ADVANCED 2068 MACHINE CODE

VOLUME 1

by

Dr. Lloyd Dreger

Advanced Screen Techniques and Graphics
Discussed in a Down to Earth Terms Even
The Novice M/C Programmer Can Understand

(c) 1987 by Dr. Lloyd Dreger

Distributed by S.M.U.G.--Sinclair Milwaukee Users Group
Box 101, Butler, WI 57007

This book was typed on the Timex/Sinclair 2068 Computer using MSCRIPT. An Aerco Printer Interface sent the text to a Gemini Star 10X Printer. It was stored on disks using an AERCO disk drive system.

DEDICATION

To the Timex/Sinclair 2068 Users whose dedication to their computer hve not let it die. Without their insistance this book would never have been written.

INDEX

INTRODUCTION.....1
CHAPTER 1 Compilers and Assemblers.....3
To Compile or Not to Compile.....3
Timachine-A 2068 Compiler.....3
Assemblers--Zeus and Hot-Z.....6
CHAPTER 2 Some Preliminaries.....13
Intel Mnemonics vs. Zilog Mnemonics.....13
Which Registers to Use?.....18
Standard Screen Routines.....20
Clear The Screen.....20
Locate Address in DFILE.....20
Find Attr.....21
Find Attr from DFILE ADDR.....21
Find DFILE Addr from ATTR Addr.....21
Saving SCREEN$ Space.....22
Relocatable Program Technique.....26
Easy Print Message Routine.....35
CHAPTER 3 Activating AERCO Banks for Basic.....37
Activating Dual Screen Modes for Other Banks.....44
CHAPTER 4 64 and 80 Column Modes.....47
64 Column Screen.....47
Definitions.....50
Inputs and Entries.....50
Other Variables.....50
Write Character.....51
Write String.....53
Position Control.....54
Scroll.....54
Clear Screen.....56
Attribute Control.....58
Set Mask.....58
Get Char.....59
Get Attr Byte.....60
Get Cursor Position.....60
Video Mode Control.....61
Get Value.....62
Enable Home.....63
Test Parameter.....63
Convert Parameter.....63
Calculate Position.....64
64.....65
A Start on a Word Processor.....67
Scroll 64 Screen Left and Right.....73
80 Column Mode.....75
Rewrite of WRCHAR.....75

Rewrite of GETCHAR....78
Rewrite of CALCPOS....80
Compressed Character Set....81
Code Check....84
CHAPTER 5 TWO SCREEN MODE....89
Definitions....90
Inputs and Entries....90
Variables....91
Write Character....91
Write String....93
Position Control....94
Scroll....94
Clear Screen....96
GET Character....98
Get ATTR Byte....99
Get Cursor Position....100
Video Mode Control....100
Copy Screen....102
Exchange Screens....105
Get Value....108
Update Position....109
Basic for Using Dual Screen Mode....112
CHAPTER 6 Hi Res Graphics Mode....119
Change Video....120
Fill....120
Extended Paint....121
CHAPTER 7 Playing With Borders and Sprites....129
Everything You Ever Wanted to Ask About Interrupts.129
American vs. British TV....133
The Interrupt Handler....135
The Horizon Handler....140
Initialization Routine....146
Horizon Set Routine....147
Horizon Move Routine....150
HRZNMK Routine....152
Moving Horizon Demo Routine....153
Part 1. MASTDEMO....154
Horizon Color Routine....155
Using the Print Buffer for Stationary Items....156
Alter Buffer Routine....157
Refresh of the RO-Buffer....158
Send Data to RO-Buffer Routine....159
Adding the Gunsight to the Moving Horizon....161
The OR-Printing Routines....163
Print to the Normal Buffer Routine....165
Plotting and Drawing....167
Plot....167
Draw....169
Moire Program....172
Sprites....174
Padout....179
Sprite Expand....181
Sprint....183
Sprite Move....189

Sprite Test.....192
Putting it All Together--Ships.....196
Writing to the Border.....204
Hiron.....204
Borpic.....205
Attset.....207
BFDemo.....209
General Purpose Border Writing Routine.....210
MDEMO.....213
Countdown.....215
Tips to Game Designers.....218
CHAPTER 8 Factuals and Adventure Program Graphics....221
Terrain--A Factual Program.....221
Adventure Graphics.....223
BIBLIOGRAPHY and Copyrights.....243

INTRODUCTION

When I first wrote "Introduction to 2068 Machine Code" it was an effert to provide a teaching manual for my SMUG students in Beginning Machine Code class. As with every "Introduction", many topics are beyond the grasp of the beginning student of assembly language and had to be skipped with the proviso that they would be covered in a follow on book.

