 39 up 1 position. To do these operations in BASIC can be very time consuming for a large array.

The IDARRAY USR routine lets you use the speed of Z-80 machine language programming to perform insert and delete operations for any singly dimensioned integer, single precision, double precision or string array.

To delete an element, you simply specify the array to be altered and the element to be deleted. All subsequent elements are moved down 1 position and the top element is loaded with zero.

To insert an element, you specify the array and the element number. The USR routine moves up all elements at and above that position. You can then load the element with the value to be inserted. (If an element was at the top position of the array before the insertion, it is deleted.)

To call the IDARRAY USR routine, you must have first loaded it and used the DEFUSR command so that BASIC will know where to find it. Then you load a 3-element integer control array with the parameters for your insert or delete operation:

Element 0 = 1 to insert and 0 if you want to delete.
Element 1 = VARPTR for element 0 of the array to alter.
Element 2 = element number to be inserted or deleted.
(0 is the first element.)

When you make the USR call, your argument is the VARPTR of the first element of the control array. If P%(0), P%(1) and P%(2) contain the control information, your call is:

```
J=USR(VARPTR(P%(Ø)))
```

If we've defined USR4 to point to the IDARRAY subroutine and we want to delete element 5 from string array S$, we would use the following commands:

```
P%(Ø)=Ø:P%(1)=VARPTR(S$(Ø)):P%(2)=5:J%=USR4(VARPTR(P%(Ø)))
```

To delete the 5th element from double precision array, D#, our commands would be:

```
P%(Ø)=Ø:P%(1)=VARPTR(D#(Ø)):P%(2)=4:J%=USR4(VARPTR(P%(Ø)))
```

To insert the string, 'JONES' at the 7th position of string array, S$, we would use the following commands:

```
P%(0)=1:P%(1)=VARPTR(S$(0)):P%(2)=6:J%=USR4(VARPTR(P%(0)))
S$(6)="JONES"
```

IDARRAY/DEM is a BASIC program that you can use to demonstrate and test the IDARRAY USR routine:

IDARRAY/DEM

Array Element Insertion & Deletion Demonstration Program

M 2 Note # 23

```
0 'IDARRAY/DEM

10'LOAD IDARRAY USR ROUTINE INTO A MAGIC ARRAY
11 DATA 32717,-6902,-7715,28381,-8958,870,11237,11094,11102,1105
1,11051,32299,28381,-8956,1382,-6699,-13489,-13343
12 DATA 10553,10731,-13333,12345,-13320,10311,-16120,-5367,2497,
6379,-16126,-15935,-5367,1545,20224,-13347,17920,4896
13 DATA-5163,-6903,-18453,21229,-15899,-11807,552,-20243,6187,11
027,-18459,17133,9189,-4681,-6830,-7743,10449,-4862,-5192,15943,
30464,4139,-13828
14 DIMUS%(58):FORX=0TO58:READUS%(X):NEXT

100 DEFINTA-Z:J=0
110 DEFSTRA       'DEMONSTRATE USING A STRING ARRAY
120 DIMA(11)      'DIMENSION THE DEMONSTRATION ARRAY
130 DIMP(2)       'DIMENSION THE CONTROL ARRAY

150 'LOAD DEMONSTRATION DATA
151 DATA 100,101,102,103,104,105,106,107,108,109,110,111
152 FORX=0TO11:READA(X):NEXT
170 GOSUB1000
180 PRINT@832,CHR$(31);:INPUT"D=DELETE, I=INSERT   ";A$
181 P(0)=INSTR("DI",A$)-1:IFP(0)<0ORLEN(A$)=0THEN180
190 PRINT@864,CHR$(31);:INPUT"ELEMENT# ";P(2)
191 IFP(2)>11ORP(2)<0THEN190
200 IFP(0)=0THEN210ELSEPRINT@896,CHR$(31);:INPUT"NEW CONTENTS
";AN
210 P(1)=VARPTR(A(0)):DEFUSR=VARPTR(US%(0)):J=USR(VARPTR(P(0)))
220 IFP(0)=1THENA(P(2))=AN
230 GOSUB1000:GOTO180
1000 CLS:PRINT"ARRAY CONTENTS...":FORX=0TO11:PRINTUSING"###";X;:
PRINTTAB(20)A(X):NEXT:RETURN
```

The array element insertion and deletion demonstration shows how the IDARRAY USR routine works with a string array. To see how it works with a integer array, single precision array or double precision array, simply change the 'DEFSTR' in line 110 to a DEFINT, DEFSNG or a DEFDBL.

There are a few things you must remember when calling the IDARRAY subroutine:

1. Element 1 of your control array must be the VARPTR to element 0 of a singly dimensioned array. Any other value will cause dangerous results because the routine doesn't check the validity of the control arguments you give it.

2. Element 2 of your control array must not be greater than the dimension that you've assigned to the array to be altered and it must not be less than zero. Again, the USR routine does no validation, so it is up to you in your BASIC program. (Line 191 does this validation in our demo program.)

3. As with all USR routine control arrays, your control array must be defined as integer. In our sample program, the P(0), P(1) and P(2) are

the control array elements. The DEFINT in line 100 defined all variables as integers, so we satisfied the requirment.

In application programs, you'll probably want to set up a variable that keeps track of the next element number in your array. When the array is empty, the next element number will be zero. Each time you add an element, add 1 to the next element pointer. Each time you delete an element, subtract 1. When you want to add an element to your array just after the last active element, you can add it at the position shown by your next element pointer. Then you can add 1 to the pointer.

The IDARRAY USR routine is 118 bytes long. Because of its length, your preference should be to store it on disk, rather than poking it into memory or using the magic array method.

IDARRAY
Array Element Insertion & Deletion USR Subroutine

M 2 Note # 23

```
Magic Array Format, 59 elements:
 32717  -69Ø2  -7715  28381  -8958    87Ø  11237  11Ø94  111Ø2
 11Ø51  11Ø51  32299  28381  -8956   1382  -6699 -13489 -13343
 1Ø553  1Ø731 -13333  12345 -1332Ø  1Ø311 -1612Ø  -5367   2497
  6379 -16126 -15935  -5367   1545  2Ø224 -13347  1792Ø   4896
 -5163  -69Ø3 -18453  21229 -15899 -118Ø7    552 -2Ø243   6187
 11Ø27 -18459  17133   9189  -4681  -683Ø  -7743  1Ø449  -4862
 -5192  15943  3Ø464   4139 -13828
Poke Format, 118 bytes:
  2Ø5 127  1Ø 229 221 225 221 11Ø   2 221 1Ø2   3 229  43  86  43
   94  43  43  43  43  43  43 126 221 11Ø   4 221 1Ø2   5 213 229
   79 2Ø3 225 2Ø3  57  41 235  41 235 2Ø3  57  48 248 2Ø3  71  4Ø
    8 193   9 235 193   9 235  24   2 193 193 193   9 235   9   6
    Ø  79 221 2Ø3   Ø  7Ø  32  19 213 235   9 229 235 183 237  82
  229 193 225 2Ø9  4Ø   2 237 176  43  24  19  43 229 183 237  66
  229  35 183 237  82 229 193 225 2Ø9  4Ø   2 237 184 235  71  62
    Ø 119  43  16 252 2Ø1
```

```
                 ØØØØØ ;IDARRAY
                 ØØØØ1 ;
FFØØ             ØØØ9Ø        ORG     ØFFØØH      ;ORIGIN - RELOCATABLE
FFØØ  CD7FØA     ØØ1ØØ        CALL    ØA7FH       ;PUT ARGUMENT IN HL
FFØ3  E5         ØØ11Ø        PUSH    HL          ;
FFØ4  DDE1       ØØ12Ø        POP     IX          ;IX POINTS TO CONTROL ZERO
FFØ6  DD6EØ2     ØØ13Ø        LD      L,(IX+2)    ;
FFØ9  DD66Ø3     ØØ14Ø        LD      H,(IX+3)    ;HL POINTS TO ARRAY ELEMENT Ø
FFØC  E5         ØØ15Ø        PUSH    HL          ;SAVE ON STACK
FFØD  2B         ØØ16Ø        DEC     HL          ;
FFØE  56         ØØ17Ø        LD      D,(HL)      ;
FFØF  2B         ØØ18Ø        DEC     HL          ;
FF1Ø  5E         ØØ19Ø        LD      E,(HL)      ;DE HAS DIMENSION
FF11  2B         ØØ2ØØ        DEC     HL          ;
FF12  2B         ØØ21Ø        DEC     HL          ;
FF13  2B         ØØ22Ø        DEC     HL          ;
FF14  2B         ØØ23Ø        DEC     HL          ;
FF15  2B         ØØ24Ø        DEC     HL          ;
FF16  2B         ØØ25Ø        DEC     HL          ;
FF17  7E         ØØ26Ø        LD      A,(HL)      ;ACCUM HAS TYPE: 2,3,4, OR 8
FF18  DD6EØ4     ØØ27Ø        LD      L,(IX+4)    ;
FF1B  DD66Ø5     ØØ28Ø        LD      H,(IX+5)    ;HL HAS ELEMENT #
FF1E  D5         ØØ29Ø        PUSH    DE          ;SAVE DIMENSION ON STACK
FF1F  E5         ØØ3ØØ        PUSH    HL          ;SAVE ELEMENT # ON STACK
FF2Ø  4F         ØØ31Ø        LD      C,A         ;TYPE 2,3,4, OR 8 TO C
FF21  CBE1       ØØ32Ø        SET     4,C         ;BIT 4 WILL STOP MULT LOOP
FF23  CB39       ØØ33Ø        SRL     C           ;SHIFT
FF25  29         ØØ34Ø MLOOP  ADD     HL,HL       ;MULT ELEMENT # BY 2
```

```
FF26 EB       00350          EX    DE,HL         ;
FF27 29       00360          ADD   HL,HL         ;MULTIPLY DIMENSION BY 2
FF28 EB       00370          EX    DE,HL         ;
FF29 CB39     00380          SRL   C             ;SHIFT UNTIL BIT FOUND
FF2B 30F8     00390          JR    NC,MLOOP      ;REPEAT
FF2D CB47     00400          BIT   0,A           ;TYPE CODE 3?
FF2F 2808     00410          JR    Z,JMP1        ;IF NOT, SKIP
FF31 C1       00420          POP   BC            ;BC HAS ELEMENT #
FF32 09       00430          ADD   HL,BC         ;HL HAS ELEMENT # * 3
FF33 EB       00440          EX    DE,HL         ;
FF34 C1       00450          POP   BC            ;BC HAS DIMENSION
FF35 09       00460          ADD   HL,BC         ;HL HAS DIMENSION * 3
FF36 EB       00470          EX    DE,HL         ;
FF37 1802     00480          JR    JMP2          ;
FF39 C1       00490 JMP1     POP   BC            ;RELIEVE STACK
FF3A C1       00500          POP   BC            ;RELIEVE STACK
FF3B C1       00510 JMP2     POP   BC            ;BC POINTS TO ARRAY ELEMENT 0
FF3C 09       00520          ADD   HL,BC         ;HL POINTS TO TARGET ELEMENT
FF3D EB       00530          EX    DE,HL         ;
FF3E 09       00540          ADD   HL,BC         ;HL POINTS TO TOP OF ARRAY
              00550 ;
              00560 ;AT THIS POINT, A CONTAINS TYPE: 2, 3, 4, OR 8
              00570 ;DE POINTS TO ELEMENT, HL POINTS TO TOP OF ARRAY
              00580 ;STACK IS CLEAR
FF3F 0600     00590          LD    B,0           ;
FF41 4F       00600          LD    C,A           ;BC HAS ELEMENT LENGTH
FF42 DDCB0046 00610          BIT   0,(IX+0)      ;TEST ON COMMAND
FF46 2013     00620          JR    NZ,INSERT     ;
FF48 D5       00630 DELETE   PUSH  DE            ;SAVE "TO" ADDRESS
FF49 EB       00640          EX    DE,HL         ;
FF4A 09       00650          ADD   HL,BC         ;HL HAS "FROM" ADDRESS
FF4B E5       00660          PUSH  HL            ;SAVE "FROM" ADDRESS
FF4C EB       00670          EX    DE,HL         ;
FF4D B7       00680          OR    A             ;
FF4E ED52     00690          SBC   HL,DE         ;SUBTRACT TOP - "FROM"
FF50 E5       00700          PUSH  HL            ;
FF51 C1       00710          POP   BC            ;BC HAS # BYTES TO MOVE
FF52 E1       00720          POP   HL            ;HL HAS "FROM" ADDRESS
FF53 D1       00730          POP   DE            ;DE HAS "TO" ADDRESS
FF54 2802     00740          JR    Z,NOMOVE      ;SKIP MOVE IF ZERO TO MOVE
FF56 EDB0     00750          LDIR                ;MOVE
FF58 2B       00760 NOMOVE   DEC   HL            ;HL POINTS TO TOP - 1
FF59 1813     00770          JR    ZFILL         ;GO FILL ZEROS TO TOP ELEMENT
FF5B 2B       00780 INSERT   DEC   HL            ;HL HAS "TO" ADDRESS
FF5C E5       00790          PUSH  HL            ;SAVE "TO" ADDRESS
FF5D B7       00800          OR    A             ;
FF5E ED42     00810          SBC   HL,BC         ;HL HAS "FROM" ADDRESS
FF60 E5       00820          PUSH  HL            ;SAVE "FROM" ADDRESS
FF61 23       00830          INC   HL            ;
FF62 B7       00840          OR    A             ;
FF63 ED52     00850          SBC   HL,DE         ;HL HAS # BYTES TO MOVE
FF65 E5       00860          PUSH  HL            ;
FF66 C1       00870          POP   BC            ;BC HAS # BYTES TO MOVE
FF67 E1       00880          POP   HL            ;HL HAS "FROM" ADDRESS
FF68 D1       00890          POP   DE            ;DE HAS "TO" ADDRESS
FF69 2802     00900          JR    Z,JMP4        ;SKIP MOVE IF ZERO
FF6B EDB8     00910          LDDR                ;MOVE
FF6D EB       00920 JMP4     EX    DE,HL         ;
FF6E 47       00930 ZFILL    LD    B,A           ;B HAS # ZEROS TO FILL
FF6F 3E00     00940          LD    A,0           ;
FF71 77       00950 LOOP     LD    (HL),A        ;ZERO ELEMENT BYTE
FF72 2B       00960          DEC   HL            ;
FF73 10FC     00970          DJNZ  LOOP          ;REPEAT FOR EACH BYTE
FF75 C9       00980          RET                 ;RETURN TO BASIC
FF71          00990          END                 ;
00000 TOTAL ERRORS
```

Super String Array Searcher

The SEARCH1 USR routine lets your BASIC program search a string array based on a string that you provide as a search key. Based on your commands, you can search for the first string in the array that is less than or equal to, greater than or equal to, or not equal to the search key. You can start your search at any element in the array and you can specify the number of elements that are to be searched. The USR routine returns the element number, relative to your starting element, for the first string that qualifies. If no string in the array meets the conditions, −1 is returned to your BASIC program.

For large string arrays of about 1000 elements, your search time will be just a fraction of a second with the SEARCH1 routine, so the 133 bytes required for the machine language subroutine can be a good investment of memory. If you'd like to keep a string array in sequence, you can use the SEARCH1 routine in conjunction with the insert and delete capabilities of the IDARRAY USR routine. To add a key, just search for the first element that is greater, then insert at that point. You've got an interactive insertion sort for string arrays!

To call the SEARCH1 subroutine, you load an integer control array with the following:

Element 0 = VARPTR to the string array to be searched.
Normally this will be the VARPTR to element 0, but you can
start the search at any element.

Element 1 = The number of elements to be searched minus 1.
To search from element 0 to element 9, (10 elements),
you would load control array element 1 with 9.

Element 2 = VARPTR to the string that contains the search key.

Element 3 = Your command indicating the search mode:

1 = Find first element equal to search key.
2 = Find first element less than.
3 = Find first element less than or equal.
4 = Find first element greater than.
5 = Find first element greater than or equal.
6 = Find first element not equal.

When the control array has been loaded, you call the USR routine with the argument being the VARPTR to the control array. The USR subroutine returns the relative element number if one is found. If no element in the array qualifies for your search key and command, a −1 is returned to BASIC.

The SEARCH1 demonstration program sets up a sample array so that you can see how it works:

```
0  APPLE
1  BASKET
2  BAT
3  BERRY
4  CAT
5  CATTLE
6  DOG
```

Here are some sample searches:

```
START SEARCH AT ELEMENT # ? 0
SEARCH HOW MANY ELEMENTS  ? 7
SEARCH KEY                ? CAR
MODE                      ? 2
SEARCH RESULT = 0

START SEARCH AT ELEMENT # ? 3
SEARCH HOW MANY ELEMENTS  ? 4
SEARCH KEY                ? CATTLE
MODE                      ? 1
SEARCH RESULT = 2

START SEARCH AT ELEMENT # ? 0
SEARCH HOW MANY ELEMENTS  ? 3
SEARCH KEY                ? DOG
MODE                      ? 1
SEARCH RESULT = -1
```

Note that the P% array is the control array in the demonstration program. We load it in line 100. Line 110 calls the USR routine, with the results of the call being returned in the variable, 'J'. The magic array method is used for convenience of demonstration, so that you don't need to reserve memory for the USR routine. In most cases, though, it's preferable to load the routine into protected memory from a disk file so that you won't waste the memory taken by the data statements.

SEARCH1/DEM
String Array Search Demonstration Program

M 2 Note # 21
M 2 Note # 23
M 2 Note # 37

```
1 DEFINTA-Z:J=0

10 'LOAD SEARCH1 USR ROUTINE INTO A MAGIC ARRAY
11 DATA 32717,-6902,-7715, 20189,-8958, 838, 17, 2048, 32477,
 2054,-8743, 1134, 26333, 19973, 24099, 22051
12 DATA 28381,-8960, 358,-10811, 18149, 9173, 9054,-5290, 1233,
 8197, 3078, 8205, 6205, 3121, 10253, 6668
13 DATA 8382, 8966, 1299, 6157, 12520, 2091, 2293,-13327,
 8279,-9939,-20359, 2856, 4875,-7719, 8995,-11997
14 DATA 6337, 3011,-7711,-14879,-15391, 2714,-2808,-3832, 18379,
 3104,-8936,-2808,-3832, 20427, 544,-11496
15 DATA-10791, 6337, 223
16 DIMUS(66):FORX=0TO66:READUS(X):NEXT

30 'READ TEST DATA INTO A STRING ARRAY
31 DATA APPLE,BASKET,BAT,BERRY,CAT,CATTLE,DOG
32 FORX=0TO6:READSA$(X):NEXT

40 CLS:FORX=0TO6:PRINTX,SA$(X):NEXT
50 PRINT@640,CHR$(31);:INPUT"START SEARCH AT ELEMENT #    ";SS
60 PRINT@704,CHR$(31);:INPUT"SEARCH HOW MANY ELEMENTS     ";SN
70 PRINT@768,CHR$(31);:INPUT"SEARCH KEY                   ";SK$
80 PRINT@832,CHR$(31);"
1=EQUAL     2=LESS          3=LESS/EQUAL
4=GREATER   5=GREATER/EQUAL  6=NOT EQUAL";
81 PRINT@832,CHR$(30);:INPUT"MODE:                        ";MO

100 P(0)=VARPTR(SA$(SS)):P(1)=SN-1:P(2)=VARPTR(SK$):P(3)=MO
110 DEFUSR=VARPTR(US(0)):J=USR(VARPTR(P(0)))
120 PRINT@896,CHR$(31);:PRINT"SEARCH RESULT = ";J

130 LINEINPUT"PRESS <ENTER>...";A$:GOTO40
```

SEARCH1
String Array Search USR Subroutine

Magic Array Format, 67 ELEMENTS

```
 32717  -6902  -7715  20189  -8958    838     17   2048  32477
  2054  -8743   1134  26333  19973  24099  22051  28381  -8960
   358 -10811  18149   9173   9054  -5290   1233   8197   3078
  8205   6205   3121  10253   6668   8382   8966   1299   6157
 12520   2091   2293 -13327   8279  -9939 -20359   2856   4875
 -7719   8995 -11997   6337   3011  -7711 -14879 -15391   2714
 -2808  -3832  18379   3104  -8936  -2808  -3832  20427    544
-11496 -10791   6337    223
```

Poke Format, 133 BYTES

```
205 127  10 229 221 225 221  78   2 221  70   3  17   0   0   8
221 126   6   8 217 221 110   4 221 102   5  78  35  94  35  86
221 110   0 221 102   1 197 213 229  70 213  35  94  35  86 235
209   4   5  32   6  12  13  32  61  24  49  12  13  40  12  26
190  32   6  35  19   5  13  24 232  48  43   8 245   8 241 203
 87  32  45 217 121 176  40  11  11  19 217 225  35  35  35 209
193  24 195  11 225 225 225 197 225 195 154  10   8 245   8 241
203  71  32  12  24 221   8 245   8 241 203  79  32   2  24 211
217 213 193  24 223
```

```
                   ;SEARCH1
                   ;
F000                          ORG     0F000H      ;ORIGIN - RELOCATABLE
F000 CD7F0A                   CALL    0A7FH       ;HL POINTS TO CONTROL ARRAY
F003 E5                       PUSH    HL          ;PREPARE TO COPY TO IX
F004 DDE1                     POP     IX          ;IX POINTS TO CONTROL ARRAY
F006 DD4E02                   LD      C,(IX+2)    ;
F009 DD4603                   LD      B,(IX+3)    ;BC HAS # RECORDS TO SEARCH
F00C 110000                   LD      DE,0        ;DE HAS # RECORDS SEARCHED
F00F 08                       EX      AF,AF'      ;EXCHANGE TO AF'
F010 DD7E06                   LD      A,(IX+6)    ;A' HAS COMMAND
F013 08                       EX      AF,AF'      ;EXCHANGE BACK TO AF
F014 D9                       EXX                 ;EXCHANGE REGISTERS
F015 DD6E04                   LD      L,(IX+4)    ;
F018 DD6605                   LD      H,(IX+5)    ;HL' POINTS TO SKEY VARPTR
F01B 4E                       LD      C,(HL)      ;C' HAS SKEY LENGTH
F01C 23                       INC     HL          ;
F01D 5E                       LD      E,(HL)      ;
F01E 23                       INC     HL          ;
F01F 56                       LD      D,(HL)      ;DE' POINTS TO SKEY DATA
F020 DD6E00                   LD      L,(IX+0)    ;
F023 DD6601                   LD      H,(IX+1)    ;HL' HAS FIRST VARPTR
F026 C5              SLOOP    PUSH    BC          ;SAVE SKEY LENGTH
F027 D5                       PUSH    DE          ;SAVE SKEY POINTER
F028 E5                       PUSH    HL          ;SAVE CURRENT ARRAY VARPTR
F029 46                       LD      B,(HL)      ;B' HAS ARRAY STRING LEN
F02A D5                       PUSH    DE          ;SAVE SKEY POINTER
F02B 23                       INC     HL          ;
F02C 5E                       LD      E,(HL)      ;
F02D 23                       INC     HL          ;
F02E 56                       LD      D,(HL)      ;DE' POINTS TO ARRAY STRING
F02F EB                       EX      DE,HL       ;HL' POINTS TO ARRAY STRING
F030 D1                       POP     DE          ;DE' POINTS TO SKEY
F031 04              CPLOOP   INC     B           ;TEST ARRAY STRING LENGTH
F032 05                       DEC     B           ;
F033 2006                     JR      NZ,CMP1     ;IF IT'S NONZERO, GOTO CMP1
F035 0C                       INC     C           ;OTHERWISE TEST SKEY LENGTH
```

```
F036 0D                 DEC    C          ;
F037 203D               JR     NZ,SGR     ;IF SKEY LEN NONZERO, JUMP
F039 1831               JR     EQ         ;BOTH LENGTHS ARE ZERO SO JUMP
F03B 0C        CMP1     INC    C          ;ARRAY STR LEN >0, TEST SKEY
F03C 0D                 DEC    C          ;
F03D 280C               JR     Z,SLS      ;ARRAY STR >0, SKEY=0, SO SKEY IS LESS
F03F 1A                 LD     A,(DE)     ;BOTH LENGTHS >0, LOAD FOR COMPARE
F040 BE                 CP     (HL)       ;COMPARE
F041 2006               JR     NZ,NOTEQ   ;END LOOP IF NOT EQUAL
F043 23                 INC    HL         ;POINT TO NEXT BYTE
F044 13                 INC    DE         ;POINT TO NEXT BYTE
F045 05                 DEC    B          ;SUBTRACT FROM LENGTH COUNT
F046 0D                 DEC    C          ;SUBTRACT FROM LENGTH COUNT
F047 18E8               JR     CPLOOP     ;GO REPEAT FOR NEXT PAIR
F049 302B      NOTEQ    JR     NC,SGR     ;SKEY IS GREATER IF NC
F04B 08        SLS      EX     AF,AF'     ;EXCHANGE TO GET COMMAND
F04C F5                 PUSH   AF         ;
F04D 08                 EX     AF,AF'     ;EXCHANGE BACK
F04E F1                 POP    AF         ;AF HAS COMMAND
F04F CB57               BIT    2,A        ;WILL WE ACCEPT A LESS?
F051 202D               JR     NZ,FOUND   ;IF SO, WE'VE FOUND ONE.
F053 D9        CONT     EXX               ;EXCHANGE REGISTERS
F054 79                 LD     A,C        ;
F055 B0                 OR     B          ;ELEMENTS LEFT = 0?
F056 280B               JR     Z,RNF      ;RETURN NOT FOUND IF ZERO
F058 0B                 DEC    BC         ;OTHERWISE, DECREMENT # LEFT
F059 13                 INC    DE         ;INCREMENT # SEARCHED
F05A D9                 EXX               ;EXCHANGE REGISTERS
F05B E1                 POP    HL         ;HL' HAS PRIOR ARRAY VARPTR
F05C 23                 INC    HL         ;ADD 3
F05D 23                 INC    HL         ;CONTINUE...
F05E 23                 INC    HL         ;HL' HAS NEXT ARRAY VARPTR
F05F D1                 POP    DE         ;DE' POINTS TO SKEY DATA
F060 C1                 POP    BC         ;C' HAS SKEY LENGTH
F061 18C3               JR     SLOOP      ;REPEAT THE SEARCH LOOP
F063 0B        RNF      DEC    BC         ;BC HAS -1  (FFFF)
F064 E1        RF       POP    HL         ;RELIEVE STACK
F065 E1                 POP    HL         ;RELIEVE STACK
F066 E1                 POP    HL         ;RELIEVE STACK
F067 C5                 PUSH   BC         ;
F068 E1                 POP    HL         ;HL HAS RETURN VALUE
F069 C39A0A             JP     0A9AH      ;RETURN HL TO BASIC
F06C 08        EQ       EX     AF,AF'     ;EXCHANGE TO CHECK ON COMMAND
F06D F5                 PUSH   AF         ;
F06E 08                 EX     AF,AF'     ;EXCHANGE BACK
F06F F1                 POP    AF         ;AF HAS COMMAND
F070 CB47               BIT    0,A        ;DO WE WANT AN EQUAL?
F072 200C               JR     NZ,FOUND   ;IF SO, WE'VE FOUND ONE.
F074 18DD               JR     CONT       ;OTHERWISE, CONTINUE SEARCH
F076 08        SGR      EX     AF,AF'     ;EXCHANGE TO CHECK ON COMMAND
F077 F5                 PUSH   AF         ;
F078 08                 EX     AF,AF'     ;EXCHANGE BACK
F079 F1                 POP    AF         ;AF HAS COMMAND
F07A CB4F               BIT    1,A        ;WILL WE ACCEPT A GREATER?
F07C 2002               JR     NZ,FOUND   ;IF SO, WE'VE FOUND ONE.
F07E 18D3               JR     CONT       ;OTHERWISE, CONTINUE SEARCH
F080 D9        FOUND    EXX               ;EXCHANGE REGISTERS BACK
F081 D5                 PUSH   DE         ;
F082 C1                 POP    BC         ;BC HAS ELEMENT NUMBER
F083 18DF               JR     RF         ;RETURN TO BASIC
F064                    END               ;
00000 TOTAL ERRORS
```

Speedy String Array Sort

The SORT1 USR routine will sort any singly dimensioned string array into ascending sequence. Typically, it will take less than 15 seconds to sort a 1000 element array. (To do the same job in BASIC, it could take from 15 minutes to hours, depending on the method you use!) The routine is fully relocatable, and it only takes 188 bytes. In sequencing the array elements, only the pointers are swapped. The actual data contained in each string in the array does not move.

To call the SORT1 USR routine, load a 2-element control array with the following parameters:

Element 0 = VARPTR to the string array to be sorted.
Element 1 = Number of elements to sort − 1.

Then call the USR routine. Your argument will be the VARPTR to your control array. For example, to sort element 0 through element 567 of the string array, SA$, using P%(0) and P%(1) as our control array, our commands will be:

```
P%(Ø)=VARPTR(SA$(Ø))
P%(1)=567
J=USRØ(VARPTR(P%(Ø)))
```

There is no argument returned from the SORT1 USR routine, so 'J' in this case is just a dummy variable. You can substitute USR0 with USR1 through USR9 if you wish, but in any case, you will need a DEFUSR command to identify the calling address.

SORT1
String Array Sort
USR Subroutine

M 2 Note # 23

```
Magic Array Format, 94 ELEMENTS

 32717  -69Ø2  -7715  2Ø189  -8958    838   1Ø48  -6695 -15911
    33 -18688  17133 -1336Ø -13512 -15Ø79  -7719  -8743    622
 26333 -18685  17133  -9755  -9775 -1356Ø   2183  2Ø189  -896Ø
   326   8645      1  -9755  -6719 -11815  -6887  1Ø7Ø5  -8935
    94  22237   64Ø1 -1Ø799   6373  -7924   2273   2293 -13327
 1Ø311   6321   6863  17999   9173   9Ø54  -529Ø  -67Ø3   9195
  9Ø54  -785Ø   1284   1568   334Ø  12Ø64   412Ø   334Ø   3112
-1687Ø   1568   4899   3333  -612Ø   7472 -1Ø791  -9787  -7727
 -4681  1Ø322   5Ø54  -9771  -9791      6    782  -7727  -69Ø3
  2539   6373  -7752 -1Ø799   1765   6659  3Ø542   4729   4899
 -2288 -1356Ø   2247 -12776

Poke Format, 188 BYTES

2Ø5 127  1Ø 229 221 225 221  78   2 221  7Ø   3  24   4 217 229
217 193  33   Ø   Ø 183 237  66 2Ø8 2Ø3  56 2Ø3  25 197 217 225
217 221 11Ø   2 221 1Ø2   3 183 237  66 229 217 2Ø9 217   8 2Ø3
135   8 221  78   Ø 221  7Ø   1 197  33   1   Ø 229 217 193 229
217 2Ø9  25 229 2Ø9  41  25 221  94   Ø 221  86   1  25 2Ø9 213

229  24  12 225 225   8 245   8 241 2Ø3  71  4Ø 177  24 2Ø7  26
 79  7Ø 213  35  94  35  86 235 2Ø9 229 235  35  94  35  86 225
  4   5  32   6  12  13  32  47  24  16  12  13  4Ø  12  26 19Ø
 32   6  35  19   5  13  24 232  48  29 217 213 197 217 2Ø9 225
183 237  82  4Ø 19Ø  19 213 217 193 217   6   Ø  14   3 2Ø9 225
  9 229 235   9 229  24 184 225 2Ø9 213 229   6   3  26  78 119
121  18  35  19  16 247   8 2Ø3 199   8  24 2Ø6
```

SORT1

String Array Sort
USR Subroutine

```
                00000 ;SORT1
                00001 ;
F000            00080         ORG     0F000H        ;ORIGIN - RELOCATABLE
F000 CD7F0A     00090         CALL    0A7FH         ;HL POINTS TO CONTROL ARRAY
F003 E5         00100         PUSH    HL            ;PREPARE FOR COPY TO IX
F004 DDE1       00110         POP     IX            ;IX POINTS TO CONTROL ARRAY
F006 DD4E02     00120         LD      C,(IX+2)      ;
F009 DD4603     00130         LD      B,(IX+3)      ;BC HAS # RECORDS
F00C 1804       00140         JR      JMP1          ;
F00E D9         00150 LOOP1   EXX                   ;
F00F E5         00160         PUSH    HL            ;
F010 D9         00170         EXX                   ;
F011 C1         00180         POP     BC            ;BC HAS CURRENT GAP
F012 210000     00190 JMP1    LD      HL,0000H      ;PREPARE FOR TEST IF GAP <=0
F015 B7         00200         OR      A             ;CLEAR CARRY
F016 ED42       00210         SBC     HL,BC         ;SUBTRACT: 0 - GAP
F018 D0         00220         RET     NC            ;BACK TO BASIC IF GAP <=0
F019 CB38       00230         SRL     B             ;DIVIDE GAP BY 2
F01B CB19       00240         RR      C             ;DIVIDE GAP BY 2, CONT.
F01D C5         00250         PUSH    BC            ;
F01E D9         00260         EXX                   ;
F01F E1         00270         POP     HL            ;HL' HAS CURRENT GAP
F020 D9         00280         EXX                   ;
F021 DD6E02     00290         LD      L,(IX+2)      ;
F024 DD6603     00300         LD      H,(IX+3)      ;HL HAS # RECORDS
F027 B7         00310         OR      A             ;CLEAR CARRY
F028 ED42       00320         SBC     HL,BC         ;SUBTRACT: #RECS - GAP
F02A E5         00330         PUSH    HL            ;
F02B D9         00340         EXX                   ;
F02C D1         00350         POP     DE            ;DE' HAS DIFFERENCE
F02D D9         00360         EXX                   ;
F02E 08         00370 LOOP2   EX      AF,AF'        ;
F02F CB87       00380         RES     0,A           ;
F031 08         00390         EX      AF,AF'        ;FLAG BIT = 0
F032 DD4E00     00400         LD      C,(IX+0)      ;
F035 DD4601     00410         LD      B,(IX+1)      ;BC POINTS TO FIRST RECORD
F038 C5         00420         PUSH    BC            ;SAVE IT ON STACK
F039 210100     00430         LD      HL,0001H      ;
F03C E5         00440         PUSH    HL            ;
F03D D9         00450         EXX                   ;
F03E C1         00460         POP     BC            ;BC' HAS LOWER COMPARE REC#
F03F E5         00470         PUSH    HL            ;
F040 D9         00480         EXX                   ;
F041 D1         00490         POP     DE            ;DE HAS CURRENT GAP
F042 19         00500         ADD     HL,DE         ;COMPUTE UPPER REC# FOR COMPARE
F043 E5         00510         PUSH    HL            ;
F044 D1         00520         POP     DE            ;
F045 29         00530         ADD     HL,HL         ;
F046 19         00540         ADD     HL,DE         ;UPPER RECORD# MULTIPLIED BY 3
F047 DD5E00     00550         LD      E,(IX+0)      ;HL HAS # BYTES FROM BASE TO UPPER REC
F04A DD5601     00560         LD      D,(IX+1)      ;DE HAS MEMORY BASE
F04D 19         00570         ADD     HL,DE         ;HL POINTS TO UPPER RECORD
F04E D1         00580         POP     DE            ;DE HAS LOWER REC POINTER
F04F D5         00590         PUSH    DE            ;SAVE LOWER REC POINTER ON STACK
F050 E5         00600         PUSH    HL            ;SAVE UPPER REC POINTER ON STACK
F051 180C       00610         JR      LOOP3         ;
F053 E1         00620 JMP2    POP     HL            ;RELIEVE STACK
F054 E1         00630         POP     HL            ;RELIEVE STACK
F055 08         00640         EX      AF,AF'        ;
F056 F5         00650         PUSH    AF            ;
F057 08         00660         EX      AF,AF'        ;
F058 F1         00670         POP     AF            ;
F059 CB47       00680         BIT     0,A           ;ANY SWAPS MADE?
F05B 28B1       00690         JR      Z,LOOP1       ;IF NO SWAPS, LOOP1
```

```
FØ5D 18CF      ØØ7ØØ          JR      LOOP2        ;OTHERWISE, LOOP2
FØ5F 1A        ØØ71Ø  LOOP3   LD      A,(DE)       ;
FØ6Ø 4F        ØØ72Ø          LD      C,A          ;C HAS LOWER REC LENGTH
FØ61 46        ØØ73Ø          LD      B,(HL)       ;B HAS UPPER REC LENGTH
FØ62 D5        ØØ74Ø          PUSH    DE           ;SAVE LOWER REC VARPTR
FØ63 23        ØØ75Ø          INC     HL           ;
FØ64 5E        ØØ76Ø          LD      E,(HL)       ;
FØ65 23        ØØ77Ø          INC     HL           ;
FØ66 56        ØØ78Ø          LD      D,(HL)       ;DE POINTS TO UPPER REC
FØ67 EB        ØØ79Ø          EX      DE,HL        ;HL POINTS TO UPPER REC
FØ68 D1        ØØ8ØØ          POP     DE           ;DE HAS LOWER REC VARPTR
FØ69 E5        ØØ81Ø          PUSH    HL           ;SAVE POINTER TO UPPER REC
FØ6A EB        ØØ82Ø          EX      DE,HL        ;HL HAS LOWER REC VARPTR
FØ6B 23        ØØ83Ø          INC     HL           ;
FØ6C 5E        ØØ84Ø          LD      E,(HL)       ;
FØ6D 23        ØØ85Ø          INC     HL           ;
FØ6E 56        ØØ86Ø          LD      D,(HL)       ;DE POINTS TO LOWER REC
FØ6F E1        ØØ87Ø          POP     HL           ;HL POINTS TO UPPER REC
FØ7Ø Ø4        ØØ88Ø  CPLOOP  INC     B            ;TEST UPPER REC LENGTH
FØ71 Ø5        ØØ89Ø          DEC     B            ;
FØ72 2ØØ6      ØØ9ØØ          JR      NZ,CMP1      ;IF IT'S NONZERO, GOTO CMP1
FØ74 ØC        ØØ91Ø          INC     C            ;OTHERWISE, TEST LOWER REC LENGTH
FØ75 ØD        ØØ92Ø          DEC     C            ;
FØ76 2Ø2F      ØØ93Ø          JR      NZ,SWAP      ;IF LOWER=NONZERO, UPPER=Ø, SWAP
FØ78 181Ø      ØØ94Ø          JR      NOSWAP       ;BOTH ARE ZERO, SO NO SWAP
FØ7A ØC        ØØ95Ø  CMP1    INC     C            ;UPPER LEN IS NON ZERO, TEST LOWER
FØ7B ØD        ØØ96Ø          DEC     C            ;
FØ7C 28ØC      ØØ97Ø          JR      Z,NOSWAP     ;LOWER=Ø, UPPER=NONZERO, NO SWAP
FØ7E 1A        ØØ98Ø          LD      A,(DE)       ;BOTH NONZERO. LOAD BYTE FOR COMPARE
FØ7F BE        ØØ99Ø          CP      (HL)         ;COMPARE
FØ8Ø 2ØØ6      Ø1ØØØ          JR      NZ,NOTEQ     ;IF NOT EQUAL WE CAN END LOOP
FØ82 23        Ø1Ø1Ø          INC     HL           ;POINT TO NEXT IN UPPER REC
FØ83 13        Ø1Ø2Ø          INC     DE           ;POINT TO NEXT IN LOWER REC
FØ84 Ø5        Ø1Ø3Ø          DEC     B            ;SUBTRACT FROM LENGTH COUNT
FØ85 ØD        Ø1Ø4Ø          DEC     C            ;SUBTRACT FROM LENGTH COUNT
FØ86 18E8      Ø1Ø5Ø          JR      CPLOOP       ;GO REPEAT FOR NEXT 2 BYTES
FØ88 3Ø1D      Ø1Ø6Ø  NOTEQ   JR      NC,SWAP      ;LOWER IS GREATER IF NC, SO SWAP
FØ8A D9        Ø1Ø7Ø  NOSWAP  EXX                  ;
FØ8B D5        Ø1Ø8Ø          PUSH    DE           ;
FØ8C C5        Ø1Ø9Ø          PUSH    BC           ;
FØ8D D9        Ø11ØØ          EXX                  ;
FØ8E D1        Ø111Ø          POP     DE           ;DE HAS LOWER COMPARE REC #
FØ8F E1        Ø112Ø          POP     HL           ;HL HAS UPPER COMPARE BASE #
FØ9Ø B7        Ø113Ø          OR      A            ;CLEAR CARRY
FØ91 ED52      Ø114Ø          SBC     HL,DE        ;TEST IF EQUAL
FØ93 28BE      Ø115Ø          JR      Z,JMP2       ;MORE TO GO IF NOT EQUAL
FØ95 13        Ø116Ø          INC     DE           ;ADD 1 TO LOWER COMPARE REC#
FØ96 D5        Ø117Ø          PUSH    DE           ;
FØ97 D9        Ø118Ø          EXX                  ;
FØ98 C1        Ø119Ø          POP     BC           ;SAVE IT IN BC'
FØ99 D9        Ø12ØØ          EXX                  ;
FØ9A Ø6ØØ      Ø121Ø          LD      B,Ø          ;
FØ9C ØEØ3      Ø122Ø          LD      C,3          ;BC HAS RECORD LENGTH
FØ9E D1        Ø123Ø          POP     DE           ;GET UPPER REC POINTER
FØ9F E1        Ø124Ø          POP     HL           ;GET LOWER REC POINTER
FØAØ Ø9        Ø125Ø          ADD     HL,BC        ;POINT TO NEXT LOWER REC
FØA1 E5        Ø126Ø          PUSH    HL           ;PUT IT ON STACK
FØA2 EB        Ø127Ø          EX      DE,HL        ;
FØA3 Ø9        Ø128Ø          ADD     HL,BC        ;POINT TO NEXT UPPER REC
FØA4 E5        Ø129Ø          PUSH    HL           ;PUT IT ON STACK
FØA5 18B8      Ø13ØØ          JR      LOOP3        ;REPEAT
FØA7 E1        Ø131Ø  SWAP    POP     HL           ;GET POINTER TO UPPER REC
FØA8 D1        Ø132Ø          POP     DE           ;GET POINTER TO LOWER REC
FØA9 D5        Ø133Ø          PUSH    DE           ;SAVE AGAIN ON STACK
FØAA E5        Ø134Ø          PUSH    HL           ;SAVE AGAIN ON STACK
```

```
F0AB 0603      01350         LD      B,3        ;3 BYTES TO EXCHANGE
F0AD 1A        01360 SWLOOP  LD      A,(DE)     ;SWAP THE STRING POINTERS
F0AE 4E        01370         LD      C,(HL)     ;CONTINUE...
F0AF 77        01380         LD      (HL),A     ;CONTINUE...
F0B0 79        01390         LD      A,C        ;CONTINUE...
F0B1 12        01400         LD      (DE),A     ;CONTINUE...
F0B2 23        01410         INC     HL         ;CONTINUE...
F0B3 13        01420         INC     DE         ;CONTINUE...
F0B4 10F7      01430         DJNZ    SWLOOP     ;REPEAT IF LESS THAN 3 BYTES SWAPPED
F0B6 08        01440         EX      AF,AF'     ;
F0B7 CBC7      01450         SET     0,A        ;SET SWAP FLAG
F0B9 08        01460         EX      AF,AF'     ;
F0BA 18CE      01470         JR      NOSWAP     ;
F08A           01480         END                ;
00000 TOTAL ERRORS
```

The logic used in this sort is based on the Shell sort algorithm. Array elements are compared in pairs across a 'gap' which initally spans half the size of the array. When the lower element of a pair is greater than the upper element of the pair, the pointers for the two elements are swapped. Then the next 2 elements are compared. If at least one swap was made during the comparison of each set of pairs, the process of comparisons and swaps across the gap is repeated. If no swaps have been made, the gap is divided by 2 and the comparison and swap phase is repeated. When the gap is finally less than or equal to 1, the sort is complete.

Making Numeric Data Sortable

The need to sort numbers presents a special problem. Integers, for example, are stored in 2 bytes, the least significant byte, 'LSB', preceding the most significant byte, 'MSB'. Negative integers, in 2-byte mode, are greater than positive integers. To illustrate the problem, here are the hex values of some integers, as they are normally stored, in LSB-MSB format:

```
-1 = FFFF, 1 = 0100, 17 = 1100, 4097 = 0110, 32512 = 007F
```

As you can see, an attempt to sort these while in 2-byte format will give useless results. Here are two function calls that you can use to convert integers into 'sortable integers'. The first, FNIX$(A%), converts an integer to a 2-byte string. It is analogous to the MKI$ function, except that the resulting 2 bytes are sortable. The second, FNIX%(A$), converts a sortable 2-byte integer string, back to an integer. The valid range is from −32767 to 32767.

Sortable Integer Functions

```
Convert A% to a 2-byte sortable string:
40 DEFFNIX$(A%)=RIGHT$(MKI$(-SGN(A%)*(32768-ABS(A%))),1)+LEFT$(M
KI$(-SGN(A%)*(32768-ABS(A%))),1)

Convert a 2-byte sortable string, A$, back to an integer:
41 DEFFNIX%(A$)=(32768-ABS(CVI(RIGHT$(A$,1)+LEFT$(A$,1))))*-SGN(
CVI(RIGHT$(A$,1)+LEFT$(A$,1)))
```

Now, to sort an integer array, we can convert each integer to a sortable string with the FNIX$ function, load it into a string array, sort the string array and then load the results back into the integer array using the FNIX% function to convert back. For example, to sort the 200 element integer array, IA%, we can load it into a string array, SA$, using:

```
FORX=ØTO199
SA$(X)=FNIX$(IA%(X))
NEXT
```

We then use the SORT1 USR routine to sort the string array. Finally we reload the integer array:

```
FORX=ØTO199
IA%(X)=FNIX%(SA$(X))
NEXT
```

Or, we can convert each element in the integer array to the corresponding integer in sortable format and then sort the integer array with the SORT2 USR routine we shall be discussing. Now we can convert back. Let's say we have a 200 element array, IA%. To convert it to a sortable integer array we can use the following logic:

```
FORX=ØTO199
IA%(X)=CVI(FNIX$(IA%(X)))
NEXT
```

Now we have an array we can sort with the SORT2 routine. After the sort, we can convert back with:

```
FORX=ØTO199
IA%(X)=FNIX%(MKI$(IA%(X)))
NEXT
```

Single precision and double precision numbers present even bigger problems in sorting. The best method is to convert them into strings in ASCII format. The FNSA$ function call does this for you.

Sortable Numeric ASCII String Function

```
42 DEFFNSA$(A1#,A2#,A3%)=MID$("-Ø",(A1#<Ø)+2,1)+RIGHT$(STRING$(A
3%,"Ø")+MID$(STR$(INT(A2#*A1#)),2),A3%)
```

FNSA$(A1#,A2#,A3%) converts a single or double precision number to a sortable ASCII string, where:

Argument 1 is the number to be converted.

Argument 2 is a multiplier, such as 1, 10 or 100, to indicate how many places to the right of the decimal are to be allowed for. (1 indicates none, 100 indicates 2, etc.)

Argument 3 indicates the number of significant digits to allow in the string to be created. For example, if you are going to deal with numbers up to 9999.99, argument 3 would be 6. The length of the string created will be the number you specify as argument 3, plus 1 byte for the sign.

Here are some examples:

```
If D# =   23.45, FNSA$(D#,100,6) = "0002345"
If D# =  -23.45, FNSA$(D#,100,6) = "-002345"
If D# =     100, FNSA$(D#,100,6) = "0010000"
If D# =     100, FNSA$(D#,1,6)   = "0000100"
```

Notice that we've taken out the decimal by multiplying each number by 100. Then we right-justified the number and filled in zeros to the left of the most significant digit. In the first position, we used '0' if the number is positive or '–' if the number is negative, because in ASCII collating sequence, '0' is greater than '–', (but '+' isn't.) After sorting these numbers as strings, we can then convert back to single precision if necessary, by taking the VAL function of each and dividing by the number we used as argument 2.

This method is sufficient for most purposes. But be aware that negative numbers will sort in descending sequence. An array sorted in ascending sequence will yield:

Negative numbers in descending sequence
– Zero –
Positive Numbers in ascending sequence

In accounting applications, where credit balances may be stored as negatives, this is fine. In applications where you need negatives sorted in ascending sequence, you'll need to do some other manipulations.

Sorting With Assorted Keys

Let's suppose that you have data for several retail stores. Working at each store you have several salesmen and your computer program has accumulated total sales for each salesman:

```
STORE LOCATION      SALESMAN      SALES
----------------    -----------   -----------
CHINO                 JR           532.40
AZUSA                 DJ           221.28
UPLAND                MS           223.32
UPLAND                JJ           332.22
ONTARIO               SA            52.48
ONTARIO               BW           299.40
```

To sort the data in alphabetical order by store and within each store, in alphabetical order by salesman initials, you simply add each of the strings together before sorting, making sure that the fields line up. This way you can create a single array to be sorted. Here's what the array would contain before the sort:

```
CHINO  JR053240
AZUSA  DJ022128
UPLAND MS022332
UPLAND JJ033222
ONTARIOSA005248
ONTARIOBW029940
```

After the data is sorted in ascending sequence, you can split out the fields with the MID$ function and here's what you get:

```
AZUSA          DJ        221.28
CHINO          JR        532.40
ONTARIO        BW        299.40
ONTARIO        SA         52.48
UPLAND         JJ        332.22
UPLAND         MS        223.32
```

Now suppose you want to sort so that the salesman with the lowest sales total is shown at the top of the list and if more than 1 salesman has the same sales figure, they will be listed alphabetically. To do this, you just arrange the strings to be sorted so that the sales figures come first:

```
053240JRCHINO
022128DJAZUSA
022332MSUPLAND
033222JJUPLAND
05248SAONTARIO
029940BWONTARIO
```

After the data is sorted in ascending sequence and you've separated it with the MID$ function, here's what you get:

```
 52.48         SA        ONTARIO
221.28         DJ        AZUSA
223.32         MS        UPLAND
299.40         BW        ONTARIO
332.22         JJ        UPLAND
532.40         JR        CHINO
```

Now, let's suppose you want the salesman with the highest sales total to be shown at the top of the list. In other words, you want the list sorted in descending sequence by sales total, ascending sequence by salesman and ascending sequence by store location. One method that you can use is to sort in ascending sequence, as we did above and then print the data from our sorted array or disk file by starting at the last element, working up toward the first. With this method, one sort lets us handle two possible sequences for printing the file. The only problem is that, when we read the file or array in reverse, the salesman initials and store locations will also be in descending sequence, in the event more than one salesman has the same total.

A better solution that provides for the possibility of any combination of ascending and descending sort keys is to 'complement' those strings that we want to be sorted in descending sequence.

When we complement a string, we simply subtract the code for each byte in the string from 255. Thus, a CHR$(0) within the string becomes a CHR$(255). A CHR$(255) becomes a CHR$(0). A CHR$(1) becomes a CHR$(254). The complement of 'AAA' is greater than the complement of 'BBB'.

In our example, we would want to complement the sales amount strings before concatenating them with the salesman and store location strings. Then we do the sort. After the sort, we separate the strings and we complement the sales amount strings again to restore them to their original contents.

To complement a string in BASIC could be quite slow. Here's a 19-byte USR routine that complements any string instantly. To use it, you simply load it into protected memory or a magic array and do a DEFUSR. Then, whenever you want to complement a string, you call the USR routine, with your argument being the string's VARPTR.

Suppose that we've loaded the STRCOMPL USR routine at location **FF00** in protected memory. Our logic to sort the 100-element SA$ array in descending sequence is:

```
110 DEFUSR=&HFF00                       'DEFINE USR ROUTINE ADDRESS
120 FOR X = 1 TO 100                    'FOR EACH ELEMENT OF THE STRING ARRAY
130 J=USR(VARPTR(SA$(X)))               'COMPLEMENT IT
140 NEXT                                'REPEAT

150 'Call a subroutine that sorts in sequence here...

160 FOR X = 1 TO 100                    'FOR EACH ELEMENT OF THE STRING ARRAY
170 J=USR(VARPTR(SA$(X)))               'COMPLEMENT IT AGAIN TO RESTORE
180 NEXT                                'REPEAT

190 FOR X = 1 TO 100                    'PRINT EACH ELEMENT OF THE ARRAY
200 PRINT SA$(X)                        'IT'S IN DESCENDING SEQUENCE!
210 NEXT                                'REPEAT
```

STRCOMPL
String Complement USR Subroutine

```
Magic Array Format, 10 Elements:

32717  17930  24099  22051   1259 -14331  12158   9079  -1520
  201

Poke Format, 19 Bytes:

 205 127  10  70  35  94  35  86 235   4   5 200 126  47 119  35
  16 250 201   0
```

```
                  00000 ;STRCOMPL
                  00001 ;
FF00              00060          ORG     0FF00H         ;ORIGIN - RELOCATABLE
FF00 CD7F0A       00070          CALL    0A7FH          ;HL HAS STRING VARPTR
FF03 46           00080          LD      B,(HL)         ;B HAS STRING LENGTH
FF04 23           00090          INC     HL             ;
FF05 5E           00100          LD      E,(HL)         ;
FF06 23           00110          INC     HL             ;
FF07 56           00120          LD      D,(HL)         ;DE POINTS TO STRING
FF08 EB           00130          EX      DE,HL          ;HL POINTS TO STRING
FF09 04           00140          INC     B              ;
FF0A 05           00150          DEC     B              ;INC & DEC B TO TEST IF ZERO
FF0B C8           00160          RET     Z              ;RETURN IF ZERO LENGTH
FF0C 7E           00170 LOOP     LD      A,(HL)         ;PUT BYTE IN ACCUM
FF0D 2F           00180          CPL                    ;COMPLEMENT IT
FF0E 77           00190          LD      (HL),A         ;PUT IT BACK
FF0F 23           00200          INC     HL             ;POINT TO NEXT BYTE
FF10 10FA         00210          DJNZ    LOOP           ;DECREMENT COUNT & REPEAT
FF12 C9           00220          RET                    ;RETURN TO BASIC
FF0C              00230          END                    ;
00000 TOTAL ERRORS
```

More – Arrays, Searches & Sorts

'Pointing' a String Array

Have you ever tried to load a large amount of data into a string array, finding that after a certain point, your computer freezes up for a few minutes to reorganize the string data you've fed it before it will take any more? Or, have you had problems in knowing how much memory to reserve for string storage with the CLEAR command? Do you risk 'out of string space' errors because you don't know the total length of the string data that will be entered by the operator? Do you sometimes need to pass string data from one program to another?

The ARPOINT USR routine gives you a method to handle all of these problems. The string reorganization problem is a side-effect of BASIC's dynamic string allocation feature. With ARPOINT, we can bypass the dynamic allocation, and pre-allocate an array of uniform length strings. Since your array is pre-allocated, you'll know exactly how much information the operator will be able to enter, so there's no guesswork with CLEAR statements, and you can prevent 'out of string space' errors. With ARPOINT, we specify a starting memory location in protected memory for the data to be stored in the array. This lets us pass the contents of a string array from one program to another.

Here are the steps required to call ARPOINT:

1. Load the ARPOINT routine and do a DEFUSR that points to the routine's address.

2. Dimension the string array that you will want to 'point'.

3. Load a 3-element control array with the following arguments:

   Element 0 = VARPTR to the string array.
   Element 1 = Memory location at which array data will start.
   Element 2 = Uniform length of each element in the array, 1 to 255 bytes.

4. Call the ARPOINT USR routine, with your argument being the VARPTR to the control array.

5. To put data into any array element, use LSET or RSET. This prevents the computer from changing the address or length of the element.

Let's assume, for example, you've got a 48K TRS-80 and you need a 500 element array, AA$, each element being 20 bytes long. The string data will take 10,000 bytes, so you decide to store it at memory address **D8F0**. (**D8F0** equals −10000 in decimal integer format.)

Upon loading BASIC, you specify a memory size of 55536 or less to protect the memory for your array. (Or you can change the memory size while in BASIC.)

Now, in your program, you dimension the string array, and load your control array:

```
DIM AA$(499)
P%(Ø)=VARPTR(AA$(Ø))
P%(1)=&HD8FØ
P%(2)=2Ø
```

Next, assuming the ARPOINT routine has been loaded and DEFUSRed as USR routine 0, you call it, using a dummy integer variable, such as 'J':

```
J=USRØ(VARPTR(P%(Ø)))
```

To load the string, A$, into array element 5, you can say, LSETAA$(5)=A$. To load the string, 'ABCDEF' into array element 400, you can say, LSET(AA$(400))='ABCDEF'.

To pass the contents of the AA$ array to another program, you can simply:

1. Load the other program.
2. Dimension the string array again, as you did in the first program.
3. Call the ARPOINT routine again, with control array elements 1 and 2 being the same as they were in the first program. You've passed the data!

Within a program, you can point as many string arrays as you wish by changing the control array and executing ARPOINT again. You can also repoint an array or change the length of the elements. You may, in certain applications, want to point a 16 element array to the video display with each element being 64 characters. That way, each string in the array will point to a line on the screen, and the contents of that string will be the current contents of the display line. Here's how to do it:

```
DIM VD$(15)             'DIMENSION VIDEO DISPLAY STRING ARRAY
P%(Ø)=VARPTR(VD$)       'CONTROL Ø IS VARPTR TO STRING ARRAY
P%(1)=1536Ø             'ARRAY ADDRESS WILL EQUAL VIDEO ADDRESS
P%(2)=64                'EACH ELEMENT OF THE ARRAY IS 64 BYTES
J=USRØ(VARPTR(P%(Ø)))   'CALL ARPOINT USR ROUTINE
```

Now we can LSET or RSET to the display. For example, to right-justify and print the word 'TEST' on the 3rd line, we can RSET VD$(2)='TEST'. To LPRINT the top 3 lines of the display, we can say,

M 2 Note # 38

```
FORX=ØTO2 : LPRINT VD$(X) : NEXT
```

You'll find the ARPOINT routine especially useful when you want to load a large amount of data from disk to memory for a sort. You can use the SORT1 routine, which sorts a BASIC string array. Or, if you wish, you can use the SORT2 routine, which sorts uniform length records within a contiguous block of memory.

ARPOINT
String Array Pointer USR Subroutine

M 2 Note # 23

Magic Array Format, 21 elements

```
 32717  24074  22051  -6877  -6677  17963  20011  -7719  24291
 22051   1571  19968  29153  29475  29219  -5341  -5367   3033
-20359  -9784  -4328
```

Poke Format, 42 bytes

```
205 127  10  94  35  86  35 229 235 229  43  70  43  78 217 225
227  94  35  86  35   6   0  78 225 113  35 115  35 114  35 235
  9 235 217  11 121 176 200 217  24 239
```

```
                00000 ;ARPOINT
                00001 ;
F000            00090         ORG    0F000H        ;ORIGIN - RELOCATABLE
F000 CD7F0A     00100         CALL   0A7FH         ;HL POINTS TO CONTROL 0
F003 5E         00110         LD     E,(HL)        ;
F004 23         00120         INC    HL            ;
F005 56         00130         LD     D,(HL)        ;DE POINTS TO STRING ARRAY
F006 23         00140         INC    HL            ;HL POINTS TO CONTROL 1
F007 E5         00150         PUSH   HL            ;SAVE ON STACK
F008 EB         00160         EX     DE,HL         ;HL POINTS TO STRING ARRAY
F009 E5         00170         PUSH   HL            ;SAVE WHILE GETTING DIM
F00A 2B         00180         DEC    HL            ;
F00B 46         00190         LD     B,(HL)        ;
F00C 2B         00200         DEC    HL            ;
F00D 4E         00210         LD     C,(HL)        ;BC HAS DIMENSION +1
F00E D9         00220         EXX                  ;EXCHANGE REGISTERS
F00F E1         00230         POP    HL            ;HL' POINTS TO STRING ARRAY
F010 E3         00240         EX     (SP),HL       ;HL' POINTS TO CONTROL ARRAY
F011 5E         00250         LD     E,(HL)        ;
F012 23         00260         INC    HL            ;
F013 56         00270         LD     D,(HL)        ;DE' HAS STARTING LOCATION
F014 23         00280         INC    HL            ;
F015 0600       00290         LD     B,0           ;
F017 4E         00300         LD     C,(HL)        ;BC' HAS ELEMENT LENGTH
F018 E1         00310         POP    HL            ;HL' POINTS TO FIRST ELEMENT
F019 71         00320 NXTELE  LD     (HL),C        ;LOAD THE LENGTH
F01A 23         00330         INC    HL            ;
F01B 73         00340         LD     (HL),E        ;LOAD LSB OF ADDRESS
F01C 23         00350         INC    HL            ;
F01D 72         00360         LD     (HL),D        ;LOAD MSB OF ADDRESS
F01E 23         00370         INC    HL            ;HL' POINTS TO NEXT
F01F EB         00380         EX     DE,HL         ;
F020 09         00390         ADD    HL,BC         ;COMPUTE NEXT ADDRESS
F021 EB         00400         EX     DE,HL         ;DE HAS NEXT ADDRESS
F022 D9         00410         EXX                  ;EXCHANGE REGISTERS
F023 0B         00420         DEC    BC            ;DECREMENT COUNT
F024 79         00430         LD     A,C           ;
F025 B0         00440         OR     B             ;SET Z FLAG IF COUNT IS 0
F026 C8         00450         RET    Z             ;BACK TO BASIC IF DONE
F027 D9         00460         EXX                  ;OTHERWISE, EXCHANGE
F028 18EF       00470         JR     NXTELE        ;REPEAT FOR NEXT ELEMENT
F019            00480         END                  ;
00000 TOTAL ERRORS
```

Saving Thousands of Bytes for Large Arrays

A string array of 1000 elements requires more than 3000 bytes of overhead. This overhead is the space allocated by BASIC to keep track of the length and address of each string in the array. If we decide on a uniform length for each element in a string array and a block of protected memory in which to store the elements, we can save all that overhead. But equally important in many applications, we can significantly improve program execution speed because BASIC will not have to manage the array.