This is the first part of that follow on book and covers primarily advanced screen techniques. I have tried to continue in the vein of the first book in giving routines with enough explainations so that even the most novice of machine code programers can, with effort, grasp the concepts and adopt the programs for their own use. I have sprinkled demonstrations liberally throughout the book.

Unfortunately, it takes a lot of time and energy to write all the programs, debug them, write demonstration programs that show how they work and then write it all up in a manner that is understandable by the average m/c programmer. By the time I was half way through the discussion of just the graphics I realized that I would not be able to cover all the subjects that I wanted to cover in less then 400 pages and that it would take me another year to finish it. Rather than hold up the first half of the book for yet another year while writing the second half, I have decided to publish the first half. Hence Volume 1. Such is the fate of those of us who have other full time jobs.

Volume 2 will cover the topics of music and sound in depth and go into the double precision floating point calculator also promised in the "Introduction".

The biggest headache of assembly programers is debugging programs. Hours can be spent getting them to just run, much less to run correctly. For those of you who wish to bypass this agony of imperfect assembly, I am making available copies of the final version of all the main programs and demos discussed in this book either on Aerco or Olliger Disk or tape for $9.95 + $1.50 S&H (WI residents kindly add 5% sales tax). Send your requests to:

SMUG, Box 101, Butler, WI 53007.

Indicate which media you prefer.

CHAPTER 1 COMPILERS AND ASSEMBLERS

TO COMPILE OR NOT TO COMPILE

Why do I start a book on machine code with a subject like this? It's very simple. I believe in doing things the easiest way possible and for some of my readers compiling their basic program into a faster running compiled program may be the easiest solution to their slow running program. Don't bet on it being shorter than the basic program. For most programs lack of memory is not a problem. If compilation fits your needs it's much easier and faster than writing and debugging code. Unfortunately most of us are going to find that we still need to learn code so that we can make the 2068 do things that it can't do with even its powerful basic.

First, what is compiling? If you owned or ran a mini or main frame computer you already know the answer. Most non-PC machines compile programs from some language or another into a machine-running type code. Set routines for the commands of the language are compiled into machine code routines which are then loaded into the machine and run rather than the language program. In this sense of the word, the 2068 is an interpretive machine in that it interprets each basic statement as the program runs. Well, not quite. The use of tokens and slugs in the use of numbers and the pretesting of syntax is a partial use of compilation technique.

From this you can see that the 2068 already has some advantage in running speed over the completely interpretive computers like the IBM PC Basic programs. Using the Z80 CPU also speeds things up over the slower 6500 series CPU's and the 8080 CPU's. The two main reasons to compile are to speed things up and to save space but don't bet on saving much space.

TIMACHINE--A 2068 Compiler.

(Timachine is the work of NovelSoft, 106 7th St., Toronto, Ontario, Canada M8V 3B4. Copyright 1986 by Camron Hayes.)

In case you haven't heard, the Spectrum compiler has been converted to work on the 2068. It generally does speed up running times of programs and if the program is long, it saves space as well. And it even can do floating point numbers. This makes it a good compiler. Some compilers I've seen are so limited they aren't worth using. This one is. It can handle all the Basic commands except some obvious system commands like CLEAR, CONTINUE, ERASE, FORMAT, LIST, LLIST, MERGE, MOVE, NEW, RESET, RUN, SAVE, FREE, ONERR and VERIFY. A compiled program requires a RANDOMIZE USR # statement and in that sense is just

like any other m/c program. Thus, it needs a small basic program to go along with it that can load the code and run the program and thus handle all these commands just like a true machine code program. Most of these comands aren't usually handled in m/c programs either. Since it is a Spectrum program converted to the 2068, there is no support of ONERR and FREE as the Spectrum does not have these commands. FREE we can all live without in a compiled program as the only time it is of use is to ask how much memory is left as you are writing a program. ONERR, on the other hand, could be very useful. Unfortunately, most of the time I have seen it used to cover program errors rather than a more useful purpose. STICK and SOUND ARE supported by the 2068 compiler even though they are not Spectrum commands.

There are a few other limitations to Timachine as well:

1. No calculated GOTO's, GOSUB's or RESTORE's. Must be exact line numbers.
2. Some restrictions on data lines and FN.
3. No calculated DIM statments and not more than 2 dimensions.
4. Strings are all set to 255 characters unless a previous LEN directive is given shortening or lengthening them.
5. The BREAK key WILL work as compiled code is running.

The only real limit is the need for REAL DIM (no variables in DIM statements) and GOTO or GOSUBS (no equivalent of ON GOTO or ON GOSUB). The program is loaded first and occupies from 26688 to 38145. Your Basic loads above that and there must be room for compiling the code above that so the limit is a Basic program of about 27k maximum length. Your program has to be error free as you can't use ONERR to trap errors and continue the program.

This is not a lot of limitations. Let's admit it, many programs can fit that mold. If your basic program fits, go ahead, buy the compiler and compile your programs and forget about reading this book and learning m/c.

Now for some things that NO compiler can do no matter how good. There will be a day when you want your computer to do something that it can't do from basic or wasn't designed to do. I hit that limit very rapidly in my computer program writing when I wanted my screen to scroll left, right and down as well as up. Some people like their computers to speak and play fine music and soon discover that their basic doesn't cut it. Music, as the 2068 Basic uses it is very heavy in numbers and really chews up memory rapidly. Others like to draw fancy pictures of art; although most first attempts at art tend to be quite pornographic, real attempts soon hit the limits of the 2068 as well. AND finally, the real computer games person finds that even a compiled Basic program doesn't run fast enough for that phaser gun to really zap the aliens.

How much faster are compiled programs? Claims range all the way up to 160 times as fast. That may be but the average increase in

speed of a compiled program over its Basic counterpart is going to be on the order of 10 times faster not 100. If you do a lot of floating point calculations you will be lucky to double your speed. Compiled programs are not as brief, precise or efficient as specifically designed m/c programs and thus always take longer to run. In other words, not quite the ultimate in speed and sometimes very far from the ultimate speed and if you are doing a lot of screen printing, quite a bit slower than m/c.

Many people find that they just want to fix an already existing m/c program so that it can do a little more or something a little faster or a little differently and just want to "patch" a program. This is even more difficult than taking the slow part of a Basic program and writing it in code. They have a dual headache on their hands because they first have to disassemble the code, then study it and find out how it works and what is doing what and finally write the modification. It takes a lot of time and effort. Just ask Jack Dohaney about his struggle with MSCRIPT although I really don't call Jack's modifications a patch but more in the category of a general overhaul. It nonetheless takes a lot of time. In-line or inside-the-m/c modifications can't be fixed with writing the program patch in Basic and then compiling around it.

While we are on the subject, most program writers and book writers never make enough money on their efforts to even earn the minimum wage for all the time spent on their efforts. You have to be in it for the pleasure it brings to you helping other people or for the challenge it takes writing the program or the reputation you get for doing a good job writing a good program or game. Most remain poor and unsung heros.

Using TIMACHINE requires the use of quite a few directives written into REM statements that tell the program the type of variables you are using. One must declare quite a few as integer, positive integer or real and make sure you don't divide one integer by another to get a fraction which must be handled as both real numbers giving a real number answer. Reusing variables runs into a problem as well. Defining the lengths of strings can save a lot of memory as undefined strings have 255 bytes reserved for them. A much larger area of memory is used for program variables than in the normal type of machine code routine.