The KWKARRAY ('quick array') USR routine lets you create one or more arrays in protected memory, composed of uniform length strings. You have 3 commands that let you put data into the array, and retrieve data from it:

Command 0 moves the the data from any element in the quick array to a regular BASIC string.

Command 1 moves a BASIC string to the top-most element of a quick array and adds 1 to the count of active elements.

Command 2 lets you move a BASIC string into any element of a quick array.

Your BASIC program communicates with the KWKARRAY routine using a 6-element control array:

Element 0 specifies the element number within your quick array that you want to 'get' (with command 0) or 'put' (with command 2). The first element in a quick array is 1.

Element 1 specifies your command:

0 = get a string from a specific element of the array.
1 = move a string to the top of the array.
2 = put a string into a specific element of the array.

Element 2 specifies the starting address of your quick array in memory.

Element 3 specifies the next address at the top of the quick array. When you start out with an empty array, control element 3 equals control element 2. Each time you put a string onto the top of the array with command 1, the length of that string is added to control 3.

Element 4 specifies the number of active elements in the array. You preset it to zero. Then each time you put a string onto the top of the array with command 1, element 4 is incremented.

Element 5 is the VARPTR to the string that you've selected for the purpose of passing data to and from the quick array. The length of this string determines the length of each element in the array, so you should create this string with your desired element length, then LSET into this string before using commands to put data into the quick array.

Here's an example of how you might use the quick array in a programming application. Suppose we want to set up an array that maintains the prices and descriptions of 1000 products. Each single precision price will be stored in 4-bytes, and each description will be stored in 12 bytes. Since each product will require 16 bytes, we need to protect at least 16000 bytes of memory. We can do this with our response to the MEMORY SIZE question, or we can change the memory size while in BASIC. Let's assume that we are using a 48K TRS-80 and we want to use the top 16000 bytes of memory for our quick array. Therefore, its starting address will be C180

We load the 133-byte KWKARRAY USR routine into memory with any of the available procedures for loading USR routines. We then do a DEFUSR to point one of the USR addresses (USR0 through USR9) to our KWKARRAY routine. For the remainder of this example, let's assume that we've pointed USR5 to the KWKARRAY routine. Now, before using the KWKARRAY routine, we must set up our 6-element control array and initialize the BASIC string we'll use to pass data. Let's use ST$ as our data-passing string. To initialize it, we use the command:

```
ST$=STRING$(16," ")
```

Let's use C%(0) through C%(5) for our control array. We can initialize our control array with the following commands:

```
C%(2) = &HC180          'LOAD QUICK-ARRAY START ADDRESS
C%(3) = &HC180          'LOAD NEXT ADDRESS, TOP OF ARRAY
C%(4) = 0               'NUMBER OF ACTIVE ELEMENTS = 0
C%(5) = VARPTR(ST$)     'ST$ WILL BE USED TO PASS STRINGS
```

Now, to load a price stored in PR! and a description, stored in DE$, to the next element in the quick array, we can use this subroutine:

```
LSET ST$=MKS$(PR!)+DE$  'PUT DATA INTO THE STRING
C%(1) = 1               'COMMAND IS 1, MOVE-TO-TOP
J=USR5(VARPTR(C%(0)))   'CALL THE KWKARRAY USR ROUTINE
RETURN                  '
```

At this point, J contains the new count of elements in the quick array. C%(4) also contains the new count, and C%(3) has been incremented by the length of the string we passed, 16. We should test J to see that we have not reached our limit, 1000 elements, using something like:

```
IF J>999 THEN PRINT "ARRAY IS FULL" : GOTO 1090
```

The quick array USR routine doesn't check on a limit for the number of entries, so your BASIC program should prevent adding too many elements.

When we want to recall the contents of any element that we have added to the quick array we can put the desired element number in control 0 and use a command 0. The following logic puts the contents of array element 29 into the string ST$:

```
C%(0)=29              'DESIRED ELEMENT NUMBER
C%(1)=0               'COMMAND IS MOVE-TO-STRING
J=USR5(VARPTR(C%(0))) 'CALL KWKARRAY ROUTINE
```

Now to get the price and description:

```
PR!=CVS(ST$)            'GET PRICE FROM STRING
DE$=MID$(ST$,5)         'GET DESCRIPTION FROM STRING
```

To sequentially retrieve the contents of each element in the array we can use a FOR-NEXT loop:

```
FOR X = 1 TO C%(4)       'FROM FIRST ELEMENT TO LAST ACTIVE
C%(0) = X                'LOAD DESIRED ELEMENT NUMBER
C%(1) = 0                'LOAD COMMAND
J=USR5(VARPTR(C%(0)))    'CALL THE USR ROUTINE
GOSUB 5000               'INSERT LOGIC HERE TO USE THE
                         'DATA THAT HAS BEEN RETRIEVED INTO ST$
NEXT                     'REPEAT
```

We can update or replace the data stored in any element of our quick array with command 2. A call to the KWKARRAY routine with command 2 alters control 4, (the count of active elements). If we've extended the array, it alters control 3, the pointer to the next address at the top of the array.

To load a price, PR! and a description, DE$, into element 40, we can use:

```
C%(0) = 40               'SPECIFY DESIRED ELEMENT NUMBER
C%(1) = 2                'SET COMMAND MODE - "MOVE-TO"
LSET ST$=MKS$(PR!)+DE$   'LOAD DATA TO BE PASSED
J=USR5(VARPTR(C%(0)))    'PASS THE DATA
```

The KWKARRAY routine is relocatable, and it is designed to be modular. If all 3 commands are required, it is 134 bytes long. If you just need commands 0 and 1, the routine is designed so that only the first 98 bytes are required. For applications that are simply loading data into a quick array with command 1, only the first 56 bytes are required. When executing a command 1, our USR routine avoids a multiplication to be especially fast. Here is the information you'll need to implement the KWKARRAY routines, if you don't already have them on disk:

KWKARRAY
Quick Array USR Subroutine
M 2 Note # 23
M 2 Note # 37

```
Magic Array Format, 67 elements

 32717  -6902  -7715  28381  -8950   2918   1614   8960   9054
 -8874    715  10310  -5345  24285  -8954   1878 -20243  29661
 -8954   1906  28381  -8952   2406  -8925   2165  29917 -15607
  2714 -14891  24285  -8960    342   8475      0  14795    304
 10265  -5371  -5335  -3048  24285  -8956   1366 -16103  -8751
   715   8270  -4861 -13904  -4629  -8784   1646  26333 -18681
 21229   2104  28381  -8952   2406 -17128  29661  -8954   1906
 28381  -8960    358 -22248

Poke Format, 134 bytes

205 127  10 229 221 225 221 110  10 221 102  11  78   6   0  35
 94  35  86 221 203   2  70  40  31 235 221  94   6 221  86   7
237 176 221 115   6 221 114   7 221 110   8 221 102   9  35 221
117   8 221 116   9 195 154  10 213 197 221  94   0 221  86   1
 27  33   0   0 203  57  48   1  25  40   5 235  41 235  24 244
221  94   4 221  86   5  25 193 209 221 203   2  78  32   3 237
176 201 235 237 176 221 110   6 221 102   7 183 237  82  56   8
221 110   8 221 102   9  24 189 221 115   6 221 114   7 221 110
  0 221 102   1  24 169
```

The following program demonstrates how the KWKARRAY USR subroutine works. For the demo, we will use the top portion of our video display as an array of 88 strings, each being 8 bytes long. You can use commands 0, 1 or 2 to pass strings to and from the array:

KWKARRAY/DEM
Quick Array Demonstration Program
M 2 Note # 21
M 2 Note # 23
M 2 Note # 37
M 2 Note # 39

```
1 CLEAR1000:DEFINTA-Z:J=0

10 'LOAD THE KWKARRAY ROUTINE INTO A MAGIC ARRAY
11 DATA 32717,-6902,-7715, 28381,-8950, 2918, 1614, 8960, 9054,-
8874, 715, 10310,-5345, 24285,-8954, 1878
12 DATA-20243, 29661,-8954, 1906, 28381,-8952, 2406,-8925, 2165,
 29917,-15607, 2714,-14891, 24285,-8960, 342
13 DATA 8475, 0, 14795, 304, 10265,-5371,-5335,-3048, 24285,-895
6, 1366,-16103,-8751, 715, 8270,-4861
14 DATA-13904,-4629,-8784, 1646, 26333,-18681, 21229, 2104,
 28381,-8952, 2406,-17128, 29661,-8954, 1906, 28381
15 DATA-8960, 358,-22248
16 DIMUS%(66):FORX=0TO66:READUS%(X):NEXT

100 'INITIALIZE SCREEN AS A QUICK-ARRAY WITH 8-BYTE ELEMENTS
101 ST$=STRING$(8," "):C%(2)=15360:C%(3)=C%(2):C%(4)=0:C%(5)=VAR
PTR(ST$)
110 CLS

200 PRINT@768,CHR$(31);"ACTIVE ELEMENTS =";C%(4);"   NEXT ADDRES
S =";C%(3)
201 IFC%(1)<0ORC%(1)>2THEN200

210 PRINT@832,CHR$(31);:INPUT"COMMAND";C%(1)
211 IFC%(1)<0ORC%(1)>2THEN210
212 IFC%(1)=1THEN230

220 PRINT@864,CHR$(31);:INPUT"ELEMENT";C%(0)
221 IFC%(0)<1ORC%(0)>89THEN220
222 IFC%(1)=0THEN250

230 PRINT@896,CHR$(31);:INPUT"STRING";A$
240 LSETST$=A$:DEFUSR=VARPTR(US%(0)):J=USR(VARPTR(C%(0)))
241 GOTO200

250 DEFUSR=VARPTR(US%(0)):J=USR(VARPTR(C%(0)))
251 PRINT@896,CHR$(31);"STRING IS ";ST$;"   PRESS <ENTER>...";
252 LINEINPUTA$:GOTO200
```

The KWKARRAY routine is especially useful if you want to load data from disk to memory for a sort. You'll see that SORT2 and SORT3 are designed to work with arrays organized as contiguous fixed-length records in protected memory. That's exactly how a quick array is organized. Once the data is sorted, KWKARRAY gives you a convenient way to retrieve and use the data.

KWKARRAY
Quick Array USR Subroutine

```
                 00000 ;KWKARRAY
                 00001 ;
FE00             00150        ORG     0FE00H         ;ORIGIN - RELOCATABLE
FE00 CD7F0A      00160        CALL    0A7FH          ;HL POINTS TO CONTROL ARRAY
FE03 E5          00170        PUSH    HL             ;
FE04 DDE1        00180        POP     IX             ;IX POINTS TO CONTROL ARRAY
FE06 DD6E0A      00190        LD      L,(IX+10)      ;
FE09 DD660B      00200        LD      H,(IX+11)      ;HL POINTS TO STRING VARPTR
```

```
FEØC 4E        ØØ21Ø          LD     C,(HL)            ;
FEØD Ø6ØØ      ØØ22Ø          LD     B,Ø               ;BC HAS STRING LENGTH
FEØF 23        ØØ23Ø          INC    HL                ;
FE1Ø 5E        ØØ24Ø          LD     E,(HL)            ;
FE11 23        ØØ25Ø          INC    HL                ;
FE12 56        ØØ26Ø          LD     D,(HL)            ;DE POINTS TO SKEY
FE13 DDCBØ246  ØØ27Ø          BIT    Ø,(IX+2)          ;TEST FOR MOVE-TO-TOP COMMAND
FE17 281F      ØØ28Ø          JR     Z,TEST2           ;TEST BIT 1 IF BIT 2 IS ZERO
FE19 EB        ØØ29Ø          EX     DE,HL             ;SKEY POINTER TO HL
FE1A DD5EØ6    ØØ3ØØ          LD     E,(IX+6)          ;
FE1D DD56Ø7    ØØ31Ø          LD     D,(IX+7)          ;DE POINTS TO NEXT POSITION
FE2Ø EDBØ      ØØ32Ø          LDIR                     ;COPY SKEY INTO ARRAY
FE22 DD73Ø6    ØØ33Ø          LD     (IX+6),E          ;
FE25 DD72Ø7    ØØ34Ø          LD     (IX+7),D          ;PUT NEW TOP BYTE IN CONTROL 3
FE28 DD6EØ8    ØØ35Ø          LD     L,(IX+8)          ;
FE2B DD66Ø9    ØØ36Ø          LD     H,(IX+9)          ;HL HAS OLD COUNT
FE2E 23        ØØ37Ø          INC    HL                ;HL HAS NEW COUNT
FE2F DD75Ø8    ØØ38Ø JMP3     LD     (IX+8),L          ;
FE32 DD74Ø9    ØØ39Ø          LD     (IX+9),H          ;PUT NEW COUNT IN CONTROL 4
FE35 C39AØA    ØØ4ØØ REBAS    JP     ØA9AH             ;RETURN TO BASIC
               ØØ4Ø1 ;
               ØØ4Ø2 ;NOTE:   FOLLOWING LOGIC IS ONLY NEEDED FOR COMMANDS 1 & 2
FE38 D5        ØØ41Ø TEST2    PUSH   DE                ;SAVE POINTER TO SKEY
FE39 C5        ØØ42Ø          PUSH   BC                ;SAVE STRING LENGHT
FE3A DD5EØØ    ØØ43Ø          LD     E,(IX+Ø)          ;
FE3D DD56Ø1    ØØ44Ø          LD     D,(IX+1)          ;DE HAS DESIRED ELEMENT#
FE4Ø 1B        ØØ45Ø          DEC    DE                ;ELEMENT 1 = ELEMENT Ø
FE41 21ØØØØ    ØØ46Ø          LD     HL,Ø              ;MULTIPLY DE BY C GIVING HL
FE44 CB39      ØØ47Ø MUL1     SRL    C                 ;CONTINUE...
FE46 3ØØ1      ØØ48Ø          JR     NC,MUL2           ;CONTINUE...
FE48 19        ØØ49Ø          ADD    HL,DE             ;CONTINUE...
FE49 28Ø5      ØØ5ØØ MUL2     JR     Z,MUL9            ;CONTINUE...
FE4B EB        ØØ51Ø          EX     DE,HL             ;CONTINUE...
FE4C 29        ØØ52Ø          ADD    HL,HL             ;CONTINUE...
FE4D EB        ØØ53Ø          EX     DE,HL             ;CONTINUE...
FE4E 18F4      ØØ54Ø          JR     MUL1              ;CONTINUE...
FE5Ø DD5EØ4    ØØ55Ø MUL9     LD     E,(IX+4)          ;
FE53 DD56Ø5    ØØ56Ø          LD     D,(IX+5)          ;DE HAS MEMORY BASE
FE56 19        ØØ57Ø          ADD    HL,DE             ;HL POINTS TO ARRAY ELEMENT
FE57 C1        ØØ58Ø          POP    BC                ;BC HAS MOVE LENGTH
FE58 D1        ØØ59Ø          POP    DE                ;DE POINTS TO SKEY
FE59 DDCBØ24E  ØØ6ØØ          BIT    1,(IX+2)          ;TEST ON COMMAND
FE5D 2ØØ3      ØØ61Ø          JR     NZ,JMP1           ;JUMP IF COMMAND WAS 2
FE5F EDBØ      ØØ62Ø          LDIR                     ;MOVE ARRAY ELEMENT TO SKEY
FE61 C9        ØØ63Ø          RET                      ;RETURN TO BASIC
               ØØ631 ;
               ØØ632 ;NOTE:   FOLLOWING LOGIC IS ONLY NEEDED FOR COMMAND 2
FE62 EB        ØØ64Ø JMP1     EX     DE,HL             ;
FE63 EDBØ      ØØ65Ø          LDIR                     ;MOVE SKEY TO ARRAY ELEMENT
FE65 DD6EØ6    ØØ66Ø          LD     L,(IX+6)          ;
FE68 DD66Ø7    ØØ67Ø          LD     H,(IX+7)          ;HL HAS OLD TOP ADDRESS
FE6B B7        ØØ68Ø          OR     A                 ;CLEAR CARRY
FE6C ED52      ØØ69Ø          SBC    HL,DE             ;
FE6E 38Ø8      ØØ7ØØ          JR     C,JMP2            ;IF CARRY, WE'VE EXTENDED ARRAY
FE7Ø DD6EØ8    ØØ71Ø          LD     L,(IX+8)          ;
FE73 DD66Ø9    ØØ72Ø          LD     H,(IX+9)          ;HL HAS # ELEMENTS FOR PASS-BACK
FE76 18BD      ØØ73Ø          JR     REBAS             ;RETURN TO BASIC
FE78 DD73Ø6    ØØ74Ø JMP2     LD     (IX+6),E          ;
FE7B DD72Ø7    ØØ75Ø          LD     (IX+7),D          ;RECORD NEW TOP ADDRESS
FE7E DD6EØØ    ØØ76Ø          LD     L,(IX+Ø)          ;
FE81 DD66Ø1    ØØ77Ø          LD     H,(IX+1)          ;HL HAS NEW # OF ELEMENTS
FE84 18A9      ØØ78Ø          JR     JMP3              ;RECORD IT AND RETURN TO BASIC
FE2F           ØØ79Ø          END                      ;
ØØØØØ TOTAL ERRORS
```

A High-Speed Memory Sort

The SORT2 USR routine lets you quickly sort data that is stored in protected memory. That data can be arranged in records of up to 255 bytes and you can specify that a specific 'field' within each record be used as the sort key. Though it uses much of the same logic as the SORT1 routine, in this case, we are actually swapping records in memory. You can use the KWKARRAY routine to get the data into memory, either from disk or operator entry. Then, after calling SORT2, you can retrieve each record in ascending sequence with the KWKARRAY routine.

Here are some typical timings for random data sorted with the SORT2 USR routine on a TRS-80 Model 1:

250 4-byte keys – 2 seconds
1000 1-byte keys – 10 seconds
1000 8-byte keys – 16 seconds

In sorting data from disk files, you're main time consumption is in loading that data into memory and in recording the results back onto the disk when the sort is complete. Here's where the SORT2 routine, used in conjunction with the KWKARRAY routine gives you some big time savings over sorts that use standard BASIC string arrays.

The sort parameters are passed to the SORT2 routine using a 10-element control array. Elements 0, 1, 3 and 5 are not used by SORT2 but they are defined so that the KWKARRAY USR routine can share the same control array.

Load your parameters into the control array as follows:

Element 2 specifies the starting memory address of the array to be sorted.

Element 4 specifies the number of elements within the array that you want to sort.

Element 6 specifies the record length of each array element.

Element 7 specifies the offset from the start of each record to the field containing the sort key. If, for instance, you've got 16-byte records and you want to ignore the first 4 bytes, element 7 would be 4. If you want comparisons to start at the first byte of each record, element 7 is specified as 0.

Element 8 specifies the length of the field that is to be used in comparisons. If you have 16-byte records, but just want to sort based on the first 3 bytes, element 8 should be 3 and element 7 should be 0. If you have 16-byte records and you want every byte to be considered in the sort, element 8 should be 16 and element 7 should be 0.

Element 9 specifies the address of a work area. This work area is used as temporary storage by SORT2 when it swaps the records in your array. The work area required is equal to your record length. You can specify an area just above or below your array in memory or if you've got upper-lower case capability, you can specify part of your video display as a work area. (This way, your operator has something to look at while the computer is sorting.)

Let's suppose you have an array of 1000 product prices and descriptions stored in upper memory, starting at C180. Each record contains a 4-byte price followed by a 12-byte product description. To sort in alphabetical order by product description, you could set up your control array as follows:

```
C%(2)=&HC180       'ARRAY BASE
C%(4)=1000         'SORT 1000 RECORDS
C%(6)=16           'LENGTH OF EACH RECORD
C%(7)=4            'COMPARE OFFSET
C%(8)=12           'COMPARE LENGTH
C%(9)=&HC170       'WORK AREA, 16 BYTES BELOW ARRAY BASE
```

Now, to sort the memory array, assuming you have loaded and defined your SORT2 routine as USR6, your command is:

```
J=USR6(VARPTR(C%(0)))
```

'J' in this case is a dummy variable. No argument is passed back from the SORT2 subroutine.

The SORT2 routine is 212 bytes and fully relocatable. You can load it anywhere in memory using any of the procedures we've described for loading USR routines.

SORT2
Memory Sort USR Subroutine

M 2 Note # 2 3

```
Magic Array Format, 106 elements

 32717  -6902  -7715  20189  -8952   2374   1048  -6695 -15911
   289 -18688  17133 -13360 -13512 -15079  -7719  -8743   2158
 26333 -18679  17133  -9755  -9775 -13560   2183  20189  -8956
  1350   8645      1  -9755  -6719 -11815  -5351  20189   6924
    33 -13568  12345   6401   1320  10731   6379  -8716   1118
 22237   6405 -10799   6373  -7924   2273   2293 -13327  10311
  6305  -8769   3662      6  -5367  -5367  18141   6672  10430
  6146   8966   4115   6390  14340   6146  -9954 -14891 -11815
-18463  21229 -13016 -10989 -15911   1753  -8960   3150  -7727
 -6903   2539   6373  -7738  -8731   4702  22237  -8941   3150
     6  -4667 -15952  -7727 -10779  -4667 -15952 -11807  -6699
 28381  -8942   4966 -20243 -13560   2247 -18664


Poke Format, 212 bytes

205 127  10 229 221 225 221  78   8 221  70   9  24   4 217 229
217 193  33   1   0 183 237  66 208 203  56 203  25 197 217 225
217 221 110   8 221 102   9 183 237  66 229 217 209 217   8 203
135   8 221  78   4 221  70   5 197  33   1   0 229 217 193 229
217 209  25 235 221  78  12  27  33   0   0 203  57  48   1  25
 40   5 235  41 235  24 244 221  94   4 221  86   5  25 209 213
229  24  12 225 225   8 245   8 241 203  71  40 161  24 191 221
 78  14   6   0   9 235   9 235 221  70  16  26 190  40   2  24
  6  35  19  16 246  24   4  56   2  24  30 217 213 197 217 209
225 183 237  82  40 205  19 213 217 193 217   6   0 221  78  12
209 225   9 229 235   9 229  24 198 225 229 221  94  18 221  86
 19 221  78  12   6   0 197 237 176 193 209 225 229 213 197 237
176 193 225 209 213 229 221 110  18 221 102  19 237 176   8 203
199   8  24 183
```

To see how the SORT2 routine works, we can generate random data on the video display and then sort the display. If you've never seen a Shell sort in action, seeing the sort on the video display is quite a sight and it gives you a feel for the pattern of comparisons and swaps that is used. The following program first generates 1000 random letters on the screen and sorts them into alphabetical order. Then it generates 250 random 4-byte records and sorts them. Finally, it sorts the contents of the video display again as 1000 1-byte records. The bottom-right corner of the screen is used as a work area for swaps.

You'll see that it takes longer for the computer to generate the random data than it takes for the SORT2 routine to rearrange the data in alphabetical sequence!

SORT2/DEM
Demonstrating a Memory Sort on the Video Display

M 2 Note # 23
M 2 Note # 40

```
20 'LOAD THE SORT2 ROUTINE INTO A MAGIC ARRAY
21 DATA 32717,-6902,-7715, 20189,-8952, 2374, 1048,-6695,-15911,
 289,-18688, 17133,-13360,-13512,-15079,-7719
22 DATA-8743, 2158, 26333,-18679, 17133,-9755,-9775,-13560,
2183, 20189,-8956, 1350, 8645, 1,-9755,-6719
23 DATA-11815,-5351, 20189, 6924, 33,-13568, 12345, 6401, 1320,
10731, 6379,-8716, 1118, 22237, 6405,-10799
24 DATA 6373,-7924, 2273, 2293,-13327, 10311, 6305,-8769, 3662,
6,-5367,-5367, 18141, 6672, 10430, 6146
25 DATA 8966, 4115, 6390, 14340, 6146,-9954,-14891,-11815,-18463
, 21229,-13016,-10989,-15911, 1753,-8960, 3150
26 DATA-7727,-6903, 2539, 6373,-7738,-8731, 4702, 22237,-8941, 3
150, 6,-4667,-15952,-7727,-10779,-4667
27 DATA-15952,-11807,-6699, 28381,-8942, 4966,-20243,-13560, 224
7,-18664
28 DIMUX%(105):FORX=0TO105:READUX%(X):NEXT
100 'CREATE DEMONSTRATION DATA ON THE SCREEN AND SORT
101 CLS:FORX=0TO999:PRINTCHR$(64+RND(26));:NEXT
110 C%(2)=15360:C%(4)=1000:C%(6)=1:C%(7)=0:C%(8)=1:C%(9)=16372
111 J=0:DEFUSR=VARPTR(UX%(0)):J=USR(VARPTR(C%(0)))
115 FORX=1TO1000:NEXT
120 'CREATE 250 4-BYTE SORT KEYS ON THE SCREEN AND SORT
121 CLS:FORX=0TO249:FORY=1TO3:PRINTCHR$(64+RND(13));:NEXT:PRINT"
";:NEXT
130 C%(2)=15360:C%(4)=250:C%(6)=4:C%(7)=0:C%(8)=4:C%(9)=16372
131 J=0:DEFUSR=VARPTR(UX%(0)):J=USR(VARPTR(C%(0)))
132 FORX=1TO1000:NEXT

140 'RE-SORT THEM AS 1-BYTE KEYS
150 C%(2)=15360:C%(4)=1000:C%(6)=1:C%(7)=0:C%(8)=1:C%(9)=16372
151 J=0:DEFUSR=VARPTR(UX%(0)):J=USR(VARPTR(C%(0)))
160 FORX=1TO1000:NEXT

170 GOTO100
```

SORT2
Memory Sort USR Subroutine

```
                00000 ;SORT2
                00001 ;
F000            00200         ORG     0F000H     ;ORIGIN - RELOCATABLE
F000 CD7F0A     00210         CALL    0A7FH      ;HL POINTS TO CONTROL ARRAY
F003 E5         00220         PUSH    HL         ;PREPARE FOR COPY TO IX
F004 DDE1       00230         POP     IX         ;IX POINTS TO CONTROL ARRAY
F006 DD4E08     00240         LD      C,(IX+8)   ;
F009 DD4609     00250         LD      B,(IX+9)   ;BC HAS # RECORDS
F00C 1804       00260         JR      JMP1       ;
F00E D9         00270 LOOP1   EXX                ;
F00F E5         00280         PUSH    HL         ;
F010 D9         00290         EXX                ;
```

```
F011 C1         00300         POP     BC           ;BC HAS CURRENT GAP
F012 210100     00310 JMP1    LD      HL,0001H     ;PREPARE FOR TEST IF GAP <=1
F015 B7         00320         OR      A            ;CLEAR CARRY
F016 ED42       00330         SBC     HL,BC        ;SUBTRACT: 1 - GAP
F018 D0         00340         RET     NC           ;BACK TO BASIC IF GAP <=1
F019 CB38       00350         SRL     B            ;DIVIDE GAP BY 2
F01B CB19       00360         RR      C            ;DIVIDE GAP BY 2, CONT.
F01D C5         00370         PUSH    BC           ;
F01E D9         00380         EXX                  ;
F01F E1         00390         POP     HL           ;HL' HAS CURRENT GAP
F020 D9         00400         EXX                  ;
F021 DD6E08     00410         LD      L,(IX+8)     ;
F024 DD6609     00420         LD      H,(IX+9)     ;HL HAS # RECORDS
F027 B7         00430         OR      A            ;CLEAR CARRY
F028 ED42       00440         SBC     HL,BC        ;SUBTRACT: #RECS - GAP
F02A E5         00450         PUSH    HL           ;
F02B D9         00460         EXX                  ;
F02C D1         00470         POP     DE           ;DE' HAS DIFFERENCE
F02D D9         00480         EXX                  ;
F02E 08         00490 LOOP2   EX      AF,AF'       ;PREP TO RESET SWAP FLAG
F02F CB87       00500         RES     0,A          ;SWAP FLAG BIT = 0
F031 08         00510         EX      AF,AF'       ;RESTORE AF
F032 DD4E04     00520         LD      C,(IX+4)     ;
F035 DD4605     00530         LD      B,(IX+5)     ;BC POINTS TO FIRST RECORD
F038 C5         00540         PUSH    BC           ;SAVE IT ON STACK
F039 210100     00550         LD      HL,0001H     ;
F03C E5         00560         PUSH    HL           ;
F03D D9         00570         EXX                  ;
F03E C1         00580         POP     BC           ;BC' HAS LOWER COMPARE REC#
F03F E5         00590         PUSH    HL           ;
F040 D9         00600         EXX                  ;
F041 D1         00610         POP     DE           ;DE HAS CURRENT GAP
F042 19         00620         ADD     HL,DE        ;COMPUTE UPPER REC# FOR COMPARE
F043 EB         00630         EX      DE,HL        ;DE HAS UPPER REC#
F044 DD4E0C     00640         LD      C,(IX+12)    ;C  HAS RECORD LENGTH
F047 1B         00650         DEC     DE           ;DE HAS UPPER REC# -1
F048 210000     00660         LD      HL,0         ;MULTIPLY DE BY C GIVING HL
F04B CB39       00670 MUL1    SRL     C            ;CONTINUE...
F04D 3001       00680         JR      NC,MUL2      ;CONTINUE...
F04F 19         00690         ADD     HL,DE        ;CONTINUE...
F050 2805       00700 MUL2    JR      Z,MUL9       ;CONTINUE...
F052 EB         00710         EX      DE,HL        ;CONTINUE...
F053 29         00720         ADD     HL,HL        ;CONTINUE...
F054 EB         00730         EX      DE,HL        ;CONTINUE...
F055 18F4       00740         JR      MUL1         ;CONTINUE...
F057 DD5E04     00750 MUL9    LD      E,(IX+4)     ;HL HAS # BYTES FROM BASE
F05A DD5605     00760         LD      D,(IX+5)     ;DE HAS MEMORY BASE
F05D 19         00770         ADD     HL,DE        ;HL POINTS TO UPPER RECORD
F05E D1         00780         POP     DE           ;DE HAS LOWER REC POINTER
F05F D5         00790         PUSH    DE           ;SAVE LOWER REC POINTER ON STACK
F060 E5         00800         PUSH    HL           ;SAVE UPPER REC POINTER ON STACK
F061 180C       00810         JR      LOOP3        ;
F063 E1         00820 JMP2    POP     HL           ;RELIEVE STACK
F064 E1         00830         POP     HL           ;RELIEVE STACK
F065 08         00840         EX      AF,AF'       ;PREP TO TEST FOR SWAP
F066 F5         00850         PUSH    AF           ;
F067 08         00860         EX      AF,AF'       ;
F068 F1         00870         POP     AF           ;A HAS SWAP FLAG
F069 CB47       00880         BIT     0,A          ;ANY SWAPS MADE?
F06B 28A1       00890         JR      Z,LOOP1      ;IF NO SWAPS, LOOP1
F06D 18BF       00900         JR      LOOP2        ;OTHERWISE, LOOP2
F06F DD4E0E     00910 LOOP3   LD      C,(IX+14)    ;
F072 0600       00920         LD      B,0000H      ;BC HAS COMPARE OFFSET
F074 09         00930         ADD     HL,BC        ;POINT TO COMPARE PORTION
F075 EB         00940         EX      DE,HL        ;
F076 09         00950         ADD     HL,BC        ;POINT TO COMPARE PORTION
```

```
FØ77 EB       ØØ96Ø          EX    DE,HL       ;DE & HL ARE ADJUSTED FOR COMPARE
FØ78 DD461Ø   ØØ97Ø          LD    B,(IX+16)   ;B HAS COMPARE LENGTH
FØ7B 1A       ØØ98Ø CPLOOP   LD    A,(DE)      ;ACCUM HAS LOWER REC BYTE
FØ7C BE       ØØ99Ø          CP    (HL)        ;COMPARE TO UPPER REC BYTE
FØ7D 28Ø2     Ø1ØØØ          JR    Z,NXCHAR    ;IF EQUAL, LOOK AT NEXT BYTE
FØ7F 18Ø6     Ø1Ø1Ø          JR    NOTEQ       ;OTHERWISE, GO PROCESS INEQUALITY
FØ81 23       Ø1Ø2Ø NXCHAR   INC   HL          ;POINT TO NEXT BYTE FOR COMPARE
FØ82 13       Ø1Ø3Ø          INC   DE          ;POINT TO NEXT BYTE FOR COMPARE
FØ83 1ØF6     Ø1Ø4Ø          DJNZ  CPLOOP      ;SUBTRACT FROM COUNT, REPEAT
FØ85 18Ø4     Ø1Ø5Ø          JR    NOSWAP      ;IF COUNT REACHED Ø, ARE EQUAL
FØ87 38Ø2     Ø1Ø6Ø NOTEQ    JR    C,NOSWAP    ;NO SWAP IF UPPER GREATER
FØ89 181E     Ø1Ø7Ø          JR    SWAP        ;OTHERWISE, SWAP UPPER & LOWER
FØ8B D9       Ø1Ø8Ø NOSWAP   EXX               ;
FØ8C D5       Ø1Ø9Ø          PUSH  DE          ;
FØ8D C5       Ø11ØØ          PUSH  BC          ;
FØ8E D9       Ø111Ø          EXX               ;
FØ8F D1       Ø112Ø          POP   DE          ;DE HAS LOWER COMPARE REC #
FØ9Ø E1       Ø113Ø          POP   HL          ;HL HAS UPPER COMPARE BASE #
FØ91 B7       Ø114Ø          OR    A           ;CLEAR CARRY
FØ92 ED52     Ø115Ø          SBC   HL,DE       ;TEST IF EQUAL
FØ94 28CD     Ø116Ø          JR    Z,JMP2      ;MORE TO GO IF NOT EQUAL
FØ96 13       Ø117Ø          INC   DE          ;ADD 1 TO LOWER COMPARE REC#
FØ97 D5       Ø118Ø          PUSH  DE          ;
FØ98 D9       Ø119Ø          EXX               ;
FØ99 C1       Ø12ØØ          POP   BC          ;SAVE IT IN BC'
FØ9A D9       Ø121Ø          EXX               ;
FØ9B Ø6ØØ     Ø122Ø          LD    B,Ø         ;
FØ9D DD4EØC   Ø123Ø          LD    C,(IX+12)   ;BC HAS RECORD LENGTH
FØAØ D1       Ø124Ø          POP   DE          ;GET UPPER REC POINTER
FØA1 E1       Ø125Ø          POP   HL          ;GET LOWER REC POINTER
FØA2 Ø9       Ø126Ø          ADD   HL,BC       ;POINT TO NEXT LOWER REC
FØA3 E5       Ø127Ø          PUSH  HL          ;PUT IT ON STACK
FØA4 EB       Ø128Ø          EX    DE,HL       ;
FØA5 Ø9       Ø129Ø          ADD   HL,BC       ;POINT TO NEXT UPPER REC
FØA6 E5       Ø13ØØ          PUSH  HL          ;PUT IT ON STACK
FØA7 18C6     Ø131Ø          JR    LOOP3       ;REPEAT
FØA9 E1       Ø132Ø SWAP     POP   HL          ;GET POINTER TO UPPER REC
FØAA E5       Ø133Ø          PUSH  HL          ;PUT IT BACK ON STACK
FØAB DD5E12   Ø134Ø          LD    E,(IX+18)   ;
FØAE DD5613   Ø135Ø          LD    D,(IX+19)   ;DE POINTS TO WORK AREA
FØB1 DD4EØC   Ø136Ø          LD    C,(IX+12)   ;
FØB4 Ø6ØØ     Ø137Ø          LD    B,ØØH       ;BC HAS # BYTES TO MOVE
FØB6 C5       Ø138Ø          PUSH  BC          ;SAVE IT FOR NEXT MOVE
FØB7 EDBØ     Ø139Ø          LDIR              ;MOVE UPPER REC TO WORK AREA
FØB9 C1       Ø14ØØ          POP   BC          ;RESTORE # BYTES TO MOVE
FØBA D1       Ø141Ø          POP   DE          ;DE HAS POINTER TO UPPER REC
FØBB E1       Ø142Ø          POP   HL          ;HL HAS POINTER TO LOWER REC
FØBC E5       Ø143Ø          PUSH  HL          ;SAVE ON STACK
FØBD D5       Ø144Ø          PUSH  DE          ;SAVE ON STACK
FØBE C5       Ø145Ø          PUSH  BC          ;SAVE ON STACK
FØBF EDBØ     Ø146Ø          LDIR              ;MOVE LOWER REC TO UPPER REC
FØC1 C1       Ø147Ø          POP   BC          ;GET # BYTES TO MOVE
FØC2 E1       Ø148Ø          POP   HL          ;
FØC3 D1       Ø149Ø          POP   DE          ;DE POINTS TO LOWER RECORD
FØC4 D5       Ø15ØØ          PUSH  DE          ;
FØC5 E5       Ø151Ø          PUSH  HL          ;
FØC6 DD6E12   Ø152Ø          LD    L,(IX+18)   ;
FØC9 DD6613   Ø153Ø          LD    H,(IX+19)   ;HL POINTS TO TEMP WORK AREA
FØCC EDBØ     Ø154Ø          LDIR              ;MOVE FROM WORK AREA TO LOWER REC
FØCE Ø8       Ø155Ø          EX    AF,AF'      ;PREP TO SET SWAP FLAG
FØCF CBC7     Ø156Ø          SET   Ø,A         ;SWAP FLAG IN A' IS SET
FØD1 Ø8       Ø157Ø          EX    AF,AF'      ;RESTORE AF REGISTER
FØD2 18B7     Ø158Ø          JR    NOSWAP      ;SWAP IS DONE
FØ8B          Ø159Ø          END               ;
ØØØØØ TOTAL ERRORS
```

Interactive Sorting by Insertion

The SORT3 USR routine lets you maintain an array in sequence as you add data to it. Upon receiving a key, this subroutine searches for the first record in the array that is greater. It then moves all remaining records up and inserts the new key. The parameters for SORT3 are designed to be compatible with the KWKARRAY USR routine. Instead of using the KWKARRAY command 1, which adds a new entry to the top of the array, you can call SORT3 to insert the new key in sequence and update the count of active elements.

Where does SORT3 fit in with the other techniques we've discussed? Its main application is in programs where the operator may be entering data and you want to keep the array sorted as data is entered. The average time taken to insert an element, once you've got about 1000 elements in the array, is about a quarter second, so it will still be unnoticeable to the operator.

In applications where you are sorting data being read from a disk file, you should use the SORT2 routine for the greatest speed, unless you need the memory that is saved by the shorter SORT3 routine. Your savings is about 59 bytes plus the length of 1 record.

The parameters for SORT3 are passed using a control array. This control array can be shared with the control array you may be using with the KWKARRAY routine. Elements 0 and 1 are not used. Elements 2 through 7 are loaded as follows:

> **Element 2** specifies the starting address of your array in protected memory.
>
> **Element 3** specifies the next address at the top of the array. Upon beginning with an empty array, you should load element 3 so that it is equal to element 2. The SORT3 subroutine automatically adds the record length to element 3 each time you add to the array.
>
> **Element 4** is a count of the number of elements in the array. When starting with an empty array, you should load 0 into element 4. Each call to the SORT3 routine increments this counter by 1.
>
> **Element 5** is the VARPTR to the string you are adding to the array. The length of the string specifies the record length to be used for each element in the array. You LSET data into this string before calling SORT3.
>
> **Element 6** is the compare offset. It specifies the position of the sort field within each element of the array. If element 6 is 0, comparisons begin at the first byte.
>
> **Element 7** is the length of the sort field. If only a portion of each record is used for sequencing purposes, you will use elements 6 and 7 to define that portion.

Let's suppose you want to maintain a sorted array of up to 500 8-byte elements, starting at memory location **F000**. Each element consists of a 6-byte alphanumeric product number and a 2-byte pointer which indicates where that product can be found on disk. You want to maintain the product numbers in sequence as you add new products, but the 2-byte pointer is not to be used in the sequencing. To initialize the array, your commands are:

```
ST$=STRING$(8," ")   'INITIALIZE KEY-PASSING STRING
C%(2)=&HF000         'ARRAY BASE
C%(3)=C%(2)          'NEXT ADDRESS = ARRAY BASE
C%(4)=0              '0 ACTIVE ELEMENTS TO START
C%(5)=VARPTR(ST$)    'RECORDS WILL BE INSERTED VIA ST$
C%(6)=2              'COMPARE OFFSET
C%(7)=6              'COMPARE LENGTH
```

To add a 2-byte pointer, A% and a product number, PN$ to the array, maintaining the proper sequence, your command is:

```
LSETST$=MKI$(A%)+PN$    'PUT THE NEW KEY IN A STRING
J=USR4(VARPTR(C%(0)))   'INSERT THE KEY IN SEQUENCE
IFJ>500THENPRINT "THAT'S IT - YOU CAN'T ADD ANY MORE"
```

In this case we would have earlier defined USR4 to point to the SORT3 routine. The variable, 'J', upon return to BASIC from SORT3, contains the updated count of active entries in the array. It is the responsibility of the BASIC program to insure that we don't add more elements than we've allowed space for.

To get the keys back in sequence, we can use command 0 of the KWKARRAY USR routine. Assuming we've loaded KWKARRAY as USR5, we can display all the keys that have been added in ascending sequence:

```
FOR X = 1 TO C%(4)      'FROM FIRST ELEMENT TO LAST ACTIVE
C%(0) = X               'LOAD DESIRED ELEMENT NUMBER
C%(1) = 0               'LOAD COMMAND
J=USR5(VARPTR(C%(0)))   'CALL KWKARRAY ROUTINE
PRINTMKI$(ST$);         'PRINT THE POINTER WE'VE STORED
PRINTMID$(ST$,3)        'PRINT THE PRODUCT NUMBER
NEXT                    '
```

SORT3

Insertion Sort USR Subroutine

M 2 Note # 23

M 2 Note # 41

Magic Array Format, 77 elements:

```
32717  -6902  -7715  20189  -8952   2374  28381  -8950   2918
 9086   9054  -8874   1134  26333   2053 -20359  19496 -15093
-6699  20189  -8948   3398  -5367  -5367  18141   6670  10430
14340   6160   8964   4115  -7692  24328     22 -12007   6337
-8748   1646  26333 -12025  -6699   9143  21229 -15899  -6687
24328     22  -5351  -8735   1651  29405  -4857  -7752 -15919
 5400 -10779  -8952   1646  26333   1543  20224  -8951   1653
29917 -12025  -5151      6  -4785  -8784   2158  26333   8969
30173  -8952   2420 -25917     10
```

Poke Format, 153 bytes:

```
205 127  10 229 221 225 221  78   8 221  70   9 221 110  10 221
102  11 126  35  94  35  86 221 110   4 221 102   5   8 121 176
 40  76  11 197 213 229 221  78  12 221  70  13   9 235   9 235
221  70  14  26 190  40   4  56  16  24   4  35  19  16 244 225
  8  95  22   0  25 209 193  24 212 221 110   6 221 102   7 209
213 229 183  35 237  82 229 193 225 229   8  95  22   0  25 235
225 221 115   6 221 114   7 237 184 225 209 193  24  21 229 213
  8 221 110   6 221 102   7   6   0  79   9 221 117   6 221 116
  7 209 225 235   6   0  79 237 176 221 110   8 221 102   9  35
221 117   8 221 116   9 195 154  10
```

The SORT3 demonstration program shows an insertion sort of random data. Video display memory is used as the base for our array so you can see the sort in action. First, 1000 random letters are generated and inserted at the proper position on the screen. Then the demo is repeated, this time with 250 4-byte records.

SORT3/DEM
Demonstrating an Insertion Sort on the Video Display

M 2 Note # 23
M 2 Note # 41
M 2 Note # 42

```
20 'LOAD THE SORT3 ROUTINE INTO A MAGIC ARRAY
21 DATA 32717,-6902,-7715, 20189,-8952, 2374, 28381,-8950, 2918,
 9086, 9054,-8874, 1134, 26333, 2053,-20359
22 DATA 19496,-15093,-6699, 20189,-8948, 3398,-5367,-5367, 18141
, 6670, 10430, 14340, 6160, 8964, 4115,-7692
23 DATA 24328, 22,-12007, 6337,-8748, 1646, 26333,-12025,-6699,
9143, 21229,-15899,-6687, 24328, 22,-5351
24 DATA-8735, 1651, 29405,-4857,-7752,-15919, 5400,-10779,-8952,
 1646, 26333, 1543, 20224,-8951, 1653, 29917
25 DATA-12025,-5151, 6,-4785,-8784, 2158, 26333, 8969, 30173,-89
52, 2420,-25917, 10
26 DIMUX%(76):FORX=0TO76:READUX%(X):NEXT

100 DEFINTA-Z:J=0
110 CLS
120 ST$=STRING$(1," "):C%(2)=15360:C%(3)=15360:C%(4)=0:C%(5)=VAR
PTR(ST$):C%(6)=0:C%(7)=0
130 FORX=0TO999
140 LSETST$=CHR$(64+RND(26))
150 DEFUSR=VARPTR(UX%(0)):J=USR(VARPTR(C%(0)))
160 NEXT

170 FORX=1TO1000:NEXT

200 DEFINTA-Z:J=0
210 CLS
220 ST$=STRING$(4," "):C%(2)=15360:C%(3)=15360:C%(4)=0:C%(5)=VAR
PTR(ST$):C%(6)=0:C%(7)=0
230 FORX=0TO249
240 A$="":FORY=0TO2:A$=A$+CHR$(64+RND(26)):NEXT:LSETST$=A$
250 DEFUSR=VARPTR(UX%(0)):J=USR(VARPTR(C%(0)))
260 NEXT

270 FORX=1TO1000:NEXT

280 GOTO100
```

High-Speed Memory Search

The SEARCH2 USR subroutine lets you search memory for any string. Based on the parameters you load into a control array, you can search byte-by-byte from any starting location, or you can define a record length greater than 1 to search record-by-record. Within each record you can specify the position of the search key. If the search key is found, SEARCH2 returns the record number and the actual memory address. If you wish, you can continue the search to find the next match.

SEARCH2 is designed for use with the KWKARRAY USR routine, and it can share the same control array. But you can use it for most any memory searching job. I've found it very helpful in finding the memory addresses used by the TRS-80 and its operating systems.

SORT3

Insertion Sort USR Subroutine

```
                00000 ;SORT3
                00001 ;
F000            00180         ORG     0F000H          ;ORIGIN - RELOCATABLE
                00190 ;THE FOLLOWING LOGIC POINTS IX TO BASE OF CONTROL ARRAY
F000 CD7F0A     00200         CALL    0A7FH           ;LOAD PARAMETER ARRAY VARPTR HL
F003 E5         00210         PUSH    HL              ;PREPARE FOR COPY TO IX
F004 DDE1       00220         POP     IX              ;IX POINTS TO PARAMETER ARRAY
                00230 ;THE FOLLOWING LOGIC LOADS CONTROL INFO TO Z80 REGISTERS
F006 DD4E08     00240         LD      C,(IX+8)        ;
F009 DD4609     00250         LD      B,(IX+9)        ;BC HAS RECORD COUNT
F00C DD6E0A     00260         LD      L,(IX+10)       ;
F00F DD660B     00270         LD      H,(IX+11)       ;HL HAS SKEY VARPTR
F012 7E         00280         LD      A,(HL)          ;A HAS KEY LENGTH
F013 23         00290         INC     HL              ;
F014 5E         00300         LD      E,(HL)          ;
F015 23         00310         INC     HL              ;
F016 56         00320         LD      D,(HL)          ;DE POINTS TO SKEY DATA
F017 DD6E04     00330         LD      L,(IX+4)        ;
F01A DD6605     00340         LD      H,(IX+5)        ;HL POINTS TO BASE OF ARRAY
                00350 ;THE FOLLOWING LOGIC PREPARES FOR NEXT COMPARE
F01D 08         00360 LOOP1   EX      AF,AF'          ;SAVE KEY LENGTH
F01E 79         00370         LD      A,C             ;PREP TO TEST BC FOR ZERO
F01F B0         00380         OR      B               ;SET Z FLAG IF BC IS ZERO
F020 284C       00390         JR      Z,EOF           ;IF EOF THEN GO ADD AT END
F022 0B         00400         DEC     BC              ;DECREMENT COUNT FOR NEXT PASS
F023 C5         00410         PUSH    BC              ;SAVE RECORD COUNT ON STACK
F024 D5         00420         PUSH    DE              ;SAVE SKEY POINTER
F025 E5         00430         PUSH    HL              ;SAVE CURRENT ARRAY POINTER
F026 DD4E0C     00440         LD      C,(IX+12)       ;
F029 DD460D     00450         LD      B,(IX+13)       ;BC HAS COMPARE OFFSET
F02C 09         00460         ADD     HL,BC           ;HL POINTS TO COMPARE PORTION
F02D EB         00470         EX      DE,HL           ;
F02E 09         00480         ADD     HL,BC           ;
F02F EB         00490         EX      DE,HL           ;DE POINTS TO COMPARE PORTION
F030 DD460E     00500         LD      B,(IX+14)       ;B HAS COMPARE LENGTH
                00510 ;THE FOLLOWING LOGIC COMPARES SKEY TO ARRAY KEY
F033 1A         00520 CPLOOP  LD      A,(DE)          ;LOAD SKEY DATA TO ACCUM
F034 BE         00530         CP      (HL)            ;COMPARE TO KEY DATA
F035 2804       00540         JR      Z,NXCHAR        ;NEXT CHARACTER IF EQUAL
F037 3810       00550         JR      C,SKEYLS        ;IF C,  SKEY IS LESS
F039 1804       00560         JR      GROREQ          ;SKEY IS GREATER
F03B 23         00570 NXCHAR  INC     HL              ;POINT TO NEXT CHAR. IN KEY
F03C 13         00580         INC     DE              ;POINT TO NEXT CHAR. IN SKEY
F03D 10F4       00590         DJNZ    CPLOOP          ;DEC CHAR. COUNT AND REPEAT
                00600 ;AT THIS POINT SKEY IS GREATER OR EQUAL.
F03F E1         00610 GROREQ  POP     HL              ;RESTORE POINTER TO CURR. KEY
F040 08         00620         EX      AF,AF'          ;GET KEY LENGTH
F041 5F         00630         LD      E,A             ;
F042 1600       00640         LD      D,0             ;KEY LENGTH IN DE
F044 19         00650         ADD     HL,DE           ;HL POINTS TO NEXT KEY
F045 D1         00660         POP     DE              ;RESTORE SKEY POINTER
F046 C1         00670         POP     BC              ;RESTORE RECORD COUNT
F047 18D4       00680         JR      LOOP1           ;REPEAT FOR NEXT RECORD
                00690 ;THE FOLLOWING LOGIC IS USED IF SKEY IS LESS THAN CURRENT KEY
                00700 ;FIRST, WE WILL MOVE REMAINING KEYS UP
F049 DD6E06     00710 SKEYLS  LD      L,(IX+6)        ;
F04C DD6607     00720         LD      H,(IX+7)        ;HL POINTS TO LAST ACTIVE BYTE
F04F D1         00730         POP     DE              ;DE POINTS TO CURRENT RECORD
F050 D5         00740         PUSH    DE              ;RESTORE STACK
F051 E5         00750         PUSH    HL              ;SAVE HL DURING ADD
F052 B7         00760         OR      A               ;CLEAR CARRY FLAG
F053 23         00770         INC     HL              ;
F054 ED52       00780         SBC     HL,DE           ;HL HAS # BYTES TO MOVE
F056 E5         00790         PUSH    HL              ;PREPARE FOR COPY TO BC
F057 C1         00800         POP     BC              ;BC HAS # BYTES TO MOVE
```

```
FØ58 E1        ØØ81Ø        POP     HL              ;HL POINTS TO LAST ACTIVE BYTE
FØ59 E5        ØØ82Ø        PUSH    HL              ;SAVE AGAIN DURING ADD
FØ5A Ø8        ØØ83Ø        EX      AF,AF'          ;RECORD LENGTH TO A
FØ5B 5F        ØØ84Ø        LD      E,A             ;
FØ5C 16ØØ      ØØ85Ø        LD      D,Ø             ;DE HAS RECORD LENGTH
FØ5E 19        ØØ86Ø        ADD     HL,DE           ;HL POINTS TO NEW LAST BYTE
FØ5F EB        ØØ87Ø        EX      DE,HL           ;DE POINTS TO NEW LAST BYTE
FØ6Ø E1        ØØ88Ø        POP     HL              ;HL POINTS TO OLD LAST BYTE
FØ61 DD73Ø6    ØØ89Ø        LD      (IX+6),E        ;
FØ64 DD72Ø7    ØØ9ØØ        LD      (IX+7),D        ;SAVE NEW LAST BYTE IN CONTROL 3
FØ67 EDB8      ØØ91Ø        LDDR                    ;MOVE UP REST OF ARRAY
FØ69 E1        ØØ92Ø        POP     HL              ;HL POINTS TO COPY DESTINATION
FØ6A D1        ØØ93Ø        POP     DE              ;DE POINTS TO SKEY
FØ6B C1        ØØ94Ø        POP     BC              ;BC HAS RECORD COUNT
FØ6C 1815      ØØ95Ø        JR      CPYKEY          ;GO COPY THE KEY INTO ARRAY
               ØØ96Ø ;FOLLOWING LOGIC ADDS REC LENGTH TO CONTROL 3 FOR EOF ADDS
FØ6E E5        ØØ97Ø EOF    PUSH    HL              ;SAVE POINTER TO ARRAY DATA
FØ6F D5        ØØ98Ø        PUSH    DE              ;SAVE POINTER TO SKEY DATA
FØ7Ø Ø8        ØØ99Ø        EX      AF,AF'          ;RECORD LENGTH TO A
FØ71 DD6EØ6    Ø1ØØØ        LD      L,(IX+6)        ;
FØ74 DD66Ø7    Ø1Ø1Ø        LD      H,(IX+7)        ;HL POINTS TO OLD LAST BYTE
FØ77 Ø6ØØ      Ø1Ø2Ø        LD      B,Ø             ;
FØ79 4F        Ø1Ø3Ø        LD      C,A             ;BC HAS REC LENGTH
FØ7A Ø9        Ø1Ø4Ø        ADD     HL,BC           ;HL POINTS TO NEW LAST BYTE
FØ7B DD75Ø6    Ø1Ø5Ø        LD      (IX+6),L        ;
FØ7E DD74Ø7    Ø1Ø6Ø        LD      (IX+7),H        ;WRITE NEW LAST BYTE TO CONTROL 3
FØ81 D1        Ø1Ø7Ø        POP     DE              ;RESTORE POINTER TO SKEY
FØ82 E1        Ø1Ø8Ø        POP     HL              ;RESTORE POINTER TO ARRAY
               Ø1Ø9Ø ;THE FOLLOWING LOGIC COPIES EXTERNAL POINTER AND KEY
FØ83 EB        Ø11ØØ CPYKEY EX      DE,HL           ;HL=SOURCE & DE=DESTINATION
FØ84 Ø6ØØ      Ø111Ø        LD      B,Ø             ;
FØ86 4F        Ø112Ø        LD      C,A             ;BC HAS RECORD LENGTH
FØ87 EDBØ      Ø113Ø        LDIR                    ;COPY SKEY INTO ARRAY
FØ89 DD6EØ8    Ø114Ø        LD      L,(IX+8)        ;
FØ8C DD66Ø9    Ø115Ø        LD      H,(IX+9)        ;HL HAS KEY COUNT
FØ8F 23        Ø116Ø        INC     HL              ;ADD 1 TO KEY COUNT
FØ9Ø DD75Ø8    Ø117Ø        LD      (IX+8),L        ;
FØ93 DD74Ø9    Ø118Ø        LD      (IX+9),H        ;RECORD COUNT IN CONTROL 7
FØ96 C39AØA    Ø119Ø        JP      ØA9AH           ;PASS COUNT BACK TO BASIC
ØA9A           Ø12ØØ        END                     ;
ØØØØØ TOTAL ERRORS
```

Communication between your BASIC program and the SEARCH2 USR routine is done with a 10-element control array:

Element 0 specifies the starting record number for the search, and the record number that is found when the search is completed. If you want the search to begin at the first record in a memory array, you load element 0 with 1. If SEARCH2 finds a match in the 10th record, upon return to BASIC, element 0 will contain 10.

Element 1 is unused by the SEARCH2 routine. It is left unused so that SEARCH2 can share the same control array with the KWKARRAY routine.

Element 2 specifies the starting address of the memory array.

Element 3 is unused. Like element 1, it's unused so that SEARCH2 can be compatible with the KWKARRAY routine.

Element 4 specifies the number of records in the array. The search is terminated with a 'not found' condition if the search key is not found between the starting record number, specified by element 0, and the ending record number, specified by element 4.

Element 5 must be loaded with the VARPTR to a 'return' string. Before calling SEARCH2 you should create this string so that it has a length equal to the record length. When a match is found, SEARCH2 points this string to the record in the array containing the matching search key. In effect, the record that is found will be contained in this return string upon return to BASIC.

Element 6 specifies the record length, ranging from 1 to 255 bytes. SEARCH2 increments the memory address by the record length after each element is compared.

Element 7 specifies the key offset from the beginning of each record. If your memory array is composed of records that are 80 bytes long, and you want to match on the 10th byte of each record, you would use 9 as your key offset. (9 bytes precede the comparison portion of the record.)

Element 8 should be loaded with the VARPTR of a search key. This is the string that the SEARCH2 routine will look for in your memory array. If SK$ is your search key, element 8 would be specified as VARPTR(SK$). If SK$ contains 'XXX', element 7 is 10, and element 6 is 80, SEARCH2 will look for the first 80-byte record having 'XXX' in bytes 11 through 13. (If one is found, the string specified by element 5 will contain the full 80-byte record.)

Element 9 is used by SEARCH2 to return the memory address of the record that is found.

As an example, let's suppose you have an array in protected memory, starting at C180, and there are 200 records in the array. Each record is 16 bytes long, consisting of a 4-byte price, followed by a 12-byte product description. Assuming you've loaded and defined the SEARCH2 routine as USR6, you could use the following logic to find the first record whose product description starts with one or more letters entered by the operator.

First we should define our array. We only need to do this once in a program, unless we change the address or number of active records:

```
C%(2)=&HC180              'DEFINE ARRAY BASE ADDRESS
C%(4)=200                 'NUMBER OF ACTIVE RECORDS
RE$=STRING$(16," ")       'CREATE A RETURN STRING
C%(5)=VARPTR(RE$)         'LOAD RETURN STRING VARPTR
C%(6)=16                  'DEFINE RECORD LENGTH
C%(8)=VARPTR(SK$)         'WE WILL USE SK$ AS OUR SEARCH KEY
```

Now, we let the operator input the desired search key and we store it in SK$. To do the search of product descriptions we use the following logic:

```
C%(7)=4                   '4-BYTES PRECEDE THE DESCRIPTION
C%(0)=1                   'START AT FIRST RECORD IN ARRAY
J=USR6(VARPTR(C%(0)))     'CALL SEARCH2 USR ROUTINE
```

Upon return from the search routine, 'J' will equal 0 if the record was not found. Otherwise, 'J' will have the record number, as will C%(0). RE$ will have the 16-byte record that was found. C%(9) will contain the memory address of the record.

If we have found a match and want to continue the search to see if there are any other matches, we can simply add 1 to the record number contained in C% (0), and loop back to call the SEARCH2 USR routine again.

In some cases, you may wish to search memory byte-by-byte. Let's suppose we want to find the word 'RADIO' in memory, starting from byte 0 in ROM. We could use the following logic:

```
A$="RADIO"               'SEARCH KEY IS "RADIO"
C%(Ø)=1                  'START AT RECORD 1
C%(2)=Ø                  'BASE OF MEMORY ARRAY IS Ø
C%(4)=&HFFFF             'SEARCH TO TOP OF MEMORY
RE$=" "                  'SETUP A DUMMY RETURN STRING
C%(5)=VARPTR(RE$)        'LOAD VARPTR OF RETURN STRING
C%(6)=1                  'RECORD LENGTH IS 1
C%(7)=Ø                  'KEY OFFSET IS Ø
C%(8)=VARPTR(A$)         'LOAD VARPTR OF SEARCH KEY
J=USR6(VARPTR(C%(Ø)))'CALL SEARCH2
```

If 'RADIO' is found, J will be non-zero, and the address will be returned in C%(9).

SEARCH2
General Purpose Memory and Array Search USR Subroutine
M 2 Note # 23
M 2 Note # 41

```
Magic Array Format, 85 elements:
 32717  -6902  -7715  20189  -8948     94  22237   6913     33
-13568  12345   6401   1320  10731   6379  -5132  28381  -8956
  1382  -8935   4725  29917  -8941   4206  26333  17937   9032
  9054 -10922  -8763     94  22237  -8959   2158  26333 -18679
 21229  21560  28381  -8942   4966  24285   5646   6400 -11839
-14891 -16870   1568   8979  -2032   8472  28381  -8960    358
 -8925    117  29917  -8959   4718  26333  -8941   3166     22
 -8935   4725  29917   6163  -8780   2670  26333  17931  24285
 -8942   4950  29475  29219  28381  -8960    358   1048     46
    38 -15935 -25917     10

Poke Format, 169 bytes:
 205 127  10 229 221 225 221  78  12 221  94   0 221  86   1  27
  33   0   0 203  57  48   1  25  40   5 235  41 235  24 244 235
 221 110   4 221 102   5  25 221 117  18 221 116  19 221 110  16
 221 102  17  70  72  35  94  35  86 213 197 221  94   0 221  86
   1 221 110   8 221 102   9 183 237  82  56  84 221 110  18 221
 102  19 221  94  14  22   0  25 193 209 213 197  26 190  32   6
  19  35  16 248  24  33 221 110   0 221 102   1  35 221 117   0
 221 116   1 221 110  18 221 102  19 221  94  12  22   0  25 221
 117  18 221 116  19  24 180 221 110  10 221 102  11  70 221  94
  18 221  86  19  35 115  35 114 221 110   0 221 102   1  24   4
  46   0  38   0 193 193 195 154  10
```

The SEARCH2/DEM program demonstrates the use of SEARCH2. You'll want to keep it in your library as a utility program to use whenever you need to find something in memory. Since SEARCH2 is loaded into a magic array in the demo program, you don't need to specify a special memory size and any arrays that you might have in upper memory will be undisturbed.

SEARCH2

General Purpose Search USR Subroutine

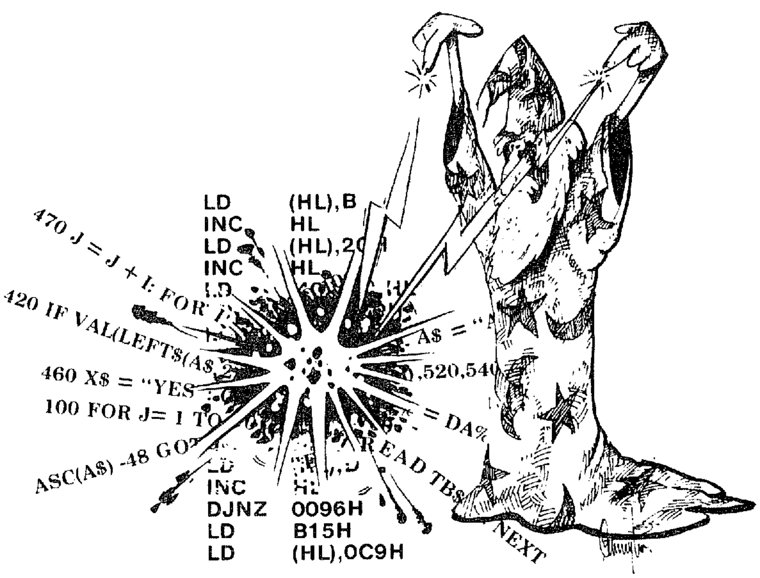
```
                ;SEARCH2
                ;
F000                    ORG     0F000H          ;ORIGIN - RELOCATABLE
                ;
                ;THE FOLLOWING LOGIC POINTS IX TO BASE OF PARAMETER ARRAY
F000 CD7F0A             CALL    0A7FH           ;CONTROL ARRAY POINTER TO HL
F003 E5                 PUSH    HL              ;PREPARE FOR COPY TO IX
F004 DDE1               POP     IX              ;IX POINTS TO BASE OF CONTROL
                ;
                ;THE FOLLOWING LOGIC COMPUTES THE MEMORY LOC OF THE START RECORD
F006 DD4E0C             LD      C,(IX+12)       ;C HAS RECORD LENGTH
F009 DD5E00             LD      E,(IX+0)        ;
F00C DD5601             LD      D,(IX+1)        ;START REC# IN DE
F00F 1B                 DEC     DE              ;REC# EXPRESSED AS 1 IS  ZERO
F010 210000             LD      HL,0            ;MULTIPLY DE BY C GIVING HL
F013 CB39       MUL1    SRL     C               ;CONTINUE...
F015 3001               JR      NC,MUL2         ;CONTINUE...
F017 19                 ADD     HL,DE           ;CONTINUE...
F018 2805       MUL2    JR      Z,MUL9          ;CONTINUE...
F01A EB                 EX      DE,HL           ;CONTINUE...
F01B 29                 ADD     HL,HL           ;CONTINUE...
F01C EB                 EX      DE,HL           ;CONTINUE...
F01D 18F4               JR      MUL1            ;CONTINUE...
F01F EB         MUL9    EX      DE,HL           ;DE HAS PRODUCT
F020 DD6E04             LD      L,(IX+4)        ;
F023 DD6605             LD      H,(IX+5)        ;HL HAS ARRAY BASE ADDRESS
F026 19                 ADD     HL,DE           ;HL POINTS TO START RECORD
F027 DD7512             LD      (IX+18),L       ;
F02A DD7413             LD      (IX+19),H       ;RECORD START RECORD ADDRESS
                ;
                ;THE FOLLOWING LOGIC GETS SKEY ADDRESS AND LENGTH FROM VARPTR
F02D DD6E10             LD      L,(IX+16)       ;
F030 DD6611             LD      H,(IX+17)       ;HL HAS SKEY VARPTR
F033 46                 LD      B,(HL)          ;B HAS SKEY LENGTH
F034 48                 LD      C,B             ;C ALSO HAS SKEY LENGTH
F035 23                 INC     HL              ;
F036 5E                 LD      E,(HL)          ;
F037 23                 INC     HL              ;
F038 56                 LD      D,(HL)          ;DE POINTS TO SKEY DATA
F039 D5                 PUSH    DE              ;
F03A C5                 PUSH    BC              ;
                ;
                ;BEGIN LOOP FOR EACH RECORD
F03B DD5E00     RCLOOP  LD      E,(IX+0)        ;
F03E DD5601             LD      D,(IX+1)        ;CURRENT REC # IN DE
F041 DD6E08             LD      L,(IX+8)        ;
F044 DD6609             LD      H,(IX+9)        ;RECORD LIMIT IN HL
F047 B7                 OR      A               ;CLEAR CARRY
F048 ED52               SBC     HL,DE           ;SUBTRACT
F04A 3854               JR      C,NOTFND        ;NOT FOUND IF WE'VE SEARCHED ALL
F04C DD6E12             LD      L,(IX+18)       ;
F04F DD6613             LD      H,(IX+19)       ;HL HAS MEMORY LOCATION
F052 DD5E0E             LD      E,(IX+14)       ;
F055 1600               LD      D,0             ;DE HAS KEY OFFSET
F057 19                 ADD     HL,DE           ;HL POINTS TO KEY DATA
                ;
                ;BEGIN LOOP FOR EACH COMPARE
F058 C1                 POP     BC              ;
F059 D1                 POP     DE              ;
F05A D5                 PUSH    DE              ;
F05B C5                 PUSH    BC              ;
F05C 1A         CPLOOP  LD      A,(DE)          ;SKEY DATA TO ACCUM
F05D BE                 CP      (HL)            ;COMPARE WITH ARRAY DATA
F05E 2006               JR      NZ,NOTEQ        ;IF NON ZERO, NO MATCH
```

```
FØ6Ø 13                      INC     DE               ;POINT TO NEXT CHARACTER
FØ61 23                      INC     HL               ;POINT TO NEXT CHARACTER
FØ62 1ØF8                    DJNZ    CPLOOP           ;GO COMPARE NEXT IF MORE
                      ;
                      ;END LOOP FOR EACH COMPARE
FØ64 1821                    JR      EQUAL            ;ALL CHARACTERS ARE EQUAL
FØ66 DD6EØØ           NOTEQ  LD      L,(IX+Ø)         ;
FØ69 DD66Ø1                  LD      H,(IX+1)         ;HL HAS RECORD COUNT
FØ6C 23                      INC     HL               ;ADD TO RECORD COUNT
FØ6D DD75ØØ                  LD      (IX+Ø),L         ;
FØ7Ø DD74Ø1                  LD      (IX+1),H         ;RE-RECORD THE COUNT
FØ73 DD6E12                  LD      L,(IX+18)        ;
FØ76 DD6613                  LD      H,(IX+19)        ;HL HAS OLD MEMORY LOCATION
FØ79 DD5EØC                  LD      E,(IX+12)        ;
FØ7C 16ØØ                    LD      D,Ø              ;DE HAS RECORD LENGTH
FØ7E 19                      ADD     HL,DE            ;HL POINTS TO NEW MEMORY LOC
FØ7F DD7512                  LD      (IX+18),L        ;
FØ82 DD7413                  LD      (IX+19),H        ;RE-RECORD MEMORY LOCATION
FØ85 18B4                    JR      RCLOOP           ;GET NEXT RECORD
                      ;
                      ;END LOOP FOR EACH RECORD
                      ;
                      ;PROCESS THE RETURN WHEN AN EQUAL IS FOUND
FØ87 DD6EØA           EQUAL  LD      L,(IX+1Ø)        ;
FØ8A DD66ØB                  LD      H,(IX+11)        ;HL HAS RETURN VARPTR
FØ8D 46                      LD      B,(HL)           ;PUT RECORD LENGTH IN B
FØ8E DD5E12                  LD      E,(IX+18)        ;
FØ91 DD5613                  LD      D,(IX+19)        ;GET RECORD ADDRESS
FØ94 23                      INC     HL               ;
FØ95 73                      LD      (HL),E           ;
FØ96 23                      INC     HL               ;
FØ97 72                      LD      (HL),D           ;RECORD RETURN VARPTR
FØ98 DD6EØØ                  LD      L,(IX+Ø)         ;
FØ9B DD66Ø1                  LD      H,(IX+1)         ;RETURN RECORD NUMBER TO BASIC
FØ9E 18Ø4                    JR      BACBAS           ;JUMP TO GO BACK TO BASIC
                      ;
                      ;THE FOLLOWING LOGIC PROCESSES THE RETURN IF THE KEY NOT FOUND
FØAØ 2EØØ             NOTFND LD      L,Ø              ;
FØA2 26ØØ                    LD      H,Ø              ;RETURN ZERO IF NONE FOUND
                      ;
                      ;THE FOLLOWING LOGIC RETURNS HL TO BASIC
FØA4 C1               BACBAS POP     BC               ;RESTORE STACK
FØA5 C1                      POP     BC               ;RESTORE STACK
FØA6 C39AØA                  JP      ØA9AH            ;RETURN HL TO BASIC
ØA9A                         END                      ;
ØØØØØ TOTAL ERRORS
```

SEARCH2/DEM

Memory Search Demonstration and Utility Program

```
1 CLEAR1ØØØ:J%=Ø

1Ø 'LOAD SEARCH2 ROUTINE INTO A MAGIC ARRAY...
11 DATA 32717,-69Ø2,-7715, 2Ø189,-8948, 94, 22237, 6913, 33,-135
68, 12345, 64Ø1, 132Ø, 1Ø731, 6379,-5132
12 DATA 28381,-8956, 1382,-8935, 4725, 29917,-8941, 42Ø6, 26333,
 17937, 9Ø32, 9Ø54,-1Ø922,-8763,  94, 22237
13 DATA-8959, 2158, 26333,-18679, 21229, 2156Ø, 28381,-8942,
 4966, 24285, 5646, 64ØØ,-11839,-14891,-1687Ø, 1568
14 DATA 8979,-2Ø32, 8472, 28381,-896Ø, 358,-8925, 117,
29917,-8959, 4718, 26333,-8941, 3166, 22,-8935
15 DATA 4725, 29917, 6163,-878Ø, 267Ø, 26333, 17931,
 24285,-8942, 495Ø, 29475, 29219, 28381,-896Ø, 358, 1Ø48
16 DATA 46, 38,-15935,-25917,1Ø
17 DIMUS%(84):FORX=ØTO84:READUS%(X):NEXT

1ØØ CLS:PRINT
11Ø INPUT"STARTING RECORD NUMBER              ";C%(Ø)
12Ø INPUT"MEMORY ARRAY BASE ADDRESS           ";C%(2)
13Ø INPUT"NUMBER OF RECORDS IN ARRAY          ";C%(4)
14Ø INPUT"RECORD LENGTH                       ";C%(6)
15Ø INPUT"KEY OFFSET FROM START OF EACH REC ";C%(7)
16Ø LINEINPUT"SEARCH KEY : ";SK$:IFLEN(SK$)=ØTHEN16Ø
17Ø PRINT

2ØØ RE$=STRING$(C%(6)," "):C%(5)=VARPTR(RE$):C%(8)=VARPTR(SK$)

3ØØ DEFUSR=VARPTR(US%(Ø)):J%=USR(VARPTR(C%(Ø)))
31Ø PRINT@64Ø,CHR$(31);:IFJ%=ØTHENPRINT"NOT FOUND.":LINEINPUT"PR
ESS <ENTER>...";A$:GOTO1ØØ
32Ø PRINT"FOUND IN RECORD";J%;"  MEMORY LOCATION = ";C%(9)
33Ø PRINT"RECORD FOUND IS: ";RE$
34Ø PRINT@896,"PRESS <C> TO CONTINUE SEARCH, OTHERWISE PRESS <EN
TER>...";
341 LINEINPUTA$
345 IFA$="C"THENC%(Ø)=C%(Ø)+1:GOTO3ØØELSE1ØØ
```



Video & Keyboard Trickery

Here are some powerful tricks, subroutines and programming ideas that can give you more control over the dialog between the TRS-80 computer and the operator. These techniques will help you make your video displays more professional in appearance, but, just as important, you will be able to better enforce valid operator entries while simplifying your programming task.

Video Display = Visible Memory

M 2 Note # 7

The first thing that you need to know is that the TRS-80 video display is in reality, a 'window', showing the contents of a block of memory 1024 bytes long. This window of memory extends from memory locations 15360 to 16383. (3C00 – 3FFF). If, for example, memory location 15360 contains a 65 decimal or 41 hex, you will see the letter 'A' in the upper left corner of your video display.

A PRINT command actually just copies data from its current memory location, into the screen memory area located at 15360 plus the current cursor position. When the screen rolls up or 'scrolls', your TRS-80 is really just moving the contents of memory locations 15424 through 16383 down 64 bytes to locations 15360 through 16319 and it is loading 64 spaces onto the bottom line of the screen, memory locations 16320 through 16383.

You can use the video display position chart as a reference in planning your video displays. The upper portion of the chart gives you the PRINT positions for every 8 positions on the screen, starting at position 0 in the upper left corner. The lower portion of the chart shows the corresponding memory locations.

Video Display POKEs

M 2 Note # 7

Knowing that the video display is just another block of memory, we have an alternate way of printing information. We can POKE one or more characters into any location between 15360 and 16383.

To use the poke command on the video display, you can simply add 15360 to the desired PRINT@ position ranging from 0 to 1023. For example, to put the letter 'A' at position 256, your command could be:

```
      POKE 15360 + 256, ASC("A")
 or,  POKE 15616,65
```

Why poke to video display memory when you can use a PRINT@ just as easily?

1. Poking video display memory gives you a method of printing one or more characters without moving the current cursor positon.

2. In some situations, (but not all), the poke command is faster than PRINT@.

3. The poke command lets you print a character in the lower right corner of your screen, position 1023, without scrolling the screen up. (Any PRINT command that prints in position 1023 will cause a line feed.)

Video Display PEEKs

M 2 Note # 7

The peek function lets us inquire into the current contents of any location on the video display. To peek a location on the video display, use 15360 plus the desired position on the display. For example:

```
     PEEK (15360+256)
or,  PEEK (15616)
```

. . . gives your program the ASCII code for the character currently displayed at position 256.

Peek is useful in 'flashing cursor' routines where you need to temporarily store the character from the current cursor position, while alternating between your cursor symbol and the character.

Note that if your TRS-80 has an 'upper case only' video display, the computer converts all characters to upper case for display. Therefore, if you type or print a lower case character, that character will be changed to a displayable (upper case) character. This change is automatically made by the system in video display memory. If you POKE 97 (lower case 'a') into memory location 15360, you will get 65 (upper case 'A') when you peek that location.

If you have installed a lower case modification in your TRS-80, be sure to load the driver program provided when using any special techniques that directly access video display memory. While your TRS-80 may appear to be operating in upper case mode without the driver, you'll find that a displayed upper case 'A' will be a 1, 'B' will be a 2, and so forth.

Radio Shack's upper/lower case driver for Model 1 TRS-80's uses the top 590 bytes of memory. The mini upper/lower case driver program that follows is a solution for you if you need that top 590 bytes for something else or you just can't afford to spend that much RAM on a ULC driver. This one takes just 38 bytes and it is relocatable, so you can put it anywhere in protected memory.

This driver is only 38 bytes because it handles just the video conversions. It does not include a keyboard driver, so to get lower case characters, you'll have to hold down the shift key. At any rate, if you've had the upper/lower case kit installed you will need to use Radio Shack's driver or this one in order to take advantage of many of the video display subroutines in this book.

VDRIVE/BAS
Mini Upper Lower Case Video Driver
M 2 Note # 7

```
0'VDRIVE/BAS
10 DATA 221,110,3,221,102,4,218,154,4,221,126,5,183,40,1,119,121
,254,32,218,6,5,254,128,210,166,4,229,38,32,188,48,1,124,225,195
,125,4
20 FORX=0TO37:READP:POKE&HFFDA+X,P:NEXT
21 A$="":A%=VARPTR(A$):POKEA%,2:POKEA%+1,&H1E:POKEA%+2,&H40
22 LSETA$=CHR$(&HDA)+CHR$(&HFF)
```

Video Display Position Chart

M 2 Note # 7

VIDEO DISPLAY -- PRINT@ POSITIONS

0	8	16	24	32	40	48	56
64	72	80	88	96	104	112	120
128	136	144	152	160	168	176	184
192	200	208	216	224	232	240	248
256	264	272	280	288	296	304	312
320	328	336	344	352	360	368	376
384	392	400	408	416	424	432	440
448	456	464	472	480	488	496	504
512	520	528	536	544	552	560	568
576	584	592	600	608	616	624	632
640	648	656	664	672	680	688	696
704	712	720	728	736	744	752	760
768	776	784	792	800	808	816	824
832	840	848	856	864	872	880	888
896	904	912	920	928	936	944	952
960	968	976	984	992	1000	1008	1016

VIDEO DISPLAY - MEMORY LOCATIONS

15360	15368	15376	15384	15392	15400	15408	15416
15424	15432	15440	15448	15456	15464	15472	15480
15488	15496	15504	15512	15520	15528	15536	15544
15552	15560	15568	15576	15584	15592	15600	15608
15616	15624	15632	15640	15648	15656	15664	15672
15680	15688	15696	15704	15712	15720	15728	15736
15744	15752	15760	15768	15776	15784	15792	15800
15808	15816	15824	15832	15840	15848	15856	15864
15872	15880	15888	15896	15904	15912	15920	15928
15936	15944	15952	15960	15968	15976	15984	15992
16000	16008	16016	16024	16032	16040	16048	16056
16064	16072	16080	16088	16096	16104	16112	16120
16128	16136	16144	16152	16160	16168	16176	16184
16192	16200	16208	16216	16224	16232	16240	16248
16256	16264	16272	16280	16288	16296	16304	16312
16320	16328	16336	16344	16352	16360	16368	16376

To link-in the mini ULC video driver, specify the proper memory size when you go into BASIC and then run the VDRIVE/BAS program. It will remain activated until you re-boot the computer.

The listing shown assumes that you have a 48K TRS-80 and you want the driver to go into the top 38 bytes. In this case, you would specify a memory size of 65497 or less.

If you've got 32K, you can load the driver into the top 38 bytes by changing the **FFDA** in line 20 to **BFDA** and the **FF** in line 22 to **BF**. Your memory size specification must be 49113 or less.

If you want to locate this 38-byte driver at any other location, simply change the FF and DA in lines 20 and 22 accordingly.

Pointing Strings at the Screen

This useful technique lets you, in effect, load up to 255 bytes of data currently displayed at any position into a string. You will find that this trick will help you:

Quickly and simply record video display screens to disk.

Create screen-to-printer routines to provide hard copy printouts of a complete screen or selected portions.

Eliminate duplication of program logic in applications where you want to provide both a video display and a line printer routine for printing the same data.

Create routines which temporarily store video display data in one or more strings, while displaying other data, with the ability to flash back the original data.

To simplify and speed-up formatted data entry routines. Your video display can serve as a temporary storage area for the data before it is loaded into a string.

To understand how this technique works, you must know that for every string variable in your program, the TRS-80 maintains a 2-byte pointer which keeps track of the location of the string's contents in memory and a 1-byte indicator of the string's length. Your program can access this information using the VARPTR function:

For string A$:

PEEK(VARPTR(A$)) = length of the string A$
PEEK(VARPTR(A$)+1) = LSB of address pointer to A$'s data
PEEK(VARPTR(A$)+2) = MSB of address pointer to A$'s data

The video display string pointer subroutine pokes the desired length and screen address into a string variable's pointers. This one-line subroutine arbitrarily uses the string, AN$ and line 40070:

Video Display String Pointer Subroutine

M 2 Note # 43

```
40070 AN$=" ":POKEVARPTR(AN$),A1%:POKEVARPTR(AN$)+2,INT(PO%/256)
+60:POKEVARPTR(AN$)+1,PO%-INT(PO%/256)*256:RETURN
```

Before calling the subroutine, load integer PO% with the desired starting position on the screen (0-1023) and load A1% with the length of the data to be loaded into the string (1-255).

Upon return from the subroutine, the string AN$ will contain the data currently displayed at position PO%, for length A1%. Note that if you subsequently print other data on the video display or if the video display scrolls, the string AN$ will then contain the new data displayed. Because of this, you may want to immediately set another string equal to AN$ so that the data won't be modified if the video display is altered.

Here is a simple program that demonstrates one application of the video display string pointer subroutine. It first points the AN$ string to the top 64 positions on the video display. Then it uses LSET to progressively move portions of another string, S$, onto the video display. The effect is horizontal scrolling of the top video display line. To use it, you will need to type in or merge subroutine 40070. You can try other values for PO% and A1% in line 210, to move your scrolling window to another location.

Horizontal Scrolling Demonstration

M 2 Note # 7

```
1 CLEAR1ØØØ
2ØØ CLS:S$="THIS IS A STRING THAT IS 219 BYTES LONG.  WE ARE SCR
OLLING IT LEFT AND RIGHT USING THE LSET COMMAND.  TO DO IT WE SI
MPLY POINT A STRING TO THE DISPLAY. THEN WE LSET A  MID-PORTION
OF THE STRING WE WANT TO SCROLL INTO IT."
21Ø PO%=Ø:A1%=64:GOSUB4ØØ7Ø
22Ø FORX=1TOLEN(S$)+1:LSETAN$=MID$(S$,X):NEXT
221 FORX=1TO2ØØ:NEXT
23Ø FORX=LEN(S$)+1TO1STEP-1:LSETAN$=MID$(S$,X):NEXT
231 FORX=1TO2ØØ:NEXT
24Ø GOTO22Ø
```

LPRINT the Video Display

You can use the video display string pointer subroutine to make a printout of the screen. This method is much faster than peeking each position and LPRINTing the CHR$ of each peek. Watch out for graphics characters, though. This routine does no conversions of graphics characters for printing.

This screen printer subroutine calls subroutine 40070, using a length of 64 and LPRINTs AN$ for each line on the video display:

Screen Printer Subroutine

M 2 Note # 44

```
573ØØ A1%=64:FORPO%=ØTO96ØSTEP64:GOSUB4ØØ7Ø:LPRINTAN$:NEXT:RETURN
```

You can modify this routine to print selected portions of the screen. For example, if you want to LPRINT the middle 10 lines of the screen only, the second command of the subroutine could be changed to read:

M 2 Note # 21

```
FORPO%=192TO768STEP64
```

Reference to the video display position chart will help you determine the 'from' and 'to' values of PO%.

If you are printing the full screen, you might want to 'frame' the video display printout by printing a string of dashes before and after calling the subroutine.

Storing Displays on Disk

The video display string pointer subroutine can also be used when you want to store a video display on disk. I've used the technique at times to record displays so that they could be merged into word processing text for writing program documentation. Here is a sample routine:

Write Video Display to Disk Subroutine

M 2 Note # 44

```
574ØØ OPEN"O",1,"DISPLAY1/SEQ":' OPEN A SEQUENTIAL DISK FILE
5741Ø FORPO%=ØTO96ØSTEP64
5742Ø A1%=64:GOSUB4ØØ7Ø:PRINT#1,AN$
5743Ø NEXT
5744Ø CLOSE1:RETURN
```

In line 57400, you may, of course, provide the file number, disk file name and drive number that you want. The part of your program that displays the screen would execute the command, 'GOSUB 57400' in response to a specific key depression.

Reading a Display from Disk

There are two things to watch out for when re-displaying a screen that you have recorded on disk in a sequential file. You must use the LINE INPUT# command to prevent problems that could be caused by ':' or ',' characters within your display. Secondly, if you recorded 16 lines of 64 characters each, you will need to avoid generating unwanted line feeds, especially after the last line. We can avoid the line feeds by 'fielding' each line of the screen using the video display string pointer subroutine and using LSET to put the line from disk onto the screen.

Read Video Display from Disk Subroutine

M 2 Note # 45

```
57450 OPEN"I",1,"DISPLAY1/SEQ":'OPEN THE SCREEN FILE
57460 FORPO%=0TO960STEP64
57470 A1%=64:GOSUB40070:'POINT AN$ TO CURRENT SCREEN LINE
57475 LINE INPUT#1,A$:LSETAN$=A$
57480 NEXT
57490 CLOSE1:RETURN
```

LSET and RSET Data to the Screen

M 2 Note # 7

In line 57475 of the routine which reads a video display from disk we used LSET to print on the video display. The TRS-80 would scroll the screen up 1 line if we tried to display 64 characters on the last line using a PRINT command. The LSET and RSET commands, while normally used to load information disk buffers, can be very useful in video display applications.

LSET and RSET load information into a string of predefined length without altering the the location of the string in memory and without changing its length. Because of this, you can set up input and output 'fields' on the video display. LSET lets you left-justify information into a field, filling trailing positions with blanks. RSET lets you right-justify information into a field, filling leading spaces with blanks. When these fields are on your video display, you can quickly flash information into them without altering other portions of the screen.

Here are the steps required:

1. Point a string to the screen. (The video display string pointer subroutine 40070 shows you how to do this for the string, AN$, position, PO% and length, A1%. You can, if necessary, change AN$ to another variable name or use a string array if you want more than one field simultaneously.)

2. LSET or RSET the string that is pointed to the screen equal to another string.

3. Note that if, after pointing a string to the screen, you load it with another value without using LSET or RSET, it will no longer point to the screen. Also, be aware that if you let the screen scroll, the contents of any string that is pointed to the screen will be the new screen data at the pointed position and length.

Pointing Disk Buffers to the Screen

M 2 Note # 7

For each disk file that you have opened, there is a 2-byte location in memory that gives the address of a 256-byte buffer area. When you GET a physical record, the data on disk is copied into this buffer area in memory. When you PUT a physical record, the data in this memory area is written to disk. With 2 simple poke commands, we can point the disk buffer directly to video display memory! Then when you GET a record, it will automatically be displayed. When you PUT a record, the contents of a 256-byte block on the video display will be written to disk.

Here's how to write the screen to random disk file 1, starting at the disk physical record specified by X:

```
P1%=PEEK(25944):P2%=PEEK(25945)
FORA%=0 TO 3
POKE 25944,0:POKE25945,60+A%
PUT 1,X+A%
NEXT
POKE 25944,P1%:POKE25945,P2%
```

To restore the video display from disk, you simply change the 'PUT' command to a 'GET' command.

The example shown above assumes that you are using file 1 with NEWDOS 2.1. To use a different file number or if you are using a different disk operating system, you can refer to appendix 4. Look up the data control block address for the file number and disk operating system you are using. Add 3 to the DCB address and replace the 25944's in the example with the number you obtain. Add 4 to the DCB address and replace the 25945's.

The first line of the example saves the previous contents of the disk buffer pointers in P1% and P2%. The last line pokes them back. These 2 lines are optional if you are using NEWDOS 2.1 or NEWDOS80. For TRSDOS 2.3 they are required.

If you are using a Model 3, you will have to use other methods, such as moving data between the disk buffer and the display in 256 byte blocks with a move-data routine. Model 3 TRSDOS doesn't let you alter the disk buffer pointers.

Video Displays to Random Files

Here's a subroutine that lets you keep a random disk file of one or more video displays. It uses the technique we described that allows us to point a disk buffer to the screen. To use the subroutine:

1. Set PF% equal to the file number you wish to use, 1 – 15.

2. Set SN% equal to the screen number. The subroutine lets you keep as many screens on disk as capacity permits, each screen requiring 4 physical records. For a standard 35-track drive, SN% could be from 1 to 80.

3. Set A$ equal to 'R' to read from disk to video display, or 'W' to write from video display to disk.

4. Call the video display / random disk read-write subroutine using the command, 'GOSUB 57400'.

Video Display to Random Disk File Subroutine

M 2 Note # 7

```
57400 OPEN"R",PF%,"DISPLAY1/RND:1":'OPEN A RANDOM DISK FILE
57401 P1%=PEEK(25944):P2%=PEEK(25945)
57410 POKE25944,0:A1%=SN%*4-3
57420 FORA%=0TO3
57422 POKE25945,60+A%
57424 IFA$="W"THENPUTPF%,A1%+A%ELSEGETPF%,A1%+A%
57426 NEXT
57429 POKE25944,P1%:POKE25945,P2%
57430 CLOSEPF%:RETURN
```

You should change the disk file name in line 57400 according to your requirements. You will also need to change the 25944's and 25945's according to the guidelines we discussed in the previous section. Lines 57401 and 57429 are optional if you are using NEWDOS 2.1 or NEWDOS80. If you want greater speed, you don't have to open and close the file each time you call the subroutine. If you wish to handle your open and close functions outside the subroutine, you'll need to change lines 57400 and 57430.

The Single-Key Subroutine

I use this neat little subroutine in just about every program I write. You'll find that it provides quite a programming convenience when you want the operator to press a single key in response to a prompt or question displayed on the screen. Subroutine 40500 simply tells the computer to wait for the operator to press any key. Upon return from the subroutine, you've got the character corresponding to the key that was pressed in A$. Here's the subroutine:

Single-Key Subroutine

```
40500 A$=INKEY$:IFA$=""THEN40500ELSERETURN
```

When you want the operator to press a single key just 'GOSUB 40500'. I use this in:

Menu routines, where I want the operator to select a program or subprogram by pressing a number or letter key, without needing to press enter.

Applications where a message or data is displayed on the screen and I want the operator to press enter to continue.

Applications where I want the operator to give a simple one-key response.

The advantage of the single-key subroutine is that:

You don't have to clutter your program logic with a two-or-more line routine to accept a single key entry. You save memory.

Your video display is not disturbed (as it could be with INPUT or LINEINPUT.) Nothing is printed until your program logic analyzes the contents of A$. The danger keys (down-arrow, clear, right-arrow) can't destroy your screen.

You provide more convenience and fewer key depressions for the operator.

The menu routine shown next is an example of one way that you can use the single-key subroutine.

Quick and Easy Menu Routines

M 2 Note # 29
M 2 Note # 30

A menu routine is a video display that gives the operator a list of alternative functions to perform and the ability to select one of those functions by letter or number. I've included this sample menu to illustrate a few tricks and system design ideas. Here's the menu to be displayed:

```
SAMPLE MENU
===============================================================
<1> ADD, CHANGE, INQUIRY
<2> TRANSACTION ENTRY
<3> PRINTED REPORTS
===============================================================
PRESS THE NUMBER OF YOUR SELECTION,
              OR PRESS <UP-ARROW> TO END...
```

Sample Menu Routine

```
1  CLEAR1ØØØ
4  SG$=STRING$(63,131)

1ØØ CLS:PRINT"
SAMPLE MENU
";SG$;
11ØPRINT"
<1> ADD, CHANGE, INQUIRY
<2> TRANSACTION ENTRY
<3> PRINTED REPORTS
";SG$
12Ø PRINT@896,"PRESS THE NUMBER OF YOUR SELECTION,
              OR PRESS <UP-ARROW> TO END...";
19Ø GOSUB4Ø5ØØ:A%=INSTR(CHR$(91)+"1234",A$):IFA%=ØTHEN19ØELSEONA
%GOTO9ØØ,1ØØØ,2ØØØ,3ØØØ

9ØØ 'END OF PROGRAM ROUTINES WOULD BE HERE
1ØØØ 'ADD, CHANGE, INQUIRY ROUTINES WOULD BE HERE
2ØØØ 'TRANSACTION ENTRY ROUTINES WOULD BE HERE
3ØØØ 'PRINTED REPORT ROUTINES WOULD BE HERE

4Ø5ØØ A$=INKEY$:IFA$=""THEN4Ø5ØØELSERETURN
```

Notice that:

1. In line 4 we created SG$, a horizontal bar to be used to help dress up video display screens within the program.

2. In lines 100, 110 and 120 the down-arrow key was used to simplify the programming of multi-line print commands.

3. Any time that the display refers to a specific key to press, it is shown

enclosed in brackets. A consistent use of brackets this way in your printed program documentation and video displays communicates 'key depression' to the operator.

4. The menu has a name. (In this case the name is 'SAMPLE MENU'.) When you write your operating instructions, it makes things much easier if you can refer to a menu by name, especially if the system has more than one menu.

5. Line 190 calls the single-key subroutine. When a key has been pressed, the INSTR function is used to validate the selection. The ON GOTO command branches the program logic to the proper routine.

6. The menu routine starts at line 100. I always put the main program menu at line 100 so that if I have troubles when debugging the program I can always press the break key and type 'GOTO100'. Line 0 has the name of the program and the date. Lines 1 to 99 perform the original 'housekeeping' functions of the program.

7. Line 40500 is the single-key subroutine.

Finding the Cursor Position

As you know, the POS(0) function tells you the current tab position of the cursor on the screen. Here's how to find the current PRINT@ position of the cursor, ranging from 0 to 1023 or the current PEEK and POKE memory location, ranging from 15360 to 16383.

```
Cursor PRINT@ position = PEEK(16417)*256+PEEK(16416)-15360
Cursor memory position = PEEK(16417)*256+PEEK(16416)
```

Now that you know how to compute the cursor position, your programs can stop the screen for viewing before information is scrolled off the top in applications where you are displaying long lists of data. Here's an example:

Cursor Inquiry Demonstration
M 2 Note # 46

```
10 CLS
20 X=X+1:PRINTX
30 IFPEEK(16417)*256+PEEK(16416)-15360>=960THENPRINT"PRESS ENTER
TO CONTINUE...";:GOSUB40500:CLS
40 GOTO20
40500 A$=INKEY$:IFA$=""THEN40500ELSERETURN
```

A more important application of cursor position inquiries is in disk error handling. In your ON ERROR GOTO routine, you can save the cursor position, display the error message and then re-poke the cursor position when you resume.

Flashing Cursors

Flashing cursors are useful in word processors and other applications where you want to have variable cursor movement without erasing the character currently displayed at the cursor postion. The big challenge is to make the cursor flashing routine fast enough so that it doesn't interfere with the typing speed of the operator. To make it fast enough in BASIC, I've found that its best to forget about delay routines. Just flash it – then immediately restore the original character.

Here's a routine that you can try. It's a variation on the single-key subroutine. Before calling subroutine 40600, load PZ% with the current cursor position in video display memory, ranging from 15360 to 16383. Load PC% with the ASCII value of the character at the current cursor position. This will be PEEK(PZ%). Upon depression of any key, your program will return from the subroutine, with the result of the key depression in A$.

Flashing Cursor Single-Key Subroutine

M 2 Note # 47

```
40600 A$=INKEY$:IFA$<>""THENRETURNELSEPOKEPX%,95:POKEPX%,PC%:GOT
O40600
```

Note that we are using the underline character, CHR$(95), as the cursor character in this routine. If you prefer a graphics block for your cursor character, replace '95' in the subroutine with '132'.

Locking Out the 'BREAK' Key

To make your programs truly 'operator-proof' you may want to lock out the break key. You can use some simple poke commands to prevent accidental or intentional interruption of a program. Be certain though, that you provide ways to get back to 'READY' if your program is not fully debugged yet.

Here are the pokes for the most popular TRS80 Model 1 disk operating systems:

M 2 Note # 48

DOS	LOCK OUT BREAK	RESTORE BREAK
TRSDOS 2.3	POKE 23886,0	POKE 23886,1
NEWDOS 2.1	POKE 23461,0	POKE 23461,1
NEWDOS/80 1.0	POKE 19408,0	POKE 19408,1

For any Model 1 or Model 3 you can lock out the break key by poking 16396 with 175 and 16397 with 201. To restore you can poke 16396 with 201. This method is given in the Model 3 manual, but watch out! If you've got the break key locked out with this method and you try to do a disk command, your computer will 'freeze up'. The only escape is the reset button.

Repeating Keys and Combinations

Did you ever want to make a function repeat as long as you are holding a key down? Here's some information that will help you:

M 2 Note # 7

```
IF PEEK(14591) = 0, then no key is being pressed.
IF PEEK(14591) > 0, then one or more keys are being pressed.
```

Type in this program and run it:

```
10 PRINTPEEK(14591);:GOTO10
```

Now press any key or key-combination and notice the numbers that are displayed. To set up repeat keys in your programs, simply test on PEEK(14591) and direct the program logic to the desired routine!

Free-Form Video Displays

Here is a program that demonstrates repeating key capabilities, a flashing cursor, character insertions and deletions, plus line insertions and deletions. The free-form video display program lets you type anything on your screen. You may also use the following special keys:

```
<UP-ARROW>          Move up  (repeating)
<DOWN-ARROW>        Move down  (repeating)
<LEFT-ARROW>        Move left  (repeating)
<RIGHT-ARROW>       Move right  (repeating)
<ENTER>             Move to beginning of next line  (repeating)
<SHIFT-UP-ARROW>    Delete current line
<SHIFT-DOWN-ARROW>  Insert line
                    (For Model 3 and late Model 1's use
                     <SHIFT-DOWN-ARROW-Z>)
<SHIFT-LEFT-ARROW>  Delete character
<SHIFT-RIGHT-ARROW> Insert character
<CLEAR>             Print underline character
```

Free-Form Video Display Program

```
0 'FREE-FORM VIDEO DISPLAY PROGRAM

10 DEFINTA-Z:PX=0:J=0
20 SC$=CHR$(9)+CHR$(8)+CHR$(91)+CHR$(10)+CHR$(13)+CHR$(25)+CHR$(
24)+CHR$(26)+CHR$(27)
30 DIMUS(7):US(0)=8448:US(2)=4352:US(4)=256:US(7)=201
100 CLS:PO=0
120 PX=15360+PO:PC=PEEK(PX):POKEPX,95
125 IFA%>0ANDPEEK(14591)>0THEN131ELSEGOSUB40600
130 A%=INSTR(SC$,A$):IFA%=0THEN140ELSEIFA%>5THEN150
131 POKEPX,PC:ONA%GOSUB1001,1002,1003,1004,1006
132 GOTO120
140 POKEPX,ASC(A$):GOSUB1001:GOTO120
150 POKEPX,PC
155 ONA%-5GOSUB2001,2002,2003,2004
160 A%=0:GOTO120
1001 PO=PO-(PO+1<1024):RETURN
1002 PO=PO+(PO-1>-1):RETURN
1003 PO=PO+64*(PO-64>-1):RETURN
1004 PO=PO-64*(PO+64<1024):RETURN
1006 PO=-((PO>=960)*PO)-(PO<960)*(INT(PO/64)*64+64):RETURN
2001 US(6)=-18195:US(1)=15360+INT(PO/64)*64+62:US(3)=US(1)+1:US(
5)=US(3)-(PX):IFUS(5)=0THENRETURNELSEGOSUB2010:POKEPX,32:RETURN

2002 US(6)=-20243:US(1)=PX+1:US(3)=PX:US(5)=64-(POANDNOT-64)-1:I
FUS(5)=0THENRETURNELSEGOSUB2010:POKEPX+64-(POANDNOT-64)-1,32:RET
URN

2003 US(6)=-18195:US(1)=16319:US(3)=16383:US(5)=960-INT(PO/64)*6
4:IFUS(5)=0THENRETURNELSEGOSUB2010:PRINT@INT(PO/64)*64,CHR$(30);
:RETURN

2004 US(6)=-20243:US(1)=15360+INT(PO/64)*64+64:US(3)=US(1)-64:US
(5)=960-INT(PO/64)*64:IFUS(5)=0THENRETURNELSEGOSUB2010:PRINT@960
,CHR$(30);:RETURN

2010 DEFUSR=VARPTR(US(0)):J=USR(0):RETURN
40600 A$=INKEY$:IFA$<>""THENRETURNELSEPOKEPX,95:POKEPX,PC:GOTO40
600
```

Line comments:

```
  10  Define variables as integers, unless otherwise
      specified.
     :Initialize variable PX for faster access
     :Initialize variable J as USR routine dummy variable
  20  Load SC$ with a table of special characters
      for processing arrow and enter key depressions.
  30  Dimension the integer array US% for 7 elements
     :Load integer array US% for use as a "move-data magic
      array".
 100  Clear the screen.
     :Set starting cursor position to zero (upper left
      corner).
 120  Load variable PX with the memory address corresponding to
      the current cursor position.
     :Store ASCII code for character at current cursor position
      in variable PO.
     :Print cursor character at current cursor position.
 125  If previous key pressed was a special character and a
      key is still being pressed then go to 131,
      otherwise GOSUB 40600 to await depression of a key.
 130  Now that a key has been pressed and the result is in A$,
      scan the special character string, SC$.
     :A% is zero if not a special character. (GOTO 140.)
      A% is > 5 if an insert/delete character. (GOTO 150.)
 131  The key pressed indicates a cursor movement command.
      Restore character at current position before moving cursor.
     :Call proper cursor movement subroutine based on A%.
 132  Go back to line 120 to get next key depression.
 140  Print the character corresponding to the current key
      depression at the current cursor position.
     :Call subroutine 1001 to advance the cursor 1 position.
     :Go back to line 120 to get next key depression.
 150  Restore character at current position before performing
      an insert or delete operation.
 155  Call proper insert/delete subroutine based on A%.
 160  Load A% with zero to prevent repetitions of the insert
      or delete operation without pressing key again.
     :Go back to line 120 to get next key depression.

1001 (Right-arrow routine)
      Add 1 to cursor position to move forward,
      enforcing a maximum of 1023.
     :Return from the subroutine.
1002 (Left-arrow routine)
      Subtract 1 from cursor position to move backward,
      enforcing a minimum of zero.
     :Return from the subroutine.
1003 (Up-arrow routine)
      Subtract 64 from cursor position to move up 1 line,
      enforcing a minimum of zero.
     :Return from the subroutine.
1004 (Down-arrow routine)

      Add 64 to cursor position to move down 1 line,
      enforcing a maximum of 1023.
     :Return from the subroutine.
1006 (ENTER routine)
      Compute beginning of next line based on cursor position,
      enforcing a maximum of 960.
     :Return from the subroutine.
```

```
 2001 (Shift-right-arrow routine - Insert space)
       Set "move-data" routine to LDDR mode.
      :Load "from" address.
      :Load "to" address.
      :Load number of bytes.
      :Return if 0, otherwise call move-data subroutine.
      :Load space at current cursor position.
      :Return.
 2002 (Shift-left-arrow routine - Delete character)
       Set "move-data" routine to LDIR mode.
      :Load "from" address.
      :Load "to" address.
      :Load number of bytes.
      :Return if 0, otherwise call move-data subroutine.
      :Load space at end of line.
      :Return.
 2003 (Shift-down-arrow routine - Insert line)
       Set "move-data" routine to LDDR mode.
      :Load "from" address.
      :Load "to" address.
      :Load number of bytes.
      :Return if 0, otherwise call move-data subroutine.
      :Clear current line.
      :Return.
 2004 (Shift-up-arrow routine - Delete line)
       Set "move-data" routine to LDIR mode.
      :Load "from" address.
      :Load "to" address.
      :Load number of bytes.
      :Return if 0, otherwise call move-data subroutine.
      :Clear bottom line.
      :Return.

 2010 (Move data subroutine)
       Define USR routine address as current base of US% array.
      :Call the "move-data" USR routine.
      :Return.

40600 (Await key depression and flash-cursor subroutine)
       Load A$ with character for key currently pressed, if any.
      :If a key was pressed then return,
       otherwise display cursor at current cursor position.
      :Re-display previous character at current cursor position.
      :Repeat line 40600.
```

Computing Video Display Positions

In lines 1001 through 1006 of the free-form video display program we used some unusual methods for computing PO%, the variable representing the PRINT@ position. Program line 1001 adds 1 to PO%, while enforcing a maximum of 1023.

The expression:

```
PO%=PO%-(PO%+1<1024)
```

. . . is really the same as:

```
PO%=PO%+1:IFPO%>1023THENPO%=1023
```

The video display computation chart gives you a list of 9 expressions for computing video display positions, based on the current position, PO%. For

Video Display Computation Chart

M 2 Note # 44

```
VIDEO DISPLAY COMPUTATIONS:

Integer PO% is the current position ranging from Ø to 1Ø23.

Space forward 1 position:
                   PO=PO-(PO+1<1Ø24)
Space back 1 position:
                   PO=PO+(PO-1>-1)
Move up 1 line, same column:
                   PO=PO+64*(PO-64>-1)
Move down 1 line, same column:
                   PO=PO-64*(PO+64<1Ø24)
Move to beginning of current line:
                   PO=INT(PO/64)*64
Move to beginning of next line:
                   PO=-((PO>=96Ø)*PO)-(PO<96Ø)*(INT(PO/64)*64+64)
Move to beginning of previous line:
                   PO=-((PO<64*PO)-(PO>=64)*(INT(PO/64)*64-64)
Move to top of screen, same column:
                   PO=PO-INT(PO/64)*64
Move to bottom of screen, same column:
                   PO=PO-INT(PO/64)*64+96Ø


(X,Y) expressions where X is the column ranging from Ø to 63,
and Y is the row, ranging from Ø to 15, and PO is the position,
ranging from Ø to 1Ø23.

When "X=Ø, Y=Ø" indicates the upper left corner:

Convert line Y, column X to PO:
                   PO=Y*64+X
Convert PO to column X and line Y:
                   X=PO-INT(PO/64)*64
                   Y=INT(PO/64)

When "X=Ø, Y=Ø" indicates the lower left corner:

Convert line Y, column X to PO:
                   PO=(15-Y)*64+X
Convert PO to column X and line Y:
                   X=PO-INT(PO/64)*64
                   Y=15-INT(PO/64)
```

applications where you might prefer to express video display print positions based on 'X' and 'Y' coordinates, the lower portion of the chart gives you a reference for conversions.

An Easy Way to Plan Video Displays

Are you tired of designing your video display layouts by trial and error? Here's a simple modification to the free-form video display routine that will turn it into a 'video display planner'. Add these two program lines:

M 2 Note # 30

```
121 PRINT@1Ø17,PO;
151 PRINT@1Ø17,CHR$(3Ø);
```

The video display planner lets you lay out your screen, while, in the lower right corner, the PRINT@ position is constantly indicated for each position that you may move your cursor. Just move the cursor to the first character of each planned print command and jot down the PRINT@ position.

You can also add the screen printer subroutine to get a hard-copy printout of your planned video display. Or, if you are using the NEWDOS disk operating system, just press JKL when you want a printout. With the Model 3 you can press shift-down-arrow-*.

Special Keys and Their Codes

Here's a list of the most important special keys found on the TRS-80 keyboard and the effect that you will get by printing the CHR$ function for the code generated:

Special Keys and Their Character Codes

KEY	CHR$()	PRINT CHR$()
Left-arrow	8	Backspaces and erases
Shift-left-arrow	24	Backspaces without erasing
Right-arrow	9	Space forward
Shift-right-arrow	25	Space forward without erasing
Enter	13	Line-feed and return to left Clears line below current line
Clear	31	Clears from current position to bottom of screen
Down-arrow	1Ø	Line-feed and return to left
Shift-down-arrow	26	Move down, same column
Up-arrow	91	Prints an up-arrow
Shift-up-arrow	27	Move up, same column

Shift-down-arrow, when combined with another key from A to Z, generates a character code from 1 to 26. On the Model 3 and the late Model 1's with the new ROM you will need to use shift-down-arrow-z to generate a CHR$(26).

Video Display Planning Sheets

This short program will print video display planning sheets for you on your line printer. Why buy planning sheets when you can make your own?

VSHEETS/BAS

Video Display Planning Sheets Program

M 2 Note # 50

```
Ø 'VSHEETS  - VIDEO DISPLAY PLANNING SHEET PRINTER
1Ø CLEAR1ØØØ
2Ø POKE16425,1     'SET LINE PRINTER
3Ø LPRINT" ";:FORX=ØTO63STEP2:LPRINTUSING"  ##";X;:NEXT:LPRINT"
":LPRINT" "
4Ø FORY=ØTO96ØSTEP64:LPRINTUSING"###";Y;:FORX=ØTO63:LPRINT"";CHR
$(95);:NEXT:LPRINT" ":LPRINT" ":NEXT
5Ø LPRINTCHR$(12)
```

String Graphics

For your convenience, Appendix 7 gives you the TRS-80 graphics characters. You'll find that it is often useful to load the graphics that you want to display into one or more strings. I often print 2 horizontal bars, 63 bytes long, to 'frame' my video displays. To do this, I use the command 'SG$=STRING$(63,131)' early in my programs. Then I just print SG$ when ever I want a horizontal bar.

You can also load a vertical graphics bar into a string and print it whenever and where required. Simply create a string that contains CHR$(170)+CHR$(24)+CHR$(26) up to 16 times. Here's a program line that sets up a vertical bar string, VB$, 10 positions 'high':

M 2 Note # 30

```
20 A$=CHR$(170)+CHR$(24)+CHR$(26):VB$="":FORX=1TO10:VB$=VB$+A$:NEXT
```

The CHR$(170) is a vertical bar graphics character. (You could use 149 or 191 instead.) The CHR$(24) backspaces without erasing and the CHR$(26) moves down one line in the same column.

Here's another trick. Suppose you want to clear the middle 10 lines of the screen without affecting the rest of the display. Simply print a string of 10 CHR$(13)'s:

```
PRINT@128,STRING$(10,13);
```

Refer to the chart showing the special keys and their character codes for more ideas on codes to insert in strings for graphics effects.

Alphanumeric Inkey Routine

This is a simple subroutine that provides an input field for the operator on the video display, allowing entry of a specified number of characters. It's called an inkey routine because it employs the INKEY function instead of LINEINPUT. It gives you the same capabilities as the standard LINEINPUT command, but with several major improvements:

1. The subroutine displays a string of underline characters on the screen to show the operator the field length and location.

2. Entry is limited to the field length. The operator can't ruin your display by typing too many characters.

3. You, the programmer, have control over the characters that may be typed. You can lock out or redefine the function of any key. You can prevent the screen-destroying effects of the clear key, the left-arrow key and the down-arrow key that can be a problem with LINEINPUT or INPUT. This subroutine also lets you, if you wish, limit input to upper case letters only, (a particularily helpful feature in applications where you will be sorting the data or using the entry as an access key to disk file records.)

4. Unlike INPUT and LINEINPUT, this subroutine does not generate a line-feed after enter is pressed. You have full control over your video display. You can pre-print information on the line below the data being entered without erasing it. You can allow typing on the bottom line of the video display without getting an unwanted scroll when the enter key is pressed.

5. You can set up one or more single key 'escapes' from the input routine. For example, you may wish to permit the operator to press the up-arrow key to abort the entry and return to the previous input field. You can also use keys other than the enter key as 'termination keys'.

The alphanumeric inkey subroutine occupies lines 40130 through 40139. The video display string pointer subroutine, 40070, must be present in your program if you want the result of the input to be loaded into the AN$ string. Upon calling the subroutine, just set PO% equal to the desired beginning position on the screen for the input, (0-1023) and load A1% with the desired length of the input, (1-255).

Line comments:

```
40130  Set count of characters entered (variable A%) to 0.
      :Print a string of (variable A1%) underline characters,
       starting at the beginning positon of the input field,
       specified by variable PO%.
40131  If the count of the characters entered equals the maximum
       number of characters permitted, go to 40134 to force entry
       of the enter, backspace, or any other special key,
       otherwise print an underline character at the current
       position.
40132  Check to see whether a key has been pressed.
      :If no key has been pressed, start line 40132 and check
       again, otherwise the result of the key depression is stored
       in A$.  If the key pressed represents a valid character
       then print it at the current position,
      :add 1 to the count of characters entered, and
      :go back to 40131.
40133  The key pressed does not represent a valid character, so
       check to see if it is a special key.  Based on its position
       in the list of special keys, go to the proper routine,
      :but, if it is not in the list of special keys, ignore this
       key depression and go back to line 40131.

40134 (We have reached the maximum number of characters, therefore
       we can only accept a special character)
       Load new key pressed, if any, into A$.
      :If no key was pressed, start line 40134 again,
       otherwise go back to line 40133 to see if it is a special
       key.

40135 (Process a backspace (CHR$(8)) key depression)
       If number of characters entered is less than the maximum
       then print an underline character at the current position.
40136  Subtract 1 from the count of characters entered.
      :If the subtraction gave us a negative number then restore
       the count back to zero and return to 40131 to accept another
       character,
       otherwise return to 40131 anyway.

40137 (Process those special characters for which we want to
       restore the count of characters entered back to zero
       before returning from the subroutine)
       Set count of characters entered back to zero, and fall
       through to line 40138.
40138  If the special character entered was an up-arrow, reprint
       the string of underline characters before returning,
       otherwise, fill the remaining positions of the field
       with spaces.

40139  Call the video display string pointer subroutine to load the
       data that was entered into the variable, AN$.
      :Return from the alpha-numeric inkey subroutine.
```

Alphanumeric Inkey Subroutine

M 2 Note # 30
M 2 Note # 43

```
40130 A%=0:PRINT@PO%,STRING$(A1%,95);

40131 IFA%=A1%THEN40134ELSEPRINT@PO%+A%,CHR$(95);

40132 A$=INKEY$:IFA$=""THEN40132ELSEIFINSTR(" !#$%&'()*+,-./0123
456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ",A$)THENPRINT@PO%+A%,A$;
:A%=A%+1:GOTO40131

40133 ONINSTR(CHR$(8)+CHR$(31)+CHR$(13)+CHR$(91),A$)GOTO40135,40
130,40138,40137:GOTO40131

40134 A$=INKEY$:IFA$=""THEN40134ELSE40133

40135 IFA%<A1%THENPRINT@PO%+A%,CHR$(95);

40136 A%=A%-1:IFA%<0THENA%=0:GOTO40131ELSE40131

40137 A%=0

40138 IFA$=CHR$(91)THENPRINT@PO%,STRING$(A1%,95);ELSEPRINT@PO%+A
%,STRING$(A1%-A%," ");

40139 GOSUB40070:RETURN
```

Alphanumeric Inkey Subroutine Modifications

Here are several modifications that you may want to make to the alphanumeric inkey subroutine:

1. On applications where you wish to create a 'fill in the blanks' form on the screen, it is helpful to provide an indicator that points to the current input field. I like to print a right-arrow in front of the field. A right-arrow can be displayed with CHR$(94) on the Model 1. On the Model 3, you can use CHR$(62). This is the modification:

To display the arrow, insert the following as the first command in line 40130:

```
PRINT@PO%-1,CHR$(94);:
```

To erase the arrow before returning from the subroutine, insert the following as the first command in line 40139:

```
PRINT@PO%-1," ";
```

Note that the arrow is printed at PO%-1. When you use this modification, PO% must be greater than 1 and you should avoid starting any input fields in the leftmost column of the screen.

2. There may be times when you will want to allow the operator to press a special character, either as an 'escape' key to be pressed instead of typing any data or as a 'termination' key, to be used as an alternative to the enter key. Here's how to make the subroutine recognize other special character keys:

Modify line 40133 to include the ASCII code in the list of special characters.

Modify line 40133 so that the ON GOTO command directs the program to the proper routine for the code you've added.

If you are adding a new termination key for the operator, the ON GOTO command should direct the program to line 40138, (the same path followed by the enter key logic.) Upon return from the subroutine, your program should analyze A$ for the termination key that was used. (AN$ will contain the data that was entered and A% will specify the length.)

If you are adding a new termination key for the operator, the ON GOTO command should direct the program to line 40138, (the same path followed by the ENTER key logic.) Upon return from the subroutine, your program should analyze A$ for the termination key that was used. (AN$ will contain the data that was entered and A% will specify the length.)

3. If you prefer 'boxes' instead of underline characters, replace all 95's in the subroutine with 132's.

4. The subroutine as shown will return the inputed data in AN$ with a length of A1%. If you want trailing spaces, if any, to be stripped, from the returned variable, AN$, insert the following command just before the 'RETURN' in line 40139:

```
AN$=LEFT$(AN$,A%):
```

5. The list of valid characters in line 40132 can be modified to include lower case characters also. Or you can replace the string of characters shown with a variable, making it possible for you to specify the valid character set elsewhere in your program.

Numeric Inkey Subroutine

The numeric inkey subroutine provides a video display input field for the operator, allowing entry of numeric data. It is much like the alphanumeric inkey routine, except:

Only the digits 0 through 9, the decimal, and '—' are accepted as input into the field. You can, however, set up special characters to be used as termination keys or escape keys.

Numeric Inkey Subroutine

M 2 Note # 30
M 2 Note # 43

```
40160 S%=1:AN$="":PRINT@PO%,STRING$(A1%,95);" ";

40161 A$=INKEY$:IFA$=""THEN40161ELSEIFINSTR("0123456789",A$)THEN
40162ELSEONINSTR(CHR$(8)+CHR$(31)+"."+"-"+CHR$(13)+CHR$(91),A$)G
OTO40160,40160,40165,40163,40166,40168:GOTO40161

40162 AN$=AN$+A$:IFLEN(AN$)>A1%THENAN$=LEFT$(AN$,A1%):GOTO40161E
LSEPRINT@PO%+A1%-LEN(AN$),AN$;:GOTO40161

40163 S%=-S%:PRINT@PO%+A1%,"";:IFS%=-1THENPRINT"-";ELSEPRINT" ";

40164 GOTO40161

40165 IFINSTR(AN$,".")=0THEN40162ELSE40161

40166 IFAN$=""THEN40168ELSEPRINT@PO%,STRING$(A1%-LEN(AN$)," ");

40167 IFS%=-1THENAN$="-"+AN$:GOTO40169ELSE40169

40168 IFA$=CHR$(91)THENPRINT@PO%,STRING$(A1%,95);" ";ELSEPRINT@P
O%,STRING$(A1%," ");" ";

40169 RETURN
```

As they are entered, the digits are shown on the screen 'calculator style'. That is, each new digit keyed is added at the rightmost position and all previous digits slide to the left.

Upon entry to the subroutine, A1% should specify the number of digits permitted, including decimal. One position beyond the input field is used to display the sign. The sign position is not included in the number of digits indicated by A1%.

Upon return from the subroutine, AN$ will contain the STR$ of the number entered. To use it as a number, simply use the VAL(AN$) function. If no digits were entered, AN$ will be null upon return from the subroutine.

Line comments:

```
40160  Set the sign indicator, (variable S%) to 1.
      :Clear the number string, (variable AN$).
      :Print a string of (variable A1%) underline characters,
       starting at the beginning position of the input field,
       specified by PO%  follow with a space to blank out the
       sign position.
40161  Check to see whether a key has been pressed.
      :If no key has been pressed, repeat line 40161, otherwise,
       if the key is a numeric digit, GOTO 40162, otherwise,
       check to see if the key is a special key.
       If it is a special key, go to the proper routine, otherwise,
      :repeat line 40161.
40162  The key pressed, now stored in A$, is a numeric or a decimal.
       Append the character onto the number string, AN$.
      :If the length of AN$ is now greater than the maximum number
       of digits requested, strip off the last character, and
      :go back to 40161 to await another key depression, otherwise,
       compute the position and redisplay the number string.
      :Go back to 40161 to await another key depression.

40163 (Change sign routine)
       Change the sign indicator, S%
      :Move the cursor to the sign position on the screen.
      :If S%= -1 then print a minus sign, otherwise,
       print a space.
40164 :Go back to 40161 to await another key depression.

40165 (Decimal processing routine)
       If the number string does not yet have a decimal in it, then
       goto 40162 to append the decimal to the number string,
       otherwise, go back to 40161 to await another key depression.

40166 (Termination key processing)
       If the number string is empty, go to 40168, otherwise
       erase any underline characters that may precede the number.
40167  If the sign is minus, add a minus sign to the number string
       and go to 40169, otherwise
       go to 40169 anyway.

40168 (Decide whether to leave spaces or underline characters)
       If the key pressed was an up-arrow, restore underlines.
       otherwise, leave spaces at the input field position.

40169  Return from the subroutine.
```

The numeric inkey subroutine occupies lines 40160 through 40169. Before calling the subroutine, just load PO% with the starting screen position and set A1% equal to the number of digits. Note that S% is used within the subroutine to keep track of the sign.

Numeric Inkey Subroutine Modifications

Here are several modifications that you may want to make to the numeric inkey subroutine:

1. To print a right-arrow that directs the operator's attention to the current input field and to erase the arrow after input is completed, make these changes:

To display the arrow, insert the following as the first command in line 40160:

```
PRINT@PO%-1,CHR$(94);:
```

To erase the arrow before returning from the subroutine, insert the following as the first command in line 40169:

```
PRINT@PO%-1," ";:
```

Note that the arrow is printed at PO%-1. When you use this modification, PO% must be greater than 1 and you should avoid starting any input fields in the leftmost column of the screen.

For the Model 3, you can use CHR$(62) instead of CHR$(94).

2. You can modify the subroutine to accept special characters to be used as escape or termination keys. The last character pressed is always returned from the subroutine as A$. The standard version of subroutine 40160 that is shown recognizes up-arrow as an escape key and the enter key as a termination key. Here's how to make the numeric inkey subroutine recognize other special characters:

Modify line 40161 to include the code for the special character you are adding.

Modify line 40161 so that the ON GOTO command directs the program to the proper routine for the code you've added. If you are adding a new termination key for the operator, the ON GOTO command should direct the program to line 40166. If you are adding a new escape key, the ON GOTO should direct the program to line 40168.

Modify line 40168 to control the input field display after the key is pressed. You can restore the string of underline characters or you can display blanks across the complete input field.

3. If you prefer 'boxes' instead of underline characters, replace all '95 's in the subroutine with '132''s.

4. You can change the minus sign display. (In accounting applications, you might want a 'CR' instead of the '–'). To make this change, modify line 40163. If your sign indicator is more than 1 character, you will also need to modify the subroutine every place where a space is displayed, increasing the number of spaces displayed to equal the length of the minus indicator.

Formatted Inkey Subroutine

This subroutine lets you give the operator a formatted 'template' for the entry of numeric dates, social security numbers and telephone numbers. You supply the format to the subroutine using a format string, AF$. The subroutine inserts the number entered, from left to right, filling in the blanks specified by the underline character, CHR$(95). Here are somesample format strings that can be used:

```
DATE:  __/__/__

      AF$=STRING$(2,95)+"/"+STRING$(2,95)+"/"+STRING$(2,95)

TELEPHONE NUMBER:  (___) ___-____

      AF$="("+STRING$(3,95)+") "+STRING$(3,95)+"-"+STRING$(4,95)

SOCIAL SECURITY NUMBER:  ___-__-____

      AF$=STRING$(3,95)+"-"+STRING$(2,95)+"-"+STRING$(4,95)
```

The formatted inkey subroutine enforces entry of numeric and special characters only, but you can modify it to allow alpha characters also. The clear key and the left-arrow key both allow the operator to erase the entry and start over. The enter key terminates the entry and the up-arrow operates as an escape key.

The result of the entry is returned from the subroutine in the string, AN$, without any formatting characters. If, for example, you are using a date format and the operator fills it in so that '06/15/81' is displayed, AN$ will contain '061581' upon return from the subroutine. An optional modification explained below will let you return the complete string, including format characters.

Before calling the subroutine, load AF$ with the desired format and set PO% to the starting position on the video display. A1% is automatically set to the length of the format string, AF$, within the subroutine.

Upon return, A% specifies the number of characters entered, AN$ contains the actual characters entered and A$ contains the character corresponding to the last key pressed.

Formatted Inkey Modifications

Here are several modifications that you may want to make to the formatted inkey subroutine:

1. To display a right-arrow on the screen to direct the operator's attention to the current input field and to erase the arrow after the entry is complete, make this change:

To display the arrow, insert the following as the first command in line 40150:

```
PRINT@PO%-1,CHR$(94);:
```

To erase the arrow before returning from the subroutine, insert the following as the first command in line 40159:

```
PRINT@PO%-1," ";:
```

Note that the arrow is printed at PO%−1. When you use this modification,

Formatted Inkey Subroutine

M 2 Note # 30
M 2 Note # 43

```
40150 AN$="":A%=0:PRINT@PO%,AF$;:A1%=LEN(AF$)

40151 IFA%>=LEN(AF$)THEN40156ELSEA%=INSTR(A%+1,AF$,CHR$(95)):PRI
NT@PO%+A%-1,"";

40152 A$=INKEY$:IFA$=""THEN40152ELSEIFINSTR("1234567890",A$)THEN
PRINTA$;:AN$=AN$+A$:GOTO40151

40153 ONINSTR(CHR$(8)+CHR$(31)+CHR$(13)+CHR$(91),A$)GOTO40150,40
150,40159,40158

40154 GOTO40151

40156 A$=INKEY$:IFA$=""THEN40156ELSE40153

40158 A%=0:AN$="":PRINT@PO%,AF$;

40159 RETURN
```

Line comments:

```
40150  Clear the entry-holding string, AN$.
      :Set entry position pointer, A%, to 0.
      :Print the format, AF$, at the desired position, PO%.
      :Set A1% equal to the length of the format string.
40151  If current position is greater than the length of the format
       string then go to 40156 to await entry of a special key,
       otherwise, set entry position pointer, A%, equal to the
       position of the next underline character.
      :Move the cursor to that position.
40152  Check to see whether a key has been pressed.
      :If no key has been pressed, start at line 40152 and check
       again, otherwise the result of the key depression is stored
       in A$.  If it is in the valid character string,
       then print it and append it to the entry-holding string, AN$.
      :Go back to 40151 to check for another character.
40153 (Special key processing)
       Check to see if it is a special key.  If it is, go to the
       proper line, otherwise,
40154  go back to 40151 to check for another character.

40156 (We have reached the maximum number of characters, therefore
       we can only accept a special character)
       Check to see if a key has been pressed.
      :If no key has been pressed, restart line 40156, otherwise
       go back to 40153 to see if it's a special character.

40158 (Process escape special characters)
       Clear the position pointer, A%.
      :Clear the entry-holding string, AN$.
      :Re-display the format string, AF$.
40159  Return from the formated inkey subroutine.
```

PO% must be greater than 1 and you should avoid starting any input fields in the left-most column of the screen.

For the Model 3, you can replace the CHR$(94) with CHR$(62).

2. You can modify the subroutine to accept special characters to be used as escape or termination keys. The last character pressed is always returned from the subroutine as A$. The standard version of subroutine 40150 that is shown recognizes up-arrow as an escape kty and the enter key as a termination key. Here's how to make the formatted inkey subroutine recognize other special characters:

Modify line 40153 to include the CHR$ code for the special character you wish to add.

Modify line 40153 so that the ON GOTO command directs the program to the proper routine for the code you've added. If you are adding a new termination key for the operator, the ON GOTO command should direct the program to line 40159. If you are adding a new escape, the ON GOTO command in line 40153 should direct theprogram to line 40158 so that the entry-holding string, AN$, is cleared and the format is redisplayed before returning.

3. If you prefer 'boxes' instead of underline characters, set up your format string, AF$, using '132' instead of '95'. Within the subroutine, replace all 95's with 132's.

4. If you want to allow entry of non-numeric characters, you can change line 40152 so that the valid character string includes all characters that you want the subroutine to accept. Or, you can replace '1234567890' with a string variable and set up the valid character set elsewhere in the program.

5. If you want to return the complete formatted input from the subroutine as AN$, rather than the numbers only, add the following command just before the 'RETURN' in line 40159:

```
GOSUB40070:
```

Be sure to include the video display string pointer subroutine, 40070, in your program.

The Dollar Inkey Subroutine

The dollar inkey subroutine provides an input field for the entry of dollars and cents. The amounts are entered in 'adding machine style'. As each new digit is entered, it is added at the rightmost position and all previous digits slide to the left, with the decimal point remaining 2 positions from the right.

Only the digits 0 through 9 and '–' are recognized as valid data entries. The enter key is used as the termination key and up-arrow is accepted as an escape key. You can, of course, add other termination and escape characters if you wish.

Upon entry to the subroutine, A1% specifies the length of the input field, including the decimal position, but not including the dollar sign or minus indicator. PO% specifies the starting position for the field, where the subroutine prints a '$'. The actual data starts at position PO% + 1.

Upon return from the subroutine, AN$ contains the STR$ of the dollar amount entered so that you can use the VAL(AN$) function. If no digits were entered, AN$ will have a length of 0. A$ contains the character corresponding to the last key pressed.

The dollar inkey subroutine occupies lines 40140 through 40149. S% is used within the subroutine to indicate whether the entry is positive or negative.

Line comments:

```
40140  Set sign indicator, (variable S%) to 1.
      :Clear the amount-holding string, (variable AN$).
      :Print a dollar sign and a string of (variable A1%) underline
       characters, starting at position PO%.
      :Print the decimal point.
40141  Check to see whether a key has been pressed.
      :If no key has been pressed, repeat line 40141, otherwise,
       if the key pressed is a number then go to 40143, otherwise,
       check to see if the key is a "-" or a special key.
       If it is within the list of special keys, go to the proper
       routine, otherwise,
40142  go back to 40141 to await another key depression.

40143 (Process a new digit entered)
       Add the digit onto the end of the amount-holding string, AN$.
       If the length of AN$ is now 1, then make the length 2 by
       adding a dummy underline character to the right side.
       If the length of AN$ is greater than the number of digits
       requested, then strip off the last digit.  Otherwise,
       if the length of AN$ is 3, and the dummy underline character
       is present as the first digit, strip it off.
40144  Print the new contents of the amount-holding string at the
       proper position, with the decimal inserted.
      :Go back to line 40141 for another key depression.

40145 (Change sign routine)
       Change the sign of indicator S%.
      :Move the cursor to the sign position.
      :If the sign is minus then print the minus sign, and
       return to line 40141 for another key depression, otherwise,
       print a space to blank out the minus sign, and
       return to line 40141 for another key depression.

40146  If an escape key was pressed, restore the input field
       underline characters, and
       restore the decimal, and
       go to 40149 to return from the subroutine, otherwise,
       for all other special characters, blank out the input field
       before going to 40149 to return.

40147  If no numeric keys were pressed, clear out the input field,
       and go to 40149 to return, otherwise,
       blank out the underline characters between the dollar sign
       and the first digit.
      :If the dummy underline character is present as the left-most
       digit, replace it with a "0" on the screen, and
      :replace it with a "0" in the amount-holding string, AN$.
40148  Insert the decimal in the amount-holding routine to prepare
       for a return from the subroutine.
      :If the sign is minus, add a "-" to the left side of the
       string.
40149  Return from the subroutine.
```

Dollar Inkey Subroutine

M 2 Note # 30
M 2 Note # 43

```
40140 S%=1:AN$="":PRINT@PO%,"$";STRING$(A1%,95);" ";:PRINT@PO%+A
1%-2,".";

40141 A$=INKEY$:IFA$=""THEN40141ELSEIFINSTR("0123456789",A$)THEN
40143ELSEONINSTR("-"+CHR$(8)+CHR$(31)+CHR$(13)+CHR$(91),A$)GOTO4
0145,40140,40140,40147,40146

40142 GOTO40141

40143 AN$=AN$+A$:IFLEN(AN$)=1THENAN$=CHR$(95)+AN$ELSEIFLEN(AN$)>
A1%-1THENAN$=LEFT$(AN$,A1%-1)ELSEIFLEN(AN$)=3ANDLEFT$(AN$,1)=CHR
$(95)THENAN$=RIGHT$(AN$,2)

40144 PRINT@PO%+A1%-LEN(AN$),LEFT$(AN$,LEN(AN$)-2);".";RIGHT$(AN
$,2);:GOTO40141

40145 S%=-S%:PRINT@PO%+A1%+1,"";:IFS%=-1THENPRINT"-";:GOTO40141E
LSEPRINT" ";:GOTO40141

40146 IFA$=CHR$(91)THENPRINT@PO%+1,STRING$(A1%,95);" ";:PRINT@PO
%+A1%-2,".";:GOTO40149ELSEPRINT@PO%,STRING$(A1%+2," ");:GOTO4014
9

40147 IFLEN(AN$)=0THENPRINT@PO%,STRING$(A1%+2," ");:GOTO40149ELS
EPRINT@PO%+1,STRING$(A1%-1-LEN(AN$)," ");:IFLEFT$(AN$,1)=CHR$(95
)THENPRINT@PO%+A1%-1,"0";:MID$(AN$,1,1)="0"

40148 AN$=MID$(AN$,1,LEN(AN$)-2)+"."+RIGHT$(AN$,2):IFS%=-1THENAN
$="-"+AN$

40149 RETURN
```

Dollar Inkey Subroutine Modifications

Here are some changes that you might want to make to the dollar inkey subroutine:

1. You can print a right-arrow to direct the operator's attention to the input field by adding commands to lines 40140 and 40149. The complete explanation for this change is given with the numeric inkey subroutine.

2. To add other escape or termination keys:

   Modify line 40143 to include the code for the special character you are adding.
   Modify line 40143 so that the ON GOTO command will direct the program logic to the proper routine. Escape keys should direct the logic to line 40146. Termination keys should direct the logic to line 40147.

If you have added an escape key, you can modify line 40146 to control whether the input field underline characters remain on the screen or the input area is replaced by spaces.

3. If you prefer 'boxes' instead of underline characters, replace all 95's in the subroutine with 132's.

4. If you want the numeric data in AN$ to be returned from the subroutine with an 'assumed' decimal, (no decimal inserted), delete the first command from line 40148.

5. If you want to display 'CR' instead of '–' for negative entries, change the PRINT"–" in line 40145 to PRINT"CR". Enlarge the " " in lines 40140, 40145 and 40146 to 2 spaces. Change the second '2' in line 40146 to a '3'. Change the first '2' in line 40147 to a '3'.

6. If you want the complete input field, including dollar sign and trailing 'minus' indicator to be returned in AN$, insert:

```
A1%=A1%+2:GOSUB40070:
```

. . . as the first command in line 40149. (Subroutine 40070, the video display string pointer subroutine, must be present.)

7. You may wish to remove the dollar sign. Simply change it to a space in line 40140.

Poking Graphics Into Program Text

This powerful technique can give you speed improvements in routines that display graphics and routines where you are scanning a list of special characters. For example, in the alphanumeric inkey routine, line 40133, we are scanning the string:

```
CHR$(8)+CHR$(31)+CHR$(13)+CHR$(91)
```

BASIC has to interpret and create the string each time we use it. To get greater speed, we can create a dummy string of 4 '*' characters in our program text, find the dummy string in memory and then poke an 8, 31, 13 and 91 into each position of the dummy string.

The LINEMOD/BAS utility program shown below gives you an easy way to poke into program text. Let's say you want to create the string VB$ in line 10, containing the graphics characters:

```
CHR$(170)+CHR$(24)+CHR$(26)+CHR$(170)+CHR$(24)+CHR$(26)
```

Here are the steps:

- Set up a dummy string in line 10 that reads:

```
VB$="******"
```

- Merge the LINEMOD/BAS utility into your program.
- Enter the command, 'RUN 64000', (without quotes.).
- The program will request the line number desired. Enter 10.
- The program will find the memory location of line 10.
- Press enter or down-arrow-enter until you see a 42 in the column labeled 'CONTENTS'. (42 is the ASCII code for '*').
- Type the 6 character codes you want to POKE, pressing enter after each. (170,24,26,170,24,26)
- Delete the LINEMOD/BAS utility, lines 64000 through 64059.

There are four things you should know before using the LINEMOD/BAS utility:

1. You will not be able to save the program on disk in ASCII format.
2. LISTing or LLISTing the modified line will usually give confusing results.
3. Always save your original program before using the LINEMOD utility to modify program lines.
4. Never poke a zero into program text. (Zero indicates 'end-of-line'. It will usually invalidate the internal text pointers, causing you to lose other program lines.)

Here's the utility:

LINEMOD/BAS
Program Text Poking Subroutine

M 2 Note # 30
M 2 Note # 16

```
64000 CLS:ML%=0:LF%=0:A%=0
64001 DEFFNIS!(A1%)=-((A1%<0)*(65536+A1%)+((A1%>=0)*A1%))
64002 DEFFNSI%(A1!)=-((A1!>32767)*(A1!-65536))-((A1!<32768)*A1!)
64010 PRINT@64,"";:INPUT"LINE NUMBER";LN!
64020 PRINT"SEARCHING...";
64021 POKEVARPTR(ML%),PEEK(16548):POKEVARPTR(ML%)+1,PEEK(16549)
64022 POKEVARPTR(LF%),PEEK(ML%+2):POKEVARPTR(LF%)+1,PEEK(ML%+3)
64023 LF!=FNIS!(LF%):PRINT@140,LF!;:IFLF!>LN!THEN64030ELSEIFLF=L
N!THEN64040
64024 POKEVARPTR(A%),PEEK(ML%):POKEVARPTR(A%)+1,PEEK(ML%+1)
64025 IFA%=0THEN64030ELSEML%=A%:GOTO64022
64030 PRINT@140," NOT FOUND"
64031 PRINT:LINEINPUT"PRESS <ENTER>...";A$:GOTO64000
64040 PRINT:PRINT"FOUND AT MEMORY LOCATION ";ML%
64041 PRINT@512,"PRESS <M><ENTER> TO BEGIN MODIFICATIONS...";:LI
NEINPUTA$:IFA$<>"M"THEN64000
64045 PRINT@512,CHR$(31);"MEM LOC     CONTENTS     CHANGE-TO":PR
INT@896,"<UP-ARROW><ENTER> = PREVIOUS       <DOWN-ARROW><ENTER> =
 NEXT
NEW CONTENTS<ENTER> TO CHANGE     <E><ENTER> TO END";
64050 PRINT@576,CHR$(30);USING"#######        ###";ML%,PEEK(ML%);
64055 PRINT@604,"";:LINEINPUTA$
64056 IFA$=CHR$(91)THENML%=FNSI%(FNIS!(ML%)+1):GOTO64050
64057 IFA$=""ORA$=CHR$(10)THENML%=FNSI%(FNIS!(ML%)+1):GOTO64050
64058 IFA$="E"THEN64000
64059 IFVAL(A$)<0ORVAL(A$)>255THEN64050ELSEPOKEML%,VAL(A$):ML%=F
NSI%(FNIS!(ML%)+1):GOTO64050
```

Saving Screens in Memory with Instant Recall

You'll be amazed at the speed at which you can save the current contents of the video display and then, flash the screen back with this subroutine. You simply reserve space in memory to hold 1024 contiguous bytes of video display data for each screen you want to save. This can be protected memory, reserved by your response to the MEMORY SIZE question or it can be an integer array, dimensioned with 512 elements for each screen you want to save and flash back.

The screen save and flashback subroutine employs the 'move-data magic array.' When we save a screen, we are simply moving 1024 bytes of data from memory location 15360 to another memory location. When we flash it back, we just reverse the 'from' and 'to' addresses.

Here's how to use the subroutine with an integer array for screen storage:

1. Your program must initialize variable J and the 'move-data magic array' early in your program.
2. Dimension an integer array, with 512 elements for each screen you'll be saving.
3. Set A$ equal to 'S' to save the current screen or 'D' to re-display a screen that is currently stored in memory.
4. Set SN% equal to the screen number.
5. Issue a 'GOSUB 40200' command.

Screen Save and Recall Subroutine

M 2 Note # 51

```
30 J=0:DIMUS%(7):US%(0)=8448:US%(2)=4352:US%(4)=256:US%(7)=201
40 DIMSS%(1023)

40200 DEFUSR=VARPTR(US%(0)):US%(5)=1023:US%(6)=-20243:IFA$="S"TH
ENUS%(1)=15360:US%(3)=VARPTR(SS%(SN%*512))ELSEUS%(1)=VARPTR(SS%(
SN%*512)):US%(3)=15360
40201 J=USR(0):RETURN
```

If you want to use a protected area of memory, rather than an array to save your screen, replace both occurrences of 'VARPTR(SS%(SN%*512))' in line 40200 with an integer expression indicating your memory storage area.

Here is a program that demonstrates the screen save and flash back subroutine. Type in the lines shown and merge lines 40200 and 40201 listed above.

FLASH/DEM
Screen Save and Recall Demonstration Program

M 2 Note # 30
M 2 Note # 51

```
1 CLEAR1000
30 J=0:DIMUS%(7):US%(0)=8448:US%(2)=4352:US%(4)=256:US%(7)=201
40 DIMSS%(1023)

100 'DISPLAY AND SAVE DEMO SCREEN 1
110 CLS:PRINT"
THIS IS SCREEN #1
";STRING$(64,131)
120 FORX=1TO64:PRINTUSING"####";X;:NEXT
130 PRINT:PRINTSTRING$(64,131)
140 PRINT@896,"PRESS <ENTER> TO FLASH TO SCREEN #2...";
150 SN%=0:A$="S":GOSUB40200

200 'DISPLAY AND SAVE DEMO SCREEN 2
210 CLS:PRINT"
THIS IS SCREEN #2
";STRING$(63,"*")
220 FORX=1TO10:PRINTTAB(X)STRING$(63-X,131):NEXT
240 PRINT@896,"PRESS <ENTER> TO FLASH TO SCREEN #1...";
260 SN%=1:A$="S":GOSUB40200

300 GOSUB40500:A$="D":IFSN%=1THENSN%=0ELSESN%=1
301 GOSUB40200:GOTO300

40200 'MERGE THE SCREEN SAVE AND FLASHBACK SUBROUTINE HERE

40500 A$=INKEY$:IFA$=""THEN40500ELSERETURN
```

You can, if you want, modify the screen save and flashback subroutine to save and flashback partial screens. Simply change '15360', where it appears, to the desired starting position ranging from 15360 to 16382 and '1023', where it appears, to the number of bytes to be saved, from 1 to 1023.

Swapping Screens

Here's a screen-swapping technique that you can use if you have two screens to alternate and you don't want to allocate a 1024-byte storage area for each. You just need one storage area of 1024 bytes.

The technique uses a 'swap-memory' magic array. You simply load the addresses of the two memory locations to be swapped, (one of which will be screen memory starting at 15360) and the number of bytes to swap. The elements of the swap-memory magic array are listed in line 20 of the demonstration program that follows. Before executing the magic array, element 1 is loaded with one address and element 3 is loaded with the other. Element 5 is loaded with the number of bytes to swap.

This demonstration program shows how we can swap between the top half of the screen and the bottom half:

SWAP/DEM
Swap Memory Demonstration Program

M 2 Note # 52

```
10 DIMUS%(10):J=0
20 DATA8448,0,4352,0,256,0,-4838,11168,9079,6368,247
30 FORX=0TO10:READUS%(X):NEXT

40 CLS:PRINT@0,"TOP HALF":FORX=1TO48:PRINTUSING"   ##    ";X;:NEX
T
41 PRINT@512,"BOTTOM HALF":FORX=1TO48:PRINT"   ";CHR$(48+X);"
 ";:NEXT

50 US%(1)=15360:US%(3)=15872:US%(5)=512
51 DEFUSR=VARPTR(US%(0)):J=USR(0)

70 FORX=1TO500:NEXT:GOTO51
```

Line comments:

```
10    Dimension the array to hold the swap-memory USR routine.
     :Initalize integer J.
20    Data to be loaded into the magic array.
30    Initialize the magic array.
40    Generate a demonstration screen.
41    Generate bottom half of demonstration screen.
50    Load swapping addresses and number of bytes to swap.
51    Call the USR routine.
70    Delay for viewing, then repeat from line 51.
```

Data Entry Handlers

To come up with an attractive, easy-to-use, and 'water-tight' system, you can easily spend 75 percent or more of your programming time on data entry. Once you've got good 'clean' information in the computer, processing the information, and printing it out is comparatively easy.

To provide good data entry, you want prompting messages to guide a new operator. But those prompting messages shouldn't slow down an experienced operator.

In addition you want data validation. With validation of entries, you can catch errors when they happen. Your job of processing the information becomes much simpler. For a really good entry program, you need to control each key that might be pressed by the operator. You've got to avoid the screen-destroying effects of the clear key, the down arrow key and the break key.

Finally, you need to provide consistent ways for the operator to correct entry errors. The operator should always be able to go back and correct the previous entry. Ignore this requirement and you've got built-in operator frustration!

The Horizontal I/O Subroutine

The horizontal input/output subroutine lets you easily input or display multi-column lists of data on the screen. It provides the computation of the PRINT@ positon and moves the cursor based on a count of the current row number (from 0 to 32767) and the horizontal tab specified. The screen illustrated below shows the type of data input and output problem that this subroutine solves:

```
LINE #  DESCRIPTION.............     QUANTITY
==================================================================
  1     NOTEBOOKS                           5
  2     TABLETS                            32
  3     PENCILS                            15
  4     PENS                               24
  5     ERASERS                            30
  6     REFILLS                            30
  7     RULERS                             22
  8     TEMPLATES                      ......


==================================================================
ENTER THE QUANTITY,
OR PRESS <UP-ARROW> TO RETURN TO THE DESCRIPTION COLUMN...
```

The need for the horizontal input/output subroutine arises from:

- **The fact** that a LINEINPUT or INPUT generates a line feed after you press the enter key. You can't just tab over to the next column during data entry if you are using 'normal' input methods. Many times you'll want to override this line feed.
- **The need** to provide the alphanumeric, numeric, formatted and dollar inkey routines presented in this book with a PRINT@ position. The horizontal input/output subroutine computes it for you.
- **A desire** to print prompting messages and error messages at the bottom of the screen, without disturbing the data-entry portion of the screen.

Here's the subroutine:

Horizontal Input Output Subroutine

```
40100 PO%=LI%+LT%+64*(LZ%-INT(LZ%/LV%)*LV%):IFPO%=LI%ANDLZ%>0THE
NPRINT@1000,"PRESS <ENTER>...";CHR$(30);:GOSUB40500:PRINT@1000,C
HR$(30);:PRINT@PO%,STRING$(LV%-1,13);
40101 PRINT@PO%,CHR$(30);:RETURN
```

Note that the horizontal input/output subroutine calls the single-key subroutine, 40500, when the data entry portion of the screen is filled. Subroutine 40500 must be present in your program.

Before using the subroutine, you must pre-load the following constants in your program:

```
LI%   Starting line PRINT@ position.
      (Example:  If you want the first data entry line to be the 4th
      line on the screen, you would use the command, LI%=192).

LV%   Number of vertical lines.
```

Example:

To display data at tab position 10 for the current line, LZ%, your command is:

```
LT=10:GOSUB40100:
```

This command is followed by your print or input command. When the screen is filled, the computer displays 'PRESS (ENTER)' in the bottom right corner of the video display. Press any key and the input/output portion of the screen will be cleared, with data entry resuming at the top line, specified by LI%.

To get a feel for the horizontal input/output subroutine, type in the following demonstration program and merge:

```
Lines 40100-40101   Horizontal input/output subroutine
Lines 40160-40169   Numeric inkey subroutine
Lines 40130-40139   Alphanumeric inkey subroutine
Line  40070         Video display string pointer subroutine
Line  40500         Single key subroutine
```

The horizontal input/output demonstration program provides input and output in the format illustrated at the beginning of this section.

HZIO/DEM
Horizontal Input Output Demonstration Program

M 2 Note # 30

```
0 'HZIO/DEM
1 CLEAR1000:DEFINTA-Z
3 DIMAR$(100),AR!(100)
4 SG$=STRING$(63,131)

100 CLS:PRINT@256,"
HORIZONTAL INPUT/OUTPUT SUBROUTINE DEMONSTRATION
";SG$;
110 PRINT"
<1> DATA ENTRY
<2> DATA RECALL
";SG$
190 PRINT@896,"PRESS THE NUMBER OF YOUR SELECTION...";
200 GOSUB40500:A%=INSTR("12",A$):IFA%=0THEN200ELSEONA%GOTO1000,2
000

1000 GOSUB30000
1010 LT=0:GOSUB40100:PRINTUSING"###";LZ+1;
1020 PRINT@896,"TYPE THE DESCRIPTION AND PRESS <ENTER>,
  OR PRESS <UP-ARROW> TO RETURN TO THE PROGRAM MENU...";
1021 LT=8:GOSUB40100:A1%=24:GOSUB40130:IFA$=CHR$(91)THEN100ELSEA
R$(LZ)=AN$
1030 PRINT@896,CHR$(31);"ENTER THE QUANTITY,
OR PRESS <UP-ARROW> TO RETURN TO THE DESCRIPTION COLUMN...";
1031 LT=36:GOSUB40100:A1%=6:GOSUB40160:IFA$=CHR$(91)THEN1020ELSE
AR!(LZ)=VAL(AN$)
1040 PRINT@896,CHR$(31);
1090 LZ=LZ+1:IFLZ>100THEN100ELSE1010

2000 GOSUB30000
2010 LT=0:GOSUB40100:IFA$=CHR$(91)THEN100ELSEPRINTUSING"###";LZ+
1;
2020 LT=8:GOSUB40100:PRINTAR$(LZ);
2030 LT=36:GOSUB40100:PRINTUSING"######";AR!(LZ);
2090 LZ=LZ+1:IFLZ>99THEN100ELSE2010

30000 CLS:PRINT"
LINE #  DESCRIPTION............   QUANTITY
";SG$;:PRINT@832,SG$;
30010 LI=192:LV=10:LZ=0:RETURN
```

Line comments:

```
Lines 0 - 4         Housekeeping
Lines 100-200       Menu
Lines 1000-1090     Input data into arrays AN$() and AN!()
Lines 2000-2090     Display data from array AN$() and AN!()
Lines 30000-30010   Print video display heading and load parameters
                    for the horizontal input subroutine.
```

Scrolling a Split Screen

The scroll-up subroutine lets you roll up, line by line, any area on the screen, while leaving the rest of the screen unscrolled.

This lets you, for instance, set up heading lines on the top of your screen and prompting lines on the bottom of your screen, while allowing operator input or displays of data, on the middle portion of the screen. Optionally, you can scroll the top portion only, the bottom portion only or the full screen, all under program control.

The scroll-up subroutine uses the 'move-data magic array' in LDIR mode, while providing all computations for PRINT@ positions.

Scroll-Up Subroutines

M 2 Note # 54

```
30 DIMUS%(7):US%(0)=8448:US%(2)=4352:US%(4)=256:US%(7)=201

40700 IFLZ>LV-1THENPL=LI+(LV-1)*64:PO=PL+LT:PRINT@PO,"";:RETURNE
LSEPL=LI+LZ*64:PO=PL+LT:PRINT@PO,"";:RETURN

40710 IFLZ<LVTHENGOSUB40700:RETURNELSEJ=0
40711 US%(1)=15424+LI:US%(3)=15360+LI:US%(5)=(LV-1)*64:US%(6)=-2
0243
40712 DEFUSR=VARPTR(US%(0)):J=USR(0):GOSUB40700:PRINT@PL+LT,CHR$
(30);:RETURN
```

Note that:

1. All variables within the scroll-up subroutines are integers. You should 'DEFINT' J, P and L early in your program or you may insert '%' after each variable in the subroutine.

2. The program must initialize the constants in the move-data magic array early in the program, before you call the scrolling subroutines. Line 30 shows how to do this, but you can use any line number.

3. Line 40700 is actually a variation on the horizontal input/output subroutine. It computes and prints at the desired position, based on the values you pre-load into the following variables:

Variables used:

```
LI%   Position of the top line of the scrolling area.
      (Example:  If you want to scroll the middle 10 lines of your
      screen, LI% would be 192.  If you want to scroll the top of
      your screen, LI% would be 0.

LV%   LV% is the number of vertical lines within the scrolling area
      of your screen.  LV% must be between 1 and 16.  (If you want to
      scroll the middle 10 lines of your screen, for example, LV%
      would be 10.

LZ%   LZ% is a count of the number of lines that have been displayed.
      LZ% starts at 0.  After displaying or inputting each line, add
      1 to LZ% and "GOSUB 40710".

LT%   LT% is the requested tab position, 0 to 63.  Before displaying
      data on a scrolling line, set LT% to the horizontal tab
      position and "GOSUB 40700".  PO% will be returned with the
      computed PRINT@ position, PL% will be returned containing the
      PRINT@ position of the beginning of the current line, and your
      cursor will have been moved to the desired printing position.
```

Lines 40710 through 40712 roll the scrolling portion of the screen up 1 line if more lines of data have been displayed than can fit within the scrolling portion. Add 1 to LZ% and 'GOSUB40710' when you want to input or display data on the next line.

The following short program demonstrates the scroll-up subroutines and how they are used. The program displays a fixed heading and footing on the screen and scrolls data in the middle 10 lines. You will need to add or merge lines 30, 40700 and 40710 through 40712 as shown above.

Scroll-Up Demonstration Program

M 2 Note # 30

M 2 Note # 54

```
0'SCROLLUP/DEM
1 CLEAR1000:DEFINTA-Z
4 SG$=STRING$(63,131)

1000 CLS
1001 PRINT"
 LINE #  DESCRIPTION...........    QUANTITY
";SG$
1002 PRINT@832,SG$;"
YOU MAY PRESS <UP-ARROW> TO END...";

1005 LI=192:LV=10:LZ=0

1010 LT=1:GOSUB40700:PRINTLZ+1;
1020 LT=8:GOSUB40700:PRINTSTRING$(24,RND(26)+65);
1030 LT=36:GOSUB40700:PRINTUSING"######";RND(10000);
1080 A$=INKEY$:IFA$=CHR$(91)THENPRINT@896,CHR$(31);:END
1090 LZ=LZ+1:GOSUB40710:GOTO1010
```

Note that:

- The houskeeping tasks are performed in lines 0 through 30.
- Lines 1000 through 1002 print the screen heading and footing.
- Line 1005 loads the scrolling parameters and sets the line count, LZ% to 0.
- The scrolling subroutines occupy lines 40700 through 40712.

After you've run the scroll-up demo program, you can try the following modifications:

To scroll the top portion only:
Delete line 1001.
Change line 1005 so that LI=0 and LV=13.

To scroll the bottom portion only:Add line 1001 again.
Delete line 1002.
Change line 1005 so that LI=192 and LV=13.

The Up-Down Scroller

The up-down scroller subroutine, 40800, provides a handler that you can use when you want to display data from arrays or disk files. The up and down arrow keys let the operator roll the data display up and down, line by line or continuously. You can specify any group of display lines as your scrolling area or you can use the whole screen.

To use the up-down scroller in a program:

1. Print the display heading and footing or clear the display.

2. Set up the scrolling parameters, LI% and LV%, using the rules explained in the section about scrolling up with a split screen. Set LT% and LZ% to zero to start.

3. Provide a subroutine that prints one line of display data (from your disk file or array), based on LZ%, the line counter. (Each print command in this subroutine must use the ';' option to avoid generating line feeds). This subroutine will be called by the up-down scroller subroutine.

4. Call the up-down scrolling subroutine, using the command, 'GOSUB40800'.

5. Provide logic to end the program or perform other functions after the up-down scrolling subroutine is exited.

The operator can view the data by using the arrow keys:

```
<Down-arrow>            Roll display down (toward end of data)
                        Repeat until key is released.
<Shift down-arrow>      Roll display down (toward end of data)
                        Repeat until another key is pressed.
                        (For Model 3, use <shift-down-arrow-Z>)
<Up-arrow>              Roll display up (toward beginning of data)
                        Repeat until key is released.
<Shift up-arrow>        Roll display up (toward beginning of data)
                        Repeat until another key is pressed.
<E>                     End the display   (return from subroutine)
```

Up-Down Scroller Subroutine

M 2 Note # 30

M 2 Note # 54

```
40800 GOSUB40500
40801 A%=INSTR("E"+CHR$(91)+CHR$(10)+CHR$(27)+CHR$(26),A$):ONA%G
OTO40802,40803,40804,40805,40806:GOTO40800
40802 RETURN
40803 GOSUB40820:IFPEEK(14591)>0THEN40803ELSE40800
40804 GOSUB40830:IFPEEK(14591)>0THEN40804ELSE40800
40805 GOSUB40820:A$=INKEY$:IFA$=""THEN40805ELSE40801
40806 GOSUB40830:A$=INKEY$:IFA$=""THEN40806ELSE40801
40820 IFLZ<=LVTHENRETURNELSELZ=LZ-1
40821 US%(1)=15360+LI+LV*64-65:US%(3)=US%(1)+64:US%(5)=(LV-1)*64
:US%(6)=-18195
40822 DEFUSR=VARPTR(US%(0)):J=USR(0):J=LZ:LZ=LZ-LV:PRINT@LI,CHR$
(30);
40823 GOSUB4000: 'CALL THE LINE DISPLAY ROUTINE
40824 LZ=J:RETURN

40830 IFLZ>LMTHENRETURNELSEGOSUB40710:PRINTCHR$(30);
40831 GOSUB4000:  'CALL THE LINE DISPLAY ROUTINE
40832 LZ=LZ+1:RETURN
```

For the Model 2, change each reference to "J=LZ" and "LZ=J" to "J1=LZ" and "LZ=J1", respectively. The lines affected are 40822, 40824, 40923, 40924, 40931, 40932, 40972, 40974.

Before you can use the up-down scroller subroutine these other subroutines must be present in your program:

```
40500        Single-key subroutine
40700-40712  Scroll-up subroutines
```

You also must preload the 'move-data magic array' early in your program. This line does the job:

```
30 US%(0)=8448:US%(2)=4352:US%(4)=256:US%(7)=201
```

You must modify the 'GOSUB4000' in lines 40823 and 40831 to call the subroutine you've provided for the purpose of displaying a line of data on the screen.

The 'UPDOWN/DEM' program demonstrates the up-down scroller subroutine. It creates random data and stores it in the arrays, AR$ and AR!. Then it allows you to display the data, rolling it up and down for viewing.

UPDOWN/DEM
Up-Down Scroller Demonstration Program

M 2 Note # 30
M 2 Note # 29

```
0 'UPDOWN/DEM
1 CLEAR1000:DEFINTA-Z
3 DIMAR$(49),AR!(49)
4 SG$=STRING$(63,131)
30 DIMUS%(7):US%(0)=8448:US%(2)=4352:US%(4)=256:US%(7)=201
1000 CLS
1001 PRINT"
 LINE #   DESCRIPTION..........   QUANTITY
";SG$;
1002 PRINT@832,SG$;"
CREATING TWO ARRAYS OF DEMONSTRATION DATA...";
1005 LI=192:LV=10:LZ=0
1006 FORLZ=0TO49:A$="":FORY=0TORND(14):A$=A$+CHR$(64+RND(26)):NE
XT:AR$(LZ)=A$:AR!(LZ)=RND(9999):LT=0:GOSUB40710:GOSUB4000:NEXT
1009 PRINT@896,CHR$(31);"PRESS <UP-ARROW> TO ROLL UP, <DOWN-ARRO
W> TO ROLL DOWN,
      <E> TO END THE DISPLAY....";
1010 LM=49:GOSUB40800
1020 PRINT@896,CHR$(31);:END
4000 PRINTUSING"###";LZ+1;
4020 PRINTTAB(8)AR$(LZ);
4030 PRINTTAB(36)USING"######";AR!(LZ);
4040 RETURN
```

Note that:

1. Lines 0 through 30 perform the housekeeping tasks.
2. Lines 90 through 91 load two arrays with data for the demonstration.
3. Lines 1000 through 1002 print the screen heading and footing.
4. Line 1005 loads the scrolling parameters.
5. Line 1010 calls the up-down scroller.
6. Lines 4000 through 4040 print a single line of data on the display. This is where you put the custom subroutine for your application.

You may wish to experiment with the program. You can change the scrolling parameters with the same modifications described for the scroll-up demo program. Simple modifications can specify scrolling on the upper screen lines only, the lower screen lines only or the whole screen.

A Scrolled Video Entry to Memory Handler

This video entry handler lets you design operator-friendly programs for the entry of transactions, lists or other line-oriented data. I've used variations on this subroutine in inventory transaction entry programs, invoicing programs and many others. The beauty of the handler is that you can use it by calling one subroutine. Here are the features of the scrolled video entry to memory handler:

- A portion of the screen is designated as a scrolling area. In most applications I scroll the middle 10 lines of the screen, using the top 2 lines for screen and column headings and the bottom 2 lines for operator prompting messages. I normally display a horizontal bar on the 3rd line and 14th line to frame the scrolling area.

- The operator enters data in columnar format. After each line of data, the scrolling portion of the screen, if full, is rolled up to allow entry of the next line. You the programmer, provide a subroutine which controls the entry of each field of data on the line, according to the special requirements of your application. You have full control over operator prompting and data validation.

- Instead of entering the next line of data, the operator may elect to perform special command functions by pressing up-arrow. Upon pressing the up-arrow key, a right-arrow 'pointer' is displayed in the leftmost column of the screen, pointing to the current line and a list of special commands is shown at the bottom of the screen. The special commands are:

<Up-arrow>	Rolls the display up to review previous line entries. Each depression of the up-arrow will move the pointer to the previous line. Holding the key down will provide a continuous upward scrolling until you release it. <Shift><Up-arrow> scrolls the display until any other key is pressed.
<Down-arrow>	Rolls the display down toward the last line entered. Each depression of the down-arrow will move the pointer to the next line, until the last line is reached. The continuous rolling functions operate as they do with the up-arrow.
<I>	Allows the insertion of a line of data at the position indicated by the pointer. All lines starting at the pointer and below are moved down to make room for the inserted line.
<D>	Allows the deletion of a line of data at the position indicated by the pointer. All lines below the pointer are moved up.
<L>	Loads a previously saved file from disk.
<S>	Saves the data that has been entered onto disk into the sequential file, "SAVEDATA/SEQ". (You may wish to change the file name, or to provide logic that allows operator entry of a file name.)
<R>	Resumes the data entry function, by rolling down, if necessary, to the line below the last line entered.
<E>	Ends the data entry functions, and returns control to the main program.

- Each line of data, when entered, is copied into a protected area of memory. You may specify that each line of data be from 1 to 63 characters. You also specify the maximum number of lines that may be entered. A prompting message is provided by the subroutine that informs the operator when the maximum has been reached.

For the Model 2, change each reference to "J=LZ" and "LZ=J" to "J1=LZ" and "LZ=J1", respectively. The lines affected are 40822, 40824, 40923, 40924, 40931, 40932, 40972, 40974.

Scrolled Video Entry to Memory Handler

M 2 Note # 55

```
4Ø9ØØ GOSUB3ØØØ
4Ø9Ø1 IFA$=CHR$(91)THENPRINT@PL,CHR$(3Ø);:GOSUB4Ø96Ø:GOSUB4Ø9Ø5:
IFA$="E"THENRETURNELSE4Ø9Ø3
4Ø9Ø2 GOSUB4Ø96Ø:LZ=LZ+1:LN=LZ:GOSUB4Ø71Ø
4Ø9Ø3 IFLN<LMTHEN4Ø9ØØELSEPRINT@896,CHR$(31);"LIMIT OF";LM;" ENT
RIES HAS BEEN REACHED.
PRESS <ENTER>...";:GOSUB4Ø5ØØ:A$=CHR$(91):GOTO4Ø9Ø1

4Ø9Ø5 PRINT@896,CHR$(31);"<";CHR$(91);">MOVE UP      <I>INSERT
 <L>LOAD FROM DISK    <R>RESUME
<";CHR$(92);">MOVE DOWN    <D>DELETE     <S>SAVE ON DISK       <E>E
ND";
4Ø91Ø GOSUB4Ø99Ø:GOSUB4Ø5ØØ:GOSUB4Ø991
4Ø911 A%=INSTR(CHR$(91)+CHR$(1Ø)+CHR$(27)+CHR$(26)+"RIDLSE",A$):
ONA%GOTO4Ø913,4Ø914,4Ø915,4Ø916,4Ø917,4Ø92Ø,4Ø93Ø,4Ø94Ø,4Ø95Ø,4Ø
912:GOTO4Ø91Ø
4Ø912 RETURN
4Ø913 GOSUB4Ø97Ø:IFPEEK(14591)>ØTHEN4Ø913ELSE4Ø91Ø

4Ø914 GOSUB4Ø991:GOSUB4Ø98Ø:GOSUB4Ø99Ø:IFPEEK(14591)>ØTHEN4Ø914E
LSE4Ø91Ø

4Ø915 GOSUB4Ø991:GOSUB4Ø97Ø:GOSUB4Ø99Ø:A$=INKEY$:IFA$=""THEN4Ø91
5ELSE4Ø911

4Ø916 GOSUB4Ø991:GOSUB4Ø98Ø:GOSUB4Ø99Ø:A$=INKEY$:IFA$=""THEN4Ø91
6ELSE4Ø911

4Ø917 IFLZ=LNTHENGOSUB4Ø991:RETURNELSEGOSUB4Ø98Ø:GOTO4Ø917
4Ø92Ø IFLN>=LMTHEN4Ø917ELSEGOSUB4Ø991:IFPL<>LI+LV*64-64THENUS%(1
)=1536Ø+LI+LV*64-65:US%(3)=US%(1)+64:US%(5)=(LI+LV*64-64)-PL:US%
(6)=-18195:DEFUSR=VARPTR(US%(Ø)):J=USR(Ø)
4Ø921 PRINT@PL,CHR$(3Ø);:GOSUB3ØØØ
4Ø922 IFA$<>CHR$(91)THEN4Ø925ELSEIFPL<>LI+LV*64-64THENUS%(1)=PL+
1536Ø+64:US%(3)=US%(1)-64:US%(6)=-2Ø243:DEFUSR=VARPTR(US%(Ø)):J=
USR(Ø)
4Ø923 J=LZ:A1%=PL:LZ=LZ+((LI+LV*64-64)-PL)/64:PL=LI+LV*64-64:IFL
Z>LNTHENPRINT@PL,CHR$(3Ø);ELSEGOSUB4Ø961
4Ø924 LZ=J:PL=A1%:GOTO4Ø9Ø5
4Ø925 US%(1)=LN*LE+LE+MB%:US%(3)=US%(1)+LE:US%(5)=(LN-LZ)*LE+LE:
US%(6)=-18195:DEFUSR=VARPTR(US%(Ø)):J=USR(Ø):LN=LN+1
4Ø926 GOSUB4Ø96Ø:GOSUB4Ø98Ø:GOTO4Ø9Ø5
4Ø93Ø IFLZ=LNTHEN4Ø91ØELSEIFPL<>LI+LV*64-64THENUS%(1)=PL%+15424:
US%(3)=US%(1)-64:US%(5)=(LI+LV*64-64)-PL:US%(6)=-2Ø243:DEFUSR=VA
RPTR(US%(Ø)):J=USR(Ø)
4Ø931 J=LZ:A1%=PL:LZ=LZ+((LI+LV*64)-PL)/64:PL=LI+LV*64-64:IFLZ>L
NTHENPRINT@PL,CHR$(3Ø);ELSEGOSUB4Ø961
4Ø932 LZ=J:PL=A1%:US%(1)=MB%+1+LZ*LE+LE:US%(3)=US%(1)-LE:US%(5)=
```

```
(LN-LZ)*LE:DEFUSR=VARPTR(US%(0)):J=USR(0):LN=LN-1:GOTO40910

40940 LZ=0:LN=0:PRINT@LI,CHR$(30);STRING$(LV-1,13);:PRINT@896,CH
R$(31);"LOADING FROM DISK...";
40941 ONERRORGOTO40947:OPEN"I",1,"SAVEDATA/SEQ:1":ONERRORGOTO0
40942 IFEOF(1)THEN40945ELSELINE INPUT#1,AN$
40943 LT=1:GOSUB40710:PRINTAN$;:GOSUB40960
40944 LZ=LZ+1:GOTO40942
40945 CLOSE1:LZ=LZ-1:LN=LZ:GOTO40905
40947 LZ=0:LN=0:RESUME40905

40950 LZ=0:PRINT@LI,CHR$(30);STRING$(LV-1,13);:PRINT@896,CHR$(31
);"SAVING ON DISK...";
40951 OPEN"O",1,"SAVEDATA/SEQ:1"
40952 LT=1:GOSUB40710:GOSUB40961:A1%=LE:GOSUB40070
40953 PRINT#1,AN$
40954 IFLZ=LNTHENCLOSE1:GOTO40905ELSELZ=LZ+1:GOTO40952

40960 US%(1)=PL+15361:US%(3)=LZ*LE+MB%+1:US%(5)=LE:US%(6)=-20243
:GOTO40962
40961 US%(1)=LZ*LE+MB%+1:US%(3)=PL+15361:US%(5)=LE:US%(6)=-20243
:GOTO40962
40962 A%=0:DEFUSR=VARPTR(US%(0)):A%=USR(0):RETURN

40970 GOSUB40991:LZ=(LZ-1)*-((LZ-1)>0):IFLZ<LV-1THEN40975
40971 US%(1)=15360+LI+LV*64-65:US%(3)=US%(1)+64:US%(5)=(LV-1)*64
:US%(6)=-18195
40972 DEFUSR=VARPTR(US%(0)):J=USR(0):J=LZ:LZ=LZ+1-LV:PL=LI
40973 GOSUB40961
40974 LZ=J
40975 GOSUB40990:RETURN

40980 LZ=LZ+1:IFLZ>LNTHENLZ=LN:RETURNELSEIFLZ<LVTHEN40982ELSEGOS
UB40711:PL=LI+LV*64-64
40981 GOSUB40961
40982 RETURN

40990 GOSUB40700:PRINT@PL,CHR$(94);:RETURN
40991 GOSUB40700:PRINT@PL," ";:RETURN
```

Line comments:

40900 Call the line entry subroutine starting at line 3000.
(You provide the line entry subroutine, customized according to your specific application)

40901 If, upon return from the line entry subroutine, A$ equals up-arrow then clear the current line,
:call subroutine 40960 to copy the cleared line to the memory storage area, and
:call subroutine 40905 to perform special command functions.
:If, upon return from the special command subroutine, A$ equals "E" then return to the main program, otherwise go to 40903.

40902 Upon return from the line entry subroutine, A$ was not equal to up-arrow, so call subroutine 40960 to copy the entered line to the memory storage area, and
:add 1 to the current line pointer, integer LZ, and
:set integer LN, the highest line indicator equal to LZ, and
:call subroutine 40710 to scroll up if necessary.

40903 If integer LN, the highest line indicator, is less than integer LM, the maximum permited line number then go back to line 40900 to get another entry.
Otherwise print a message at the bottom of the screen, indicating that the limit has been reached.
:Call subroutine 40500 to await depression of a key.
:Set A$ equal to up-arrow, and
:go to line 40901 to force a return to special command mode.

```
40905 (Special command selection menu).
       Print the special command menu on the bottom 2 lines of the
       screen.
40910  Call subroutine 40990 to display an arrow to point to the
       current line.
      :Call subroutine 40500 to await a key depression, the results
       of which will be returned as A$.
      :Now that a key has been pressed, call subroutine 40991 to
       erase the pointer arrow.
40911  Scan a list of valid characters for the character
       corresponding to the key that was pressed in A$.
      :A% contains the relative position within the valid character
       list.  Based on A%, go to the proper routine.
      :but if the key pressed wasn't a valid command character, go
       back to 40910 to force another key depression.

40912 (Process the "E" command - End)
       Return from the special function subroutine.

40913 (Process the up-arrow command - Move up)
       Call the scroll up subroutine, 40970.
      :If a key is still being pressed, then repeat line 40913,
       otherwise, go back to 40910 for another command.

40914 (Process the down-arrow command - Move down)
       Erase the arrow pointing to the current line.
      :Call the scroll down subroutine, 40980.
      :Re-display the pointer arrow at the (new) current line.
      :If a key is still being pressed, then repeat line 40914,
       otherwise, go back to 40910 for another command.
40915 (Process the shift up-arrow command - Continuous move up)
       Erase the arrow pointing to the current line.
      :Call the scroll up subroutine, 40970.
      :Re-display the pointer arrow at the (new) current line.
      :Load A$ with the code for the current key being pressed.
      :If A$ is null, then no key is being pressed.  Repeat 40915.
       Otherwise, go to 40911 and process the key depression as the
       next command.

40916 (Process the shift down-arrow command - Continuous move down)
       Perform same logic as in line 40915, except call subroutine
       40980 to scroll down.

40917 (Process the "R" command - Resume)
       If the current line is equal to the highest line, then erase
       the pointer arrow, and
      :return from the special command subroutine.
       Otherwise, call the scroll down subroutine, 40980, and
      :repeat line 40917.

40920 (Process the "I" command - Insert line)
       If number of lines entered is greater than or equal to the
       maximum number of lines allowed, abort the insertion by going
       to the resume routine, line 40917, otherwise
      :Erase the pointer arrow.
      :If the current line is not the last line on the scrolling
       portion of the screen, then load parameters into the move-data
       magic array.  Define it as a USR routine, and call it, to
       move down the video display data below the line to be
       inserted.
40921  Clear the current video display line.
       Call subroutine 3000 to allow entry of the line to be
       inserted.
40922  If A$ is not equal to up-arrow then go to line 40925.
       If A$ is an up-arrow, restore the data on the screen by moving
       it back up.
```

40923 Temporarily store the current line pointer as integer J.
:Temporarily store the position of the current line as A1%.
:Set current line pointer to the line at bottom of the screen.
:Set line position indicator, PL to point to last line of data entry area.
:If we are now (temporarily) beyond the last line entered, then clear the last line of the screen entry area, otherwise
:call subroutine 40961 to transfer the data back from memory to the screen.

40924 Restore the current line pointer, LZ.
:Restore the position pointer of the current line, PL.
:Go back to 40905 to await a special command.

40925 Load the move-data magic array with the parameters to move the data beyond the current line in the memory storage area, and call the routine to open up a space in memory for the insertion.
:Add 1 to LN to increment the highest line number.

40926 Call subroutine 40960 to move the inserted line from the screen to the newly created space in the memory storage area.
:Call subroutine 40980 to scroll up 1 line.
:Go back to 40905 to await a special command.

40930 (Process the "D" command - Delete line)
:If the current line is equal to the highest line then a delete is not necessary, so go back to 40910 to await another command.

40931 Temporarily store the current line pointer as integer J.
:Temporarily store the position pointer of the current line as A1%.
:Set current line pointer to the line at the bottom of the data entry area.
:Set line position indicator, PL to point to last line of the data entry area.
:If we are now (temporarily) beyond the last line entered, then clear the last line of the data entry area, otherwise call subroutine 40961 to transfer the next line back onto the screen.

40932 Restore the current line pointer, integer LZ.
:Restore the position pointer of the current line, PL.
:Set up the prameters in the move-data magic array and move the data in the memory storage area.
:Subtract 1 from the highest line indicator, integer LN.
:Go back to 40910 to await another special command.

40940 (Process the "L" command - Load from disk)

40950 (Process the "S" command - Save to disk)

40960 (Move a line from the screen to memory storage)
40961 (Move a line from memory storage to the screen)
40962 (Call the move-data USR routine to process the moves)

40970 (Move up - Scroll down subroutine.)
:Erase the pointer arrow if any.
:Subtract 1 from the current line pointer, enforcing a minimum result of zero.
:If the result is less than the number of lines in the scrolling portion of the screen then no scroll is necessary, so bypass the routine and go to 40975.

40971 Load from address, to address, and number of bytes into the move-data magic array.

40972 Call the move data USR routine.
:Temporarily store the current line pointer as integer J.
:Compute the line pointer for the top line of the scrolling area.

40973 Call subroutine 40961 to move data stored in memory to the top line of the video display scrolling area.

40974 Restore the current line pointer as integer LZ.
40975 Call subroutine 40990 to re-display the pointer arrow.
:Return

```
40980  (Move down - scroll up subroutine.)
      :Add 1 to the current line pointer.
      :If it is now greater than the number of lines entered, then
       set it equal to the number of lines entered and return.
       Otherwise, if its less than the number of lines in our
       scrolling area, then skip the scroll.
       Otherwise, call the scroll up subroutine, 40711,
      :and set the line position pointer, PL to the last line on the
       display.
40981  Call subroutine 40961 to move the line from memory storage to
       the screen.
40982  Return.

40990 (Display a pointer arrow to indicate the current line)
       Call subroutine 40700 to compute the position, PL, based on
       the current line pointer, LZ.
      :Print the arrow.
      :Return
40991 (Erase the pointer arrow from the current line)
       Same logic as line 40990, but a blank is printed.
```

How to Use the Scrolled Video Handler

1. Type-in or merge the scrolled video entry handler subroutine. It occupies lines 40900 through 40991.

2. Type-in or merge the following subroutines, as they are listed in this book:

```
40070           Video display string pointer subroutine
40500           Single-key subroutine
40700 - 40712   Scroll-up subroutines
40130 - 40139   Alphanumeric inkey routine (Optional)
40140 - 40149   Dollar inkey routine (Optional)
40150 - 40159   Formatted inkey routine (Optional)
40160 - 40169   Numeric inkey routine (Optional)
```

3. Decide on the length of your input line, ranging from 1 byte to 63 bytes. Decide on the limit of line entries that you will allow. You will need commands early in your program that specify the line length as variable LE% and the limit as variable LM%. For example, to allow entry of 100 lines, each having a length of 63, your commands are:

```
LE%=63:LM%=100
```

4. Multiply the line length by the limit. The result will be the amount of memory, in bytes, that you must reserve. Subtracting the amount of memory to be reserved from 65536 (for a 48K TRS-80) or 49152 (for a 32K TRS-80) gives you the maximum memory size you can specify upon going into BASIC from DOS READY. Or, if you wish you can insert logic to reserve the memory while in BASIC by following the instructions given in the section on 'how to change the memory size from BASIC'.

5. You will need to load the variable MB% early in your program. It specifies the beginning address of your memory storage area for lines that have been scrolled off the screen. Normally, you will want to use the upper-most area of RAM for your storage area. Let's assume you've got a 48K TRS-80 and you will be needing 100 lines of 63 bytes each. Your total storage area will be 6300 bytes, so you could use the command:

```
MB%=-6300
```

To specify 6300 bytes of storage at the top of a 32K TRS-80, your command is:

```
MB%=-22686
```

As you can see, we're just subtracting the number of bytes we'll require from the top memory address plus 1. (Therefore, we're subracting from 0 for a 48K TRS-80 or −16386 for a 32K TRS-80.)

6. You will need to load the contents of the move-data magic array early in your program. The handler assumes that you have used the US% array for this purpose. Your logic to do this, if you use line 30 is:

```
30 DIMUS%(7):US%(0)=8448:US%(2)=4352:US%(4)=256:US%(7)=201
```

7. You will need to provide program lines that display your video display 'frame', if any. This is done by clearing the screen and displaying the headings. You can display a horizontal bar just above and just below your planned scrolling area if you wish.

8. You will need a program line that specifies and initializes the scrolling parameters.

- LI% specifies the leftmost PRINT@ position of the first scrollable line. If, for example your scrolling area begins on the 3rd video display line, LI% will be 192.
- LV% specifies the number of lines in the scrolling area. If you want to scroll the middle 10 lines, LV% is specified as 10.
- LZ% and LN% should be initialized as zero. LZ%, during execution, contains the current line number. LN% contains the number of the highest line entered.

9. You will need a line that calls the video entry to memory subroutine. Upon return from the subroutine, you may wish to provide logic that ends the program. The following 3 commands do the job:

```
GOSUB40900 : CLS : END
```

10. You must provide a subroutine at line 3000 that handles the entry of one video display line. Within this subroutine, you should call the alphanumeric, numeric, dollar or formatted inkey routines for entry of data. (Or you should provide another method, so as to avoid a line feed after the input.)

To position to the correct column before each entry, you should set LT% to the tab position, from 1 to 63 and GOSUB 40700. Subroutine 40700 moves the cursor to the proper position, based on the line you are entering and it computes PO%, the PRINT@ position.

You should design your subroutine so that A$ will equal CHR$(91), the up-arrow character, upon return, if the operator has chosen to go into command mode. Upon return from your subroutine, A$ should not contain CHR$(91) if the operator wants to continue with entry of the next line.

You may begin your line entry subroutine at a line number other than 3000. To do so you must change the '3000' in line 40900 and 40921 to the line number you are using.

11. If you are using a Model 3, the up-arrow, down-arrow and right-arrow are not displayable characters. You may wish to replace the CHR$(91), CHR$(92) and CHR$(94) with other symbols.

12. The 'save' command that is provided stores the data, line by line, into a disk file. You can read the data back into any program for processing as a sequential file. Or, you can read it back into your data entry program with the 'load' command that is provided.

13. If you want to add a print-out capability from command mode, you can test on the entry of 'P' in line 40911, adding another line number to the 'ON GOTO' list. You can put your printing routine at any line, but it will look something like this:

```
5000 LZ=0:PRINTLI,CHR$(30);STRING$(LV-1,13);
5010 LT=1:GOSUB40710:GOSUB40961:A1%=LE:GOSUB40070
5020 LPRINTAN$
5030 IFLZ=LNTHEN40905ELSE LZ=LZ+1:GOTO5010
```

14. Many other modifications are possible, once you are familiar with the inner workings of the video entry to memory handler.

Video Entry Demo

VETOM/DEM is a program that demonstrates the scrolled video entry to memory handler. For the demonstration, we'll show a program that could be used as the basis for a disk file layout planner. VETOM/DEM lets you enter up to 100 lines of data. Each line has 4 entry columns. From each entry column, you can press the up-arrow key to go to the previous column. When you are in the first column, up-arrow takes you to command mode. In command mode, you can scroll up or down, insert or delete lines, save your entries to disk, load previous entries or end the program.

Shown below, is an example of the entry screen as it appears after 6 lines of data have been entered. The prompting message for entry of the first field of the 7th line is shown on the bottom 2 lines of the screen. The alphanumeric inkey subroutine has displayed 24 underline characters to show the operator how many characters can be typed:

```
  FIELD NAME..............  TYPE    VARIABLE     BYTES
=================================================================
  CUSTOMER NUMBER             A       FH(1)          6
  NAME                        A       FH(2)         24
  ADDRESS                     A       FH(3)         24
  CITY, STATE                 A       FH(4)         24
  ZIP CODE                    N       FH(5)          4
  TELEPHONE NUMBER            N       FH(6)         12
  ........................

=================================================================
ENTER A DESCRIPTION OF THE DATA FIELD,
OR PRESS <UP-ARROW> TO GO TO COMMAND MODE...
```

Shown below is the entry screen as it appears in command mode. The command menu is shown on the bottom 2 lines. In this example, you can see that more than 10 lines have been entered and the first 2 lines were scrolled off the top. The arrow in the left-most column is currently pointing to the line where the operator has typed 'PURCHASES TO DATE'. To delete that line, the operator could press 'D' at this point. Or with up-arrow or down arrow, the operator may roll up or down to insert or delete other lines.

```
 FIELD NAME.............  TYPE    VARIABLE     BYTES
===============================================================
 ADDRESS                   A        FH(3)        24
 CITY, STATE               A        FH(4)        24
 ZIP CODE                  N        FH(5)         4
 TELEPHONE NUMBER          N        FH(6)        12
 BEST HOURS TO CALL        A        FH(7)        10
 DATE OF LAST CONTACT      D        FH(8)         2
 LAST PAYMENT DATE         D        FH(9)         2
 BALANCE OWING             $        FH(10)        8
 AMOUNT PAST DUE           $        FH(11)        8
→PURCHASES TO DATE         $        FH(12)        8
===============================================================
<↑>MOVE UP    <I>INSERT     <L>LOAD FROM DISK        <R>RESUME
<↓>MOVE DOWN  <D>DELETE     <S>SAVE ON DISK          <E>END
```

To enter the VETOM/DEM program, you'll need the lines shown below in addition to the standard subroutines we've discussed. Lines 0 through 30 provide the program startup 'housekeeping'. Lines 1000 through 1010 print the video display 'frame' and set up the scrolling parameters. Lines 3000 through 3040 handle the input and prompting for the 4 entry columns. The pokes in line 1 automatically set up a memory size of 42852.

VETOM/DEM
Scrolled Video Entry to Memory Demonstration Program

M 2 Note # 30
M 2 Note # 55
M 2 Note # 56

```
0 'VETOM/DEM
1 POKE16561,100:POKE16562,167:CLEAR1000:DEFINTA-Z:J=0
2 LE=63:LM=100:MB=-22686
4 SG$=STRING$(63,131)
30 DIMUS%(7):US%(0)=8448:US%(2)=4352:US%(4)=256:US%(7)=201

1000 CLS
1001 PRINT"
 FIELD NAME.............  TYPE    VARIABLE     BYTES
";SG$;
1002 PRINT@832,SG$;
1005 LI=192:LV=10:LZ=0:LN=0
1010 GOSUB40900 :CLS:END

3000 PRINT@896,CHR$(31);"ENTER A DESCRIPTION OF THE DATA FIELD,
OR PRESS <UP-ARROW> TO GO TO COMMAND MODE...";
3001 LT=1:A1%=24:GOSUB40700:PRINTCHR$(30);:GOSUB40130:IFA$=CHR$(
91)THENRETURN

3010 TC$="<D>,<N>,<A>, OR <$>":PRINT@896,CHR$(31);"ENTER THE TYP
E-CODE, ";TC$;"
OR PRESS <UP-ARROW> TO RE-ENTER THE FIELD NAME...";
3011 LT=28:A1%=1:GOSUB40700:GOSUB40130:IFA$=CHR$(91)THEN3000
```

```
3020 PRINT@896,CHR$(31);"ENTER THE FIELD-VARIABLE TO BE USED,
OR PRESS <UP-ARROW> TO RE-ENTER THE TYPE CODE...";
3021 LT=35:A1%=6:GOSUB40700:GOSUB40130:IFA$=CHR$(91)THEN3010

3030 PRINT@896,CHR$(31);"ENTER THE NUMBER OF BYTES FOR THIS FIEL
D,
OR PRESS <UP-ARROW> TO RE-ENTER THE FIELD-VARIABLE...";
3031 LT=47:A1%=3:GOSUB40700:GOSUB40160:IFA$=CHR$(91)THEN3020
3032 IFVAL(AN$)>255THEN3030

3040 RETURN

40070 'MERGE VIDEO DISPLAY STRING POINTER SUBROUTINE HERE
40130 'MERGE ALPHA NUMERIC INKEY SUBROUTINE HERE
40160 'MERGE NUMERIC INKEY SUBROUTINE HERE
40500 'MERGE SINGLE-KEY SUBROUTINE HERE
40700 'MERGE SCROLL-UP SUBROUTINES HERE (MODIFY AS NOTED)
40900 'MERGE VIDEO ENTRY TO MEMORY SUBROUTINE HERE
```

Unscrolled Video Entry Handler

The unscrolled video entry handler is a set of powerful and flexible subroutines that control the entry of data to a formatted video display. The handler provides for:

- Display of fill-in-the-blanks input fields for enforced entry of alphanumeric, numeric or dollars and cents data. The capability for specially formatted fields for dates, telephone numbers or other special numeric data.
- Controlled operator entry to those input fields in any predefined sequence.
- Customized subroutines that you can call before any entry, (normally for operator prompting).
- Customized subroutines that you can call after any entry, (normally for data validation).
- Standardized input procedures that allow the operator to press the up-arrow key to go back to the previous input field.
- The creation of a string array containing the contents of the operator's entries. The array element to be used for any input field is under the programmer's control. The array elements to be used need not correspond to the sequence of input.
- The capability to automatically transfer the results of the input to disk file fields in any sequence. Automatic handling of MKI$, MKS$ and MKD$ conversions before the data is LSET into the disk fields. Optional automatic handling for user-customized data types.
- An optional 'redisplay' mode that handles the redisplay of alpha data from disk fields. The redisplay of compressed numeric or alpha data is under programmer control.
- A 'change' mode that lets the operator change the desired field. The up-arrow or down-arrow key is used to move to the field to be changed. By holding down the arrow key, the operator can quickly move to the desired field for changes.

- Programmer controlled capability to enter and exit the input, redisplay or forms sequence at any point. Ability to exit the input sequence based on the results of operator entries. Ability to skip input fields based on the results of operator entries.
- The capability to handle any number of input fields and any number of different screens.

To get a feel for the power of the unscrolled video entry handler, let's look at a sample screen that demonstrates many of its capabilities.

Normally, you'll want to start your program with a menu that lets the operator select the function to be performed. Upon entry to the video input and inquiry portion of the program, the operator sees a complete screen containing the 'fill in the blanks' input fields. This is illustrated as sample screen 1.

Sample Screen 1

```
ACCOUNT# =>......
================================================================

NAME:        ........................
ADDRESS:     ........................
CITY,ST:     ........................     ZIP:       .........

PHONE NO:    (...) ...-....               DATE:      ../../..

QUANTITY:    ......                       AMOUNT:    $....,..

================================================================
ENTER THE CUSTOMER ACCOUNT NUMBER,
     OR PRESS <UP-ARROW> TO RETURN TO THE MENU...
```

As you can see, a prompt that tells the operator what to do is displayed on the bottom two lines of the screen. Also, an arrow is pointing to the first input field, the customer account number. At this point, the operator may simply press the up-arrow key, which will allow return to the program menu or the customer account number may be entered.

Now, let's assume that the operator types the customer account number, 'A101' and presses enter. The video entry handler automatically calls a subroutine, provided by you, the programmer, that looks up the account number from a disk file. If the account is found, the data from disk is retrieved and displayed in the proper blanks. For now, though, let's look at the process that follows if the account is not found on disk. The video entry handler continues with the next input field and its prompting message, as illustrated by sample screen 2.

As you can see, the arrow is pointing to the 'NAME' field. At the bottom of the screen is a prompt telling the operator the options that are available. If an error was made on the account number, the operator can press the up-arrow key to go back. Otherwise, the name can be typed and a maximum length of 24 characters will be enforced.

Sample Screen 2

```
ACCOUNT#   A101
=============================================================

NAME:     =>.......................
ADDRESS:    .......................
CITY,ST:    .......................       ZIP:        .........

PHONE NO:   (...) ...-....                DATE:       ../../..

QUANTITY:   ......                        AMOUNT:     $....,..


=============================================================
ENTER THE CUSTOMER NAME,
     OR PRESS <UP-ARROW> TO RE-ENTER THE ACCOUNT NUMBER...
```

The process continues for each input field. The operator can always press up-arrow to go back. Repeated pressing of the up-arrow will take the operator all the way back to the menu.

When the operator gets down to the phone number and date fields, entry of numeric data is enforced. The data field automatically fills the phone number and date 'template' from left to right. At the date field, the operator is forced to enter a valid month and day number.

When the operator gets down to the quantity field, the numbers are filled in 'calculator style' from right to left and a decimal point may be used. In the dollar amount field, the numbers are filled in from right to left, 'adding machine style' and the decimal remains 2 places from the right.

After the operator has pressed enter for the last field, a final chance is provided to use the up-arrow key for corrections. Sample screen 3 illustrates the way the video display might appear after filling in all the fields:

Sample Screen 3

```
ACCOUNT#   A101
=============================================================

NAME:       ARTHUR ADAMS
ADDRESS:    12345 MAIN STREET
CITY,ST:    CENTERVILLE, CA               ZIP:        93293

PHONE NO:   (751) 123-5432                DATE:       04/25/81

QUANTITY:      241                        AMOUNT:     $ 321.32


=============================================================
PRESS ENTER TO RECORD,
     OR PRESS <UP-ARROW> TO MAKE CORRECTIONS...
```

At this point, pressing the up-arrow returns the operator to the the amount field. Repeated pressing of the up-arrow key would back-step through every entry.

If the operator views the data and decides that it has been entered correctly, the enter key can be pressed to record it onto disk. The video entry handler then takes the data, which is currently stored in a string array, converts it to disk storage format and puts it into the proper disk fields. Under program control, the new data may then be recorded onto the disk.

Then, the input fields, as they appear to the operator, are converted back to blanks, so that the video display again looks like sample screen 1, where the operation can be repeated.

Now, let's suppose that upon entry of an account number, the disk was searched and the record was found. At that point, the video entry handler, with the proper program commands, can exit from input mode and go into redisplay mode. Under redisplay mode, the alphanumeric data fields are retrieved from disk storage and printed at the proper positions on the video display. The other fields, which may require special formatting, are redisplayed with routines provided by the programmer, outside control of the video entry handler.

The resulting screen might look like sample screen 4:

Sample Screen 4

```
ACCOUNT#    W132
================================================================

NAME:       JOHN WILLIAMS
ADDRESS:    90900 OAK BLVD.
CITY,ST:    CENTERVILLE, CA                ZIP:       93233

PHONE NO:   (751) 987-6543                 DATE:      04/10/81

QUANTITY:      308                         AMOUNT:    $ 472.21


================================================================
PRESS <C> FOR CHANGES,
     OR JUST PRESS <ENTER> TO EXIT...
```

At this point, the operator may press enter, which will erase the data from the display, returning to the format illustrated by sample screen 1.

Or the operator may wish to change one or more fields on the display. Pressing the 'C' key puts the display in change mode. It will appear as illustrated by sample screen 5.

Sample Screen 5

```
ACCOUNT#    W132
================================================================

NAME:     =>JOHN WILLIAMS
ADDRESS:    90900 OAK BLVD.
CITY,ST:    CENTERVILLE, CA                ZIP:       93233

PHONE NO:   (751) 987-6543                 DATE:      04/10/81

QUANTITY:      308                         AMOUNT:    $ 472.21


================================================================
PRESS <C> TO CHANGE THE FIELD INDICATED BY THE "=>"
<UP-ARROW> OR <DOWN-ARROW> FOR ANOTHER FIELD, OR <E> TO END...
```

Notice that the pointer is to the left of the 'NAME' field. By the parameters that the programmer has given to the video entry handler, he has prevented changes to the account number.

At this point, the operator can press the down-arrow key once and the pointer will move to the left of the 'ADDRESS' field. Or, the operator can press the down-arrow continuously and the pointer will move past each field, until it is to the left of the field to be changed. If the pointer has moved past the desired field, the operator can press up-arrow to move back to it.

Let's assume the operator has moved the pointer to the date field. Upon depression of the 'C' key again, the screen will look like sample screen 6, and the date can be changed:

Sample Screen 6

```
ACCOUNT#    W132
================================================================

NAME:       JOHN WILLIAMS
ADDRESS:    90900 OAK BLVD.
CITY,ST:    CENTERVILLE, CA                 ZIP:       93233

PHONE NO:   (751) 987-6543                  DATE:     =>../../..

QUANTITY:      308                          AMOUNT:    $ 472.21

================================================================
 ENTER THE DATE OF LAST CONTACT...
```

Upon re-entry of the date, the operator can move the pointer to any other field for changes. If the operator moves the pointer up, past the first field or down, past the last field, the changes are transferred to the disk file fields. The operator may also end changes to the account by pressing the 'E' key.

After changes have been made, the operator may press 'C' again, to make more changes to the same account. Or, by pressing enter, the blank formatted screen illustrated as sample screen 1 will be shown. From that point the operator may enter another account number or press up-arrow to return to the menu.

The example we have discussed shows how the video entry handler can be used for disk file additions, inquiries and changes. You'll find, however, that it can be useful for any data input application where you have multiple fields to be entered and you want operator-oriented, validity enforced input.

Using the Unscrolled Entry Handler

The unscrolled video entry handler operates in conjunction with one or more of the inkey routines we've discussed. Depending on whether you'll need alphanumeric, numeric, dollars and cents format or specially formatted input, you will need to have the the following subroutine lines present in your program:

```
40130 - 40139  Alphanumeric inkey routine.
40140 - 40149  Dollar inkey routine.
40150 - 40159  Formatted inkey routine.
40160 - 40169  Numeric inkey routine.
```

The video entry handler occupies lines 46010 through 46064, but for many applications you won't be needing all capabilities, so we'll be mentioning groups of lines that can be deleted. Two other standard subroutines are required. They are:

```
40500          Single-key subroutine.
40070          Video display string pointer subroutine.
```

Your application program must define variables beginning with 'F' as strings. You can do this with the 'DEFSTR F' command. All other variables within the video entry handler and the standard subroutines it calls, are explicitly defined as integer or string with the '%' or '$' symbol.

Specifying Parameters

Your application program specifies the input fields and the sequence in which they are to be requested. The parameters for input are specified in one or more control strings that occupy the F9$ array. For simple input programs with 12 or fewer data fields, you'll probably only need F9$(0), but you can use up to F9$(99). Each string in the F9$ array contains 16 characters of information for each of up to 12 input fields. Each 16-character substring is separated by a comma.

To handle the input and inquiry for the sample screens we've been discussing, our program specified the parameters for the 9 input fields in line 60:

```
60 F9(0)="075A0060101$0101,267A0240202$0200,331A0240303$0300,395
A0240404$0400,431A0090505$0500,523F0000606$0600,559F0010707$0702
,651N0060808!0800,687$0070909!0900"
```

The data before the first comma specifies the parameters for entry of the first field. The second field's parameters follow the first comma. The third field's parameters follow the second comma and so forth. When handling any input field, the video display handler pulls out the current 17-byte substring of F9$(0) and stores it temporarily as the F9$ string.

Therefore, while processing input from the first field, our F9$ string was:

```
075A0060101$0101,
```

Looking at the illustration of sample screen 1, you'll see that the first field was the account number. The video entry handler interpreted the F9$ string to mean:

> 'At video display position 75, use the alphanumeric inkey subroutine for the entry of up to 6 characters, storing the results of the input in the F1$(1) string. When storing the data on disk, LSET it into the FH$(1) field as a normal ASCII string. Before the input, call prompting subroutine number 1. After the input, call validation subroutine number 1.

As required by the formatted inkey subroutine, 40150, each input position is specified as an underline character, CHR$(95). The video entry handler loads the specified format string into AF$ just before calling the formatted inkey subroutine. The 17-byte control substring for the date field was specified as follows:

```
559F0010707$0702,
```

You can see that formatted input was requested at position 559. The '001', following the 'F', told the handler to use the F2$(1) string as its format for the date.

Video Entry Handler F9$ Format

```
Bytes  1 - 3    Video display PRINT@ position
Byte   4        Entry type code, indicating the inkey subroutine to
                be used:

                A = Alphanumeric         (Subroutine 40130)
                $ = Dollars and cents    (Subroutine 40140)
                N = Numeric              (Subroutine 40160)
                F = Special Format       (Subroutine 40150)

Bytes  5 - 7    Input length (if type code is A, $, or N)
                Template string number (if type code is F)
Bytes  8 - 9    Disk file field number within FH$ array
Bytes 10 - 11   Entry array element number within F1$ array
Byte  12        Disk field type code:

                $ = Normal ASCII string
                % = MKI$ - compressed integer format
                ! = MKS$ - compressed single precision format
                # = MKD$ - compressed double precision format

Bytes 13 - 14   Prompting subroutine number
                (Called with ON GOSUB prior to input of the field)
Bytes 15 - 16   Validation subroutine number
                (Called with ON GOSUB after input of the field)
Byte  17        Comma (for separation)
```

Since the F2$(1) string was 8 bytes long, the input length for the date was 8 bytes. The '07' just before the '$' symbol told the handler to store the results of the input, ('04/25/81' in the case of sample screen 3), in F1$(7). The '$' symbol specified that the whole 8-byte string was to be LSET into disk field FH$(7) without any compression.

Notice that bytes 5 through 7 specify the input length. For type 'A', alphanumeric, the input length specifies the maximum number of characters that may be typed. For numeric and dollar format, the input length is specified as the number of digits including the decimal, but not including the sign. For formated input, type 'F', bytes 5 through 7 refer to the F2$ array, which contains each template string that will be required in the program. In our example, we have two special format fields, the telephone number and the date. To handle these, F2$(0) and F2$(1) were used:

```
F2(0)="("+STRING$(3,95)+")  "+STRING$(3,95)+"-"+STRING$(4,95)
F2(1)=STRING$(2,95)+"/"+STRING$(2,95)+"/"+STRING$(2,95)
```

Prompting Subroutines

The '0702' in the F9$ string for the date field specified that prompting subroutine 7 was to be used, with validation subroutine 2. The prompting and validation subroutines are custom programmed for each application. They are numbered based on the way you set up an ON GOTO command within 2 subroutines you provide. You provide subroutine 25000 to handle your prompting subroutines. You may wish to use line 25000 to clear a prompting area on the bottom 2 lines of the screen:

```
25000 PRINT@896,CHR$(31);
```

Then you can use line 25001 for your ON GOTO list:

```
25001 ONVAL(MID$(F9,13,2))GOTO25010,25020,25030,25040,25050,2506
0,25070,25080,25090
```

Then at line 25010 you have prompting subroutine 1, at 25020 you have prompting subroutine 2 and so forth. Prompting subroutine 7 in our example was simply:

```
25070 PRINT"ENTER THE DATE OF LAST CONTACT,
OR PRESS <UP-ARROW> TO RE-ENTER THE TELEPHONE NUMBER...";:RETURN
```

Validation Subroutines

You'll need to provide subroutine 26000 to handle your data validation. For convenience, we'll refer to any subroutine that follows the input of a field, as a 'data validation' subroutine. In practice though, you may wish to take actions other than data validation after the entry of a field. Line 26000 contains your ON GOTO list:

```
26000 FE$="":ONVAL(MID$(F9,15,2))GOTO26010,26020
```

In our example, we used validation subroutine 2 for the date entry field. Since our ON GOTO list in 26000 directs the logic to 26020 for validation subroutine 2, our validation logic is found starting at line 26020:

```
26020 IF ASC(F1(7))=95THENF1(7)="00/00/00":PRINTPO%,F1(7);
26021 IF MID$(F1(7),1,2)>"12"ORMID$(F1(7),4,2)>"31"THENFE="X"
26022 RETURN
```

In this case, line 26020 checks the first byte of the date that was entered. If it is still an underline character, 95, no date was entered and the date '00/00/00' is automatically replaced.

Line 26021 checks the month and day. If an invalid month or day is found, it sets FE$="X" before the return. FE$ is a special string that is used by the handler in interpreting the results of the validation subroutines. If a validation subroutine sets FE$ equal to 'X', the handler forces the operator to re-enter the current field.

If a validation subroutine sets FE$="E", the handler ends input processing at that point and returns control to your mainline program. After the first input field of our example, (the account number), we used this method. Validation subroutine 1 searched the disk for the account number that was entered. If it was found, the disk was accessed, FE$ was set equal to 'E' and the input was terminated so that the existing data from disk could be displayed. If the account number was not found, FE$ remained a null string and input continued with the second field.

Video Entry Handler Commands

Your program always enters the video display handler with a 'GOSUB 46010' command. Before entering the handler, though, you must load the command string, FX$, with your handler command. FX$ is a 9-byte string, in the following format:

Video Entry Handler F3$ Format

```
Byte 1         Command code:

                F = "Forms" mode
                N = "New" mode
                C = "Change" mode
                W = "Write-to-disk-fields" mode
                R = "Redisplay-from-disk-fields" mode

Bytes 2 - 3     Parameter string number (from the F9$ array.)
Bytes 4 - 5     First field number (1 through 12) of the parameter
                string.  This specifies the first of a range of input
                fields.
Bytes 6 - 7     Last field number (1 through 12) of the parameter
                string.  This specifies the last of a range of input
                fields.
Bytes 8 - 9     Starting field number (1 through 12) of the parameter
                string, within the range specified.
```

The 'Forms' Command

The first handler command that was executed in our example was a 'forms' command:

```
FX="F00010901":GOSUB46010
```

The effect of this command was to display the input fields as underline characters. The '00' following the 'F' told the handler to refer to our F9$(0) parameter string. The '0109' told the handler to generate input areas on the screen for parameter substrings 1 through 9 of our F9$(0) parameter string. The final '01' told the handler to start with parameter number 1, within the range 1 through 9 that was specified.

The 'New' Command

The second handler command that was executed in our example was a 'new' command:

```
FX="N00010901":GOSUB46010
```

The effect of this command was to allow input to fields 1 through 9, as specified by the F9$(0) parameter string, starting at field 1. Following this command, our mainline program tested the contents of FE$. If FE$ was equal to 'E', our program knew that the operator entered an account number that was found on disk, so we branched to another part of our program to handle the redisplay of the data. If FE$ was not equal to 'E' upon return from the handler, our program knew that the operator entered all 9 input fields.

You'll remember that, after entry of the last field, we gave the operator a final chance to use the up-arrow key to make corrections. This was done by displaying the prompt:

```
"PRESS ENTER TO RECORD,
    OR PRESS <UP-ARROW> TO MAKE CORRECTIONS..."
```

At that point within our program, we called the single-key subroutine, 40500, to let the operator respond. The single-key waits for the operator to press a key and returns with A$ equal to the code corresponding to the key. If A$ was equal to CHR$(91), the up-arrow code, we re-executed a 'new' command:

```
FX="N00010909"
```

This time, however, the starting field number was 9, our last input field. The effect was to resume the original 'new' command, but to start with the last input field instead of the first.

The Write to Disk Fields

When the operator pressed ENTER to record, we executed a 'write to disk fields' handler command:

```
FX="W00010901"
```

The action taken by the handler in response to this command was to take the input, stored in array elements F1$(1) through F1$(9) and LSET it into the disk fields, FH$(1) through FH$(9), according to parameter string, F9$(0). Each field was LSET according its disk field type code in the parameter string. The first 7 fields had a type code of '$', so for fields 1 through 7, the handler LSET the FH$ array element equal to the corresponding F1$ array element. Fields 8 and 9 had a type code of '!'. For fields 8 and 9, the handler LSET the requested FH$ array element equal to the MKS$ of the VAL of the corresponding F1$ array element.

For each input field in our example, the F1$ array element was transfered to the same element number of FH$ array. F1$(1) was LSET into FH$(2), F1$(2) was LSET into FH$(2) and so forth. It's important to note, though, that the handler doesn't require a one-to-one correspondence. Bytes 8-9 of the 17-byte parameter substring specify the FH$ element number, while bytes 10-11 specify the F1$ element number. They don't have to be the same.

The Redisplay Fields Command

When the operator entered a valid account number that was found on disk, a different sequence of events occurred. After entry of the account number, validation subroutine 1 loaded FE$ with 'E'. This told the handler to abort input processing and return control to the main program. Upon receiving FE$ equal to 'E', the mainline program branched to its redisplay routines. The command given to the handler was:

```
FX="R00020902":GOSUB46010
```

This caused the handler to display the alphanumeric data from disk fields FH$(2) through FH$(7) at the proper PRINT@ positions, as specified by the parameter string F9$(0). We started at field 2 because the account number was already on the screen. The 'R' handler command only redisplays disk field data with a type code of '$'. That's why only fields 2 through 7 were automatically redisplayed. It was up to the mainline program to redisplay fields 8 and 9, because they had a type code of '!'. The mainline program displayed fields 8 and 9 with the commands:

```
PRINT@651,USING"######-";CVS(FH(8));:
PRINT@687,USING"$####.##-";CVS(FH(9));
```

The 'Change' Command

After all the data from the disk record was displayed, you'll remember that the following prompt was provided for the operator:

```
"PRESS <C> FOR CHANGES,
      OR JUST PRESS <ENTER> TO EXIT..."
```

At this point, the single-key subroutine, 40500, was called to let the operator respond. If the 'C' key was pressed for changes, the mainline program called the handler in 'change' mode:

```
FX="C00020902":GOSUB46010
```

Upon receiving this command, the handler allowed the operator to move to the desired fields for changes with the up and down arrows. Note that the range specified by the command was 2 through 9, starting at field 2. This range specification prevented changes to field 1, the account number.

The 'change' command has a built-in 'write to disk fields' command. After the last change, only those fields that were modified are LSET into the corresponding disk fields, according to the parameters specified by the F9$ string.

You should be aware that upon return from the 'change' command, each element of the F1$ array, in the range specified, will be null, unless a change was made to the field. If a change was made to a field, the corresponding F1$ element will contain the new contents.

Upon return from the handler's change mode, the mainline program issued a PUT command to record the changes to disk. All disk file PUT and GET commands are the responsibility of the mainline program.

Handling More Than 12 Fields

Since the parameter substring for each input field requires 17 bytes, a F9$ array element can provide the specifications for up to 12 fields. We can handle more than 12 fields by issuing multiple calls to the video entry handler. When issuing multiple calls, it is helpful to know the way in which input was terminated. The A$ string tells us. If A$ equals CHR$(91) after a GOSUB 46010 in 'new' or 'change' mode, the operator pressed 'up-arrow' instead of entering the first field. If A$ equals CHR$(255) after a call to the handler in 'new' or 'change' mode, the operator went through the last input field. Here's how a 20-field input sequence could be called from your mainline program:

```
1000 FX="N00011201
1010 GOSUB46010 : IFA$=CHR$(91)THEN100
1020 FX="N01010801
1030 GOSUB46010 : IFA$=CHR$(91)THEN FX="N00011212":GOTO1010
1040 PRINT@896,CHR$(31);"PRESS <UP-ARROW> FOR CORRECTIONS..."
1050 GOSUB40500 : IFA$=CHR$(91)THEN FX="N01010808":GOTO1030
```

You can see that the video entry handler was called for two different parameter strings, F9$(0) and F9$(1). F9$(0) contained the first 12 field parameters and F9$(1) specified the parameters for the last 8 fields.

Line 1010 calls the handler for entry of the first 12 fields. If up-arrow was pressed instead of entering the first field, the logic is directed back to a menu routine at line 100.

Line 1030 calls the handler for entry of the last 8 fields. If up-arrow is pressed in the first field of the last group, the logic goes back to line 1010, but the command in FX$ now specifies that field 12 is the starting point.

Lines 1040 and 1040 provide the operator with a chance to make corrections. The up-arrow key may be pressed to go back to the last field of the last group.

The 'change' logic for the same 20 fields could be organized as shown below:

```
1600 FX="C00011201
1610 GOSUB46010 : IFA$<>CHR$(255)THEN1690
1620 FX="C01010801
1630 GOSUB46010 : IFA$=CHR$(91)THEN FX="C00011212":GOTO1610
1690 PUT PF%,PR(PF%)
```

In line 1610 we are checking on the contents of A$ after changes to the first group of 12 fields. If A$ is equal to CHR$(255) we know that the operator changed the 12th field or press down-arrow at the 12th field. If A$ is equal to CHR$(91) or 'E', we know that the operator pressed up-arrow or 'E' to exit the changes.

In line 1690 we provide the logic to record the changes to disk.

It's a simple matter to use the other handler commands, 'F', 'W' and 'R', when you have more than 12 fields. Here, for example, is how you might display the 20 input fields with the 'F' command:

```
FX="F00011201 : GOSUB46010 : FX="F01010801" : GOSUB46010
```

Required Program Lines

The unscrolled video entry handler occupies lines 46010 through 46064 of your program. It requires about 1680 bytes. The following lines may be deleted, depending on the requirements of your application program:

```
Lines 46020 - 46029  if you don't need the "R" command.
Lines 46060 - 46064  if you don't need the "F" command.
Lines 46040 - 46041  if you don't need the "C" command.
Lines 46042 - 46059  if you don't need the "W" command.
```

If you delete the lines for the 'W' command, but you require the 'C' command, you should insert the following line:

```
46042 RETURN
```

A study of the unscrolled video handler listing and the line comments for it will reveal other minor deletions you can make when certain capabilities are not required.

Since the Model 2 has an automatic repeat key, you should delete the reference to PEEK(14591). From line 46031 delete: `ELSEIFPEEK(14591)>0THEN46033`

Unscrolled Video Entry Handler

M 2 Note # 30

M 2 Note # 57

```
46010 A$="":F9%=VAL(MID$(FX,2,2)):F7%=VAL(MID$(FX,4,2)):F8%=VAL(
MID$(FX,6,2)):F7%=(F7%-1)*17+1:F8%=(F8%-1)*17+1:F6%=VAL(MID$(FX,
8,2)):F6%=(F6%-1)*17+1
46011 ONINSTR("FNCWR",LEFT$(FX,1))GOTO46060,46030,46040,46042,46
020

46020 FORF4%=F7%TOF8%STEP17:F3=MID$(F9(F9%),F4%+11,1):IFF3<>"$"T
HEN46029
46021 PO%=VAL(MID$(F9(F9%),F4%,3)):A1%=VAL(MID$(F9(F9%),F4%+7,2)
)
46022 PRINT@PO%,FH(A1%);
46029 NEXT:RETURN

46030 IFF6%<F7%THENRETURNELSEF9=MID$(F9(F9%),F6%,17):F3=MID$(F9,
4,1):A1%=VAL(MID$(F9,5,3)):PO%=VAL(MID$(F9,1,3)):IFF3="F"THENAF$
=F2(A1%)
46031 PRINT@PO%-2,"=>";:IFLEFT$(FX,1)<>"C"THEN46034ELSEIFPEEK(14
591)>0THEN46033
46032 PRINT@896,CHR$(31);"PRESS <C> TO CHANGE THE FIELD INDICATE
D BY THE ";CHR$(34);"=>";CHR$(34);"
<UP-ARROW> OR <DOWN-ARROW> FOR ANOTHER FIELD, OR <E> TO END...";
:GOSUB40500
46033 IFA$=CHR$(91)ORA$=CHR$(10)THEN46035ELSEIFA$="E"THENPRINT@P
O%-2,"  ";:RETURNELSEIFA$<>"C"THEN46032
46034 GOSUB25000:ONINSTR("A$FN",F3)GOSUB40130,40140,40150,40160:
IFLEFT$(FX,1)="C"ANDA$=CHR$(91)THEN46034
46035 PRINT@PO%-2,"  ";:IFA$=CHR$(91)THENF6%=F6%-17:GOTO46030ELSE
IFA$=CHR$(10)THEN46038
46036 IFINSTR("F",F3)THENGOSUB40070
46037 F1(VAL(MID$(F9,10,2)))=AN$:GOSUB26000:IFFE="X"THENPRINT@PO
%-2,"=>";:GOTO46034ELSEIFFE="E"THENRETURN
46038 F6%=F6%+17
46039 IFF6%>F8%THENA$=CHR$(255):RETURNELSE46030

46040 FORF4%=F7%TOF8%STEP17:F1(VAL(MID$(F9(F9%),F4%+9,2)))="":NE
XT
46041 GOSUB46030

46042 FORF4%=F7%TOF8%STEP17:A%=VAL(MID$(F9(F9%),F4%+9,2)):IFLEFT
$(FX,1)="C"ANDF1(A%)=""THEN46059
46043 A1%=VAL(MID$(F9(F9%),F4%+7,2)):F3=MID$(F9(F9%),F4%+11,1)
46050 ONINSTR("$%!#",F3)GOTO46051,46052,46053,46054
46051 LSETFH(A1%)=F1(A%):GOTO46059
46052 LSETFH(A1%)=MKI$(VAL(F1(A%))):GOTO46059
46053 LSETFH(A1%)=MKS$(VAL(F1(A%))):GOTO46059
46054 LSETFH(A1%)=MKD$(VAL(F1(A%))):GOTO46059
46059 NEXT:RETURN

46060 FORF4%=F7%TOF8%STEP17:PO%=VAL(MID$(F9(F9%),F4%,3)):PRINT@P
O%,"";
46061 F3=MID$(F9(F9%),F4%+3,1):IFF3="$"THENPRINT"$";
46062 A%=VAL(MID$(F9(F9%),F4%+4,3)):IFF3="F"THENPRINTF2(A%);ELSE
PRINTSTRING$(A%,95);:IFINSTR("$N",F3)THENPRINT" ";
46063 IFF3="$"THENPRINT@PO%+A%-2,".";
46064 NEXT:RETURN
```

Variables used:

Simple Variables:
------ ---------

A$,A%,A1%	Temporary work variables
AF$	Specifies template format for formatted inkey subroutine.
AN$	Temporary storage, used to transfer data from the video display into string variables.
PO%	Stores the PRINT@ position for the beginning of the current field.
F3$	Temporary storage for the current field type code.
F4%	Used as a counter in FOR-NEXT loops within the handler.
F6%	Points to the current 17-byte parameter substring, within the current parameter string, F9$(F9%).
F7%	Points to the lowest 17-byte parameter substring, within the current parameter string, F9$(F9%), of the range specified by the current handler command.
F8%	Points to the highest 17-byte parameter substring, within the current parameter string, F9$(F9%), of the range specified by the current handler command.
F9%	Stores the current element number of the F9$ parameter array, as specified by the current handler command.
F9$	Stores the current 17-byte parameter substring for the current input field.
FE$	Loaded with "X", "E", or null by the validation subroutines you provide. FE$="X" indicates invalid entry - re-enter. FE$="E" indicates "end current handler command." FE$="" indicates that entry is OK, go to next field.
FX$	A 9-byte string, provided by your mainline program before calling the handler to specify the handler command.

Arrays Used:
------ ----

F9$( )	Provided by your mainline program to specify the parameters for the input fields. Each element within the F9$ array is a string that may specify parameters for up to 12 fields.
F2$( )	Provided by your mainline program to specify the special format templates to be used for dates, telephone numbers, etc. Each element specifies a different template. Within each template string, underline characters specify the input positions. (Not required if you don't need formatted input.)
F1$( )	Upon return from the handler after a "new" command, contains the results of each entry. Upon return from the handler after a "change" command, holds the new contents of each field that was changed.
FH$( )	Contains the disk fields to be used by the handler. You should FIELD you disk buffer before calling the handler. After a "W" command, each element of the FH$ array has been LSET with the corresponding F1$ element, according to your parameters. After a "C" command, those fields that were changed are LSET with the new value.

Line comments:

```
46010 (Initialize variables and go to desired routine)
      :Null-out working string, A$.
      :Load integer F5% with zero.
      :Load integer F9% with parameter string number from FX command.
      :Load integer F7% with first field number specified by command.
      :Load integer F8% with last  field number specified by command.
      :Convert F7% to position within F9$(F9%) parameter string.
      :Convert F8% to position within F9$(F9%) parameter string.
      :Load F6% with starting field number specified by command.
      :Convert F6% to position within F9$(F9%) paramerter string.
46011 :Go to proper routine based on first character of FX$ command.

46020 (Handle redisplay of alpha fields - "R" command)
      :Use F4% to point to first byte of each field parameter using
       a FOR-NEXT loop.
      :Load disk field type into string, F3$.
      :If it's not "$" type (alphanumeric),
       then skip the redisplay by going to 46029.
46021 :Extract PRINT@ position, PO%, from current field parameter.
      :Load disk field number into integer A1%.
46022 :Print data from the disk field at specified video position.
46029 :Repeat the process for next field, from line 46020.
      :Return to mainline program when last field has been processed.

46030 (Handle input of new data to video display - "N" command)
      :If current field is less than lowest field desired,
       then return to the mainline program,
       Otherwise, load F9$ with current 17-byte parameter string.
      :Load F3$ with with the input field type, (A,N,D,F,or $).
      :Load A1% with input field length specified.
      :Load PO% with the specified PRINT@ input field position.
      :If this is formated input, (F3$="F"),
       then load template string, AF$, with specified template from
       template array F2$.  (A1% specifies template number instead of
       length.)
46031  Display an arrow to direct operator's attention to the field.
      :If we're not in "change" mode, then skip to 46034.
       Otherwise, check if a key (up or down arrow) is still being
       pressed.  If one is, then skip to 46033.
46032  Display message, indicating that "C" can be pressed to change
       current field, and that up-arrow, down-arrow, or "E" can be
       used.
      :Call subroutine 40500 to await a key depression, the result to
       be returned in A$.
46033  If up-arrow or down-arrow key was pressed, then go to 46035.
      :Otherwise, if "E" was pressed then erase the arrow pointing to
       the input field and return to the mainline program.
      :If any other key was pressed, go back to 46032 to enforce
       entry of up-arrow, down-arow, "C", or "E".
46034  Call subroutine 25000 in mainline program.  (Display prompt
       message or execute other logic to precede the input.)
      :Based on the input field type specified, call the proper inkey
       subroutine.
      :If up-arrow was pressed instead of inputting data while in
       "change" mode, don't accept it -- repeat line 46034.
46035  Erase the arrow pointing to the input field.
      :If up-arrow was pressed,
       then point F6% to next lower field parameter in F9$(F9%)
       string, and
       set F5% equal to F6%, and
       go process the previous field again, from line 46030.
      :Otherwise, if the down arrow key was pressed,
       then skip to 46038.
```

46Ø36 This line is provided so that we can load AN$ with an image of the data that was entered if subroutine 4ØØ7Ø was not called from the inkey routine.

46Ø37 Load F1$ array string corresponding to current input field with the data that was entered.
:Call subroutine 26ØØØ in the mainline program to handle data validation or other logic for the current input field.
:If the data validation subroutine returned FE="X", then re-display the arrow pointing to the input field, and repeat the input from line 46Ø34
:Or, if the subroutine returned FE="E", then end the input here, and return to the mainline program.

46Ø38 Point F6% to the next input field parameter.
If F6% is now greater than or equal to F5%, then erase the arrow pointing to the input field, and set F5% equal to F6%.

46Ø39 If F6% now points to a input parameter higher than the highest specified by the FX$ command string, then, return to the mainline program with A$ equal to CHR$(255).
:Otherwise, go to 46Ø3Ø to process the next input field.

46Ø4Ø (Handle changes to data currently displayed - "C" command)
Null out (clear) each string in the F1$ array, corresponding to the parameters for the range to be changed. (A null F1$ string, after changes, will indicate that no change was made to the corresponding field.)

46Ø41 Point F5% to the next parameter beyond the highest input field parameter desired
:Call subroutine 46Ø3Ø to handle input of the desired changes.

46Ø42 (Handle transfer of input data in F1$ array to FH$ array for disk storage - "W" command)
For each input field in the range,
:Load A% with the F1$ array element number.
:If we're in change mode and no change was made to the field, then skip to 46Ø59 for the next field.

46Ø43 Otherwise, load A1% with the corresponding FH$ array element number.
:Load A$ with the code from the current parameter substring indicating the mode for storage on disk - alphanumeric, MKI$ format, etc.

46Ø5Ø Depending on the code now in A$, go to the proper LSET or RSET routine.

46Ø51 For code "$", LSET the entry data into the disk field.
:Go to 46Ø59.

46Ø52 For code "%", LSET the MKI$ of the numeric value of the input data into the disk field.
:Go to 46Ø59

46Ø53 For code "!", LSET the MKS$ of the numeric value of the input data into the disk field.
:Go to 46Ø59

46Ø54 For code "#", LSET the MKD$ of the numeric value of the input data into the disk field.
:Go to 46Ø59

46Ø55 ** Other data types can be handled in 46Ø55 - 46Ø58 **

46Ø59 Repeat from line 46Ø42 for the next input field.
:When all input fields are done, return to mainline program.

46Ø6Ø (Handle display of input fields - "F" command)
:For each field parameter in the desired range,
:Load PO% with the specified PRINT@ position.
:Move the cursor to the position on the display.

```
46061  Load F3$ with the input type code, A,N,$, or F.
      :If it's "$" type code (dollar format),
       then print a dollar sign at the beginning of the field.
46062  Load the length specified into A%.
      :But, if current field type is "F", (formatted), A% specifies
       the template string to use, so print it from the F2$ array.
      :Otherwise, print a string of underline characters
       corresponding to the field length.
      :If the input field type is dollar or numeric,
       follow the field with a space to blank-out the sign position.
46063  If the input field type is dollar, then print the decimal.
46064  Repeat from line 46060 for the next input field in the range
       specified.
      :When done, return to the mainline program.
```

VHANDLER/DEM is a demonstration and test program that shows the capabilities of the unscrolled video entry handler. It displays and accepts input for the sample screen we've used as our example.

To simplify matters a bit, the demonstration program does not actually access disk files, but we do open a file, 'TEST:0', so that we can simulate the use of the 'W', 'C' and 'R' handler commands. Instead of looking up account numbers on disk, the demonstration program considers any account number you enter as a new number. If you simply press ENTER, rather than typing an account number, the data for the previous account you entered will be redisplayed and you can make changes.

You'll find that the demonstration program is fully prompted. Just look at the bottom 2 lines of your display for the instructions at each step.

To use the demonstration program you will need to merge in the following subroutines:

```
40500            Single-key subroutine.
40070            Video display string pointer subroutine.
40130 - 40139    Alphanumeric inkey routine.
40140 - 40149    Dollar inkey routine.
40150 - 40159    Formatted inkey routine.
40160 - 40169    Numeric inkey routine.
46010 - 46064    Unscrolled video entry handler.
```

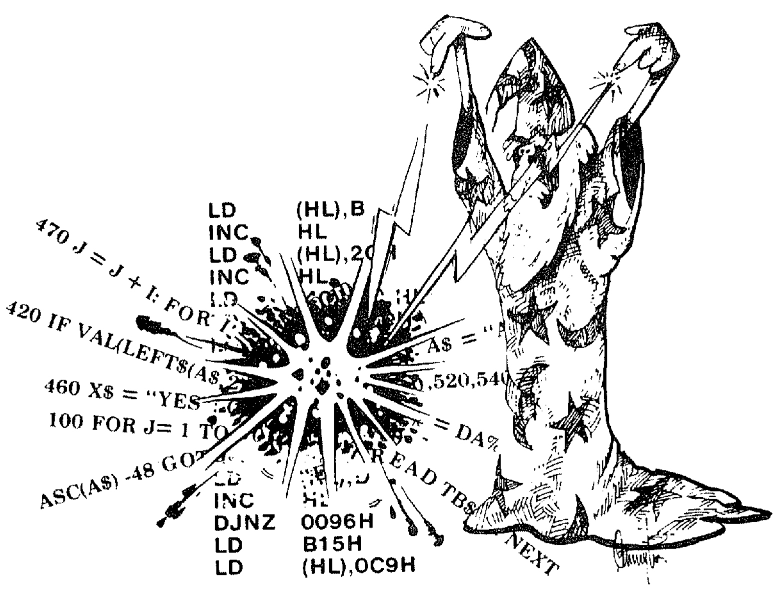
VHANDLER/DEM

Unscrolled Video Entry Handler Demonstration Program

```
0 'VHANDLER/DEM
1 CLEAR1000:DEFINTA-Z:DEFSTRF
2 A$="":A%=0:A1%=0:PO%=0:F3="":F2="":SG$=STRING$(63,131)

3 DIMF1(9),F2(1),FH(9)

20 CLOSE1:OPEN"R",1,"TEST:0":FIELD1,6ASFH(1),24ASFH(2),24ASFH(3)
,24ASFH(4),9ASFH(5),14ASFH(6),8ASFH(7),4ASFH(8),4ASFH(9)
21 LSETFH(1)=""

60 F9(0)="075A0060101$0101,267A0240202$0200,331A0240303$0300,395
A0240404$0400,431A0090505$0500,523F0000606$0600,559F0010707$0702
,651N0060808!0800,687$0070909!0900"

100 CLS
101 PRINT@256,"VIDEO ENTRY HANDLER DEMONSTRATION
";SG$
```

```
110 PRINT"
<1> BEGIN THE DEMONSTRATION
<2> END THE DEMONSTRATION

";SG$
180 PRINT@768,"PRESS THE NUMBER OF YOUR SELECTION..."
190 GOSUB40500:A%=INSTR("12",A$):IFA%=0THEN190ELSEONA%GOTO1000,2
000

1000 CLS:PRINT@128,SG$:PRINT@832,SG$
1001 PRINT@64,"ACCOUNT#";
1002 PRINT@192,"
NAME:
ADDRESS:
CITY,ST:";TAB(38);"ZIP:"
1005 PRINT@512,"PHONE NO:";TAB(38);"DATE:"
1006 PRINT@640,"QUANTITY:";TAB(38);"AMOUNT:";

1007 F2(0)="("+STRING$(3,95)+") "+STRING$(3,95)+"-"+STRING$(4,95
)
1008 F2(1)=STRING$(2,95)+"/"+STRING$(2,95)+"/"+STRING$(2,95)

1010 FX="F00010901"
1011 GOSUB46010

1020 FX="N00010901"
1021 GOSUB46010:IFA$="["THEN100ELSEIFFE="E"THEN1500
1050 PRINT@896,CHR$(31);"PRESS ENTER TO RECORD,
OR <UP-ARROW> TO MAKE CORRECTIONS...";
1051 GOSUB40500:IFA$=CHR$(91)THENFX="N00010909":GOTO1021

1080 PRINT@896,CHR$(31);"RECORDING...";:FX="W00010901":GOSUB4601
0
1090 GOTO1010

1500 FX="R00020902":GOSUB46010
1501 PRINT@651,USING"######-";CVS(FH(8));:PRINT@687,USING"$####.
##-";CVS(FH(9));
1510 PRINT@896,CHR$(31);"PRESS <C> FOR CHANGES,
   OR JUST PRESS <ENTER> TO EXIT...";
1511 GOSUB40500:IFA$="C"THEN1600ELSE1010

1600 FX="C00020902":GOSUB46010:GOTO1510

2000 CLS:CLOSE:PRINT"END OF DEMONSTRATION":END

25000 PRINT@896,CHR$(31);
25001 ONVAL(MID$(F9,13,2))GOTO25010,25020,25030,25040,25050,2506
0,25070,25080,25090
25002 RETURN

25010 IFLEFT$(FH(1),1)<>" "THENPRINT"PRESS <ENTER> TO RECALL PRE
VIOUS, OR ";
25011 PRINT"ENTER A NEW ACCOUNT #,
      OR PRESS <UP-ARROW> TO END THE DEMONSTRATION...";:RETURN

25020 PRINT"ENTER THE CUSTOMER NAME,
     OR PRESS <UP-ARROW> TO RE-ENTER THE ACCOUNT NUMBER...";:RET
URN

25030 PRINT"ENTER THE STREET ADDRESS,
     OR PRESS <UP-ARROW> TO RE-ENTER THE NAME...";:RETURN
```

```
25040 PRINT"ENTER THE CITY AND 2-LETTER STATE CODE,
     OR PRESS <UP-ARROW> TO RE-ENTER THE STREET ADDRESS...";:RET
URN

25050 PRINT"ENTER THE ZIP CODE,
     OR PRESS <UP-ARROW> TO RE-ENTER THE CITY AND STATE...";:RET
URN

25060 PRINT"ENTER THE AREA CODE AND TELEPHONE NUMBER,
     OR PRESS <UP-ARROW> TO RE-ENTER THE ZIP CODE...";:RETURN

25070 PRINT"ENTER THE DATE OF LAST CONTACT,
     OR PRESS <UP-ARROW> TO RE-ENTER THE TELEPHONE NUMBER...";:R
ETURN

25080 PRINT"ENTER THE QUANTITY OF GOODS PURCHASED TO DATE,
     OR PRESS <UP-ARROW> TO RE-ENTER THE DATE...";:RETURN

25090 PRINT"ENTER THE TOTAL AMOUNT PURCHASED,
     OR PRESS <UP-ARROW> TO RE-ENTER THE QUANTITY...";:RETURN

26000 FE="":ONVAL(MID$(F9,15,2))GOTO26010,26020
26001 RETURN

26010 IFF1(1)=STRING$(6," ")ANDF1(1)=FH(1)THENFE="X":RETURN
26011 IFA%=0THENPRINT@PO%,FH(1);:FE="E"
26012 RETURN

26020 IFASC(F1(7))=95THENF1(7)="00/00/00":PRINT@PO%,F1(7);
26021 IFMID$(F1(7),1,2)>"12"ORMID$(F1(7),4,2)>"31"THENFE="X"
26022 RETURN
```



Useful Utilities

The subroutines, functions, USR routines and utility programs that we've discussed in this book can be very valuable to you. But to make them especially valuable and easy to implement, this chapter discusses three utility programs that you'll want to keep in your disk library.

The first one, DOCLIST/BAS, gives you a way to expand and print the listings for any of the programs in this book or any BASIC program that you may have written. MERGEPRO/BAS makes it easy for you to build new programs by merging and renumbering lines from BASIC programs you already have on disk. Finally, DOSCHECK/BAS gives you a way to find the internal addresses for nearly any operating system you may be using. Though the addresses are listed in the appendix of this book, DOSCHECK/BAS should help you with any new disk operating systems you might purchase.

DOCLIST/BAS
A BASIC Program Lister and Documenter

DOCLIST/BAS lets you print classy listings for your BASIC programs. It puts each statement on a separate line, inserts spaces between each of the key words and indents IF-THEN statements and FOR-NEXT loops. Each page of the listing has a heading that shows the program name and page number and you can add a descriptive title to the heading.DOCLIST can help you understand the logic of a program because it prints a solid underline after each section in the logic. Where there is a conditional break in the logic in IF-THEN statements, a dotted underline is used to highlight them.

DOCLIST ignores blanks that may already be in your program, unless they are within quotes or within a remark statement. It also correcly processes programs in which you've used the down-arrow to provide a line feed to the next line.

Sample BASIC program before using DOCLIST/BAS

```
51 X=5712:Y=0
52 C=PEEK(X):IFC>127THENPRINT@544,RW$(Y),:Y=Y+1:IFY>123THEN55ELS
ERW$(Y)=CHR$(CANDNOT128):GOTO54
53 RW$(Y)=RW$(Y)+CHR$(C)
54 X=X+1:GOTO52
55 RW$(16)=RW$(16)+" "
```

DOCLIST/BAS expands the listing out to make it more readable:

Indented and 'Pretty-Printed' Listing, after using DOCLIST/BAS

```
51 X = 5712:
   Y = 0
52 C = PEEK (X):
   IF   C > 127
   THEN PRINT @544,RW$(Y),:
        Y = Y + 1:
        IF   Y > 123
        THEN 55: ........................................
        ELSE RW$(Y) = CHR$ (C AND NOT 128):
             GOTO 54: ..................................
53 RW$(Y) = RW$(Y) + CHR$ (C)
54 X = X + 1:
   GOTO 52
-----------------------------------------------------------
55 RW$(16) = RW$(16) + " "
```

Notice that there is an underline separating lines 54 and 55. This shows that the logic never falls through directly from 54 to 55. The dotted lines in the IF-THEN statement of line 52 show possible breaks in the logic, but since there is not a solid underline before line 53, there are some conditions in which the logic will fall through from line 52 to 53.

How to Use DOCLIST/BAS

To use DOCLIST/BAS you can RUN it, just as you'd run any other program saved on disk. You'll need to specify at least 2 files in response to the 'HOW MANY FILES?' question before going into BASIC.

Upon startup, there will be a slight pause as DOCLIST loads all the BASIC keywords (PRINT, MID$, FOR, etc.), into an array. Then your display will show the request:

```
ENTER THE NAME OF THE PROGRAM YOU WANT LISTED...
```

At this point, you should type the program name and disk drive number. For instance, if you want to list a program named, 'INVOICE/BAS' from a file on drive 1, you type:

```
INVOICE/BAS:1
```

Then, the DOCLIST/BAS program will verify that the program name you specified is on disk and that it is a BASIC program. The program must have been saved in normal compressed format. DOCLIST/BAS won't list programs that have been saved with the 'A' option.

Next, you will be permitted to select any combination of several options. The display will show:

```
<R> LINE NUMBER RANGE              <D> OUTPUT TO DISK
<W> SPECAL PAGE WIDTH              <H> SPECIAL PAGE HEADING
<S> STOP AFTER EACH PAGE           <P> NO LINE PRINTER OUTPUT

TYPE THE LETTERS CORRESPONDING TO THE OPTIONS YOU WANT, IF ANY,
```

For the normal case, you can press ENTER in response to the request. But, if for example, you want a special line number range and a special heading for each page, you can type 'RH'. Or, if you want the listing to be recorded into a sequential disk file for use in your word processing system, you can type 'D'. Any combination of the options is permitted.

The 'line number range' option lets you confine your listing to a beginning and ending line number. If you include 'R' in the list of options you specify, the program will request a 'FROM LINE' and 'TO LINE'.

If you specify the 'output to disk' option, the program will request the disk file name you want to use. Since the DOCLIST/BAS program will be reading the program file you are listing and writing the output file at the same time, both will have to be 'on-line'. You can't swap disks.

If you select the 'special page width' option, you can control the width of your listing. The default width is 80 characters, but if you may want to try other widths, especially if you have many nested FOR-NEXT loops or IF-THEN statements.

If you select 'special page heading'. you can type a one line heading that will be printed at the top of each page.

The 'S' option is especially helpful if you are using roll paper. It causes the printer to stop after each page so you can tear it off.

The 'P' option turns off the printed output. In some cases you may just want to see the listing on the display. More often, though, you may want to record your listing into a disk file, load the disk file into your word processing system, put in some additional comments and then print it with the word processing program.

DOCLIST/BAS

BASIC Program Lister and Documenter Utility

M 2 Note # 29

M 2 Note # 58

M 2 Note # 59

```
0 'DOCLIST/BAS
1 CLEAR10000:DEFINTA-Z
2 GOSUB1000
3 DIMB(1),RW$(128)
5 PW=80
50 CLS:PRINT@512,"LOADING RESERVED WORDS...";
51 X=5712:Y=0
52 C=PEEK(X):IFC>127THENPRINT@544,RW$(Y),:Y=Y+1:IFY>123THEN55ELS
ERW$(Y)=CHR$(CANDNOT128):GOTO54
53 RW$(Y)=RW$(Y)+CHR$(C)
54 X=X+1:GOTO52
55 RW$(16)=RW$(16)+"  "   'MAKE "IF"  4 CHARACTERS LONG
56 RW$(2)=RW$(2)+" "      'MAKE "FOR" 4 CHARACTERS LONG
100 GOSUB1000
110 GOSUB1100
120 GOSUB1200
130 CLS:PRINTPN$
140 GOSUB2100:GOSUB2000:IFC<>255THENPRINT"NOT A BASIC PROGRAM FI
LE...":CLOSE:GOTO100
150 PN=1:GOSUB3000:GOSUB3100
160 GOSUB4000
170 IFINSTR(OP$,"P")=0THENLPRINTCHR$(12);
171 IFINSTR(OP$,"D")THENPRINT#2,STRING$(255,0)
180 CLOSE:GOTO100
1000 'INITIALIZE SIMPLE VARIABLES
1010 C=0:P=0:BP=0:PC=0:LN$="":VB=0:NF=0:FF=0:NT=0:FX$="":QF=0:I1
=5:I2=5:FL!=0:TL!=65536:RN=1
1020 RETURN
1100 'ENTER PROGRAM NAME, OPEN AND FIELD PROGRAM FILE
```

```
1110 CLS:PRINT@64,"ENTER THE NAME OF THE PROGRAM YOU WANT LISTED
...."
1111 LINEINPUTPN$
1112 ONERRORGOTO1150:CLOSE:OPEN"I",1,PN$:CLOSE:OPEN"R",1,PN$:ONE
RRORGOTO0
1120 FIELD1,128ASB$(0),127ASB$(1):POKEVARPTR(B$(1)),128
1130 RETURN
1150 'PROGRAM FILE OPEN ERROR HANDLING
1151 IFERR=106THENPRINT"NOT FOUND."ELSEPRINT"ERROR."
1152 LINEINPUT"PRESS <ENTER>...";A$:RESUME1100
1200 'SELECT OPTIONS
1210 CLS:PRINT"
<R> LINE NUMBER RANGE          <D> OUTPUT TO DISK
<W> SPECIAL PAGE WIDTH         <H> SPECIAL PAGE HEADING
<S> STOP AFTER EACH PAGE       <P> NO LINE PRINTER OUTPUT"
1215 PRINT"
TYPE THE LETTERS CORRESPONDING TO THE OPTIONS YOU WANT, IF ANY,
AND PRESS <ENTER>..."
1220 LINEINPUTOP$
1230 IFINSTR(OP$,"R")=0THEN1240ELSEPRINT@704,CHR$(31);
1231 INPUT"FROM LINE ";FL!
1232 INPUT"TO   LINE ";TL!
1240 IFINSTR(OP$,"D")=0THEN1250ELSEPRINT@704,CHR$(31);
1241 LINEINPUT"OUTPUT DISK FILE NAME: ";A$
1242 CLOSE2:OPEN"O",2,A$
1250 IFINSTR(OP$,"W")=0THEN1260ELSEPRINT@704,CHR$(31);
1251 INPUT"PAGE WIDTH ";PW
1260 IFINSTR(OP$,"H")=0THEN1270ELSEPRINT@704,CHR$(31);
1261 PRINT"ENTER THE PAGE HEADING...":LINEINPUTPH$
1270 RETURN
2000 'GET NEXT BYTE FROM DISK FILE - RETURN AS C%
2010 P=P+1:IFP<129THEN2020ELSEP=1:BP=BP+1:IFBP<2THEN2020ELSEBP=0
:GOSUB2100

2020 C=ASC(MID$(B$(BP),P)):RETURN
2100 'GET NEXT RECORD FROM DISK FILE
2110 GET1,RN:RN=RN+1:RETURN
2200 'GET 2 BYTES FROM DISK FILE - RETURN AS A!
2210 GOSUB2000:PC=C:GOSUB2000:A!=CVI(CHR$(PC)+CHR$(C)):IFA!<0THE
NA!=65536+A!
2220 RETURN
3000 'PREPARE PRINTER
3010 IFINSTR(OP$,"P")THENRETURN
3020 LINEINPUT"PRESS <ENTER> WHEN PRINTER IS READY...";A$
3030 POKE16425,1:RETURN
3100 'PRINT PAGE HEADING
3110 IFINSTR(OP$,"P")THENRETURN
3120 LPRINTCHR$(34);PN$;CHR$(34);STRING$(PW-9-LEN(PN$)," ");"PAG
E";PN
3130 IFINSTR(OP$,"H")THENLPRINTPH$
3140 LPRINTSTRING$(PW,"."):LPRINT" "
3150 PN=PN+1:RETURN
3200 'PRINT A LINE OF TEXT
3210 PRINTLN$
3211 IFINSTR(OP$,"P")=0THENLPRINTLN$;
3212 IFINSTR(OP$,"D")THENPRINT#2,LN$;
3220 IFINSTR(" 128 141 142 146 159 167 185 187 ",STR$(VB))=0OR(P

C<>58ANDC<>0)THEN3240
3230 IFFF+NF=0THEN3235ELSENT=NT+1
3231 IFINSTR(OP$,"P")=0THENLPRINT" ";STRING$(PW-LEN(LN$)-1,".");
3232 IFINSTR(OP$,"D")THENPRINT#2," ";STRING$(PW-LEN(LN$)-1,".");
3233 IF(C=0)AND(NT/2<>INT(NT/2))THEN3235ELSE3240
3235 IFINSTR(OP$,"P")=0THENLPRINT" ":LPRINTSTRING$(PW,"-");
3236 IFINSTR(OP$,"D")THENPRINT#2," ":PRINT#2,STRING$(PW,"-");
3240 IFINSTR(OP$,"P")=0THENLPRINT" ":IFPEEK(16425)>50THENLPRINTC
```

```
HR$(12);:IFINSTR(OP$,"S")THENGOSUB3000:GOSUB3100ELSEGOSUB3100
3241 IFINSTR(OP$,"D")THENPRINT#2," "
3250 LN$=STRING$(6+NF+FF," "):RETURN
3300 'TEST ON PRINT-LINE LENGTH - PRINT IF FILLED
3310 IFLEN(LN$)+6<PWTHENRETURNELSEGOSUB3200:RETURN
4000 'PROCESS THE TEXT
4010 GOSUB2200:IFA!=0THEN4040
4020 GOSUB2200:IFA!<FL!THENPRINTA!:GOSUB4300:GOTO4010ELSEIFA!>TL
!THEN4040
4030 GOSUB4100:GOSUB3200:GOTO4010
4040 FF=0:NF=0:C=1:GOSUB3200:RETURN
4100 'PROCESS A LINE
4110 QF=0:FF=0:FX$="":C=0:VB=0:NT=0
4120 LN$=RIGHT$("     "+STR$(A!),5)+" "+STRING$(NF," ")
4130 PC=C:GOSUB2000:IFC=0THENRETURN
4135 IFC=149THENGOSUB3200:MID$(LN$,LEN(LN$)-4,4)="ELSE":VB=141:I
FFX$="ELSE"THENLN$=MID$(LN$,I1+1):FF=(FF-I1)*-(I1<=FF):GOTO4130E
LSEFX$="ELSE":GOTO4130
4140 IFPC=58ANDQF=0ANDVB<>0THENGOSUB3200
4150 IFC>127THEN4180
4160 IFC=34THENQF=NOTQF
4161 IF(C=10ANDQF=0)OR(C=32ANDQF=0)THEN4130
4162 IFC=10THENGOSUB3200:GOTO4130
4163 IFC=44ANDVB=135THENNF=(NF-I2)*-(I2<=NF):LN$=LEFT$(LN$,6)+MI
D$(LN$,7+I2)
4170 LN$=LN$+CHR$(C):GOSUB3300:GOTO4130
4180 'PROCESS RESERVED WORD
4182 IFC=202THENGOSUB3200:MID$(LN$,LEN(LN$)-4,4)="THEN":VB=141:G
OTO4130
4184 IFC=135ANDFX$=""THENMID$(LN$,LEN(LN$)-4,4)="NEXT":NF=(NF-I2
)*-(I2<=NF):VB=C:GOTO4130
4186 IFC=143THENFF=FF+I1:NT=NT+1:FX$="IF"
4188 IFC=129THENNF=NF+I2
4190 IFC=147THENQF=-2:IFPC=58THENMID$(LN$,LEN(LN$),1)="'":GOSUB3
300:GOTO4130
4200 IFRIGHT$(LN$,1)<>" "THENLN$=LN$+" "
4201 LN$=LN$+RW$(C-127)+" ":GOSUB3300
4210 IFC=141ANDVB=158THENVB=-1:GOTO4130ELSEVB=C:GOTO4130
4300 'READ TO END OF TEXT LINE - IGNORING CONTENTS
4310 GOSUB2000:IFC=0THENRETURN
4320 P=INSTR(P,B$(BP),CHR$(0)):IFP>0THENC=0:RETURNELSEP=128:GOTO
4310
```

MERGEPRO/BAS
A Program Line Merger and Renumber Utility

MERGEPRO/BAS lets you create a BASIC program by merging together lines from other BASIC programs that you've got stored on disk. You might want to store all your standard BASIC subroutines, function calls and data statements in one or more files on disk. Then with MERGEPRO/BAS, you can select them by indicating the line number ranges you want. After you've selected all the lines you want from one or more BASIC program files, MERGEPRO/BAS sorts the lines back into line number order and records them onto disk. You can then load the program that MERGEPRO/BAS created and make further modifications.

As you load lines from selected program files, you can renumber them to start at a different line number. Unlike other line renumbering utilities, MERGEPROBAS does not destroy the pattern of line numbers. If for example, your original program has a group of lines numbered 100, 101 and 110, you can renumber them to 200, 201 and 210. The increment between line numbers is not changed. You can also use the renumbering capability to change the sequence of program lines if you wish. All GOTO and GOSUB references are automatically modified, as long as they are within the range of lines you are renumbering.

How to Use MERGEPRO/BAS

To use MERGEPRO/BAS you will need to specify at least 1 file in response to the 'HOW MANY FILES?' question. Then you simply RUN MERGEPRO/BAS as you would any other program.

The first question you are asked is:

```
ALLOW HOW MANY LINES?
```

In response to this, you should enter a number that is greater than or equal to the total number of program lines that you will be merging together. MERGEPRO/BAS uses your response to dimension a string array in which the lines will be stored. In most cases it will suffice to simply enter 100, but if you have a particularly long program, you can enter a higher number.

Next, you are asked for the source program name. In response to this, you should enter the name of a program file you have stored on disk. It must be a BASIC program stored in the normal compressed format. (Your source program can not have been saved with the 'A' option.) MERGEPRO/BAS verifies that the program is present and opens it as a random file.

The next question is 'starting line number'. If you want to start from line 0 in your source program, you can just press ENTER. Otherwise, enter the first line number that you want to merge.

In response to the 'ending line number' question, you can just press ENTER if you want to merge every line to the end of the source program. Otherwise, you can enter the last line number in the range to be merged.

Then the program will ask you where you want to start renumbering. If you just press ENTER, the lines will be merged without renumbering them. Otherwise, you can enter the line number you want the first line read from the source file to be numbered.

Here's how your screen will look, assuming you are using a file named 'SROUTINE/LIB' as your source and you want to pull out lines 58000 through 58999, renumbering them to 28000 through 28999:

```
PROGRAM LINE MERGE & RENUMBER UTILITY
=================================================================
ALLOW HOW MANY LINES:    100
SOURCE PROGRAM NAME:     SROUTINE/LIB
STARTING LINE NUMBER:    58000
ENDING   LINE NUMBER:    58999
RENUMBER STARTING AT:    28000
```

After you answer the 'renumber starting at' question, MERGEPRO/BAS will read the program file and load the lines into an array. Then you will be given four options:

```
<M> MERGE MORE LINES FROM SAME PROGRAM
<P> USE ANOTHER SOURCE PROGRAM
<C> CANCEL ALL MERGES AND START OVER
<S> SAVE THE LINES THAT HAVE BEEN MERGED

PRESS THE KEY INDICATING YOUR SELECTION....
```

- The 'M' command lets you merge in another line number range from the same source program. It simply takes you back to the 'starting line number' question and repeats the process.
- The 'P' command takes you back to the 'source program name' question. From that point, you can enter another BASIC program name and merge in selected lines from it.
- The 'C' command cancels everything that you've merge so far, just as if you were using a NEW command and you can start over.
- The 'S' command lets you save all the lines that have been merged. Upon pressing 'S', the array containing the lines is sorted into numerical sequence, using the SORT1 USR routine that is described in this book. Then you are requested to enter the program name that you want to use for saving the new lines. Your prompt is:

```
SAVE USING PROGRAM NAME:
```

Simply type in the program name you want do use and the lines will be saved onto the disk you specify. The format is the same as if you were using a normal SAVE command in BASIC.

Then you are shown the prompt:

```
PRESS  <L> TO LOAD THE PROGRAM YOU JUST SAVED,
OR <ENTER> TO RE-RUN THE MERGEPRO/BAS PROGRAM...
```

If you press ENTER, the MERGEPRO/BAS program will start over. If you press 'L', the program you created will be loaded, so you can see what you've got. Then you can make further modifications to the program you've created, using

BASIC's normal procedures. Or, if you want to merge the program you've created into another program, you can save it again, this time with the 'A' option and you can use the MERGE command that is provided as part of disk BASIC.

When answering any of the questions in the MERGEPRO/BAS program, you can, instead of answering, press up-arrow and ENTER, if you want to go back to re-answer the previous question.

MERGEPRO/BAS
Program Line Merge and Renumber Utility

M 2 Note # 21
M 2 Note # 23
M 2 Note # 61

```
0 'MERGEPRO/BAS
1 CLEAR0:M!=MEM-4000:IFM!>32767THENM!=32767
2 CLEARM!
3 DEFINTA-Z:DEFSTRF:GOSUB58000:J=0:DIMP(1)
6 DEFFNIS!(A1%)=-((A1%<0)*(65536+A1%)+((A1%>=0)*A1%))
7 DEFFNSI%(A1!)=-((A1!>32767)*(A1!-65536))-((A1!<32768)*A1!)
50 DIMUS(93):FORX=0TO93:READUS(X):NEXT

100 CLS:PRINT:PRINT"PROGRAM LINE MERGE & RENUMBER UTILITY":PRINT
STRING$(63,131)
110 PRINT@192,CHR$(31);:LINEINPUT"ALLOW HOW MANY LINES:  ";A$:IF
A$=""THENA$="100":PRINT@215,A$
111 ONERRORGOTO112:LX=0:AL%=VAL(A$):DIMPT$(AL%):ONERRORGOTO0:GOT
O120
112 ONERRORGOTO0:RUN
120 PRINT@256,CHR$(31);:LINEINPUT"SOURCE PROGRAM NAME:   ";PN$
121 IFPN$=CHR$(91)THENRUNELSEONERRORGOTO128:CLOSE1:OPEN"I",1,PN$
:CLOSE1:PF=1:FS$=PN$:GOSUB58250:ONERRORGOTO0
122 PB!=1:BC%=1:GOSUB58800:IFASC(FV$)<>255THENCLOSE1:PRINT"NOT A
 BASIC PROGRAM!":FORX=1TO500:NEXT:GOTO120
123 PB!=2:LN!=0:GOTO130
128 PRINT"ERROR!":FORX=1TO500:NEXT:RESUME120
130 PRINT@320,CHR$(31);:LINEINPUT"STARTING LINE NUMBER:  ";A$
131 IFA$=CHR$(91)THEN120ELSESL!=VAL(A$):IFSL!<0THEN130ELSEIFSL!>
65535THEN130
132 PRINT@342,CHR$(30);SL!
140 PRINT@384,CHR$(31);:LINEINPUT"ENDING   LINE NUMBER:  ";A$
141 IFA$=CHR$(91)THEN130ELSEEL!=VAL(A$):IFEL!=0THENEL!=65535ELSE
IFEL!<SL!THENEL!=SL!
142 PRINT@406,CHR$(30);EL!
150 PRINT@448,CHR$(31);:LINEINPUT"RENUMBER STARTING AT:  ";A$
151 IFA$=CHR$(91)THEN140ELSERS!=VAL(A$):IFA$=""THENPRINT@471,CHR
$(30);"<NO RENUMBER>":RS!=SL!
152 OS!=RS!-SL!

200 PRINT@576,"READING LINE NUMBER: "
210 BC%=255:IFSL!<LN!THENPB!=2
220 GOSUB58800:IFCVI(FV$)=0THEN300ELSEA%=INSTR(5,FV$,CHR$(0)):FV
$=MID$(LEFT$(FV$,A%-1),3):LN%=CVI(FV$):LN!=FNIS!(LN%)
230 PRINT@598,CHR$(31);LN!:IFLN!>EL!THEN300ELSEPB!=PB!+A%:IFLN!<
SL!THEN220
240 PRINT@608,"MERGING AS LINE";LN!+OS!:IFOS!=0THEN250ELSEA%=3
241 A1%=INSTR(A%,FV$,CHR$(141)):IFA1%=0THENA%=3ELSEGOSUB1000:GOT
O241
242 A1%=INSTR(A%,FV$,CHR$(145)):IFA1%=0THENA%=3ELSEGOSUB1000:GOT
O242
243 A1%=INSTR(A%,FV$,CHR$(202)):IFA1%=0THENA%=3ELSEGOSUB1000:GOT
O243
244 A1%=INSTR(A%,FV$,CHR$(149)):IFA1%=0THENA%=3ELSEGOSUB1000:GOT
O244
250 A$=MKI$(FNSI%(LN!+OS!)):PT$(LX)=RIGHT$(A$,1)+LEFT$(A$,1)+MID
$(FV$,3)
260 LX=LX+1
```

```
280 GOTO220

300 PRINT@576,CHR$(31);"
<M> MERGE MORE LINES FROM SAME PROGRAM
<P> USE ANOTHER SOURCE PROGRAM
<C> CANCEL ALL MERGES AND START OVER
<S> SAVE THE LINES THAT HAVE BEEN MERGED"
301 PRINT"
PRESS THE KEY INDICATING YOUR SELECTION...";:GOSUB40500
305 A%=INSTR("MPCS",A$):IFA%=0THEN300ELSEONA%GOTO310,320,330,400
310 GOTO130
320 GOTO120
330 RUN

400 CLOSE:IFLX=0THENRUNELSEPRINT@192,CHR$(31);"SORTING..."
410 P(0)=VARPTR(PT$(0)):P(1)=LX-1:DEFUSR=VARPTR(US(0)):J=USR(VARP
TR(P(0)))
420 PRINT@192,CHR$(31);"SAVE USING PROGRAM NAME: ";:LINEINPUTFS$
:IFFS$=CHR$(91)THENRUN
421 PF=1:GOSUB58250:PB!=1:FV$=CHR$(255):GOSUB58810:PB!=2
430 FORX=0TOLX-1:FV$=MKI$(-1)+MID$(PT$(X),2,1)+MID$(PT$(X),1,1)+
MID$(PT$(X),3)+CHR$(0):PRINT@512,FNIS!(CVI(MID$(FV$,3))):GOSUB58
810:PB!=PB!+LEN(FV$):NEXT
440 FV$=MKI$(0):GOSUB58810:CLOSE
450 PRINT@256,CHR$(31);"
PRESS  <L> TO LOAD THE PROGRAM YOU JUST SAVED,
OR <ENTER> TO RE-RUN THE MERGEPRO/BAS PROGRAM...";
460 GOSUB40500:IFA$<>"L"THENRUN
470 CLS:FORX=1TO16:POKE15360+X-1,ASC(MID$(FS$,X,1)+" "):NEXT:CLE
AR50
471 FORX=1TO16:FS$=FS$+CHR$(PEEK(15360+X-1)):NEXT:LOADFS$

1000 A%=A1%+1
1001 A!=VAL(MID$(FV$,A%)):IFA!=0ORA!<SL!ORA!>EL!THEN1020ELSEPRIN
T@640,"RENUMBERING REFERENCE TO";A!
1010 A$=MID$(STR$(A!),2):A2%=INSTR(A%,FV$,A$)+LEN(A$):FV$=LEFT$(
FV$,A%-1)+MID$(STR$(A!+OS!),2)+MID$(FV$,A2%)
1020 A2%=INSTR(3,FV$,CHR$(161)):IFA2%=0THENRETURNELSEIF(MID$(FV$
,A1%,1)<>CHR$(141)ANDMID$(FV$,A1%,1)<>CHR$(145))THENRETURNELSEIF
A2%>A1%THENRETURN
1022 A2%=INSTR(A1%,FV$,":"):IFA2%=0THENA2%=LEN(FV$)+1
1023 A%=INSTR(A%,FV$+",",","")+1:IFA%>A2%THENA%=A1%+1:RETURNELSE1
001

10000 DATA32717,-6902,-7715,20189,-8958,838,1048,-6695,-15911
10001 DATA33,-18688,17133,-13360,-13512,-15079,-7719,-8743,622
10002 DATA26333,-18685,17133,-9755,-9775,-13560,2183,20189,-8960
10003 DATA326,8645,1,-9755,-6719,-11815,-6887,10705,-8935
10004 DATA94,22237,6401,-10799,6373,-7924,2273,2293,-13327
10005 DATA10311,6321,6863,17999,9173,9054,-5290,-6703,9195
10006 DATA9054,-7850,1284,1568,3340,12064,4120,3340,3112
10007 DATA-16870,1568,4899,3333,-6120,7472,-10791,-9787,-7727
10008 DATA-4681,10322,5054,-9771,-9791,6,782,-7727,-6903
10009 DATA2539,6373,-7752,-10799,1765,6659,30542,4729,4899
10010 DATA-2288,-13560,2247,-12776

40500 A$=INKEY$:IFA$=""THEN40500ELSERETURN

58000 A%=1:DIMPR(A%),PP(A%)
58001 RETURN
58210 IFPR(PF)=PP(PF)THENRETURN
58220 PP(PF)=PR(PF):ONERRORGOTO58900:GETPF,PR(PF):ONERRORGOTO0:R
ETURN
58250 GOSUB58290:ONERRORGOTO58910:OPEN"R",PF,FS$:ONERRORGOTO0:PP
(PF)=0:RETURN
```

```
58290 ONERRORGOTO58930:CLOSEPF:ONERRORGOTO0:RETURN
58300 ONERRORGOTO58920:PUTPF,PR(PF):ONERRORGOTO0:RETURN
58800 GOSUB58850:IFLEN(FD$)>=BC%THENFV$=LEFT$(FD$,BC%):RETURNELS
EFV$=FD$:PR(PF)=PR(PF)+1:GOSUB58210:FIELDPF,BC%-LEN(FV$)ASFD$:FV
$=FV$+FD$:RETURN
58810 GOSUB58850:IF256-LS>=LEN(FV$)THENPOKEVARPTR(FD$),LEN(FV$):
LSETFD$=FV$:GOSUB58300:RETURN
58811 LSETFD$=FV$:GOSUB58300:PR(PF)=PR(PF)+1:GOSUB58210:FIELDPF,
LEN(FV$)-LEN(FD$)ASFD$:LSETFD$=MID$(FV$,LEN(FV$)-LEN(FD$)+1):GOS
UB58300:RETURN
58850 PR(PF)=INT((PB!-1)/256)+1:LS=PB!-(PR(PF)-1)*256-1:GOSUB582
10:FIELDPF,(LS)ASA$,0ASFD$:IFLS>0THENPOKEVARPTR(FD$),256-LS:RETU
RNELSEPOKEVARPTR(FD$),255:RETURN
58900 A$="DISK READ ERROR":GOTO58990
58910 A$="CAN'T OPEN DISK FILE":GOTO58990
58920 A$="DISK WRITE ERROR":GOTO58990
58930 A$="CAN'T CLOSE DISK FILE":GOTO58990
58990 A1$="":A%=VARPTR(A1$):POKEA%,64:POKEA%+1,192:POKEA%+2,63:A
2$=A1$:A%=PEEK(16416):A1%=PEEK(16417)
58991 PRINT@960,CHR$(143);A$;TAB(22)"(E=";MID$(STR$(ERR/2),2);"
F=";MID$(STR$(PF),2);" R=";MID$(STR$(PR(PF)),2);")";TAB(41);"PRE
SS ENTER TO RETRY!";CHR$(143);
58992 A$=INKEY$:IFA$=""THEN58992
58993 PRINT@960,CHR$(31);
58994 LSETA1$=A2$:POKE16416,A%:POKE16417,A1%
58995 IFA$<>CHR$(13)THENRESUME112
58996 RESUME
```

DOSCHECK/BAS
A Disk Operating System Address Finder

DOSCHECK/BAS is a BASIC program that you can use to find the memory addresses used by your disk operating system. Although the appendix of this book lists the addresses for the most popular disk operating systems, you can be sure that others will be available, and new versions are released from time to time.

The addresses that are displayed for you by DOSCHECK/BAS are:

- USR routine pointer addresses, USR0 through USR9.
- Disk file buffer addresses for files 1 through 15.
- Disk file DCB addresses for files 1 through 15.

They are shown in decimal as well as hexadecimal format.

To use DOSCHECK/BAS, you will need to specify at least 2 files when you go into BASIC. Then you run it just as you'd run any other program. You should be aware that the program will temporarily create and then kill a file called 'XTESTX' on drive 0. Unless you modify the program, your drive 0 disk can not be write protected.

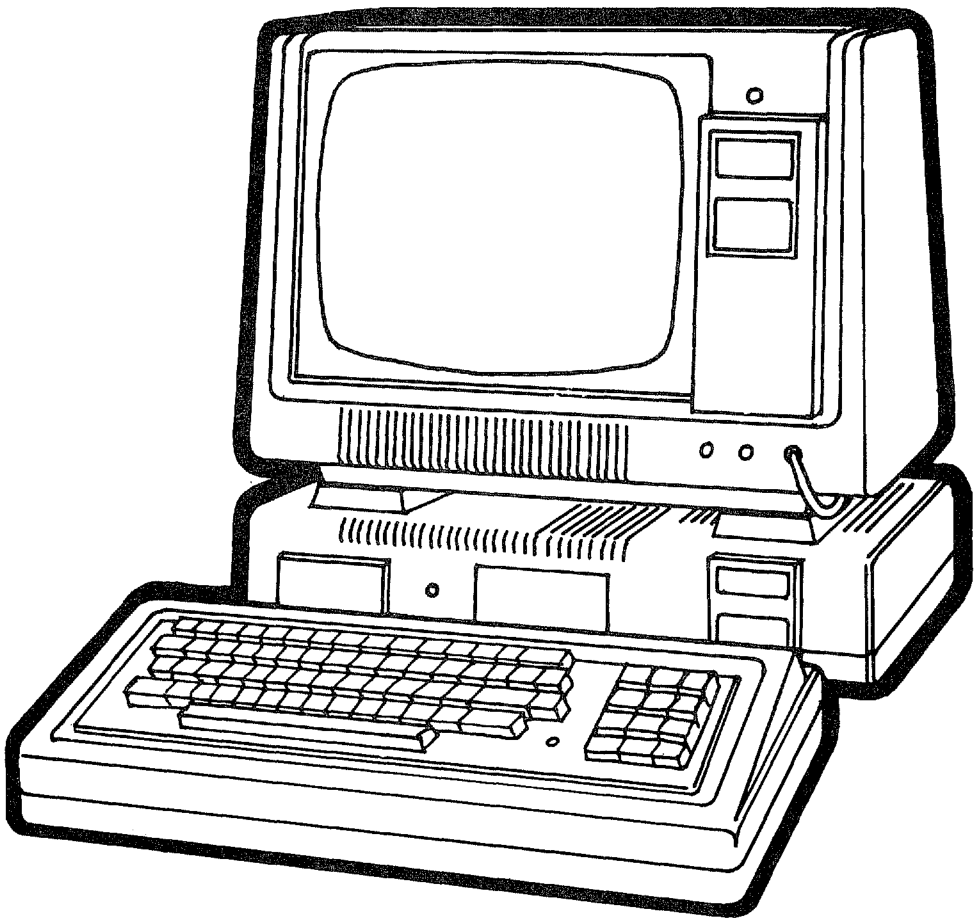
DOSCHECK/BAS finds the addresses by loading dummy values and then doing a search with the SEARCH2 USR routine. I've tried it on several different disk operating systems and it found the addresses correctly on all of them. But keep in mind, there's no way to predict the organizations that future operating systems will have, so there's no 100 percent guarantee that DOSCHECK/BAS will work with them . . .

DOSCHECK/BAS

Disk Operating System Address Finder

```
Ø 'DOSCHECK/BAS
1 CLEAR1ØØØ:DEFINTA-Z:DIMBA(2),DC(2)

1Ø 'LOAD SEARCH2 ROUTINE INTO A MAGIC ARRAY...
11 DATA 32717,-69Ø2,-7715, 2Ø189,-8948, 94, 22237, 6913, 33,-135
68, 12345, 64Ø1, 132Ø, 1Ø731, 6379,-5132
12 DATA 28381,-8956, 1382,-8935, 4725, 29917,-8941, 42Ø6, 26333,
 17937, 9Ø32, 9Ø54,-1Ø922,-8763,  94, 22237
13 DATA-8959, 2158, 26333,-18679, 21229, 2156Ø, 28381,-8942, 496
6, 24285, 5646, 64ØØ,-11839,-14891,-1687Ø, 1568
14 DATA 8979,-2Ø32, 8472, 28381,-896Ø, 358,-8925, 117, 29917,-89
59, 4718, 26333,-8941, 3166, 22,-8935
15 DATA 4725, 29917, 6163,-878Ø, 267Ø, 26333, 17931, 24285,-8942
, 495Ø, 29475, 29219, 28381,-896Ø, 358, 1Ø48
16 DATA 46, 38,-15935,-25917,1Ø
17 DIMUS(84):FORX=ØTO84:READUS(X):NEXT

6Ø DEFFNIA%(A1%,A2%)=(65536-(A1%+A2%))*((A1%+A2%)>32767)+((Ø-A1%
+A2%)*-((A1%+A2%)<-32768))+(A1%+A2%)*-(((A1%+A2%)<32768)AND((A1%
+A2%)>-32769))

61 DEFFNH2$(A1%)=MID$("Ø123456789ABCDEF",INT(A1%/16)+1,1)+MID$("
Ø123456789ABCDEF",A1%-INT(A1%/16)*16+1,1)

62 DEFFNH4$(A1%)=FNH2$(ASC(MID$(MKI$(A1%),2)))+FNH2$(ASC(MKI$(A1
%)))
1ØØ CLS:PRINT"
DOS ADDRESS FINDER
";STRING$(63,131)

2ØØ PRINT"USR ROUTINE ADDRESS POINTERS:"
21Ø DEFUSRØ=1ØØ:DEFUSR1=11Ø:DEFUSR2=12Ø:DEFUSR3=13Ø:DEFUSR4=14Ø:
DEFUSR5=15Ø:DEFUSR6=16Ø:DEFUSR7=17Ø:DEFUSR8=18Ø
211 J=Ø:RE$="":KY$="":FORX=1ØØTO18ØSTEP1Ø:KY$=KY$+MKI$(X):NEXT
22Ø C(Ø)=Ø:C(2)=&H41ØØ:C(4)=PEEK(&H4ØA4)+PEEK(&H4ØA5)*256:C(5)=V
ARPTR(RE$):C(6)=1:C(7)=Ø:C(8)=VARPTR(KY$)

23Ø DEFUSR9=VARPTR(US(Ø)):J=USR9(VARPTR(C(Ø)))
24Ø IFJ=<ØTHENPRINT"CAN'T FIND!":GOTO25ØELSEPRINT"
 USRØ   USR1   USR2   USR3   USR4   USR5   USR6   USR7   USR8   USR9"
241 FORX=C(9)TOC(9)+18STEP2:PRINTUSING"#####  ";X;:NEXT:PRINT
242 FORX=C(9)TOC(9)+18STEP2:PRINTUSING" %   % ";FNH4$(X);:NEXT:PR
INT
25Ø PRINT:PRINT"PRESS <ENTER> TO FIND DISK BUFFER ADDRESSES...":
GOSUB4Ø5ØØ

3ØØ PRINT@192,CHR$(31);"DISK FILE BUFFER ADDRESSES:"
31Ø PRINT"
NOTES:  1. THE DISK IN DRIVE Ø MUST NOT BE WRITE-PROTECTED.
        2. YOU MUST HAVE SPECIFIED AT LEAST 2 FILES UPON
           LOADING BASIC.
        3. WE WILL CREATE AND THEN KILL A FILE CALLED 'XTESTX'
           ON DRIVE Ø.

32Ø PRINT:PRINT"PRESS <ENTER> TO BEGIN SEARCH FOR DISK BUFFER AD
DRESSES...";:GOSUB4Ø5ØØ
33Ø FORX=1TO2
34Ø OPEN"R",X,"XTESTX:Ø":FIELDX,ØASA$:BF(X)=CVI(CHR$(PEEK(FNIA%(
VARPTR(A$),1)))+CHR$(PEEK(FNIA%(VARPTR(A$),2))))
35Ø C(Ø)=Ø:C(2)=FNIA%(BF(X),-6ØØ):C(4)=BF(X):C(5)=VARPTR(RE$):C(
6)=1:C(7)=Ø:C(8)=VARPTR(KY$)
351 KY$=MKI$(BF(X)):DEFUSR9=VARPTR(US(Ø)):J=USR9(VARPTR(C(Ø)))
352 IFJ>ØTHENDC(X)=FNIA%(C(9),-3)
36Ø CLOSE:KILL"XTESTX:Ø":NEXT
```

```
370 PRINT@256,CHR$(31)
371 ST=BF(2)-BF(1)
375 FORX=1TO15:PRINTUSING"## = ";X;:A%=FNIA%(BF(1),(X-1)*ST):PRI
NTA%;", ";FNH4$(A%);" HEX",:NEXT

380 PRINT:PRINT"
PRESS <ENTER> TO DISPLAY DCB ADDRESSES...";:GOSUB40500
381 PRINT@192,CHR$(31);"DISK FILE DATA CONTROL BLOCK ADDRESSES:"
382 PRINT:IFDC(1)=0ORDC(2)-DC(1)<>STTHENPRINT"CANNOT COMPUTE...
THIS DISK OPERATING SYSTEM DOESN'T FOLLOW THE PATTERN OF
MOST DISK OPERATING SYSTEMS FOR THE TRS-80!":END
385 FORX=1TO15:PRINTUSING"## = ";X;:A%=FNIA%(DC(1),(X-1)*ST):PRI
NTA%;", ";FNH4$(A%);" HEX",:NEXT
395 END
40500 A$=INKEY$:IFA$=""THEN40500ELSERETURN
```



Model 2 Modifications

I remember the ads when the TRS-80 Model 2 was first announced. The line went something like this:

'. . . not just a new TRS-80, but a whole new architecture!'

That new architecture has been a blessing to some. Since the logic in the Model 2 is not 'hard-wired' into ROM, a large body of microcomputer programs has become available. But with the new flexibilities of the Model 2 came some new challenges for those of us who wanted to use our Model 1 programs.

As we discussed in the introduction, programming is a world of trade-offs. Special techniques that give extra speed and power to one computer system often sacrifice compatibility with another. This section gives you some helpful guidelines for achieving most of the capabilites discussed in this book on your Model 2. You'll also find that the information we'll discuss will help you implement other Model 1 programs, such as those presented in magazine articles. Beyond that, we'll cover some techniques that unlock many of the unique capabilities of the Mod 2.

PEEK and POKE for the Model 2

POKEMOD/BAS is a BASIC program that temporarily patches in a peek and poke capability that is identical to that found on the Model 1 and 3. It works with Model 2 TRSDOS 2.0 and 2.0a.

To use POKEMOD/BAS you simply run it after going into BASIC. It takes less than a second and after running it you can enter, load or run any other program. Your peek and poke capabilities remain active until you go back to TRSDOS READY. POKEMOD simply overlays certain sections of BASIC in RAM with the required logic. (It replaces OCT$ and NAME.) Your system disk in drive 0 is not altered.

You may wish to execute POKEMOD/BAS from a DO file. Or, you can replace line 50 with a RUN command so that another program is chained after the modification is made. The other alternative is to imbed the logic within another program. Be aware, though, that you only need to execute POKEMOD/BAS once during any BASIC session.

```
0 'POKEMOD/BAS
10 DEFINTA-Z
20 DIMUS(46)
30 FORX=0TO46:READUS(X):NEXT
40 J=0:DEFUSR=VARPTR(US(0)):J=USR(0)
50 END

80 DATA-13023,8925,26611,15393,8917,26613,-6367,8748,26615,-13023,8938,26617,153
93,8913,26619,4641,8905,26621

81 DATA-13023,8797,26623,17441,8830,26625,-15583,8955,26627,14910,1330,8552,2043
2,-1246,15912,12875,10493,-12255

82 DATA8773,10757,17697,8779,10759,-15583,8959,23259,26430,-8910,-13990
```

Model 2 Peek & Poke Modification Program

If you'd rather, you can make the PEEK and POKE modifications permanent with the following steps. In case an error occurs though, make sure that you retain a copy of the unmodified TRSDOS 2.0 system disk as distributed by Radio Shack:

- From TRSDOS READY, enter the command: BUILD POKEPTCH
- Type the following 8 lines, pressing ENTER after each:

```
PATCH BASIC A=67F3, F=AFCD8761, C=CDDD3CD5
PATCH BASIC A=67F7, F=C5CD7166, C=E72CCDEA
PATCH BASIC A=67FB, F=E741E753, C=3CD112C9
PATCH BASIC A=67FF, F=E3011E00, C=CD5D447E
PATCH BASIC A=6803, F=09444D,   C=C3FB3A
PATCH BASIC A=2A05, F=CF435424, C=D045454B
PATCH BASIC A=28FB, F=CE414D,   C=D04F4B
PATCH BASIC A=5ADB, F=CD8A4E,   C=C3FF67
```

- Press BREAK after the last line has been entered.
- Enter the command: DO POKEPTCH
- You may KILL POKEPTCH after the process is complete.

Video Display Printing Compatibility Guidelines

The video display on the Model 2 has 24 rows of 80 columns each, while models 1 and 3 have 16 rows of 64 columns each. This gives you PRINT@ positions that range from 0 to 1919, compared to a range of 0 to 1023 for models 1 and 3. In most programs that you may wish to convert, you can look for references to 64, changing them to 80; and references to 1023, changing them to 1919 and so forth. Here is a list of numbers pertaining to video display computations as they are often found in this book and their Model 2 equivalents:

```
64 = 80         63 = 79
1024 = 1920   1023 = 1919   960 = 1840   896 = 1760   832 = 1680
```

For a quick and easy way to modify programs that use many PRINT@ statements, you can use FNP2%. It converts PRINT@ positions that assume a 64-column video line to PRINT@ positions for an 80-column video line. On a model 1 or 3, for example, 64 is the first position on the second video line. FNP2%(64) returns 80, the first position on the second line of an 80-column display. After you've defined FNP2% in your program, 'PRINT@ PO%' can be replaced by 'PRINT@ FNP2%(PO%)'. 'PRINT@ 256 ' can be replaced by 'PRINT@ FNP2%(256)' and so forth.

PRINT@ Conversion Function,

```
10 DEFFNP2%(A%)=INT(A%/64)*80+(A%ANDNOT-64)+0+0*80
```

You can replace the '+0' near the end of the function definition with '+8' if you want to center the converted positions horizontally on the 80 column screen. The '+0 *80' can be replaced with '+4 *80' if you want the converted positions to start on the 5th line for vertical centering. Or, you may delete the '+0 *80' if you're satisfied to use the upper-left 64-by-16 positions. To see which area of the screen will be used, you can try the following:

```
FOR X = 0 TO 1023 : PRINT@ FNP2%(X),"X";: NEXT
```

Special Character Conversions

You can display the character codes that are generated by specific key depressions with the following command:

```
FORX=1TO1:X=0:A$=INKEY$:IFA$=""THENNEXTELSEPRINTASC(A$):NEXT
```

It's up to you to decide which keys to use in your programs. For the inkey subroutines, video entry handlers and other programs presented in this book I prefer the following replacements:

Models 1 & 3	CHR$	Model 2	CHR$
Up-Arrow	91	F1	1
Down-Arrow	10	F2	2
Left-Arrow	8	Back Space	8
Right-Arrow	9	Tab	9
Clear	31	Escape	27
Shift-Up-Arrow	27	Up-Arrow	30
Shift-Down-Arrow	26	Down-Arrow	31
Shift-Left-Arrow	24	Left-Arrow	28
Shift-Right-Arrow	25	Right-Arrow	29

For printed special characters, as used with the CHR$ or STRING$ functions, you can make the following replacements:

FUNCTION	Models 1 & 3	Model 2
Clear remainder of current line	CHR$(30)	CHR$(23)
Clear remainder of display	CHR$(31)	CHR$(24)
Backspace without erasing	CHR$(24)	CHR$(28)
Space forward without erasing	CHR$(25)	CHR$(29)
Move Up, same column	CHR$(27)	CHR$(254)
Move Down, same column	CHR$(26)	CHR$(255)
Horizontal Bar String	STRING$(63,131)	STRING$(79,153)
Fill-in-the-blank boxes	STRING$(n,132)	STRING$(n,145)
Vertical Bar String	CHR$'s 170+24+26	CHR$'s 149+28+255

How to Use the Model 2 Supervisor Calls From BASIC

Model 2 TRSDOS has a built-in feature that lets you use a wealth of special purpose machine language subroutines. The 'supervisor call' or 'SVC' capability, as it is explained in the owner's manual, is only useful if you do machine language programming. But with a magic array technique, we can load all the arguments that are required for any supervisor call and execute it as a USR subroutine from BASIC!

Subroutine 40090 loads the required elements into the UV% magic array. It should executed only once during a BASIC program. Subroutine 40091 does the USR call for you whenever you need it. It arbitrarily uses USR2:

```
Initialize Supervisor Call Magic Array:
40090 J%=0:DIMUV%(8):UV%(0)=15872:UV%(2)=8448:UV%(4)=4352:UV%(6)=256:UV%(8)=-138
73:RETURN

Execute Supervisor Call Magic Array:
40091 DEFUSR2=VARPTR(UV%(0)):J%=USR2(0):RETURN
```

Supervisor Call Magic Array Subroutines

To load the A, HL, DE and BC registers for any supervisor call, you simply load UV%(1), UV%(3), UV%(5) and UV%(7), respectively. To load the A register with 5, for example, your statement is:

```
UV%(1)=5
```

To load the B register with 10 and the C register with 20 your command is:

```
UV%(7)=CVI(CHR$(20)+CHR$(10))
```

Once you've loaded the required registers, you simply GOSUB 40091.

Shown below are some examples for useful applications. Each of them assume that you have already executed a 'GOSUB 40090' in your program.

Preventing a Top Portion of the Screen From Scrolling

In this example we'll protect the top 10 lines. You can replace the '10' with any number from 0 to 22.

```
UV%(1)=27:UV%(7)=CVI(CHR$(0)+CHR$(10)):GOSUB40091
```

Turning Off the Flashing Cursor

We can load UV%(7) with 0 to turn it off or −1 to turn it on. Here's the call to turn it off:

```
UV%(1)=26:UV%(7)=0:GOSUB40091
```

You should be aware that the cursor comes on again when your program returns to READY.

Video Display Screen Save and Flashback

This SVC can be very important on the Model 2 because the video is not memory-maped like it is on the Models 1 and 3. You can replace subroutine 40200, as it was presented for the Model 1 and 3, with the following:

```
40200 UV%(1)=94:UV%(3)=VARPTR(SS%(SN%*960)):IFA$="S"THENUV%(7)=-1ELSEUV%(7)=0
40201 GOSUB40091:RETURN
```

Screen Save and Recall Subroutine

Note that the SS% integer array is used for storing screens. You will need to dimension it with 960 elements for each screen you wish to save. Refer back to the section that discusses the screen save and flashback subroutine for more information and a demonstration program.

Pointing Strings to the Video Display

We cannot use the same methods that we used for the Models 1 and 3. Instead, we can use the VDREAD supervisor call. Here is subroutine 40070, modified for the Model 2, so that you can load data from any position on the display, PO%, for any length up to 255 bytes, A1%, into the string variable, AN$.

```
40070 UV%(1)=11:UV%(7)=CVI(CHR$(PO%-INT(PO%/80)*80)+CHR$(INT(PO%/80))):UV%(5)=CV
I(CHR$(0)+CHR$(A1%)):AN$=STRING$(A1%,32):UV%(3)=CVI(CHR$(PEEK(VARPTR(AN$)+1))+CH
R$(PEEK(VARPTR(AN$)+2))):GOSUB40091:RETURN
```

Video Display String Pointer Subroutine

How to Maintain a Video Display Image in Memory

Many of the demonstration programs in this book take advantage of the fact that on models 1 and 3, the video display occupies memory locations 15360 through 16383. A fixed memory block that corresponds to the display makes it easy to show the results of memory sorts, block moves and special scrolling techniques.

We can have the same conveniences on the Model 2 if we reserve a specific area of memory to store an image of the video display. Just before performing a USR routine or other technique that involves the video display, we can load the current video contents into that memory area. Then we are free to use PEEK, POKE, LSET, RSET, move-data USR routines and other techniques. After we've completed our screen manipulations, we can display the modified screen. The whole process can be instantaneous and unnoticeable to the operator.

DEMOSCRN/MRG is a set of 4 subroutines that you can store on disk and merge into programs when you need the capability of treating your video display as memory. It consists of the two supervisor call magic array subroutines, 40090 and 40091 and two others. Subroutine 40080 copies the video display to protected memory. Subroutine 40081 copies from protected memory back to the video display. You should save them on disk in ASCII format, (with the 'A' option).

Video Display Memory Image Subroutines

```
40080 UV%(7)=-1:GOTO40082   'COPY SCREEN TO PROTECTED MEMORY
40081 UV%(7)=0:GOTO40082    'COPY PROTECTED MEMORY TO SCREEN
40082 UV%(1)=94:UV%(3)=-6144:GOSUB40091:RETURN
40090 'INITIALIZE SUPERVISOR CALL MAGIC ARRAY SUBROUTINE GOES HERE
40091 'EXECUTE SUPERVISOR CALL MAGIC ARRAY SUBROUTINE GOES HERE
```

As shown, the DEMOSCRN/MRG subroutines create a video display image that starts at −6144 in memory, **E800**. After a GOSUB 40080, memory location −6144 will contain the contents of PRINT@ position 0, −6143 is position 1 and so forth, up to −4225, which is position 1919. You will need to specify a memory size of 59390 or less. You can do this by specifying '-M:59390' when you load BASIC or you can use 59390 as the second argument of a CLEAR statement in a BASIC program. Several of the Model 2 program modification notes will suggest that you merge DEMOSCRN/MRG and they will assume that you've used these addresses. The notes will tell you where to put your GOSUB 40080, GOSUB 40081 and GOSUB 40090.

You can, of course, change the −6144 in line 40082 to another address, but be sure to make the appropriate memory size allowance.

Model 2 Modification Notes

The following notes describe differences that you should consider when using TRSDOS 2.0 or 2.0a on a TRS-80 Model 2. They have been referenced by number where applicable to the descriptions and illustrations in this book.

1. Replace '15360' with '**E800H**'. Replace '15361' with '**E801H**'. Replace '1023' with '1919'.

2. Merge 'DEMOSCRN/MRG'. Add line 1, GOSUB40090, line 21, GOSUB40080, line 31, GOSUB40081.

3. Replace each occurrence of '60' with '232'. Replace '255,3' with '127,7'.

4. Replace 'CHR$(191)' with 'CHR$(26);CHR$(32);CHR$(25)'

5. On Model 2, type SYSTEM instead of CMD'S' to return to DOS.

6. On Model 2 the syntax is: DUMP SFILL START=**BFF0**,

7. Does not apply to the Model 2.

8. For the Model 2, the line reads: 10 SYSTEM 'LOAD SFILL'

9. Replace '15360' with '−6144', '15361' with '−6143', '1023' with '1919'.

10. Merge 'DEMOSCRN/MRG'. Add line 6, GOSUB40090, line 31, GOSUB40080, line 51, GOSUB40081.

11. Replace '15360' with '−6144', '15364' with '−6140'.

12. Replace 'CALL **0A7FH**' with 'CALL **0445DH**'.

13. Replace 'JP **0A9AH**' with 'JP **0447AH**'.

14. From TRSDOS READY type STATUS. This gives you the top of memory address. See your owner's manual for information on conditions for using addresses above it.

15. On the Model 2 you can change the memory size from BASIC with the CLEAR command or with '−M:nnnnn' upon loading BASIC. See your owner's manual.

16. Beginning of program text pointer is at **2B4F** – **2B50**. Replace '**40A4**' with '**2B4F**', '**40A5**' with '**2B50**', '16548' with '11087', '16549' with '11088'.

17. Data statement pointer is at **2D0A** – **2D0B**. Replace '**40FF**' with '**2D0A**', '**4100**' with'**2D0B**'.

18. Pointer to beginning address for simple variables is at 11524. Replace '16633' with '11524', '16634' with '11525'. Array pointer is at 11526. Replace '16635' with '11526', '16636' with '11527'. Start of free space pointer is at 11528. Replace '16637' with '11528', '16638' with '11529'.

19. Model 2 BASIC does not reverse the 2 characters in a variable name as it does with the Model 1 and 3. In line 65130, replace 'ZZ$(0)+Z$' with 'ZZ$+Z$(0)'.

20. To use the video display for a move-data demonstration, merge 'DEMOSCRN/MRG'. Add line 11, GOSUB40090, line 79, GOSUB40080, line 81, GOSUB40081.

21. Replace PRINT@ positions according to the following:

```
  0 =    0       256 = 320       512 = 640       768 =  960
 64 =   80       320 = 400       576 = 720       832 = 1040
128 =  160       384 = 480       640 = 800       896 = 1120
192 =  240       448 = 560       704 = 880       960 = 1200
```

22. For the demonstration data, replace '15360' with '−6144', '15872' with '−5184', '512' with '960', '15392' with '−6112', '15373' with '−6131', '15378' with '−6126', '15361' with '−6143', '1023' with '1919'.

23. The following Model 2 changes are required for the first 4 bytes of USR subroutines that receive an integer argument from BASIC:

	Assembly Listing	Magic Array Format	Poke Format
As shown:	CALL 0A7FH NOP	32717,10	205,127,10,0
Change to:	CALL 0445DH NOP	24013,68	205,93,68,0
As shown:	CALL 0A7FH PUSH HL	32717,-6902	205,127,10,229
Change to:	CALL 0445DH PUSH HL	24013,-6844	205,93,68,229
As shown:	CALL 0A7FH LD B,(HL)	32717,17930	205,127,10,70
Change to:	CALL 0445DH LD B,(HL)	24013,17988	205,93,68,70
As shown:	CALL 0A7FH LD DE,0000	32717,4362	205,127,10,17
Change to:	CALL 0445DH LD DE,0000	24013,4420	205,93,68,17
As shown:	CALL 0A7FH LD E,(HL)	32717,24074	205,127,10,94
Change to:	CALL 0445DH LD E,(HL)	24013,24132	205,93,68,94

24. You may merge 'DEMOSCRN/MRG' so you can see the results of your moves on the video display. Add line 11, GOSUB40090, line 139, GOSUB40080, line 151, GOSUB40081. To see the results of your moves, your 'to' address must be between −6144 and −4225.

25. Add line 101, GOSUB40080. Add ':GOSUB40081' just before the ':RETURN' in line 200. Replace the '15360' in line 200 with '−6144'.

26. Replace '**40F9**' with '**2D04**', '**40FA**' with '**2D05**'.

27. Replace &HF9 with '&H04', '&H40' with '&H2D'.

28. Replace '&HB3' with '&HBE', '&H40' with '&H2C'.

29. Models 1 and 3 let you imbed line feeds in your PRINT statements with the down-arrow key. The Model 2 doesn't let you do this. Single PRINT statements that print on multiple video display lines should be replaced by multiple PRINT statements, one for each video display line to be printed. For example, a Model 1 or 3 program line that reads:

```
100 CLS:PRINT"
THIS IS A HEADING
";SG$

...should be replaced by:

100 CLS:PRINT:PRINT"THIS IS A HEADING":PRINTSG$
```

30. Note that some of the video display special characters and PRINT@ positions must be changed to their Model 2 equivalents. See the section on special character conversions.

31. Program text on a Model 2 with 0 files begins at 27714, so we'll need to move up our addresses for the bottom-loaded overlay demonstration. Replace 27000 with 28000, 28000 with 29000, 26999 with 27999, 27999 with 28999, 96 with 72, 109 with 113, 120 with 96, 105 with 109.

32. Make the following replacements for the SUMSNG USR routine:

```
            Assembly Listing  Magic Array Format Poke Format
            ----------------  ------------------ --------------
As shown:   CALL 09B1H        2481               177,9
Change to:  CALL 0438EH       17294              142,67

As shown:   CALL 09C2H        2498,5837,6151     194,9,205,22,7
            CALL 0716H
Change to:  CALL 0439FH       17311,-29235,6208  159,67,205,141,64
            CALL 0408DH

As shown:   LD HL,04121H      8481,321           33,65
Change to:  LD HL,02E0CH      3105,302           12,46
```

33. Make the following replacements for the SUMDBL USR routine:

```
            Assembly Listing  Magic Array Format    Poke Format
            ----------------  ------------------    ------------------
As shown:   LD (40AFH),A      16559,7457,           175,64,33,29,
            LD HL,411DH            -12991,2515          65,205,211,9
            CALL 09D3H
Change to:  LD (2CB6H),A      11446,2081,           182,44,33,8,
            LD HL,2E08H            -13010,17328         46,205,176,67
            CALL 043B0H

As shown:   LD HL,4127H       10017,-12991,         39,65,205,211,
            CALL 09D3H             2515,30669,6156      9,205,119,12
            CALL 0C77H
Change to:  LD HL,2E12H       4641,-13010,          18,46,205,176,
            CALL 043B0H          17328,-19507,6214      67,205,179,70
            CALL 046B3H

As shown:   LD HL,411DH       7457,321              29,65
Change to:  LD HL,2E08H       2081,302              8,46
```

34. Make the following replacements for the COMUNCOM USR routine:

	Assembly Listing	Magic Array Format	Poke Format
As shown:	CALL Ø2857H	22477,-728	87,4Ø
Change to:	CALL Ø5BØ8H	2253,-677	8,91
As shown:	LD DE,(Ø4ØD4H)	16596	212,64
Change to:	LD DE,(Ø2CDBH)	11483	219,44

35. The date can be accessed from BASIC as DATE$. Its format is different than that of the Models 1 & 3. You can access and change the date with peeks and pokes:

```
PEEK (72) = Day of Month          PEEK (73) = Month
PEEK (76) = Year                  PEEK (77) = Century
```

To get an 8-byte date string you can use:

```
RIGHT$(STR$(PEEK(73)),2)+"/"+
    RIGHT$(STR$(PEEK(72)),2)+"/"+RIGHT$(STR$(PEEK(76)),2)
```

36. The up-arrow is used to indicate exponentiation on the models 1 and 3. On the Model 2 you can use shift-6. Be aware that some printers display the up-arrow character as a left-bracket.

37. Make the following replacements for the BITSRCH, KWKARRAY and SEARCH1 USR routines:

	Assembly Listing	Magic Array Format	Poke Format
As shown:	JP ØA9AH	2714	154,1Ø
Change to:	JP Ø447AH	1753Ø	122,68

38. You will need to do this in an image of the video display in protected memory. If you merge 'DEMOSCRN/MRG' you can GOSUB40080 before doing a LSET or RSET and GOSUB40081 immediately after. Replace '15' with '23', '15360' with '−6144' and '64' with '80'.

39. Merge 'DEMOSCRN/MRG'. Add line 2, GOSUB40090. Add 'GOSUB40080:' as the first command in line 250, ':GOSUB40081' as the last command in line 250. Replace '15360' with '−6144'. Change each 'CHR$(31)' to 'CHR$(24)'.

40. Merge 'DEMOSCRN/MRG'. Add line 1, GOSUB40090. Add 'GOSUB40080:' as the first command and ':GOSUB40081' as the last command, in lines 111, 131 and 151. Replace each '15360' with '−6144' and each '16372' with '−5132'.

41. Make the following replacements for the SORT3 USR routines:

	Assembly Listing	Magic Array Format	Poke Format
As shown:	JP ØA9AH	-25917,1Ø	154,1Ø
Change to:	JP Ø447AH	31427,68	122,68

42. Merge 'DEMOSCRN/MRG'. Add at line 1, GOSUB40090. At line 141 and 241, add GOSUB40080. At line 151 and 251, add GOSUB40081. Change each '15360' to '−6144'.

43. Use the Model 2 version of the video display string pointer subroutine, 40070.

44. Replace '64' with '80', '960' with '1840', '1024' with '1920'.

45. Simply 'LINE INPUT' each line and PRINT it. LSET cannot be used with the Model 2 version of subroutine 40070. Use ';' following your PRINT statement. Only the top 23 lines should be displayed if you want to avoid an unwanted scroll.

46. Use the ROW(0) function to find the cursor row, POS(0) for the column. Use ROW(0) * 80 + POS(0) to find the cursor PRINT@ position.

47. Use subroutine 40500.

48. To enable and disable the BREAK key you can use supervisor call 3. Subroutines 40090 and 40091 must be present and 40090 must already have been executed.

```
To lock out the BREAK key:  UV%(1)=3 : UV%(3)=0 : GOSUB40091
To restore the BREAK key:   UV%(1)=3 : UV%(3)=24681 : GOSUB40091
```

49. The following modifications are required for the free-form video display program. During operation, F1 corresponds to up-arrow, F2 to down-arrow, tab to right-arrow and back-space to right-arrow. The arrow keys correspond to the shifted-arrow keys for the Models 1 and 3 version.

a. Merge 'DEMOSCRNMRG'. Add line 11, GOSUB40090.
b. In line 20, CHR$'s 9, 8, 91, 10, 13, 25, 24, 26 and 27 should be replaced by CHR$'s 9, 8, 1, 2, 13, 29, 28, 31 and 30, respectively.
c. Add ':GOSUB40080' as the last command in line 100.
d. Replace '15360' with '−6144' in lines 120, 2001 and 2004.
e. Delete 'POKE PX,95' from line 120.
f. Replace line 125 with PRINT@PO,"";:GOSUB40500
g. Add the single-key subroutine, 40500. Delete 40600.
h. Add 'GOSUB40081:' as the first command in line 132 and just before the 'GOTO120' in line 140. Add as line 156, 'GOSUB40080'.
i. In lines 1001 through 1006 change '1024' to '1920', '64' to '80' and '960' to '1840'.
j. In lines 2001 and 2002 insert 'GOSUB40081:' just before the final 'RETURN'. In line 2001 replace each '64' with '80', '62' with '78'.
k. In line 2002 replace each '(POANDNOT−64)' with '(PO MOD 80)'.
l. In lines 2002 through 2010 replace each '64' with '80', '960' with '1840', '16319' with '−4305', '16383' with '−4225', 'CHR$(30)' with 'CHR$(23)'.
m. In line 2010, add 'GOSUB40080:' as the first command and 'GOSUB40081:' just before the 'RETURN'.

50. Replace '1017' with '1913', 'CHR$(30)' with 'CHR$(23)'.

51. Use the modified screen save and flashback subroutine, 40200, as shown eariler in this section.

52. Merge 'DEMOSCRN/MRG'. Add at line 1, 'GOSUB40090', at line 42, 'GOSUB40080', at line 52, 'GOSUB40081'. Change each '512' to '960', '15360' to '−6144', '15872' to '−5184'.

53. Replace '64' with '80', '30' with '23', '1000' to '1896'.

54. Merge 'DEMOSCRN/MRG'. In lines 40712 and 40822 add 'GOSUB40080:' as the first command and add 'GOSUB40081:' just after the 'J=USR(0):'. Replace '64' with '80', '15360' with '−6144', '15424' with '−6064', '30' with '23', '65' with '81'. Change line 40803 to 'GOSUB40820:GOTO40800'. Change line 40804 to 'GOSUB40830:GOTO40800'. At line 5, add 'GOSUB40090.'

55. Merge 'DEMOSCRN/MRG'. Before each 'DEFUSR' insert 'GOSUB40080'. After each 'USR(0)' insert 'GOSUB40081'. Delete all tests on PEEK(14951) and replace with 'GOTO40910'. See note 54 for other modifications that may be required.

56. Delete the first 2 pokes in line 1. You can set the memory size to −22686 in with the CLEAR command in line 1.

57. The unscrolled video entry handler allows for PRINT@ positions ranging from 0 to 999. You can change the routines to allow for a 4-digit position parameter, but a simpler modification that lets you take advantage of the full Model 2 screen is to express your position parameters as the positions you want divided by 2. Then you can multiply PO% by 2 in the lines where it is assigned a value. The lines are 46021, 46030 and 46060.

58. Note that on the Model 2 you can use SYSTEM 'FORMS' to set the line printer. On Models 1 and 2, memory address 16425 maintains a count of the current line number. 'POKE 16425,1' should be replaced by the appropriate FORMS command to set the top of form. Depending on your printer type, it may be necessary to change references to 'LPRINT CHR$(12);' to the appropriate command that advances to the next page.

59. Make the following changes to 'DOCLIST/BAS'.

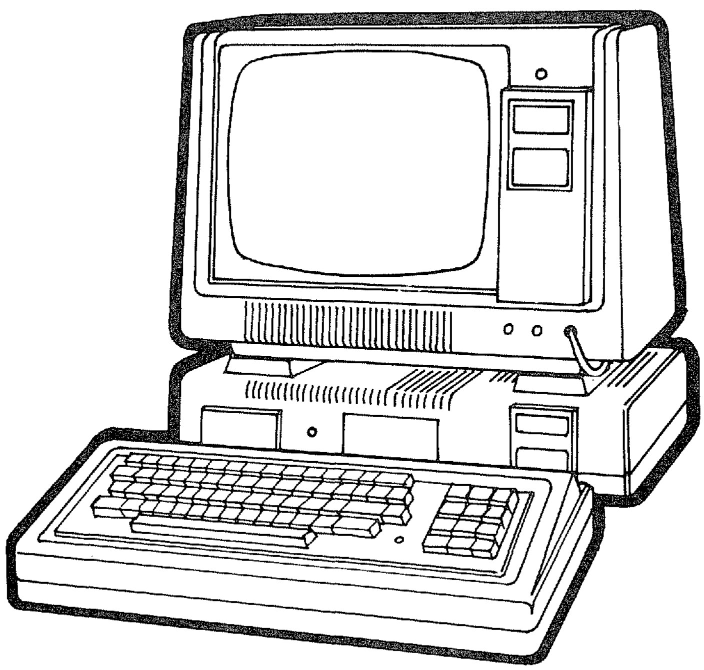
a. The reserved word list begins at 10323. In line 51 change '5712' to '10323'.
b. Change the PRINT@ commands. In line 50 change '512' to '960'. In line 52, change '544' to '992'. In line 1110, change '64' to '80'. Change each '704' to '1280'.
c. The disk error codes are different. Change '106' in line 1151 to '53'. Between the 'ELSE' and 'PRINT' in line 1151, insert 'IF ERR=54 THEN RESUME NEXT ELSE'.
d. Line 140 should simply say, 'GOSUB2100'.
e. Change each 'CHR$(31)' to 'CHR$(24)'.
f. In line 55, change '16' to '13'. In line 3220 the string should be replaced with ' 128 138 139 143 158 165 171 183'. In line 4135, change '149' to '146', '141' to '138'. In line 4182, change '202' to '199', '141' to '138'. In line 4184, change '135' to '132'. In line 4190 change '147' to '144'. In line 4210 change '141' to '138', '158' to '157'. In line 4186, '143' should be changed to '140'.

60. The disk error codes are different on the Model 2. Replace '57' with '56', '64' with '62', '67' with '56', '63' with '61', '61' with '59'.

61. Change '960' in lines 58991 and 58993 to '1840'. Delete line 58994 and change line 58990 to '58990 REM'. Replace 'CHR$(31)' with 'CHR$(24)'. Replace 'ERR/2' in line 58991 with 'ERR'.

62. The following changes are required for 'MERGEPRO/BAS' on the Model 2:

a. In line 10000, replace '32717,−6902' with '24013, −6844'.
b. Change each 'CHR$(31)' to 'CHR$(24)', 'CHR$(30)' to 'CHR$(23)'.
c. Change each 'CHR$(91)' to 'CHR$(1)'. You will use the F1 key instead of up-arrow to correct errors.
d. In line 151 change '471' to '583', in line 142 change '406' to '502', in line 133 change '342' to '422', in line 230 change '598' to '742', in line 240 change '608' to '752'.
e. In line 121, delete all between the 'ELSE' and the second 'CLOSE'. Delete lines 128 and 122.
f. Line 123, change 'PB!=1' to 'PB!=2'. In line 421, delete all after 'PB!=1'.
g. In lines 470 and 471, change '15360' to '27779'.
h. In lines 241 and 1020 change '141' to '138'. In lines 242 and 1020, change '145' to '142'. In line 243 change '202' to '199'. In line 244 change '149' to '146'. In line 1020 change '161' to '149'.



The Optional Basic Faster & Better Companion Disks

Contact the publisher for purchasing information

The 'BASIC Faster & Better' program disks contain the major subroutines, function calls, USR routines, demonstration programs and utilities, presented in this book. In addition to saving you hours of work, typing and correcting the programs, they give you a convenient library that you can merge from, whenever you want. Each disk is supplied in 35-track, single-density, format.

BFBLIB contains the following function, subroutine and utility programs:

```
ANALYZE/BAS        BASECONV/BAS       CHANGE/BAS
DATECOMP/BAS       DOCLIST/BAS        DOSCHECK/BAS
FUNCTION/LIB       KILLFILE/BAS       LINEMOD/BAS
MERGEPRO/BAS       MOVEDATA/BAS       VSHEETS/BAS
SEARCH2/BAS        SROUTINE/LIB       VIDEOGEN/BAS
VDRIVE/BAS         VDRIVE2/BAS        USRDATA1/LIB
USRDATA2/LIB       USRFILE/RND
```

BFBDEM contains the following demonstration programs:

```
BITSRCH/DEM        BITMAPFN/DEM       FREEFORM/DEM
ELEMDUP/DEM        FLASH/DEM          JOURNEY/DEM
HZIO/DEM           IDARRAY/DEM        MOVEX/DEM
KWKARRAY/DEM       MASTER/BOV         OVERLAY2/BOV
OVERLAY1/BOV       OVERLAY1/TOV       OVERLAYT/DEM
OVERLAY2/TOV       OVERLAYB/DEM       SORT2/DEM
SCROLLUP/DEM       SEARCH1/DEM        SUMDBL/DEM
SORT3/DEM          VARPASS/DEM        VARPASS/RCV
SUMSNG/DEM         VHANDLER/DEM       UPDOWN/DEM
VETOM/DEM          COMUNCOM/DEM
```

The files that have the 'BAS' and 'DEM' extensions can be run directly from BASIC. 'DEM' is used for programs whose primary purpose is to demonstrate one or more subroutines, function calls or USR routines. 'BAS' is used when the program can be used for other purposes besides demonstrations. As a general rule, you should specify 3 files when entering BASIC. You don't need to set a particular memory size, but 32K of memory, at least, is required for most of the programs to function.

The files that have the 'LIB' extension are 'library files'. They contain groups of BASIC function calls, subroutines or data statements that can be merged into your own programs. You can extract the functions that you wish to use with the MERGEPRO/BAS program. Another method is to delete all unwanted lines, and save the remainder as an ASCII file, and then merge the file into your own program.

The programs with extensions 'BOV', 'TOV', and 'RCV' are used for the overlay and variable passing demonstrations. They are BASIC programs, but cannot be executed directly. They are automatically 'RUN' by their related 'DEM' programs: 'OVERLAYB/DEM', 'OVERLAYT/DEM' and 'VARPASS/DEM'.

USRFILE/RND is the only file that is not in BASIC. It is a random disk file that contains the machine language code for each of the USR routines.

The Library Disk – BFBLIB

ANALYZE/BAS

This is the Active Variable Analyzer program. It is used to list all the variables, and arrays, that are active in any BASIC program you may be running. To use it, you will need to load it, then save it on another disk in ASCII format, (with the 'A' option).

When you are debugging a program, and you want to display all active variables, you can temporarily merge it in. To display the active variables and arrays, at any point in the program, hit 'BREAK' and then 'GOSUB 65000'.

- For more details see page 44

BASECONV/BAS

This, to save disk space, is a combination of two useful demonstration programs. The DECTOHEX/BAS program has been renumbered starting at line 1000. It lets you convert any decimal number from 32768 to 65535 to hexadecimal. The BASECONV/DEM program has been renumbered starting at line 2000. It lets you convert from decimal to any other base. When you run BASECONV/BAS a menu is displayed so that you can select either program.

- For more details see page 84

CHANGE/BAS

This program demonstrates the substring replacement subroutine. You can use it to make changes to BASIC program files that have been saved in ASCII format, You can also use it to replace selected strings within other types of sequential files, such as those created by word processing programs.

- For more details see page 95

DATECOMP/BAS

The purpose of this program is to demonstrate, and test, the date computation function calls, but it's handy to have around as a 'perpetual calendar'.

- For more details see page 112

DOCLIST/BAS

This program lets you produce 'pretty-printed' listings of any BASIC program. Be sure that the program you wish to list has been saved on disk in compressed format, (without the 'A' option).

Depending on the type of line printer you have, you may need to delete the ';' following the 'LPRINT CHR$(12)' in line 70 and 3240.

- For more details see page 231

DOSCHECK/BAS

You'll want to run this program if you've got a disk operating system that is different from those listed in appendices 2, 3, and 4. Once you've run it, you can update this book by jotting down the addresses that are produced.

Be aware that a temporary file is created on drive 0, so the disk must not be write protected!

- For more details see page 240

FUNCTION/LIB

This file contains all the function definitions explained in this book. The functions occupy lines 1 through 55. They are indexed alphabetically, and by line number, in appendix 8.

It is most convenient to merge and renumber the functions you want with the MERGEPRO/BAS program. Or, if you wish, you can load FUNCTION/LIB, delete the lines you don't want, renumber the remaining lines (if you have a RENUM program), save them in ASCII format, and then merge them into the program you are writing.

When you wish to test a particular function, you can temporarily add a few program lines above line 55. Or, you can simply load FUNCTION/LIB and type RUN. Then, while in BASIC's command mode, you can test examples as they are shown in the book or you can try your own tests.

Remember that you *must* have loaded COMUNCOM, and done a DEFUSR, if you wish to test the FNKM$ function. (This is all done for you in the COMUNCOM/DEM program).

Also, be aware that the FNBN$ function, because of its length, cannot be merged into another program. (You'll get a 'direct statement in file error'). To solve this problem, you can temporarily delete a number of characters from the end of the line. After you've merged it, you can replace the missing characters with BASIC's edit capability.

KILLFILE/BAS

This program demonstrates the command string peel-off subroutine. You can use it when you have several files that you want to KILL.

- For more details see page 94

LINEMOD/BAS

You'll need to load LINEMOD/BAS and then save it on another disk in ASCII format, (with the 'A' option). It is designed to be temporarily merged into a program so that you can poke graphics characters into the text.

- For more details see page 192

MERGEPRO/BAS

This is a utility that lets you merge and renumber selected lines from one or more BASIC program files. You can use it to pull selected lines from any programs that you have written. It is especially useful when you want to build programs by extracting lines from FUNCTION/LIB, SROUTINE/LIB, USRDATA1/LIB and USRDATA2/LIB.

Remember that you will need to specify at least 1 file when loading BASIC. If you have only 1 or 2 disk drives, you may remove the disk containing MERGEPRO/BAS when you see the prompt, 'SAVE USING PROGRAM NAME'. Then you can insert the disk on which you want to save the new program lines.

- For more details see page 236

MOVEDATA/BAS

This program demonstrates the 'Move-Data magic array'. You can use it to duplicate patterns in memory, or to copy data from one address to another.

Be sure to be careful with this one! Until you are sure of what you are doing you should write-protect, or remove, any disks that are in the drives.

- For more details see page 52

VSHEETS/BAS

This program prints video display planning sheets on your line printer. Depending on the type of printer you have, you may need to delete the ';' following the 'LPRINT CHR$(12)'.

- For more details see page 179

SEARCH2/BAS

This program demonstrates the SEARCH2 USR routine. It can be handy whenever you wish to find selected strings in memory.

- For more details see page 159

SROUTINE/LIB

This is a large BASIC program file that contains all the major subroutines .. They are indexed by line number in appendix 9.

You can load SROUTINE/LIB and delete all lines except those you need, save them in ASCII,and then merge them into your program. An alternative metod is to use the MERGEPRO/BAS program to pull out and renumber the lines you want.

If you wish, you can test many of the subroutines directly from BASIC's command mode. Lines 1 through 99 of SROUTINE/LIB contain logic to CLEAR 1000, DEFINT A-Z and to load the Move-Data magic array, (which is required by some of the subroutines). At line 99 is an END statement. You can type RUN and these 'housekeeping' functions are done for you. Then, from 'READY' you can load the required variables and GOSUB to the proper line number to test a subroutine. Or, if you wish, you can temporarily insert logic between lines 50 and 99 to test any of the subroutines.

VIDEOGEN/BAS

This is a bonus program that combines some of the routines and techniques discussed in chapter 13. It lets you draw video displays with graphics characters, and you can assign any graphics character to the CLEAR key. You can also select 'horizontal' or 'vertical' mode for graphics characters. Vertical mode makes it easy to draw vertical bars, while horizontal mode positions the cursor to the right of the last graphics character printed, making it very easy to draw horizontal patterns.

VIDEOGEN/BAS also contains a subroutine at line 57400 that lets you save, by number, the video displays you create into any random disk file, and load them back. This subroutine, unlike those listed in the book, uses the Move-Data magic array to transfer data from the screen to the disk buffer. It automatically computes the disk buffer address, so it is compatible with any DOS you may be using.

When you enter VIDEOGEN's 'command mode', to change the graphics character, (or load, or save a screen), the display you were working with is temporarily saved in an integer array. Upon returning to 'display mode', the screen is instantly recalled – flashed-back!

You also have the ability to turn on or off a position indicator in the bottom right corner of the screen. It displays the current PRINT@ position of the cursor.

All the commands available to you are explained by prompts on the screen. To use the program, specify at least 1 file upon loading BASIC and simply RUN 'VIDEOGEN/BAS'.

VDRIVE/BAS

If you have a Model 1 and you've installed an upper/lower case modification, you may need a lower case driver program. (The programs that use the video display string pointer subroutine, line 40070, will almost certainly benefit from using a driver program). You may use the driver program provided by Radio Shack, if you want to, or VDRIVE/BAS.

You may need to modify the addresses used by VDRIVE/BAS according to the instructions in the book. Also, be sure to specify a memory size so that the driver will be protected.

- For more details see page 166

VDRIVE2/BAS

This is a bonus program that uses the logic in VDRIVE/BAS in another way. It loads the video display driver below the program text and then it updates the beginning of text pointers so that the next program you load or run starts just above the driver. During execution of VDRIVE2/BAS, its line 0 is replaced by the machine language upper/lower case logic. The final command in the program is a 'NEW' so that you're ready to go. To use it, you simply RUN 'VDRIVE2/BAS'. Then you load or run the program you want. You don't need to set a special memory size and it can be used without modification for TRS-80's with any amount of memory!

VDRIVE2/BAS is documented in more detail with remark statements in the program text. You'll only need it if you've installed an upper/lower case modification your Model 1, but the same technique can be valuable in many other machine language programs.

Be aware that for some disk operating systems there may be a conflict in the memory addresses used. For example, with NEWDOS 2.1 you may need to re-boot before displaying a disk directory.

USRDATA1/LIB

This is a BASIC program file that contains DATA statements for all the USR routines discussed in this book. Each group of DATA lines contains a list of numbers that can be poked into memory. To use them, you can merge the lines you need into your program. Then your program can read the numbers and poke them into contiguous addresses in any part of protected memory. Once they are in memory, if you wish, you can go to DOS READY and 'DUMP' the desired USR routines from memory to disk.

You can use the MERGEPRO/BAS program to extract and renumber the lines you want, or you can load USRDATA1/LIB and delete the lines you don't need. (Note: In most cases, there will be no need to renumber data statements, unless you wish to change the sequence in which they will be read. Your program logic doesn't need to pass through the data statements).

Appendix 10 indexes the data statements by line number for you.

USRDATA2/LIB

This is another BASIC program file that contains DATA statements for all the USR routines discussed in this book. It contains numbers that can be read into integer arrays when you wish to use the 'magic array' technique for loading and executing USR subroutines.

You can use the MERGEPRO/BAS program to extract and renumber the lines you need or you can delete the unneeded lines and merge those that remain into your program.

Appendix 10 indexes the data statements by line number for you.

USRFILE/RND

This is a random file in which each physical record contains a USR routine. To use it, you can open 'USRFILE/RND' as a random file from any BASIC program. Then you can do a DEFUSR, specifying the memory address of the disk buffer you are using. The addresses are listed in appendix 3.

To use the routine you want, simply GET the proper record, as listed in appendix 10 and make your USR call. It will be executed in the protected memory of the disk buffer. You don't need to reserve a special memory size!

The Demonstration Disk – BFBDEM

BITSRCH/DEM demonstrates the BITSRCH USR subroutine for searching bit-map strings.

- For more details see page 123

BITMAPFN/DEM demonstrates the bit-map string function calls.

- For more details see page 120

COMUNCOM/DEM demonstrates the use of the COMUNCOM USR routine and the FNKM$ function, to compress and uncompress strings. You will need to make a minor change if you are using a disk operating system other than NEWDOS 2.1.

- For more details see page 95

ELEMDUP/DEM is the array element duplication demonstration program.

- For more details see page 125

FLASH/DEM demonstrates the screen save and instant recall subroutine.

- For more details see page 194

FREEFORM/DEM is the free-form video display program. It demonstrates repeating key capablities, a flashing cursor, insertions, and deletions.

- For more details see page 176

HZIO/DEM demonstrates the horizontal input/output subroutine for data entry and display.

- For more details see page 196

IDARRAY/DEM is a demonstration of array element insertions and deletions with the IDARRAY USR subroutine.

- For more details see page 127

JOURNEY/DEM scrolls the video display through 64K of memory, showing the current address at the bottom of the screen. It uses the MOVEX USR routine, so you'll need to make a minor modification if you are using a disk operating system other than NEWDOS 2.1.

- For more details see page 55

KWKARRAY/DEM uses the video display to demonstrate the commands of the KWKARRAY USR routine.

- For more details see page 145

MOVEX/DEM demonstrates the MOVEX USR subroutine. Again, you will need to make a minor modification if you are using a disk operating system other than NEWDOS 2.1.

- For more details see page 55

MASTER/BOV is part of the bottom-loaded overlay demonstration. You should not run it directly. It is loaded by OVERLAYB/DEM.

OVERLAY1/BOV is part of the bottom-loaded overlay demonstration. You should not run it directly. It is loaded by OVERLAYB/DEM.

OVERLAY1/TOV is part of the top-loaded overlay demonstration. You should not run it directly. It is loaded by OVERLAYT/DEM.

OVERLAY2/BOV is part of the bottom-loaded overlay demonstration. You should not run it directly. It is loaded by OVERLAYB/DEM.

OVERLAY2/TOV is part of the top-loaded overlay demonstration. You should not run it directly. It is loaded by OVERLAYT/DEM.

OVERLAYB/DEM is the bottom-loaded overlay demonstration.

- For more details see page 71

OVERLAYT/DEM is the top-loaded overlay demonstration.

- For more details see page 67

SCROLLUP/DEM demonstrates split-screen scrolling using random data.

- For more details see page 200

SEARCH1/DEM demonstrates the SEARCH1 USR subroutine for high-speed searches of string arrays.

- For more details see page 131

SORT2/DEM uses the video display to demonstrate the high-speed memory sort performed by the SORT2 USR subroutine.

- For more details see page 152

SORT3/DEM uses the video display to demonstrate the method of sorting by insertion used by the SORT3 USR subroutine.

- For more details see page 155

SUMDBL/DEM is a demonstration of the SUMDBL USR subroutine.

- For more details see page 82

SUMSNG/DEM demonstrates the SUMSNG USR subroutine.

- For more details see page 82

VARPASS/DEM shows how you can pass variables from one program to another. It creates some demonstration data and passes it to VARPASS/RCV.

- For more details see page 58

VARPASS/RCV is the receiving program in the variable passing demonstration. It is loaded and run by VARPASS/DEM. You should not run it directly.

VETOM/DEM demonstrates the scrolled video entry handler. If you wish to test the disk save and load capabilities you should specify at least 1 file upon loading BASIC, and have a formatted disk available – with several grans of free space.

Be aware that it automatically modifies the memory size setting. After running the program you can restore the original memory size by re-booting, or by poking the memory size pointers.

- For more details see page 211

VHANDLER/DEM is a demonstration of the unscrolled video handler. It also demonstrates all the INKEY subroutines. You will need a disk that isn't write protected in drive 0. The program opens but does not actually use a temporary file, 'TEST', on drive 0. You will need to specify at least 1 file upon loading BASIC. Also, if you've got a Model 1 with an upper/lower case kit installed, be sure that you've loaded a video driver such as VDRIVE/BAS or VDRIVE2/BAS.

- For more details see page 229

UPDOWN/DEM demonstrates the up and down scrolling subroutines to scroll data from an array onto the video display.

- For more details see page 202

Decimal To Hexadecimal Conversion

	00	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
00	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
01	256	272	288	304	320	336	352	368	384	400	416	432	448	464	480	496
02	512	528	544	560	576	592	608	624	640	656	672	688	704	720	736	752
03	768	784	800	816	832	848	864	880	896	912	928	944	960	976	992	1008
04	1024	1040	1056	1072	1088	1104	1120	1136	1152	1168	1184	1200	1216	1232	1248	1264
05	1280	1296	1312	1328	1344	1360	1376	1392	1408	1424	1440	1456	1472	1488	1504	1520
06	1536	1552	1568	1584	1600	1616	1632	1648	1664	1680	1696	1712	1728	1744	1760	1776
07	1792	1808	1824	1840	1856	1872	1888	1904	1920	1936	1952	1968	1984	2000	2016	2032
08	2048	2064	2080	2096	2112	2128	2144	2160	2176	2192	2208	2224	2240	2256	2272	2288
09	2304	2320	2336	2352	2368	2384	2400	2416	2432	2448	2464	2480	2496	2512	2528	2544
0A	2560	2576	2592	2608	2624	2640	2656	2672	2688	2704	2720	2736	2752	2768	2784	2800
0B	2816	2832	2848	2864	2880	2896	2912	2928	2944	2960	2976	2992	3008	3024	3040	3056
0C	3072	3088	3104	3120	3136	3152	3168	3184	3200	3216	3232	3248	3264	3280	3296	3312
0D	3328	3344	3360	3376	3392	3408	3424	3440	3456	3472	3488	3504	3520	3536	3552	3568
0E	3584	3600	3616	3632	3648	3664	3680	3696	3712	3728	3744	3760	3776	3792	3808	3824
0F	3840	3856	3872	3888	3904	3920	3936	3952	3968	3984	4000	4016	4032	4048	4064	4080
10	4096	4112	4128	4144	4160	4176	4192	4208	4224	4240	4256	4272	4288	4304	4320	4336
11	4352	4368	4384	4400	4416	4432	4448	4464	4480	4496	4512	4528	4544	4560	4576	4592
12	4608	4624	4640	4656	4672	4688	4704	4720	4736	4752	4768	4784	4800	4816	4832	4848
13	4864	4880	4896	4912	4928	4944	4960	4976	4992	5008	5024	5040	5056	5072	5088	5104
14	5120	5136	5152	5168	5184	5200	5216	5232	5248	5264	5280	5296	5312	5328	5344	5360
15	5376	5392	5408	5424	5440	5456	5472	5488	5504	5520	5536	5552	5568	5584	5600	5616
16	5632	5648	5664	5680	5696	5712	5728	5744	5760	5776	5792	5808	5824	5840	5856	5872
17	5888	5904	5920	5936	5952	5968	5984	6000	6016	6032	6048	6064	6080	6096	6112	6128
18	6144	6160	6176	6192	6208	6224	6240	6256	6272	6288	6304	6320	6336	6352	6368	6384
19	6400	6416	6432	6448	6464	6480	6496	6512	6528	6544	6560	6576	6592	6608	6624	6640
1A	6656	6672	6688	6704	6720	6736	6752	6768	6784	6800	6816	6832	6848	6864	6880	6896
1B	6912	6928	6944	6960	6976	6992	7008	7024	7040	7056	7072	7088	7104	7120	7136	7152
1C	7168	7184	7200	7216	7232	7248	7264	7280	7296	7312	7328	7344	7360	7376	7392	7408
1D	7424	7440	7456	7472	7488	7504	7520	7536	7552	7568	7584	7600	7616	7632	7648	7664
1E	7680	7696	7712	7728	7744	7760	7776	7792	7808	7824	7840	7856	7872	7888	7904	7920
1F	7936	7952	7968	7984	8000	8016	8032	8048	8064	8080	8096	8112	8128	8144	8160	8176
20	8192	8208	8224	8240	8256	8272	8288	8304	8320	8336	8352	8368	8384	8400	8416	8432
21	8448	8464	8480	8496	8512	8528	8544	8560	8576	8592	8608	8624	8640	8656	8672	8688
22	8704	8720	8736	8752	8768	8784	8800	8816	8832	8848	8864	8880	8896	8912	8928	8944
23	8960	8976	8992	9008	9024	9040	9056	9072	9088	9104	9120	9136	9152	9168	9184	9200
24	9216	9232	9248	9264	9280	9296	9312	9328	9344	9360	9376	9392	9408	9424	9440	9456
25	9472	9488	9504	9520	9536	9552	9568	9584	9600	9616	9632	9648	9664	9680	9696	9712
26	9728	9744	9760	9776	9792	9808	9824	9840	9856	9872	9888	9904	9920	9936	9952	9968
27	9984	10000	10016	10032	10048	10064	10080	10096	10112	10128	10144	10160	10176	10192	10208	10224
28	10240	10256	10272	10288	10304	10320	10336	10352	10368	10384	10400	10416	10432	10448	10464	10480
29	10496	10512	10528	10544	10560	10576	10592	10608	10624	10640	10656	10672	10688	10704	10720	10736
2A	10752	10768	10784	10800	10816	10832	10848	10864	10880	10896	10912	10928	10944	10960	10976	10992
2B	11008	11024	11040	11056	11072	11088	11104	11120	11136	11152	11168	11184	11200	11216	11232	11248
2C	11264	11280	11296	11312	11328	11344	11360	11376	11392	11408	11424	11440	11456	11472	11488	11504
2D	11520	11536	11552	11568	11584	11600	11616	11632	11648	11664	11680	11696	11712	11728	11744	11760
2E	11776	11792	11808	11824	11840	11856	11872	11888	11904	11920	11936	11952	11968	11984	12000	12016
2F	12032	12048	12064	12080	12096	12112	12128	12144	12160	12176	12192	12208	12224	12240	12256	12272

	00	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
30	12288	12304	12320	12336	12352	12368	12384	12400	12416	12432	12448	12464	12480	12496	12512	12528
31	12544	12560	12576	12592	12608	12624	12640	12656	12672	12688	12704	12720	12736	12752	12768	12784
32	12800	12816	12832	12848	12864	12880	12896	12912	12928	12944	12960	12976	12992	13008	13024	13040
33	13056	13072	13088	13104	13120	13136	13152	13168	13184	13200	13216	13232	13248	13264	13280	13296
34	13312	13328	13344	13360	13376	13392	13408	13424	13440	13456	13472	13488	13504	13520	13536	13552
35	13568	13584	13600	13616	13632	13648	13664	13680	13696	13712	13728	13744	13760	13776	13792	13808
36	13824	13840	13856	13872	13888	13904	13920	13936	13952	13968	13984	14000	14016	14032	14048	14064
37	14080	14096	14112	14128	14144	14160	14176	14192	14208	14224	14240	14256	14272	14288	14304	14320
38	14336	14352	14368	14384	14400	14416	14432	14448	14464	14480	14496	14512	14528	14544	14560	14576
39	14592	14608	14624	14640	14656	14672	14688	14704	14720	14736	14752	14768	14784	14800	14816	14832
3A	14848	14864	14880	14896	14912	14928	14944	14960	14976	14992	15008	15024	15040	15056	15072	15088
3B	15104	15120	15136	15152	15168	15184	15200	15216	15232	15248	15264	15280	15296	15312	15328	15344
3C	15360	15376	15392	15408	15424	15440	15456	15472	15488	15504	15520	15536	15552	15568	15584	15600
3D	15616	15632	15648	15664	15680	15696	15712	15728	15744	15760	15776	15792	15808	15824	15840	15856
3E	15872	15888	15904	15920	15936	15952	15968	15984	16000	16016	16032	16048	16064	16080	16096	16112
3F	16128	16144	16160	16176	16192	16208	16224	16240	16256	16272	16288	16304	16320	16336	16352	16368
40	16384	16400	16416	16432	16448	16464	16480	16496	16512	16528	16544	16560	16576	16592	16608	16624
41	16640	16656	16672	16688	16704	16720	16736	16752	16768	16784	16800	16816	16832	16848	16864	16880
42	16896	16912	16928	16944	16960	16976	16992	17008	17024	17040	17056	17072	17088	17104	17120	17136
43	17152	17168	17184	17200	17216	17232	17248	17264	17280	17296	17312	17328	17344	17360	17376	17392
44	17408	17424	17440	17456	17472	17488	17504	17520	17536	17552	17568	17584	17600	17616	17632	17648
45	17664	17680	17696	17712	17728	17744	17760	17776	17792	17808	17824	17840	17856	17872	17888	17904
46	17920	17936	17952	17968	17984	18000	18016	18032	18048	18064	18080	18096	18112	18128	18144	18160
47	18176	18192	18208	18224	18240	18256	18272	18288	18304	18320	18336	18352	18368	18384	18400	18416
48	18432	18448	18464	18480	18496	18512	18528	18544	18560	18576	18592	18608	18624	18640	18656	18672
49	18688	18704	18720	18736	18752	18768	18784	18800	18816	18832	18848	18864	18880	18896	18912	18928
4A	18944	18960	18976	18992	19008	19024	19040	19056	19072	19088	19104	19120	19136	19152	19168	19184
4B	19200	19216	19232	19248	19264	19280	19296	19312	19328	19344	19360	19376	19392	19408	19424	19440
4C	19456	19472	19488	19504	19520	19536	19552	19568	19584	19600	19616	19632	19648	19664	19680	19696
4D	19712	19728	19744	19760	19776	19792	19808	19824	19840	19856	19872	19888	19904	19920	19936	19952
4E	19968	19984	20000	20016	20032	20048	20064	20080	20096	20112	20128	20144	20160	20176	20192	20208
4F	20224	20240	20256	20272	20288	20304	20320	20336	20352	20368	20384	20400	20416	20432	20448	20464
50	20480	20496	20512	20528	20544	20560	20576	20592	20608	20624	20640	20656	20672	20688	20704	20720
51	20736	20752	20768	20784	20800	20816	20832	20848	20864	20880	20896	20912	20928	20944	20960	20976
52	20992	21008	21024	21040	21056	21072	21088	21104	21120	21136	21152	21168	21184	21200	21216	21232
53	21248	21264	21280	21296	21312	21328	21344	21360	21376	21392	21408	21424	21440	21456	21472	21488
54	21504	21520	21536	21552	21568	21584	21600	21616	21632	21648	21664	21680	21696	21712	21728	21744
55	21760	21776	21792	21808	21824	21840	21856	21872	21888	21904	21920	21936	21952	21968	21984	22000
56	22016	22032	22048	22064	22080	22096	22112	22128	22144	22160	22176	22192	22208	22224	22240	22256
57	22272	22288	22304	22320	22336	22352	22368	22384	22400	22416	22432	22448	22464	22480	22496	22512
58	22528	22544	22560	22576	22592	22608	22624	22640	22656	22672	22688	22704	22720	22736	22752	22768
59	22784	22800	22816	22832	22848	22864	22880	22896	22912	22928	22944	22960	22976	22992	23008	23024
5A	23040	23056	23072	23088	23104	23120	23136	23152	23168	23184	23200	23216	23232	23248	23264	23280
5B	23296	23312	23328	23344	23360	23376	23392	23408	23424	23440	23456	23472	23488	23504	23520	23536
5C	23552	23568	23584	23600	23616	23632	23648	23664	23680	23696	23712	23728	23744	23760	23776	23792
5D	23808	23824	23840	23856	23872	23888	23904	23920	23936	23952	23968	23984	24000	24016	24032	24048
5E	24064	24080	24096	24112	24128	24144	24160	24176	24192	24208	24224	24240	24256	24272	24288	24304
5F	24320	24336	24352	24368	24384	24400	24416	24432	24448	24464	24480	24496	24512	24528	24544	24560

	00	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
60	24576	24592	24608	24624	24640	24656	24672	24688	24704	24720	24736	24752	24768	24784	24800	24816
61	24832	24848	24864	24880	24896	24912	24928	24944	24960	24976	24992	25008	25024	25040	25056	25072
62	25088	25104	25120	25136	25152	25168	25184	25200	25216	25232	25248	25264	25280	25296	25312	25328
63	25344	25360	25376	25392	25408	25424	25440	25456	25472	25488	25504	25520	25536	25552	25568	25584
64	25600	25616	25632	25648	25664	25680	25696	25712	25728	25744	25760	25776	25792	25808	25824	25840
65	25856	25872	25888	25904	25920	25936	25952	25968	25984	26000	26016	26032	26048	26064	26080	26096
66	26112	26128	26144	26160	26176	26192	26208	26224	26240	26256	26272	26288	26304	26320	26336	26352
67	26368	26384	26400	26416	26432	26448	26464	26480	26496	26512	26528	26544	26560	26576	26592	26608
68	26624	26640	26656	26672	26688	26704	26720	26736	26752	26768	26784	26800	26816	26832	26848	26864
69	26880	26896	26912	26928	26944	26960	26976	26992	27008	27024	27040	27056	27072	27088	27104	27120
6A	27136	27152	27168	27184	27200	27216	27232	27248	27264	27280	27296	27312	27328	27344	27360	27376
6B	27392	27408	27424	27440	27456	27472	27488	27504	27520	27536	27552	27568	27584	27600	27616	27632
6C	27648	27664	27680	27696	27712	27728	27744	27760	27776	27792	27808	27824	27840	27856	27872	27888
6D	27904	27920	27936	27952	27968	27984	28000	28016	28032	28048	28064	28080	28096	28112	28128	28144
6E	28160	28176	28192	28208	28224	28240	28256	28272	28288	28304	28320	28336	28352	28368	28384	28400
6F	28416	28432	28448	28464	28480	28496	28512	28528	28544	28560	28576	28592	28608	28624	28640	28656
70	28672	28688	28704	28720	28736	28752	28768	28784	28800	28816	28832	28848	28864	28880	28896	28912
71	28928	28944	28960	28976	28992	29008	29024	29040	29056	29072	29088	29104	29120	29136	29152	29168
72	29184	29200	29216	29232	29248	29264	29280	29296	29312	29328	29344	29360	29376	29392	29408	29424
73	29440	29456	29472	29488	29504	29520	29536	29552	29568	29584	29600	29616	29632	29648	29664	29680
74	29696	29712	29728	29744	29760	29776	29792	29808	29824	29840	29856	29872	29888	29904	29920	29936
75	29952	29968	29984	30000	30016	30032	30048	30064	30080	30096	30112	30128	30144	30160	30176	30192
76	30208	30224	30240	30256	30272	30288	30304	30320	30336	30352	30368	30384	30400	30416	30432	30448
77	30464	30480	30496	30512	30528	30544	30560	30576	30592	30608	30624	30640	30656	30672	30688	30704
78	30720	30736	30752	30768	30784	30800	30816	30832	30848	30864	30880	30896	30912	30928	30944	30960
79	30976	30992	31008	31024	31040	31056	31072	31088	31104	31120	31136	31152	31168	31184	31200	31216
7A	31232	31248	31264	31280	31296	31312	31328	31344	31360	31376	31392	31408	31424	31440	31456	31472
7B	31488	31504	31520	31536	31552	31568	31584	31600	31616	31632	31648	31664	31680	31696	31712	31728
7C	31744	31760	31776	31792	31808	31824	31840	31856	31872	31888	31904	31920	31936	31952	31968	31984
7D	32000	32016	32032	32048	32064	32080	32096	32112	32128	32144	32160	32176	32192	32208	32224	32240
7E	32256	32272	32288	32304	32320	32336	32352	32368	32384	32400	32416	32432	32448	32464	32480	32496
7F	32512	32528	32544	32560	32576	32592	32608	32624	32640	32656	32672	32688	32704	32720	32736	32752
80	32768	32784	32800	32816	32832	32848	32864	32880	32896	32912	32928	32944	32960	32976	32992	33008
81	33024	33040	33056	33072	33088	33104	33120	33136	33152	33168	33184	33200	33216	33232	33248	33264
82	33280	33296	33312	33328	33344	33360	33376	33392	33408	33424	33440	33456	33472	33488	33504	33520
83	33536	33552	33568	33584	33600	33616	33632	33648	33664	33680	33696	33712	33728	33744	33760	33776
84	33792	33808	33824	33840	33856	33872	33888	33904	33920	33936	33952	33968	33984	34000	34016	34032
85	34048	34064	34080	34096	34112	34128	34144	34160	34176	34192	34208	34224	34240	34256	34272	34288
86	34304	34320	34336	34352	34368	34384	34400	34416	34432	34448	34464	34480	34496	34512	34528	34544
87	34560	34576	34592	34608	34624	34640	34656	34672	34688	34704	34720	34736	34752	34768	34784	34800
88	34816	34832	34848	34864	34880	34896	34912	34928	34944	34960	34976	34992	35008	35024	35040	35056
89	35072	35088	35104	35120	35136	35152	35168	35184	35200	35216	35232	35248	35264	35280	35296	35312
8A	35328	35344	35360	35376	35392	35408	35424	35440	35456	35472	35488	35504	35520	35536	35552	35568
8B	35584	35600	35616	35632	35648	35664	35680	35696	35712	35728	35744	35760	35776	35792	35808	35824
8C	35840	35856	35872	35888	35904	35920	35936	35952	35968	35984	36000	36016	36032	36048	36064	36080
8D	36096	36112	36128	36144	36160	36176	36192	36208	36224	36240	36256	36272	36288	36304	36320	36336
8E	36352	36368	36384	36400	36416	36432	36448	36464	36480	36496	36512	36528	36544	36560	36576	36592
8F	36608	36624	36640	36656	36672	36688	36704	36720	36736	36752	36768	36784	36800	36816	36832	36848

	00	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
90	36864	36880	36896	36912	36928	36944	36960	36976	36992	37008	37024	37040	37056	37072	37088	37104
91	37120	37136	37152	37168	37184	37200	37216	37232	37248	37264	37280	37296	37312	37328	37344	37360
92	37376	37392	37408	37424	37440	37456	37472	37488	37504	37520	37536	37552	37568	37584	37600	37616
93	37632	37648	37664	37680	37696	37712	37728	37744	37760	37776	37792	37808	37824	37840	37856	37872
94	37888	37904	37920	37936	37952	37968	37984	38000	38016	38032	38048	38064	38080	38096	38112	38128
95	38144	38160	38176	38192	38208	38224	38240	38256	38272	38288	38304	38320	38336	38352	38368	38384
96	38400	38416	38432	38448	38464	38480	38496	38512	38528	38544	38560	38576	38592	38608	38624	38640
97	38656	38672	38688	38704	38720	38736	38752	38768	38784	38800	38816	38832	38848	38864	38880	38896
98	38912	38928	38944	38960	38976	38992	39008	39024	39040	39056	39072	39088	39104	39120	39136	39152
99	39168	39184	39200	39216	39232	39248	39264	39280	39296	39312	39328	39344	39360	39376	39392	39408
9A	39424	39440	39456	39472	39488	39504	39520	39536	39552	39568	39584	39600	39616	39632	39648	39664
9B	39680	39696	39712	39728	39744	39760	39776	39792	39808	39824	39840	39856	39872	39888	39904	39920
9C	39936	39952	39968	39984	40000	40016	40032	40048	40064	40080	40096	40112	40128	40144	40160	40176
9D	40192	40208	40224	40240	40256	40272	40288	40304	40320	40336	40352	40368	40384	40400	40416	40432
9E	40448	40464	40480	40496	40512	40528	40544	40560	40576	40592	40608	40624	40640	40656	40672	40688
9F	40704	40720	40736	40752	40768	40784	40800	40816	40832	40848	40864	40880	40896	40912	40928	40944
A0	40960	40976	40992	41008	41024	41040	41056	41072	41088	41104	41120	41136	41152	41168	41184	41200
A1	41216	41232	41248	41264	41280	41296	41312	41328	41344	41360	41376	41392	41408	41424	41440	41456
A2	41472	41488	41504	41520	41536	41552	41568	41584	41600	41616	41632	41648	41664	41680	41696	41712
A3	41728	41744	41760	41776	41792	41808	41824	41840	41856	41872	41888	41904	41920	41936	41952	41968
A4	41984	42000	42016	42032	42048	42064	42080	42096	42112	42128	42144	42160	42176	42192	42208	42224
A5	42240	42256	42272	42288	42304	42320	42336	42352	42368	42384	42400	42416	42432	42448	42464	42480
A6	42496	42512	42528	42544	42560	42576	42592	42608	42624	42640	42656	42672	42688	42704	42720	42736
A7	42752	42768	42784	42800	42816	42832	42848	42864	42880	42896	42912	42928	42944	42960	42976	42992
A8	43008	43024	43040	43056	43072	43088	43104	43120	43136	43152	43168	43184	43200	43216	43232	43248
A9	43264	43280	43296	43312	43328	43344	43360	43376	43392	43408	43424	43440	43456	43472	43488	43504
AA	43520	43536	43552	43568	43584	43600	43616	43632	43648	43664	43680	43696	43712	43728	43744	43760
AB	43776	43792	43808	43824	43840	43856	43872	43888	43904	43920	43936	43952	43968	43984	44000	44016
AC	44032	44048	44064	44080	44096	44112	44128	44144	44160	44176	44192	44208	44224	44240	44256	44272
AD	44288	44304	44320	44336	44352	44368	44384	44400	44416	44432	44448	44464	44480	44496	44512	44528
AE	44544	44560	44576	44592	44608	44624	44640	44656	44672	44688	44704	44720	44736	44752	44768	44784
AF	44800	44816	44832	44848	44864	44880	44896	44912	44928	44944	44960	44976	44992	45008	45024	45040
B0	45056	45072	45088	45104	45120	45136	45152	45168	45184	45200	45216	45232	45248	45264	45280	45296
B1	45312	45328	45344	45360	45376	45392	45408	45424	45440	45456	45472	45488	45504	45520	45536	45552
B2	45568	45584	45600	45616	45632	45648	45664	45680	45696	45712	45728	45744	45760	45776	45792	45808
B3	45824	45840	45856	45872	45888	45904	45920	45936	45952	45968	45984	46000	46016	46032	46048	46064
B4	46080	46096	46112	46128	46144	46160	46176	46192	46208	46224	46240	46256	46272	46288	46304	46320
B5	46336	46352	46368	46384	46400	46416	46432	46448	46464	46480	46496	46512	46528	46544	46560	46576
B6	46592	46608	46624	46640	46656	46672	46688	46704	46720	46736	46752	46768	46784	46800	46816	46832
B7	46848	46864	46880	46896	46912	46928	46944	46960	46976	46992	47008	47024	47040	47056	47072	47088
B8	47104	47120	47136	47152	47168	47184	47200	47216	47232	47248	47264	47280	47296	47312	47328	47344
B9	47360	47376	47392	47408	47424	47440	47456	47472	47488	47504	47520	47536	47552	47568	47584	47600
BA	47616	47632	47648	47664	47680	47696	47712	47728	47744	47760	47776	47792	47808	47824	47840	47856
BB	47872	47888	47904	47920	47936	47952	47968	47984	48000	48016	48032	48048	48064	48080	48096	48112
BC	48128	48144	48160	48176	48192	48208	48224	48240	48256	48272	48288	48304	48320	48336	48352	48368
BD	48384	48400	48416	48432	48448	48464	48480	48496	48512	48528	48544	48560	48576	48592	48608	48624
BE	48640	48656	48672	48688	48704	48720	48736	48752	48768	48784	48800	48816	48832	48848	48864	48880
BF	48896	48912	48928	48944	48960	48976	48992	49008	49024	49040	49056	49072	49088	49104	49120	49136

	00	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
C0	49152	49168	49184	49200	49216	49232	49248	49264	49280	49296	49312	49328	49344	49360	49376	49392
C1	49408	49424	49440	49456	49472	49488	49504	49520	49536	49552	49568	49584	49600	49616	49632	49648
C2	49664	49680	49696	49712	49728	49744	49760	49776	49792	49808	49824	49840	49856	49872	49888	49904
C3	49920	49936	49952	49968	49984	50000	50016	50032	50048	50064	50080	50096	50112	50128	50144	50160
C4	50176	50192	50208	50224	50240	50256	50272	50288	50304	50320	50336	50352	50368	50384	50400	50416
C5	50432	50448	50464	50480	50496	50512	50528	50544	50560	50576	50592	50608	50624	50640	50656	50672
C6	50688	50704	50720	50736	50752	50768	50784	50800	50816	50832	50848	50864	50880	50896	50912	50928
C7	50944	50960	50976	50992	51008	51024	51040	51056	51072	51088	51104	51120	51136	51152	51168	51184
C8	51200	51216	51232	51248	51264	51280	51296	51312	51328	51344	51360	51376	51392	51408	51424	51440
C9	51456	51472	51488	51504	51520	51536	51552	51568	51584	51600	51616	51632	51648	51664	51680	51696
CA	51712	51728	51744	51760	51776	51792	51808	51824	51840	51856	51872	51888	51904	51920	51936	51952
CB	51968	51984	52000	52016	52032	52048	52064	52080	52096	52112	52128	52144	52160	52176	52192	52208
CC	52224	52240	52256	52272	52288	52304	52320	52336	52352	52368	52384	52400	52416	52432	52448	52464
CD	52480	52496	52512	52528	52544	52560	52576	52592	52608	52624	52640	52656	52672	52688	52704	52720
CE	52736	52752	52768	52784	52800	52816	52832	52848	52864	52880	52896	52912	52928	52944	52960	52976
CF	52992	53008	53024	53040	53056	53072	53088	53104	53120	53136	53152	53168	53184	53200	53216	53232
D0	53248	53264	53280	53296	53312	53328	53344	53360	53376	53392	53408	53424	53440	53456	53472	53488
D1	53504	53520	53536	53552	53568	53584	53600	53616	53632	53648	53664	53680	53696	53712	53728	53744
D2	53760	53776	53792	53808	53824	53840	53856	53872	53888	53904	53920	53936	53952	53968	53984	54000
D3	54016	54032	54048	54064	54080	54096	54112	54128	54144	54160	54176	54192	54208	54224	54240	54256
D4	54272	54288	54304	54320	54336	54352	54368	54384	54400	54416	54432	54448	54464	54480	54496	54512
D5	54528	54544	54560	54576	54592	54608	54624	54640	54656	54672	54688	54704	54720	54736	54752	54768
D6	54784	54800	54816	54832	54848	54864	54880	54896	54912	54928	54944	54960	54976	54992	55008	55024
D7	55040	55056	55072	55088	55104	55120	55136	55152	55168	55184	55200	55216	55232	55248	55264	55280
D8	55296	55312	55328	55344	55360	55376	55392	55408	55424	55440	55456	55472	55488	55504	55520	55536
D9	55552	55568	55584	55600	55616	55632	55648	55664	55680	55696	55712	55728	55744	55760	55776	55792
DA	55808	55824	55840	55856	55872	55888	55904	55920	55936	55952	55968	55984	56000	56016	56032	56048
DB	56064	56080	56096	56112	56128	56144	56160	56176	56192	56208	56224	56240	56256	56272	56288	56304
DC	56320	56336	56352	56368	56384	56400	56416	56432	56448	56464	56480	56496	56512	56528	56544	56560
DD	56576	56592	56608	56624	56640	56656	56672	56688	56704	56720	56736	56752	56768	56784	56800	56816
DE	56832	56848	56864	56880	56896	56912	56928	56944	56960	56976	56992	57008	57024	57040	57056	57072
DF	57088	57104	57120	57136	57152	57168	57184	57200	57216	57232	57248	57264	57280	57296	57312	57328
E0	57344	57360	57376	57392	57408	57424	57440	57456	57472	57488	57504	57520	57536	57552	57568	57584
E1	57600	57616	57632	57648	57664	57680	57696	57712	57728	57744	57760	57776	57792	57808	57824	57840
E2	57856	57872	57888	57904	57920	57936	57952	57968	57984	58000	58016	58032	58048	58064	58080	58096
E3	58112	58128	58144	58160	58176	58192	58208	58224	58240	58256	58272	58288	58304	58320	58336	58352
E4	58368	58384	58400	58416	58432	58448	58464	58480	58496	58512	58528	58544	58560	58576	58592	58608
E5	58624	58640	58656	58672	58688	58704	58720	58736	58752	58768	58784	58800	58816	58832	58848	58864
E6	58880	58896	58912	58928	58944	58960	58976	58992	59008	59024	59040	59056	59072	59088	59104	59120
E7	59136	59152	59168	59184	59200	59216	59232	59248	59264	59280	59296	59312	59328	59344	59360	59376
E8	59392	59408	59424	59440	59456	59472	59488	59504	59520	59536	59552	59568	59584	59600	59616	59632
E9	59648	59664	59680	59696	59712	59728	59744	59760	59776	59792	59808	59824	59840	59856	59872	59888
EA	59904	59920	59936	59952	59968	59984	60000	60016	60032	60048	60064	60080	60096	60112	60128	60144
EB	60160	60176	60192	60208	60224	60240	60256	60272	60288	60304	60320	60336	60352	60368	60384	60400
EC	60416	60432	60448	60464	60480	60496	60512	60528	60544	60560	60576	60592	60608	60624	60640	60656
ED	60672	60688	60704	60720	60736	60752	60768	60784	60800	60816	60832	60848	60864	60880	60896	60912
EE	60928	60944	60960	60976	60992	61008	61024	61040	61056	61072	61088	61104	61120	61136	61152	61168
EF	61184	61200	61216	61232	61248	61264	61280	61296	61312	61328	61344	61360	61376	61392	61408	61424

	00	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
F0	61440	61456	61472	61488	61504	61520	61536	61552	61568	61584	61600	61616	61632	61648	61664	61680
F1	61696	61712	61728	61744	61760	61776	61792	61808	61824	61840	61856	61872	61888	61904	61920	61936
F2	61952	61968	61984	62000	62016	62032	62048	62064	62080	62096	62112	62128	62144	62160	62176	62192
F3	62208	62224	62240	62256	62272	62288	62304	62320	62336	62352	62368	62384	62400	62416	62432	62448
F4	62464	62480	62496	62512	62528	62544	62560	62576	62592	62608	62624	62640	62656	62672	62688	62704
F5	62720	62736	62752	62768	62784	62800	62816	62832	62848	62864	62880	62896	62912	62928	62944	62960
F6	62976	62992	63008	63024	63040	63056	63072	63088	63104	63120	63136	63152	63168	63184	63200	63216
F7	63232	63248	63264	63280	63296	63312	63328	63344	63360	63376	63392	63408	63424	63440	63456	63472
F8	63488	63504	63520	63536	63552	63568	63584	63600	63616	63632	63648	63664	63680	63696	63712	63728
F9	63744	63760	63776	63792	63808	63824	63840	63856	63872	63888	63904	63920	63936	63952	63968	63984
FA	64000	64016	64032	64048	64064	64080	64096	64112	64128	64144	64160	64176	64192	64208	64224	64240
FB	64256	64272	64288	64304	64320	64336	64352	64368	64384	64400	64416	64432	64448	64464	64480	64496
FC	64512	64528	64544	64560	64576	64592	64608	64624	64640	64656	64672	64688	64704	64720	64736	64752
FD	64768	64784	64800	64816	64832	64848	64864	64880	64896	64912	64928	64944	64960	64976	64992	65008
FE	65024	65040	65056	65072	65088	65104	65120	65136	65152	65168	65184	65200	65216	65232	65248	65264
FF	65280	65296	65312	65328	65344	65360	65376	65392	65408	65424	65440	65456	65472	65488	65504	65520
80	-32768	-32752	-32736	-32720	-32704	-32688	-32672	-32656	-32640	-32624	-32608	-32592	-32576	-32560	-32544	-32528
81	-32512	-32496	-32480	-32464	-32448	-32432	-32416	-32400	-32384	-32368	-32352	-32336	-32320	-32304	-32288	-32272
82	-32256	-32240	-32224	-32208	-32192	-32176	-32160	-32144	-32128	-32112	-32096	-32080	-32064	-32048	-32032	-32016
83	-32000	-31984	-31968	-31952	-31936	-31920	-31904	-31888	-31872	-31856	-31840	-31824	-31808	-31792	-31776	-31760
84	-31744	-31728	-31712	-31696	-31680	-31664	-31648	-31632	-31616	-31600	-31584	-31568	-31552	-31536	-31520	-31504
85	-31488	-31472	-31456	-31440	-31424	-31408	-31392	-31376	-31360	-31344	-31328	-31312	-31296	-31280	-31264	-31248
86	-31232	-31216	-31200	-31184	-31168	-31152	-31136	-31120	-31104	-31088	-31072	-31056	-31040	-31024	-31008	-30992
87	-30976	-30960	-30944	-30928	-30912	-30896	-30880	-30864	-30848	-30832	-30816	-30800	-30784	-30768	-30752	-30736
88	-30720	-30704	-30688	-30672	-30656	-30640	-30624	-30608	-30592	-30576	-30560	-30544	-30528	-30512	-30496	-30480
89	-30464	-30448	-30432	-30416	-30400	-30384	-30368	-30352	-30336	-30320	-30304	-30288	-30272	-30256	-30240	-30224
8A	-30208	-30192	-30176	-30160	-30144	-30128	-30112	-30096	-30080	-30064	-30048	-30032	-30016	-30000	-29984	-29968
8B	-29952	-29936	-29920	-29904	-29888	-29872	-29856	-29840	-29824	-29808	-29792	-29776	-29760	-29744	-29728	-29712
8C	-29696	-29680	-29664	-29648	-29632	-29616	-29600	-29584	-29568	-29552	-29536	-29520	-29504	-29488	-29472	-29456
8D	-29440	-29424	-29408	-29392	-29376	-29360	-29344	-29328	-29312	-29296	-29280	-29264	-29248	-29232	-29216	-29200
8E	-29184	-29168	-29152	-29136	-29120	-29104	-29088	-29072	-29056	-29040	-29024	-29008	-28992	-28976	-28960	-28944
8F	-28928	-28912	-28896	-28880	-28864	-28848	-28832	-28816	-28800	-28784	-28768	-28752	-28736	-28720	-28704	-28688
90	-28672	-28656	-28640	-28624	-28608	-28592	-28576	-28560	-28544	-28528	-28512	-28496	-28480	-28464	-28448	-28432
91	-28416	-28400	-28384	-28368	-28352	-28336	-28320	-28304	-28288	-28272	-28256	-28240	-28224	-28208	-28192	-28176
92	-28160	-28144	-28128	-28112	-28096	-28080	-28064	-28048	-28032	-28016	-28000	-27984	-27968	-27952	-27936	-27920
93	-27904	-27888	-27872	-27856	-27840	-27824	-27808	-27792	-27776	-27760	-27744	-27728	-27712	-27696	-27680	-27664
94	-27648	-27632	-27616	-27600	-27584	-27568	-27552	-27536	-27520	-27504	-27488	-27472	-27456	-27440	-27424	-27408
95	-27392	-27376	-27360	-27344	-27328	-27312	-27296	-27280	-27264	-27248	-27232	-27216	-27200	-27184	-27168	-27152
96	-27136	-27120	-27104	-27088	-27072	-27056	-27040	-27024	-27008	-26992	-26976	-26960	-26944	-26928	-26912	-26896
97	-26880	-26864	-26848	-26832	-26816	-26800	-26784	-26768	-26752	-26736	-26720	-26704	-26688	-26672	-26656	-26640
98	-26624	-26608	-26592	-26576	-26560	-26544	-26528	-26512	-26496	-26480	-26464	-26448	-26432	-26416	-26400	-26384
99	-26368	-26352	-26336	-26320	-26304	-26288	-26272	-26256	-26240	-26224	-26208	-26192	-26176	-26160	-26144	-26128
9A	-26112	-26096	-26080	-26064	-26048	-26032	-26016	-26000	-25984	-25968	-25952	-25936	-25920	-25904	-25888	-25872
9B	-25856	-25840	-25824	-25808	-25792	-25776	-25760	-25744	-25728	-25712	-25696	-25680	-25664	-25648	-25632	-25616
9C	-25600	-25584	-25568	-25552	-25536	-25520	-25504	-25488	-25472	-25456	-25440	-25424	-25408	-25392	-25376	-25360
9D	-25344	-25328	-25312	-25296	-25280	-25264	-25248	-25232	-25216	-25200	-25184	-25168	-25152	-25136	-25120	-25104
9E	-25088	-25072	-25056	-25040	-25024	-25008	-24992	-24976	-24960	-24944	-24928	-24912	-24896	-24880	-24864	-24848
9F	-24832	-24816	-24800	-24784	-24768	-24752	-24736	-24720	-24704	-24688	-24672	-24656	-24640	-24624	-24608	-24592

	00	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
A0	-24576	-24560	-24544	-24528	-24512	-24496	-24480	-24464	-24448	-24432	-24416	-24400	-24384	-24368	-24352	-24336
A1	-24320	-24304	-24288	-24272	-24256	-24240	-24224	-24208	-24192	-24176	-24160	-24144	-24128	-24112	-24096	-24080
A2	-24064	-24048	-24032	-24016	-24000	-23984	-23968	-23952	-23936	-23920	-23904	-23888	-23872	-23856	-23840	-23824
A3	-23808	-23792	-23776	-23760	-23744	-23728	-23712	-23696	-23680	-23664	-23648	-23632	-23616	-23600	-23584	-23568
A4	-23552	-23536	-23520	-23504	-23488	-23472	-23456	-23440	-23424	-23408	-23392	-23376	-23360	-23344	-23328	-23312
A5	-23296	-23280	-23264	-23248	-23232	-23216	-23200	-23184	-23168	-23152	-23136	-23120	-23104	-23088	-23072	-23056
A6	-23040	-23024	-23008	-22992	-22976	-22960	-22944	-22928	-22912	-22896	-22880	-22864	-22848	-22832	-22816	-22800
A7	-22784	-22768	-22752	-22736	-22720	-22704	-22688	-22672	-22656	-22640	-22624	-22608	-22592	-22576	-22560	-22544
A8	-22528	-22512	-22496	-22480	-22464	-22448	-22432	-22416	-22400	-22384	-22368	-22352	-22336	-22320	-22304	-22288
A9	-22272	-22256	-22240	-22224	-22208	-22192	-22176	-22160	-22144	-22128	-22112	-22096	-22080	-22064	-22048	-22032
AA	-22016	-22000	-21984	-21968	-21952	-21936	-21920	-21904	-21888	-21872	-21856	-21840	-21824	-21808	-21792	-21776
AB	-21760	-21744	-21728	-21712	-21696	-21680	-21664	-21648	-21632	-21616	-21600	-21584	-21568	-21552	-21536	-21520
AC	-21504	-21488	-21472	-21456	-21440	-21424	-21408	-21392	-21376	-21360	-21344	-21328	-21312	-21296	-21280	-21264
AD	-21248	-21232	-21216	-21200	-21184	-21168	-21152	-21136	-21120	-21104	-21088	-21072	-21056	-21040	-21024	-21008
AE	-20992	-20976	-20960	-20944	-20928	-20912	-20896	-20880	-20864	-20848	-20832	-20816	-20800	-20784	-20768	-20752
AF	-20736	-20720	-20704	-20688	-20672	-20656	-20640	-20624	-20608	-20592	-20576	-20560	-20544	-20528	-20512	-20496
B0	-20480	-20464	-20448	-20432	-20416	-20400	-20384	-20368	-20352	-20336	-20320	-20304	-20288	-20272	-20256	-20240
B1	-20224	-20208	-20192	-20176	-20160	-20144	-20128	-20112	-20096	-20080	-20064	-20048	-20032	-20016	-20000	-19984
B2	-19968	-19952	-19936	-19920	-19904	-19888	-19872	-19856	-19840	-19824	-19808	-19792	-19776	-19760	-19744	-19728
B3	-19712	-19696	-19680	-19664	-19648	-19632	-19616	-19600	-19584	-19568	-19552	-19536	-19520	-19504	-19488	-19472
B4	-19456	-19440	-19424	-19408	-19392	-19376	-19360	-19344	-19328	-19312	-19296	-19280	-19264	-19248	-19232	-19216
B5	-19200	-19184	-19168	-19152	-19136	-19120	-19104	-19088	-19072	-19056	-19040	-19024	-19008	-18992	-18976	-18960
B6	-18944	-18928	-18912	-18896	-18880	-18864	-18848	-18832	-18816	-18800	-18784	-18768	-18752	-18736	-18720	-18704
B7	-18688	-18672	-18656	-18640	-18624	-18608	-18592	-18576	-18560	-18544	-18528	-18512	-18496	-18480	-18464	-18448
B8	-18432	-18416	-18400	-18384	-18368	-18352	-18336	-18320	-18304	-18288	-18272	-18256	-18240	-18224	-18208	-18192
B9	-18176	-18160	-18144	-18128	-18112	-18096	-18080	-18064	-18048	-18032	-18016	-18000	-17984	-17968	-17952	-17936
BA	-17920	-17904	-17888	-17872	-17856	-17840	-17824	-17808	-17792	-17776	-17760	-17744	-17728	-17712	-17696	-17680
BB	-17664	-17648	-17632	-17616	-17600	-17584	-17568	-17552	-17536	-17520	-17504	-17488	-17472	-17456	-17440	-17424
BC	-17408	-17392	-17376	-17360	-17344	-17328	-17312	-17296	-17280	-17264	-17248	-17232	-17216	-17200	-17184	-17168
BD	-17152	-17136	-17120	-17104	-17088	-17072	-17056	-17040	-17024	-17008	-16992	-16976	-16960	-16944	-16928	-16912
BE	-16896	-16880	-16864	-16848	-16832	-16816	-16800	-16784	-16768	-16752	-16736	-16720	-16704	-16688	-16672	-16656
BF	-16640	-16624	-16608	-16592	-16576	-16560	-16544	-16528	-16512	-16496	-16480	-16464	-16448	-16432	-16416	-16400
C0	-16384	-16368	-16352	-16336	-16320	-16304	-16288	-16272	-16256	-16240	-16224	-16208	-16192	-16176	-16160	-16144
C1	-16128	-16112	-16096	-16080	-16064	-16048	-16032	-16016	-16000	-15984	-15968	-15952	-15936	-15920	-15904	-15888
C2	-15872	-15856	-15840	-15824	-15808	-15792	-15776	-15760	-15744	-15728	-15712	-15696	-15680	-15664	-15648	-15632
C3	-15616	-15600	-15584	-15568	-15552	-15536	-15520	-15504	-15488	-15472	-15456	-15440	-15424	-15408	-15392	-15376
C4	-15360	-15344	-15328	-15312	-15296	-15280	-15264	-15248	-15232	-15216	-15200	-15184	-15168	-15152	-15136	-15120
C5	-15104	-15088	-15072	-15056	-15040	-15024	-15008	-14992	-14976	-14960	-14944	-14928	-14912	-14896	-14880	-14864
C6	-14848	-14832	-14816	-14800	-14784	-14768	-14752	-14736	-14720	-14704	-14688	-14672	-14656	-14640	-14624	-14608
C7	-14592	-14576	-14560	-14544	-14528	-14512	-14496	-14480	-14464	-14448	-14432	-14416	-14400	-14384	-14368	-14352
C8	-14336	-14320	-14304	-14288	-14272	-14256	-14240	-14224	-14208	-14192	-14176	-14160	-14144	-14128	-14112	-14096
C9	-14080	-14064	-14048	-14032	-14016	-14000	-13984	-13968	-13952	-13936	-13920	-13904	-13888	-13872	-13856	-13840
CA	-13824	-13808	-13792	-13776	-13760	-13744	-13728	-13712	-13696	-13680	-13664	-13648	-13632	-13616	-13600	-13584
CB	-13568	-13552	-13536	-13520	-13504	-13488	-13472	-13456	-13440	-13424	-13408	-13392	-13376	-13360	-13344	-13328
CC	-13312	-13296	-13280	-13264	-13248	-13232	-13216	-13200	-13184	-13168	-13152	-13136	-13120	-13104	-13088	-13072
CD	-13056	-13040	-13024	-13008	-12992	-12976	-12960	-12944	-12928	-12912	-12896	-12880	-12864	-12848	-12832	-12816
CE	-12800	-12784	-12768	-12752	-12736	-12720	-12704	-12688	-12672	-12656	-12640	-12624	-12608	-12592	-12576	-12560
CF	-12544	-12528	-12512	-12496	-12480	-12464	-12448	-12432	-12416	-12400	-12384	-12368	-12352	-12336	-12320	-12304

	00	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
D0	-12288	-12272	-12256	-12240	-12224	-12208	-12192	-12176	-12160	-12144	-12128	-12112	-12096	-12080	-12064	-12048
D1	-12032	-12016	-12000	-11984	-11968	-11952	-11936	-11920	-11904	-11888	-11872	-11856	-11840	-11824	-11808	-11792
D2	-11776	-11760	-11744	-11728	-11712	-11696	-11680	-11664	-11648	-11632	-11616	-11600	-11584	-11568	-11552	-11536
D3	-11520	-11504	-11488	-11472	-11456	-11440	-11424	-11408	-11392	-11376	-11360	-11344	-11328	-11312	-11296	-11280
D4	-11264	-11248	-11232	-11216	-11200	-11184	-11168	-11152	-11136	-11120	-11104	-11088	-11072	-11056	-11040	-11024
D5	-11008	-10992	-10976	-10960	-10944	-10928	-10912	-10896	-10880	-10864	-10848	-10832	-10816	-10800	-10784	-10768
D6	-10752	-10736	-10720	-10704	-10688	-10672	-10656	-10640	-10624	-10608	-10592	-10576	-10560	-10544	-10528	-10512
D7	-10496	-10480	-10464	-10448	-10432	-10416	-10400	-10384	-10368	-10352	-10336	-10320	-10304	-10288	-10272	-10256
D8	-10240	-10224	-10208	-10192	-10176	-10160	-10144	-10128	-10112	-10096	-10080	-10064	-10048	-10032	-10016	-10000
D9	-9984	-9968	-9952	-9936	-9920	-9904	-9888	-9872	-9856	-9840	-9824	-9808	-9792	-9776	-9760	-9744
DA	-9728	-9712	-9696	-9680	-9664	-9648	-9632	-9616	-9600	-9584	-9568	-9552	-9536	-9520	-9504	-9488
DB	-9472	-9456	-9440	-9424	-9408	-9392	-9376	-9360	-9344	-9328	-9312	-9296	-9280	-9264	-9248	-9232
DC	-9216	-9200	-9184	-9168	-9152	-9136	-9120	-9104	-9088	-9072	-9056	-9040	-9024	-9008	-8992	-8976
DD	-8960	-8944	-8928	-8912	-8896	-8880	-8864	-8848	-8832	-8816	-8800	-8784	-8768	-8752	-8736	-8720
DE	-8704	-8688	-8672	-8656	-8640	-8624	-8608	-8592	-8576	-8560	-8544	-8528	-8512	-8496	-8480	-8464
DF	-8448	-8432	-8416	-8400	-8384	-8368	-8352	-8336	-8320	-8304	-8288	-8272	-8256	-8240	-8224	-8208
E0	-8192	-8176	-8160	-8144	-8128	-8112	-8096	-8080	-8064	-8048	-8032	-8016	-8000	-7984	-7968	-7952
E1	-7936	-7920	-7904	-7888	-7872	-7856	-7840	-7824	-7808	-7792	-7776	-7760	-7744	-7728	-7712	-7696
E2	-7680	-7664	-7648	-7632	-7616	-7600	-7584	-7568	-7552	-7536	-7520	-7504	-7488	-7472	-7456	-7440
E3	-7424	-7408	-7392	-7376	-7360	-7344	-7328	-7312	-7296	-7280	-7264	-7248	-7232	-7216	-7200	-7184
E4	-7168	-7152	-7136	-7120	-7104	-7088	-7072	-7056	-7040	-7024	-7008	-6992	-6976	-6960	-6944	-6928
E5	-6912	-6896	-6880	-6864	-6848	-6832	-6816	-6800	-6784	-6768	-6752	-6736	-6720	-6704	-6688	-6672
E6	-6656	-6640	-6624	-6608	-6592	-6576	-6560	-6544	-6528	-6512	-6496	-6480	-6464	-6448	-6432	-6416
E7	-6400	-6384	-6368	-6352	-6336	-6320	-6304	-6288	-6272	-6256	-6240	-6224	-6208	-6192	-6176	-6160
E8	-6144	-6128	-6112	-6096	-6080	-6064	-6048	-6032	-6016	-6000	-5984	-5968	-5952	-5936	-5920	-5904
E9	-5888	-5872	-5856	-5840	-5824	-5808	-5792	-5776	-5760	-5744	-5728	-5712	-5696	-5680	-5664	-5648
EA	-5632	-5616	-5600	-5584	-5568	-5552	-5536	-5520	-5504	-5488	-5472	-5456	-5440	-5424	-5408	-5392
EB	-5376	-5360	-5344	-5328	-5312	-5296	-5280	-5264	-5248	-5232	-5216	-5200	-5184	-5168	-5152	-5136
EC	-5120	-5104	-5088	-5072	-5056	-5040	-5024	-5008	-4992	-4976	-4960	-4944	-4928	-4912	-4896	-4880
ED	-4864	-4848	-4832	-4816	-4800	-4784	-4768	-4752	-4736	-4720	-4704	-4688	-4672	-4656	-4640	-4624
EE	-4608	-4592	-4576	-4560	-4544	-4528	-4512	-4496	-4480	-4464	-4448	-4432	-4416	-4400	-4384	-4368
EF	-4352	-4336	-4320	-4304	-4288	-4272	-4256	-4240	-4224	-4208	-4192	-4176	-4160	-4144	-4128	-4112
F0	-4096	-4080	-4064	-4048	-4032	-4016	-4000	-3984	-3968	-3952	-3936	-3920	-3904	-3888	-3872	-3856
F1	-3840	-3824	-3808	-3792	-3776	-3760	-3744	-3728	-3712	-3696	-3680	-3664	-3648	-3632	-3616	-3600
F2	-3584	-3568	-3552	-3536	-3520	-3504	-3488	-3472	-3456	-3440	-3424	-3408	-3392	-3376	-3360	-3344
F3	-3328	-3312	-3296	-3280	-3264	-3248	-3232	-3216	-3200	-3184	-3168	-3152	-3136	-3120	-3104	-3088
F4	-3072	-3056	-3040	-3024	-3008	-2992	-2976	-2960	-2944	-2928	-2912	-2896	-2880	-2864	-2848	-2832
F5	-2816	-2800	-2784	-2768	-2752	-2736	-2720	-2704	-2688	-2672	-2656	-2640	-2624	-2608	-2592	-2576
F6	-2560	-2544	-2528	-2512	-2496	-2480	-2464	-2448	-2432	-2416	-2400	-2384	-2368	-2352	-2336	-2320
F7	-2304	-2288	-2272	-2256	-2240	-2224	-2208	-2192	-2176	-2160	-2144	-2128	-2112	-2096	-2080	-2064
F8	-2048	-2032	-2016	-2000	-1984	-1968	-1952	-1936	-1920	-1904	-1888	-1872	-1856	-1840	-1824	-1808
F9	-1792	-1776	-1760	-1744	-1728	-1712	-1696	-1680	-1664	-1648	-1632	-1616	-1600	-1584	-1568	-1552
FA	-1536	-1520	-1504	-1488	-1472	-1456	-1440	-1424	-1408	-1392	-1376	-1360	-1344	-1328	-1312	-1296
FB	-1280	-1264	-1248	-1232	-1216	-1200	-1184	-1168	-1152	-1136	-1120	-1104	-1088	-1072	-1056	-1040
FC	-1024	-1008	-992	-976	-960	-944	-928	-912	-896	-880	-864	-848	-832	-816	-800	-784
FD	-768	-752	-736	-720	-704	-688	-672	-656	-640	-624	-608	-592	-576	-560	-544	-528
FE	-512	-496	-480	-464	-448	-432	-416	-400	-384	-368	-352	-336	-320	-304	-288	-272
FF	-256	-240	-224	-208	-192	-176	-160	-144	-128	-112	-96	-80	-64	-48	-32	-16

USR Routine Pointer Addresses

DOS VERSION	0	1	2	3	4	5	6	7	8	9
TRSDOS 2.3	23415	23417	23419	23421	23423	23425	23427	23429	23431	23433
Radio Shack	5B77	5B79	5B7B	5B7D	5B7F	5B81	5B83	5B85	5B87	5B89
Model I										
TRSDOS 2.0	11050	11052	11054	11056	11058	11060	11062	11064	11066	11068
Radio Shack	2B2A	2B2C	2B2E	2B30	2B32	2B34	2B36	2B38	2B3A	2B3C
Model 2										
TRSDOS 1.2	22586	22588	22590	22592	22594	22596	22598	22600	22602	22604
Radio Shack	583A	583C	583E	5840	5842	5844	5846	5848	584A	584C
Model III										
TRSDOS 1.3	22632	22634	22636	22638	22640	22642	22644	22646	22648	22650
Radio Shack	5868	586A	586C	586E	5870	5872	5874	5876	5878	587A
Model III										
NEWDOS 2.1	23316	23318	23320	23322	23324	23326	23328	23330	23332	23334
Apparat	5B14	5B16	5B18	5B1A	5B1C	5B1E	5B20	5B22	5B24	5B26
NEWDOS80 1.0	22330	22332	22334	22336	22338	22340	22342	22344	22346	22348
Apparat	573A	573C	573E	5740	5742	5744	5746	5748	574A	574C
DOS PLUS 3.3D	23483	23485	23487	23489	23491	23493	23495	23497	23499	23501
Micro Systems	5BBB	5BBD	5BBF	5BC1	5BC3	5BC5	5BC7	5BC9	5BCB	5BCD
Software										
LDOS 5.0.1	23415	23417	23419	23421	23423	23425	23427	23429	23431	23433
Lobo Drives,	5B77	5B79	5B7B	5B7D	5B7F	5B81	5B83	5B85	5B87	5B89
Int'l										
ULTRADOS 4.2	20992	20994	20996	20998	21000	21002	21004	21006	21008	21010
Level IV	5200	5202	5204	5206	5208	520A	520C	520E	5210	5212
Products										
DBLDOS 4.23	23316	23318	23320	23322	23324	23326	23328	23330	23332	23334
Percom	5B14	5B16	5B18	5B1A	5B1C	5B1E	5B20	5B22	5B24	5B26

Disk Buffer Memory Locations

DISK OPERATING SYSTEM	VERSION	MODEL	1	2	3	4	5	6
TRSDOS	2.3	1	26335	26625	26915	27205	27495	27785
Radio Shack			66DF	6801	6923	6A45	6B67	6C89
TRSDOS	2.0	2	27779	28613	29447	30281	31115	31949
Radio Shack			6C83	6FC5	7307	7649	798B	7CCD
TRSDOS	1.2	3	25812	26172	26532	26892	27252	27612
Radio Shack			64D4	663C	67A4	690C	6A74	6BDC
TRSDOS	1.3	3	26232	26592	26952	27312	27672	28032
Radio Shack			6678	67E0	6948	6AB0	6C18	6D80
NEWDOS	2.1	1	25973	26263	26553	26843	27133	27423
Apparat			6575	6697	67B9	68DB	69FD	6B1F
NEWDOS/80	1.0	1	26347	26648	26949	27250	27551	27852
Apparat			66EB	6818	6945	6A72	6B9F	6CCC
DOS PLUS	3.3D	1	28053	28599	29145	29691	30237	30783
Micro Systems Software			6D95	6FB7	71D9	73FB	761D	783F
DOS PLUS - TBASIC	3.3D	1	25450	25996	26542	27088	27634	28180
Micro Systems Software			636A	658C	67AE	69D0	6BF2	6E14
DOS PLUS	3.3	3	28039	28585	29131	29677	30223	30769
Micro Systems Software			6D87	6FA9	71CB	73ED	760F	7831
LDOS	5.0.1	1	27237	27783	28329	28875	29421	29967
Lobo Drives, Int'l			6A65	6C87	6EA9	70CB	72ED	750F
ULTRADOS	4.2	1	25531	25821	26111	26401	26691	26981
Level IV Products			63BB	64DD	65FF	6721	6843	6965
DBLDOS	4.23	1	25973	26263	26553	26843	27133	27423
Percom			6575	6697	67B9	68DB	69FD	6B1F

Disk Data Control Block Addresses

DISK OPERATING SYSTEM	VERSION	MODEL	1	2	3	4	5	6
TRSDOS	2.3	1	26303	26593	26883	27173	27463	27753
Radio Shack			66BF	67E1	6903	6A25	6B47	6C69
TRSDOS	2.0	2	27715	28549	29383	30217	31051	31885
Radio Shack			6C43	6F85	72C7	7609	794B	7C8D
TRSDOS	1.2	3	25762	26122	26482	26842	27202	27562
Radio Shack			64A2	660A	6772	68DA	6A42	6BAA
TRSDOS	1.3	3	26182	26542	26902	27262	27622	27982
Radio Shack			6646	67AE	6916	6A7E	6BE6	6D4E
NEWDOS	2.1	1	25941	26231	26521	26811	27101	27391
Apparat			6555	6677	6799	68BB	69DD	6AFF
NEWDOS/80	1.0	1	26315	26616	26917	27218	27519	27820
Apparat			66CB	67F8	6925	6A52	6B7F	6CAC
DOS PLUS	3.3D	1	28021	28567	29113	29659	30205	30751
Micro Systems Software			6D75	6F97	71B9	73DB	75FD	781F
DOS PLUS - TBASIC	3.3D	1	25418	25964	26510	27056	27602	28148
Micro Systems Software			634A	656C	678E	69B0	6BD2	6DF4
DOS PLUS	3.3	3	28007	28553	29099	29645	30191	30737
Micro Systems Software			6D67	6F89	71AB	73CD	75EF	7811
LDOS	5.0.1	1	27205	27751	28297	28843	29389	29935
Lobo Drives, Int'l			6A45	6C67	6E89	70AB	72CD	74EF
ULTRADOS	4.2	1	25499	25789	26079	26369	26659	26949
Level IV Products			639B	64BD	65DF	6701	6823	6945
DBLDOS	4.23	1	25941	26231	26521	26811	27101	27391
Percom			6555	6677	6799	68BB	69DD	6AFF

Divisors of 256 – With Remainders

N	256/N	REM	N	256/N	REM	N	256/N	REM	N	256/N	REM
1**	256	0	33	7	25	65	3	61	97	2	62
2**	128	0	34	7	18	66	3	58	98	2	60
3*	85	1	35	7	11	67	3	55	99	2	58
4**	64	0	36*	7	4	68	3	52	100	2	56
5*	51	1	37	6	34	69	3	49	101	2	54
6*	42	4	38	6	28	70	3	46	102	2	52
7*	36	4	39	6	22	71	3	43	103	2	50
8**	32	0	40	6	16	72	3	40	104	2	48
9*	28	4	41	6	10	73	3	37	105	2	46
10*	25	6	42*	6	4	74	3	34	106	2	44
11*	23	3	43	5	41	75	3	31	107	2	42
12*	21	4	44	5	36	76	3	28	108	2	40
13	19	9	45	5	31	77	3	25	109	2	38
14*	18	4	46	5	26	78	3	22	110	2	36
15*	17	1	47	5	21	79	3	19	111	2	34
16**	16	0	48	5	16	80	3	16	112	2	32
17*	15	1	49	5	11	81	3	13	113	2	30
18*	14	4	50	5	6	82	3	10	114	2	28
19	13	9	51*	5	1	83	3	7	115	2	26
20	12	16	52	4	48	84	3	4	116	2	24
21*	12	4	53	4	44	85*	3	1	117	2	22
22	11	14	54	4	40	86	2	84	118	2	20
23*	11	3	55	4	36	87	2	82	119	2	18
24	10	16	56	4	32	88	2	80	120	2	16
25*	10	6	57	4	28	89	2	78	121	2	14
26	9	22	58	4	24	90	2	76	122	2	12
27	9	13	59	4	20	91	2	74	123	2	10
28*	9	4	60	4	16	92	2	72	124	2	8
29	8	24	61	4	12	93	2	70	125	2	6
30	8	16	62	4	8	94	2	68	126	2	4
31	8	8	63	4	4	95	2	66	127	2	2
32**	8	0	64**	4	0	96	2	64	128**	2	0

** Best disk logical record lengths - No bytes wasted
 * Good disk logical record lengths - Fewer than 7 bytes wasted

Divisors Of 255 – With Remainders

N	255/N	REM	N	255/N	REM	N	255/N	REM	N	255/N	REM
1**	255	0	33	7	24	65	3	60	97	2	61
2*	127	1	34	7	17	66	3	57	98	2	59
3**	85	0	35	7	10	67	3	54	99	2	57
4*	63	3	36*	7	3	68	3	51	100	2	55
5**	51	0	37	6	33	69	3	48	101	2	53
6*	42	3	38	6	27	70	3	45	102	2	51
7*	36	3	39	6	21	71	3	42	103	2	49
8	31	7	40	6	15	72	3	39	104	2	47
9*	28	3	41	6	9	73	3	36	105	2	45
10*	25	5	42*	6	3	74	3	33	106	2	43
11*	23	2	43	5	40	75	3	30	107	2	41
12*	21	3	44	5	35	76	3	27	108	2	39
13	19	8	45	5	30	77	3	24	109	2	37
14*	18	3	46	5	25	78	3	21	110	2	35
15**	17	0	47	5	20	79	3	18	111	2	33
16	15	15	48	5	15	80	3	15	112	2	31
17**	15	0	49	5	10	81	3	12	113	2	29
18*	14	3	50	5	5	82	3	9	114	2	27
19	13	8	51**	5	0	83*	3	6	115	2	25
20	12	15	52	4	47	84	3	3	116	2	23
21*	12	3	53	4	43	85**	3	0	117	2	21
22	11	13	54	4	39	86	2	83	118	2	19
23*	11	2	55	4	35	87	2	81	119	2	17
24	10	15	56	4	31	88	2	79	120	2	15
25*	10	5	57	4	27	89	2	77	121	2	13
26	9	21	58	4	23	90	2	75	122	2	11
27	9	12	59	4	19	91	2	73	123	2	9
28*	9	3	60	4	15	92	2	71	124	2	7
29	8	23	61	4	11	93	2	69	125	2	5
30	8	15	62	4	7	94	2	67	126	2	3
31	8	7	63*	4	3	95	2	65	127*	2	1
32	7	31	64	3	63	96	2	63	128	1	127

** Best disk logical record lengths - No bytes wasted
* Good disk logical record lengths - Fewer than 7 bytes wasted

TRS-80 Graphics Characters

```
129  130  131  132  133  134  135  136  137  138
 X.   .X   XX   ..   X.   .X   XX   ..   X.   .X
 ..   ..   ..   X.   X.   X.   X.   .X   .X   .X
 ..   ..   ..   ..   ..   ..   ..   ..   ..   ..

139  140  141  142  143  144  145  146  147  148
 XX   ..   X.   .X   XX   ..   X.   .X   XX   ..
 .X   XX   XX   XX   XX   ..   ..   ..   ..   X.
 ..   ..   ..   ..   ..   X.   X.   X.   X.   X.

149  150  151  152  153  154  155  156  157  158
 X.   .X   XX   ..   X.   .X   XX   ..   X.   .X
 X.   X.   X.   .X   .X   .X   .X   XX   XX   XX
 X.   X.   X.   X.   X.   X.   X.   X.   X.   X.

159  160  161  162  163  164  165  166  167  168
 XX   ..   X.   .X   XX   ..   X.   .X   XX   ..
 XX   ..   ..   ..   ..   X.   X.   X.   X.   .X
 X.   .X   .X   .X   .X   .X   .X   .X   .X   .X

169  170  171  172  173  174  175  176  177  178
 X.   .X   XX   ..   X.   .X   XX   ..   X.   .X
 .X   .X   .X   XX   XX   XX   XX   ..   ..   ..
 .X   .X   .X   .X   .X   .X   .X   XX   XX   XX

179  180  181  182  183  184  185  186  187  188
 XX   ..   X.   .X   XX   ..   X.   .X   XX   ..
 ..   X.   X.   X.   X.   .X   .X   .X   .X   XX
 XX   XX   XX   XX   XX   XX   XX   XX   XX   XX

189  190  191
 X.   .X   XX
 XX   XX   XX
 XX   XX   XX
```

Functions By Line Number

Line	Function................		Description..........................
1	FN SI%	(A1!)	Convert unsigned sgl to int
2	FN IS!	(A1%)	Convert int to unsigned sgl
3	FN IA%	(A1%,A2%)	Add and subtract int addresses
4	FN RE#	(A1#,A2#)	Remainder computation
5	FN RW#	(A1#)	Round to nearest whole number
6	FN RD#	(A1#)	Round to nearest cent
7	FN FL#	(A1#,A2#)	Round to first multiple less or equal
8	FN FM#	(A1#,A2#)	Round to first multiple greater
9	FN U3$	(A#)	Compress unsigned dbl to 3-byte str
10	FN U3#	(A$)	Uncompress 3-byte str to unsigned dbl
11	FN U4$	(A#)	Compress dbl to 4-byte str
12	FN U4#	(A$)	Uncompress 4-byte str to dbl
13	FN S3$	(A#)	Compress signed dbl to 3-byte str
14	FN S3#	(A$)	Uncompress 3-byte str to signed dbl
15	FN DI$	(A#)	Compress signed dbl to 4-byte str
16	FN DI#	(A$)	Uncompress 4-byte str to signed dbl
17	FN S4$	(A#)	Compress signed dbl to 4-byte str
18	FN S4#	(A$)	Uncompress 4-byte str to signed dbl
19	FN DF$	(A1#,A2%,A3$,A4$)	Format dbl to dollar str
20	FN BN$	(A1#,A2%)	Format dbl to dollar str with brackets
21	FN NF$	(A1#,A2%,A3$,A4$)	Format dbl to integer str
22	FN TF$	(A1#)	Format dbl to telephone number string
23	FN SO$	(A1#)	Format dbl to social security string
24	FN H2$	(A1%)	Convert int to hexadecimal (0-255)
25	FN H4$	(A1%)	Convert int to hexadecimal
26	FN DH!	(A$)	Convert hexadecimal str to sgl
27	FN SS$	(A$)	Strip trailing blanks from str
28	FN PR$	(A$,A%)	Pad blanks to right side of str
29	FN PL$	(A$,A%)	Pad blanks to left side of str
30	FN CN$	(A$,A%)	Center by padding left side of str
31	FN FL$	(A$)	Swap first and last names
32	FN RR$	(A1%,A2%,A3$)	Extract substring from a str
33	FN RC%	(A1$,A2$,A3%)	Code look-up and validation
34	FN KM$	(A$,A%)	Compress/Uncompress str
35	FN DV%	(A1$,A2%)	Validate an 8-byte date
36	FN CD$	(A1$)	Compress 8-byte date to 3-byte date
37	FN UD$	(A1$)	Uncompress 3-byte date to 8-byte date
38	FN C2$	(A1$)	Compress 3-byte date to 2-byte date
39	FN U2$	(A1$)	Uncompress 2-byte date to 3-byte date
40	FN JD%	(Y%,M%,D%)	Compute day number within year
41	FN DN!	(Y%,M%,D%)	Compute computational date
42	FN DY$	(N!)	Compute day of the week from comp. day
43	FN RY%	(N!)	Compute year from computational date
44	FN RJ%	(N!)	Compute day number from comp. date
45	FN RM%	(J%,Y%)	Compute month from day number and year
46	FN RD%	(Y%,M%,J%)	Compute day of month
47	FN SE!	(A1$)	Convert hrs, mins, secs to seconds
48	FN HM$	(A1!)	Convert seconds to hrs, mins, secs
49	FN TD!	(A1$,A2$)	Time clock subtraction
50	FN SB$	(A1$,A2%)	Set any bit in a string
51	FN RB$	(A1$,A2%)	Reset any bit in a string
52	FN TB%	(A1$,A2%)	Test any bit in a string
53	FN IX$	(A%)	Convert int to 2-byte sortable str
54	FN IX%	(A$)	Convert 2-byte sortable str to int
55	FN SA$	(A1#,A2#,A3%)	Convert number to sortable string

Functions Alphabetically

Function		Description	Line
FN BN$	(A1#,A2%)	Format dbl to dollar str with brackets	20
FN C2$	(A1$)	Compress 3-byte date to 2-byte date	38
FN CD$	(A1$)	Compress 8-byte date to 3-byte date	36
FN CN$	(A$,A%)	Center by padding left side of str	30
FN DF$	(A1#,A2%,A3$,A4$)	Format dbl to dollar str	19
FN DH!	(A$)	Convert hexadecimal str to sgl	26
FN DI#	(A$)	Uncompress 4-byte str to signed dbl	16
FN DI$	(A#)	Compress signed dbl to 4-byte str	15
FN DN!	(Y%,M%,D%)	Compute computational date	41
FN DV%	(A1$,A2%)	Validate an 8-byte date	35
FN DY$	(N!)	Compute day of the week from comp. day	42
FN FL#	(A1#,A2#)	Round to first multiple less or equal	7
FN FL$	(A$)	Swap first and last names	31
FN FM#	(A1#,A2#)	Round to first multiple greater	8
FN H2$	(A1%)	Convert int to hexadecimal (0-255)	24
FN H4$	(A1%)	Convert int to hexadecimal	25
FN HM$	(A1!)	Convert seconds to hrs, mins, secs	48
FN IA%	(A1%,A2%)	Add and subtract int addresses	3
FN IS!	(A1%)	Convert int to unsigned sgl	2
FN IX$	(A%)	Convert int to 2-byte sortable str	53
FN IX%	(A$)	Convert 2-byte sortable str to int	54
FN JD%	(Y%,M%,D%)	Compute day number within year	40
FN KM$	(A$,A%)	Compress/Uncompress str	34
FN NF$	(A1#,A2%,A3$,A4$)	Format dbl to integer str	21
FN PL$	(A$,A%)	Pad blanks to left side of str	29
FN PR$	(A$,A%)	Pad blanks to right side of str	28
FN RB$	(A1$,A2%)	Reset any bit in a string	51
FN RC%	(A1$,A2$,A3%)	Code look-up and validation	33
FN RD#	(A1#)	Round to nearest cent	6
FN RD%	(Y%,M%,J%)	Compute day of month	46
FN RE#	(A1#,A2#)	Remainder computation	4
FN RJ%	(N!)	Compute day number from comp. date	44
FN RM%	(J%,Y%)	Compute month from day number and year	45
FN RR$	(A1%,A2%,A3$)	Extract substring from a str	32
FN RW#	(A1#)	Round to nearest whole number	5
FN RY%	(N!)	Compute year from computational date	43
FN S3#	(A$)	Uncompress 3-byte str to signed dbl	14
FN S3$	(A#)	Compress signed dbl to 3-byte str	13
FN S4#	(A$)	Uncompress 4-byte str to signed dbl	18
FN S4$	(A#)	Compress signed dbl to 4-byte str	17
FN SA$	(A1#,A2#,A3%)	Convert number to sortable string	55
FN SB$	(A1$,A2%)	Set any bit in a string	50
FN SE!	(A1$)	Convert hrs, mins, secs to seconds	47
FN SI%	(A1!)	Convert unsigned sgl to int	1
FN SO$	(A1#)	Format dbl to social security string	23
FN SS$	(A$)	Strip trailing blanks from str	27
FN TB%	(A1$,A2%)	Test any bit in a string	52
FN TD!	(A1$,A2$)	Time clock subtraction	49
FN TF$	(A1#)	Format dbl to telephone number string	22
FN U2$	(A1$)	Uncompress 2-byte date to 3-byte date	39
FN U3#	(A$)	Uncompress 3-byte str to unsigned dbl	10
FN U3$	(A#)	Compress unsigned dbl to 3-byte str	9
FN U4#	(A$)	Uncompress 4-byte str to dbl	12
FN U4$	(A#)	Compress dbl to 4-byte str	11
FN UD$	(A1$)	Uncompress 3-byte date to 8-byte date	37

Index To Major Subroutines

Note: "*" indicates that minor modifications are normally required.

29000* Variable List Pointer Subroutine
Note: Renumbered from 52000 for use with top-loaded overlays.

29100 Variable Pass Subroutine
Note: Renumbered from 52100 for use with top-loaded overlays.

29200* Variable Receive Subroutine
Note: Renumbered from 52200 for use with top-loaded overlays.

29300 Overlay Loader Routine for Top-Loaded Overlays
...... Continued: 29301.

29998 End of Text Computation Subroutine
Note: For use with top-loaded overlays.

29999* Last Line Linker Subroutine
Note: For use with bottom-loaded overlays.

40070 Video Display String Pointer Subroutine

40100 Horizontal Input/Output Subroutine
...... Continued: 40101.

40130 Alphanumeric Inkey Subroutine
...... Continued: 40131,40132,40133,40134,40135,40136,40137,
...... 40138,40139.

40140 Dollar Inkey Subroutine
...... Continued: 40141,40142,40143,40144,40145,40146,40147,
...... 40148,40149.

40150 Formatted Inkey Subroutine
...... Continued: 40151,40152,40153,40154,40156,40158,40159.

40160 Numeric Inkey Subroutine
...... Continued: 40161,40162,40163,40164,40165,40166,40167,
...... 40168,40169.

40200 Screen Save and Flashback Subroutine
Note: Requires Move-Data Magic Array loaded into US%(0) through US%(7)
...... Continued: 40201.

40500 Single Key Subroutine

40600 Flashing Cursor Single Key Subroutine

By Line Number

40700 Scroll-Up PRINT@ Computation Subroutine
Note: Performs PRINT@ Computation. Normal entry is via 40710.

40710 Scroll Up Subroutine
Note: Requires Move-Data Magic Array loaded into US%(0) through US%(7)
...... Continued: 40711,40712.

40800 Up-Down Scroller Subroutine
Note: Requires Move-Data Magic Array loaded into US%(0) through US%(7)
...... Continued: 40801,40802,40803,40804,40805,40806,40820,40821,
...... 40822,40823*,40824,40830,40831*,40832.

40900* Scrolled Video Entry Handler
Note: Requires Move-Data Magic Array loaded into US%(0) through US%(7)
...... Continued: 40901,40902,40903,40905,40910,40911,40912,
...... 40913,40914,40915,40916,40917,40920,40921,40922,40923,
...... 40924,40925,40926,40930,40931,40932,40940,40941*,40942,
...... 40943,40944,40945,40947,40950,40951*,40952,40953,40954,
...... 40960,40961,40962,40970,40971,40972,40973,40974,40975,
...... 40980,40981,40982,40990,40991.

41000 String Pointer Subroutine

41100 Command String Peel-Off Subroutine
...... Continued: 41101.

41200 Substring Replacement Subroutine
...... Continued: 41201.

46010 Unscrolled Video Entry Handler
...... Continued: 46011,46020,46021,46022,46029,46030,46031,
...... 46032,46033,46034,46035,46036,46037,46038,46039,46040,
...... 46041,46042,46043,46050,46051,46052,46053,46054,46059,
...... 46060,46061,46062,46063,46064.

52000* Variable List Pointer Subroutine

52100 Variable Pass Subroutine

52200* Variable Receive Subroutine

52300* Overlay Loader Routine for Bottom-Loaded Overlays
...... Continued: 52301.

57300 Video Display Screen Printer Subroutine

57400* Video Display To Sequential Disk File Subroutine
...... Continued: 57410,57420,57430,57440.

57450* Video Display From Sequential Disk File Subroutine
...... Continued: 57460,57470,57475,57480,57490.

USR Subroutine Index

Name	Bytes	——— Line Numbers ——— USRDATA1/LIB	USRDATA2/LIB	Record-No. USRFILE/RND
MOVEDATA	16	60001	61001	1
MOVEX *	88	60021-60023	61021-61023	2
SUMSNG	47	60041-60042	61041-61042	3
SUMDBL	59	60061-60062	61061-61062	4
LSTRIP	31	60081	61081	5
RSTRIP	30	60101	61101	6
STRCOMPL	19	60121	61121	7
UPPERCON	28	60141	61141	8
BITSRCH	72	60161-60162	61161-61162	9
SORT1	188	60201-60206	61201-61206	10
SORT2	212	60221-60227	61221-61227	11
SORT3	153	60241-60245	61241-61245	12
SEARCH1	133	60261-60265	61261-61265	13
SEARCH2	169	60281-60286	61281-61286	14
ARPOINT	42	60301-60302	61301-61302	15
KWKARRAY	134	60321-60325	61321-61325	16
IDARRAY	118	60341-60344	61341-61344	17
VDRIVE	38	60401	61401	18
COMUNCOM *	416	60181-60193	61181-61193	19,20

* Modification required depending on disk operating system used. (Replace 7th and 8th bytes with USR routine pointer address from appendix 2.)

Note: USRDATA1/LIB, USRDATA2/LIB, and USRFILE/RND are files on the "BASIC Faster & Better" BFBLIB diskette. USRDATA1/LIB contains data statements in poke format. USRDATA2/LIB contains data statements in magic array format. USRFILE/RND is a random file, each physical record containing executable machine language code.

USR Routine Data – Merge Library

```
0 '"USRDATA1/LIB" - USR SUBROUTINE MERGE LIBRARY - POKE FORMAT
   (C)(P)1981  LEWIS ROSENFELDER, "BASIC FASTER & BETTER"
   IJG COMPUTER SERVICES, 1260 W. FOOTHILL, UPLAND, CA 91786
**************************************************************
60000 'MOVEDATA 16 BYTES
**************************
60001 DATA0,33,0,0,0,17,0,0,0,1,0,0,237,176,201,0
60020 'MOVEX     88 BYTES
**************************
60021 DATA205,127,10,0,221,42,20,91,221,117,49,221,116,50,221,52,10,221,52,10,221,52,13,221,52,13,221,126,10,6,49,144,221,70
60022 DATA48,144,40,1,201,221,54,10,49,221,54,13,50,24,6,0,0,0,0,0,0,229,193,221,110,49,221,102,50,229,221,94,51,221,86,52,183
60023 DATA237,82,225,56,3,237,176,201,9,43,235,9,43,235,237,184,201
60040 'SUMSNG    47 BYTES
**************************
60041 DATA205,127,10,229,43,70,43,78,225,229,197,205,177,9,193,225,11,121,176,40,14,17,4,0,25,229,197,205,194,9,205,22,7,24,235
60042 DATA17,0,0,33,33,65,1,4,0,237,176,201
60060 'SUMDBL    59 BYTES
**************************
60061 DATA205,127,10,229,43,70,43,78,209,213,197,62,8,50,175,64,33,29,65,205,211,9,193,209,11,121,176,40,18,33,8,0,25,229,197
60062 DATA235,33,39,65,205,211,9,205,119,12,24,231,17,0,0,33,29,65,1,8,0,237,176,201
60080 'LSTRIP    31 BYTES
**************************
60081 DATA205,127,10,229,78,35,94,35,86,235,121,183,40,9,62,32,190,32,4,13,35,24,243,235,225,113,35,115,35,114,201
60100 'RSTRIP    30 BYTES
**************************
60101 DATA205,127,10,229,6,0,78,35,94,35,86,235,9,43,121,183,40,9,62,32,190,32,4,13,43,24,243,225,113,201
60120 'STRCOMPL  19 BYTES
**************************
60121 DATA205,127,10,70,35,94,35,86,235,4,5,200,126,47,119,35,16,250,201
60140 'UPPERCON  28 BYTES
**************************
60141 DATA205,127,10,70,35,94,35,86,235,4,5,200,126,254,97,56,7,254,123,48,3,230,95,119,35,16,241,201
60160 'BITSRCH   72 BYTES
**************************
60161 DATA205,127,10,17,0,0,229,235,78,35,94,35,86,213,221,225,225,17,0,0,12,13,40,42,221,126,0,6,8,229,183,237,82,225,40,9,19
60162 DATA203,63,16,244,221,35,24,232,203,71,32,20,203,63,35,16,247,221,35,13,40,7,221,126,0,6,8,24,235,33,255,255,195,154,10
60180 'COMUNCOM  416 BYTES
***************************
60181 DATA205,127,10,0,221,42,34,91,221,117,49,221,116,50,221,52,10,221,52,10,221,52,13,221,52,13,221,126,10,6,49,144,221,70
60182 DATA48,144,40,1,201,221,54,10,49,221,54,13,50,24,8,0,0,0,0,0,0,0,0,221,110,53,221,102,54,35,94,35,86,221,115,53,221,114
60183 DATA54,221,70,55,221,229,253,225,221,110,49,221,102,50,78,62,0,12,13,40,24,60,60,203,72,40,1,60,13,40,14,13,40,11,203,72
60184 DATA40,2,24,237,13,40,2,24,232,221,110,51,221,102,52,229,78,35,94,35
60185 DATA86,185,40,33,56,27,245,221,229,197,253,229,205,87,40,253,225,193,221,225,237,91,212,64,241,225,119,35,115,35,114,24
60186 DATA5,225,119,24,1,225,213,217,253,110,49,253,102,50,70,35,94,35,86,213,253,225,209,4,5,217,200,221,110,53,221,102,54,203
60187 DATA72,32,115,17,39,0,25,229,225,229,253,126,0,1,40,0,237,185,17,64,6,6,0,33,0,0,203,57,48,1,25,40,5,235,41,235,24,244
60188 DATA203,64,32,26,235,217,5,217,40,61,225,229,253,35,253,126,0,1,40,0,237,185,213,17
60189 DATA40,0,6,1,24,211,209,25,235,217,5,217,40,33,225,229,253,35,253,126,0,1,40,0,237,185,235,9,235,217,5,217,40,13,217,213
60190 DATA19,19,217,225,114,35,115,253,35,24,155,225,217,213,217,225,114,35,115,201,229,217,203,128,213,217,221,225,221,43,253
60191 DATA102,0,253,35,253,110,0,253,35,253,229,14,3,22,40,125,108,38,0,30,0,6,16,253,33,0,0,41,23,48,1,44,253,41,253,35,183
60192 DATA237,82,48,3,25,253,43,16,237,124,209,225,229,213,95,22,0,25,126,221,229,6,0,221
60193 DATA9,221,119,0,221,225,13,40,5,253,229,225,24,194,253,225,221,35,221,35,221,35,217,5,5,217,40,2,24,164,225,201
60200 'SORT1     188 BYTES
***************************
60201 DATA205,127,10,229,221,225,221,78,2,221,70,3,24,4,217,229,217,193,33,0,0,183,237,66,208,203,56,203,25,197,217,225,217,221
60202 DATA110,2,221,102,3,183,237,66,229,217,209,217,8,203,135,8,221,78,0,221,70,1,197,33,1,0,229,217,193,229,217,209,25,229
60203 DATA209,41,25,221,94,0,221,86,1,25,209,213,229,24,12,225,225,8,245,8,241,203,71,40,177,24,207,26,79,70,213,35,94,35,86
60204 DATA235,209,229,235,35,94,35,86,225,4,5,32,6,12,13,32,47,24,16,12,13,40,12,26,190
60205 DATA32,6,35,19,5,13,24,232,48,29,217,213,197,217,209,225,183,237,82,40,190,19,213,217,193,217,6,0,14,3,209,225,9,229,235
60206 DATA9,229,24,184,225,209,213,229,6,3,26,78,119,121,18,35,19,16,247,8,203,199,8,24,206
60220 'SORT2     212 BYTES
***************************
60221 DATA205,127,10,229,221,225,221,78,8,221,70,9,24,4,217,229,217,193,33,1,0,183,237,66,208,203,56,203,25,197,217,225,217,221
60222 DATA110,8,221,102,9,183,237,66,229,217,209,217,8,203,135,8,221,78,4,221,70,5,197,33,1,0,229,217,193,229,217,209,25,235
60223 DATA221,78,12,27,33,0,0,203,57,48,1,25,40,5,235,41,235,24,244,221,94,4,221,86,5,25,209,213,229,24,12,225,225,8,245,8,241
60224 DATA203,71,40,161,24,191,221,78,14,6,0,9,235,9,235,221,70,16,26,190,40,2,24
60225 DATA6,35,19,16,246,24,4,56,2,24,30,217,213,197,217,209,225,183,237,82,40,205,19,213,217,193,217,6,0,221,78,12,209,225,9
60226 DATA229,235,9,229,24,198,225,229,221,94,18,221,86,19,221,78,12,6,0,197,237,176,193,209,225,229,213,197,237,176,193,225
60227 DATA209,213,229,221,110,18,221,102,19,237,176,8,203,199,8,24,183
```

```
60240 'SORT3     153 BYTES
***************************
60241 DATA205,127,10,229,221,225,221,78,8,221,70,9,221,110,10,221,102,11,126,35,94,35,86,221,110,4,221,102,5,8,121,176,40,76
60242 DATA11,197,213,229,221,78,12,221,70,13,9,235,9,235,221,70,14,26,190,40,4,56,16,24,4,35,19,16,244,225,8,95,22,0,25,209,193
60243 DATA24,212,221,110,6,221,102,7,209,213,229,183,35,237,82,229,193,225,229,8,95,22,0,25,235,225,221,115,6,221,114,7,237,184
60244 DATA225,209,193,24,21,229,213,8,221,110,6,221,102,7,6,0,79,9,221,117,6,221,116
60245 DATA7,209,225,235,6,0,79,237,176,221,110,8,221,102,9,35,221,117,8,221,116,9,195,154,10
60260 'SEARCH1   133 BYTES
***************************
60261 DATA205,127,10,229,221,225,221,78,2,221,70,3,17,0,0,8,221,126,6,8,217,221,110,4,221,102,5,78,35,94,35,86,221,110,0,221
60262 DATA102,1,197,213,229,70,213,35,94,35,86,235,209,4,5,32,6,12,13,32,61,24,49,12,13,40,12,26,190,32,6,35,19,5,13,24,232,48
60263 DATA43,8,245,8,241,203,87,32,45,217,121,176,40,11,11,19,217,225,35,35,35,209,193,24,195,11,225,225,225,197,225,195,154
60264 DATA10,8,245,8,241,203,71,32,12,24,221,8,245,8,241,203,79,32,2,24,211
60265 DATA217,213,193,24,223
60280 'SEARCH2   169 BYTES
***************************
60281 DATA205,127,10,229,221,225,221,78,12,221,94,0,221,86,1,27,33,0,0,203,57,48,1,25,40,5,235,41,235,24,244,235,221,110,4,221
60282 DATA102,5,25,221,117,18,221,116,19,221,110,16,221,102,17,70,72,35,94,35,86,213,197,221 ,94,0.221.86,1,221,110,8,221,102
60283 DATA9,183,237,82,56,84,221,110,18,221,102,19,221,94,14,22,0,25,193,209,213,197,26,190,32,6,19,35,16,248,24,33,221,110,0
60284 DATA221,102,1,35,221,117,0,221,116,1,221,110,18,221,102,19,221,94,12,22,0,25,221
60285 DATA117,18,221,116,19,24,180,221,110,10,221,102,11,70,221,94,18,221,86,19,35,115,35,114,221,110,0,221,102,1,24,4,46,0,38
60286 DATA0,193,193,195,154,10
60300 'ARPOINT   42 BYTES
**************************
60301 DATA205,127,10,94,35,86,35,229,235,229,43,70,43,78,217,225,227,94,35,86,35,6,0,78,225,113,35,115,35,114,35,235,9,235,217
60302 DATA11,121,176,200,217,24,239
60320 'KWKARRAY  134 BYTES
***************************
60321 DATA205,127,10,229,221,225,221,110,10,221,102,11,78,6,0,35,94,35,86,221,203,2,70,40,31,235,221,94,6,221,86,7,237,176,221
60322 DATA115,6,221,114,7,221,110,8,221,102,9,35,221,117,8,221,116,9,195,154,10,213,197,221,94,0,221,86,1,27,33,0,0,203,57,48
60323 DATA1,25,40,5,235,41,235,24,244,221,94,4,221,86,5,25,193,209,221,203,2,78,32,3,237,176,201,235,237,176,221,110,6,221,102
60324 DATA7,183,237,82,56,8,221,110,8,221,102,9,24,189,221,115,6,221,114,7,221,110
60325 DATA0,221,102,1,24,169
60340 'IDARRAY   118 BYTES
***************************
60341 DATA205,127,10,229,221,225,221,110,2,221,102,3,229,43,86,43,94,43,43,43,43,43,43,126,221,110,4,221,102,5,213,229,79,203
60342 DATA225,203,57,41,235,41,235,203,57,48,248,203,71,40,8,193,9,235,193,9,235,24,2,193,193,193,9,235,9,6,0,79,221,203,0,70
60343 DATA32,19,213,235,9,229,235,183,237,82,229,193,225,209,40,2,237,176,43,24,19,43,229,183,237,66,229,35,183,237,82,229,193
60344 DATA225,209,40,2,237,184,235,71,62,0,119,43,16,252,201
60400 'VDRIVE   38 BYTES
****************************
60401 DATA221,110,3,221,102,4,218,154,4,221,126,5,183,40,1,119,121,254,32,218,6,5,254,128,210,166,4,229,38,32,188,48,1,124,
      225,195,125,4



0 '"USRDATA2/LIB" - USR SUBROUTINE MERGE LIBRARY - ARRAY FORMAT
    (C)(P)1981  LEWIS ROSENFELDER, "BASIC FASTER & BETTER"
    IJG COMPUTER SERVICES, 1260 W. FOOTHILL, UPLAND, CA 91786
*******************************************************************
61000 'MOVEDATA 8 ELEMENTS
*****************************
61001 DATA8448,0,4352,0,256,0,-20243,201
61020 'MOVEX     44 ELEMENTS
*****************************
61021 DATA32717,10,10973,23316,30173,-8911,12916,13533,-8950,2612,13533,-8947,3380,32477,1546,-28623,18141,-28624
61022 DATA296,-8759,2614,-8911,3382,6194,6,0,0,-6912,-8767,12654,26333,-6862,24285,-8909,13398,-4681,-7854,824,-20243
61023 DATA2505,-5333,11017,-4629,-13896
61040 'SUMSNG    24 ELEMENTS
*****************************
61041 DATA32717,-6902,17963,20011,-6687,-12859,2481,-7743,30987,10416,4366,4,-6887,-12859,2498,5837,6151,4587,0
61042 DATA8481,321,4,-20243,201
61060 'SUMDBL    30 ELEMENTS
*****************************
61061 DATA32717,-6902,17963,20011,-10799,16069,12808,16559,7457,-12991,2515,-11839,30987,10416,8466,8,-6887,-5179
61062 DATA10017,-12991,2515,30669,6156,4583,0,7457,321,8,-20243,201
61080 'LSTRIP    16 ELEMENTS
*****************************
61081 DATA32717,-6902,9038,9054,-5290,-18567,2344,8254,8382,3332,6179,-5133,29153,29475,29219,201
61100 'RSTRIP    15 ELEMENTS
*****************************
61101 DATA32717,-6902,6,9038,9054,-5290,11017,-18567,2344,8254,8382,3332,6187,-7693,-13967
61120 'STRCOMPL  10 ELEMENTS
*****************************
61121 DATA32717,17930,24099,22051,1259,-14331,12158,9079,-1520,201
61140 'UPPERCON  14 ELEMENTS
*****************************
61141 DATA32717,17930,24099,22051,1259,-14331,-386,14433,-505,12411,-6653,30559,4131,-13839
```

```
61160 'BITSRCH   36 ELEMENTS
******************************
61161 DATA32717,4362,0,-5147,9038,9054,-10922,-7715,4577,0,3340,10792,32477,1536,-6904,-4681,-7854,2344,-13549,4159
61162 DATA-8716,6179,-13336,8263,-13548,9023,-2288,9181,10253,-8953,126,2054,-5352,-223,-15361,2714
61180 'COMUNCOM  208 ELEMENTS
******************************
61181 DATA32717,10,10973,23330,30173,-8911,12916,13533,-8950,2612,13533,-8947,3380,32477,1546,-28623,18141,-28624
61182 DATA296,-8759,2614,-8911,3382,6194,8,0,0,0,-8960,13678,26333,9014,9054,-8874,13683,29405,-8906,14150,-6691
61183 DATA-7683,28381,-8911,12902,15950,3072,10253,15384,-13508,10312,15361,10253,3342,2856,18635,552,-4840,10253
61184 DATA6146,-8728,13166,26333,-6860,9038,9054
61185 DATA-18090,8488,6968,-8715,-14875,-6659,22477,-728,-15903,-7715,23533,16596,-7695,9079,9075,6258,-7931,6263,-7935
61186 DATA-9771,28413,-719,12902,9030,9054,-10922,-7683,1233,-9979,-8760,13678,26333,-13514,8264,4467,39,-6887,-6687
61187 DATA32509,256,40,-17939,16401,1542,8448,0,14795,304,10265,-5371,-5335,-3048,16587,6688,-9749,-9979,15656,-6687
61188 DATA9213,32509,256,40,-17939,4565
61189 DATA40,262,-11496,6609,-9749,-9979,8488,-6687,9213,32509,256,40,-17939,2539,-9749,-9979,3368,-10791,4883,-7719
61190 DATA9074,-653,6179,-7781,-10791,-7719,9074,-13965,-9755,-32565,-9771,-7715,11229,26365,-768,-733,110,9213,-6659
61191 DATA782,10262,27773,38,30,4102,8701,0,5929,304,-724,-727,-18653,21229,816,-743,4139,31981,-7727,-10779,5727
61192 DATA6400,-8834,1765,-8960
61193 DATA-8951,119,-7715,10253,-763,-7707,-15848,-7683,9181,9181,9181,1497,-9979,552,-23528,-13855
61200 'SORT1      94 ELEMENTS
******************************
61201 DATA32717,-6902,-7715,20189,-8958,838,1048,-6695,-15911,33,-18688,17133,-13360,-13512,-15079,-7719,-8743,622,26333
61202 DATA-18685,17133,-9755,-9775,-13560,2183,20189,-8960,326,8645,1,-9755,-6719,-11815,-6887,10705,-8935,94,22237
61203 DATA6401,-10799,6373,-7924,2273,2293,-13327,10311,6321,6863,17999,9173,9054,-5290,-6703,9195,9054,-7850,1284
61204 DATA1568,3340,12064,4120,3340,3112,-16870
61205 DATA1568,4899,3333,-6120,7472,-10791,-9787,-7727,-4681,10322,5054,-9771,-9791,6,782,-7727,-6903,2539,6373,-7752
61206 DATA-10799,1765,6659,30542,4729,4899,-2288,-13560,2247,-12776
61220 'SORT2      106 ELEMENTS
******************************
61221 DATA32717,-6902,-7715,20189,-8952,2374,1048,-6695,-15911,289,-18688,17133,-13360,-13512,-15079,-7719,-8743,2158
61222 DATA26333,-18679,17133,-9755,-9775,-13560,2183,20189,-8956,1350,8645,1,-9755,-6719,-11815,-5351,20189,6924,33
61223 DATA-13568,12345,6401,1320,10731,6379,-8716,1118,22237,6405,-10799,6373,-7924,2273,2293,-13327,10311,6305
61224 DATA-8769,3662,6,-5367,-5367,18141,6672,10430,6146
61225 DATA8966,4115,6390,14340,6146,-9954,-14891,-11815,-18463,21229,-13016,-10989,-15911,1753,-8960,3150,-7727,-6903
61226 DATA2539,6373,-7738,-8731,4702,22237,-8941,3150,6,-4667,-15952,-7727,-10779,-4667,-15952,-11807,-6699,28381,-8942
61227 DATA4966,-20243,-13560,2247,-18664
61240 'SORT3      77 ELEMENTS
******************************
61241 DATA32717,-6902,-7715,20189,-8952,2374,28381,-8950,2918,9086,9054,-8874,1134,26333,2053,-20359,19496,-15093
61242 DATA-6699,20189,-8948,3398,-5367,-5367,18141,6670,10430,14340,6160,8964,4115,-7692,24328,22,-12007,6337,-8748
61243 DATA1646,26333,-12025,-6699,9143,21229,-15899,-6687,24328,22,-5351,-8735,1651,29405,-4857,-7752,-15919,5400,-10779
61244 DATA-8952,1646,26333,1543,20224,-8951,1653,29917
61245 DATA-12025,-5151,6,-4785,-8784,2158,26333,8969,30173,-8952,2420,-25917,10
61260 'SEARCH1     67 ELEMENTS
******************************
61261 DATA32717,-6902,-7715,20189,-8958,838,17,2048,32477,2054,-8743,1134,26333,19973,24099,22051,28381,-8960,358
61262 DATA-10811,18149,9173,9054,-5290,1233,8197,3078,8205,6205,3121,10253,6668,8382,8966,1299,6157,12520,2091
61263 DATA2293,-13327,8279,-9939,-20359,2856,4875,-7719,8995,-11997,6337,3011,-7711,-14879,-15391,2714,-2808,-3832,18379
61264 DATA3104,-8936,-2808,-3832,20427,544,-11496
61265 DATA-10791,6337,223
61280 'SEARCH2   85 ELEMENTS
******************************
61281 DATA32717,-6902,-7715,20189,-8948,94,22237,6913,33,-13568,12345,6401,1320,10731,6379,-5132,28381,-8956,1382
61282 DATA-8935,4725,29917,-8941,4206,26333,17937,9032,9054,-10922,-8763, 94,22237,-8959,2158,26333,-18679,21229
61283 DATA21560,28381,-8942,4966,24285,5646,6400,-11839,-14891,-16870,1568,8979,-2032,8472,28381,-8960,358,-8925,117
61284 DATA29917,-8959,4718,26333,-8941,3166,22,-8935
61285 DATA4725,29917,6163,-8780,2670,26333,17931,24285,-8942,4950,29475,29219,28381,-8960,358,1048,46,38,-15935
61286 DATA-25917,10
61300 'ARPOINT   21 ELEMENTS
******************************
61301 DATA32717,24074,22051,-6877,-6677,17963,20011,-7719,24291,22051,1571,19968,29153,29475,29219,-5341,-5367,3033
61302 DATA-20359,-9784,-4328
61320 'KWKARRAY  67 ELEMENTS
******************************
61321 DATA32717,-6902,-7715,28381,-8950,2918,1614,8960,9054,-8874,715,10310,-5345,24285,-8954,1878,-20243,29661,-8954
61322 DATA1906,28381,-8952,2406,-8925,2165,29917,-15607,2714,-14891,24285,-8960,342,8475,0,14795,304,10265,-5371
61323 DATA-5335,-3048,24285,-8956,1366,-16103,-8751,715,8270,-4861,-13904,-4629,-8784,1646,26333,-18681,21229,2104,28381
61324 DATA-8952,2406,-17128,29661,-8954,1906,28381
61325 DATA-8960,358,-22248
61340 'IDARRAY   59 ELEMENTS
******************************
61341 DATA32717,-6902,-7715,28381,-8958,870,11237,11094,11102,11051,11051,32299,28381,-8956,1382,-6699,-13489,-13343
61342 DATA10553,10731,-13333,12345,-13320,10311,-16120,-5367,2497,6379,-16126,-15935,-5367,1545,20224,-13347,17920,4896
61343 DATA-5163,-6903,-18453,21229,-15899,-11807,552,-20243,6187,11027,-18459,17133,9189,-4681,-6830,-7743,10449,-4862
61344 DATA-5192,15943,30464,4139,-13828
61400 'VDRIVE   19 ELEMENTS
******************************
61401 DATA28381,-8957,1126,-25894,-8956,1406,10423,30465,-391,-9696,1286,-32514,-22830,-6908,8230,12476,31745,-15391,1149
```

Index

H

I

J

K

L

M

N

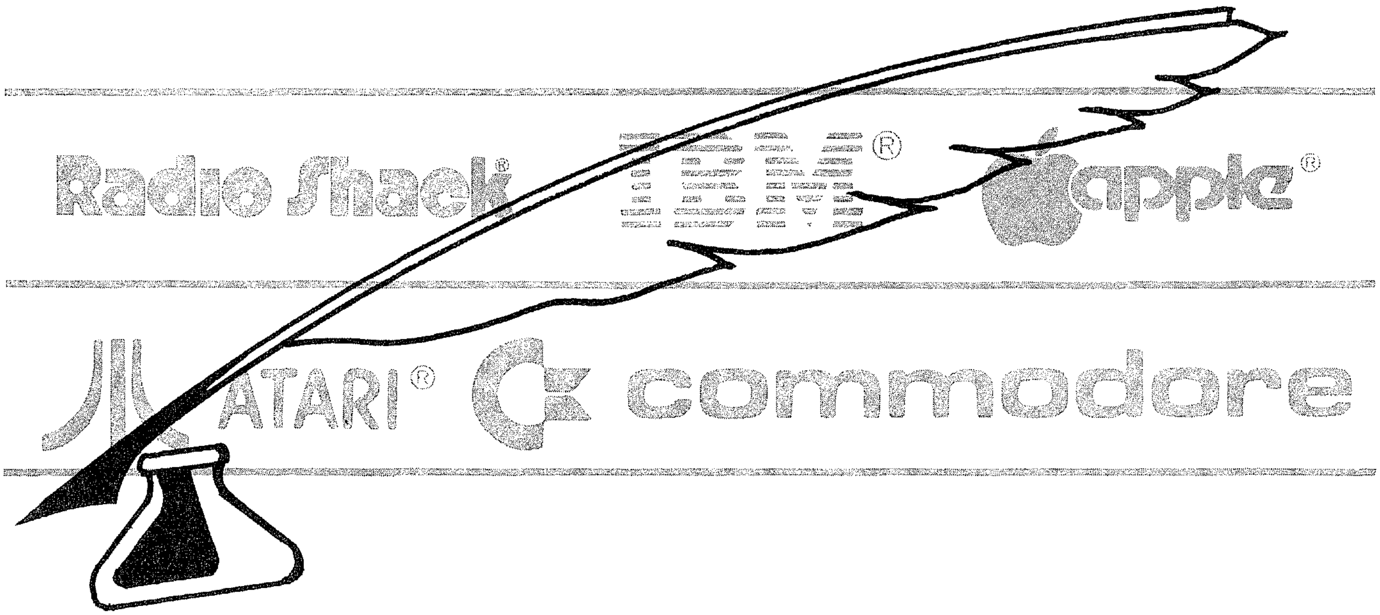
O

P

Q

R

WANTED



Book & Software
AUTHORS

IJG INC. has become a world wide recognized leader in computer publishing. We take pride in publishing only the best in computer oriented books and software. If you have an idea, and really know your subject, we would like to talk with you.

Qualifying manuscripts once submitted, will be read and evaluated by our professional editorial staff (who are themselves published authors), and a few selected writers will be invited in for a personal evaluation of their work.

Contact Mr. Harvard Pennington or Mr. David Moore.

IJG

1953 West
11th Street
Upland,CA
91786 (714)
946-5805