As with m/c, you don't have to compile a whole program but can choose where you want to start and where to end the compilation. You can do the program in sections if you like. Unfortunately, each compilation again saves the same run routines each and every time and as such generally is much longer than the Basic. Each compilation also would have its own section of variables and as such there would be no transfer of variables for one compilation to another. However, the compilation does put the variables into the Basic variable table so they become available for further Basic manipulations. For diagnostic purposes you can

even request a listing of the run routines used. Sorry, but no onestep (See Hot Z below for a definition) through the program allowed. However, since the BREAK key works, any compilation run can be interrupted at any time.

All this sounds like a lot of work. It may take several hours to convert a program for compilation but that is short compared to the amount of time you would spend doing it in your own code.

But if compilation won't work for you for whatever reason, read on. Much of what follows is going to be things that are unsupported from the present 2068 Basic and so require m/c.

ASSEMBLERS

Before we discuss assemblers, let's give a few definitions that most oldtimers in computering grew up with and are still used from time to time.

Source code: The labels, mnemonics and comments of a program which an assembler uses to create the operating code which is called the object code.

Object code: The assembled set of numbers poked into ram. But NOT necessarily at the correct address. In other words it is made to run at one address although presently stored at a different address. This is a necessity with some assemblers as they are using the space where you want to put your code.

Running code: The Object code in its right address space.

Therefore, the sequence generally needed is to type in the source code, assemble the source code into object code and then transfer the object code to the right address for running.

In the Introduction to Machine Code I recommended that you assemble your code by hand and encode it. Then, using a basic loader, enter it into memory and run it. However, this is the big leagues now and your code is getting longer so the use of an assembler saves a lot of time and prevents a lot of stupid encoding errors. There are several assemblers available but I will only discuss two of them: Hot Z and Zeus. Which one you use depends upon the way you like your final code to look. The difference between the way these two operate is about the difference in driving a Corvette and an Escort.

The corvette is Hot Z. It contains everything you would want--as they say, power everything. But the first time behind the wheel, pardon me, keyboard, can be a little nerve-wracking. I strongly recommend that you just load Hot Z and get aquainted with some of its commands. It has three separate functions: assembly, disassembly and running. The running function can be

broken down into running all the code, just a section of code or single-stepping it an instruction line at a time with a full display of all the values of the flags and the registers if you like. It comes in 3 versions: 2.5, 1.8 and 1.9, all of which are loaded in at 32768 and are up to 24576 bytes long. Version 1.8 only supports the 2040 printer. A printer is a must with Hot Z. The low loading of Hot Z allows the entry of code into high memory where it is usually found so that it can be worked on or run--unless you are working with a Spectrum m/c program which has a habit of being loaded at 32768 to accommodate a 48k ram. Hot Z can even accommodate this abnormality by telling it to reassemble the code to run at the addresses you have loaded it into, i.e., reassemble an object code to a running code.

The Escort is Zeus. It's reliable transportation with few frills and easy to learn how to operate. If you have never used an assembler start on Zeus. It is easy to learn as it has a top row of entry manipulations that can easily be held on one side of a 3x5 card you can place on the top part of the 2068 as you are using the program. It has advantages and disadvantages. If you are like me, I have a lot of comments to help me remember what is where and what is supposed to be happening. These are all saved in the source code which can be saved to tape or disk. Reloading in Zeus makes changes very easy. Telling Zeus, which can be modified to work with the Aerco print driver program, gets your program to paper--at least the labels, mnemonics and comments. You then have to add the addresses and codes by hand. Zeus occupies space from 57344 to nearly the top of ram (without the monitor and disassembler) which means that most of the time we will only be assembling an object program--displaced to some other address (unless our program is very short and fits in about 700 bytes and you want it to run in the very top of ram.)

Zeus uses arbitrary line numbers for its source code and allows for renumbering them so insertion or deletion of code is easy. A very fast 2 pass assembler gives code in seconds. A monitor mode allows you to inspect the object code (in hex) and even change it on the spot. A full label table is generated at the time of assembly giving the address of each label which makes it very easy to copy down addresses of the starts of routines.

Hot Z, on the other hand, uses a one-line-at-a-time assembly. It assembles each line as you finish entering it. Nice for backward jumps and calls but not so nice for forward ones especially if you don't know how much forward. There is no saving of a source code. What you see on the screen is the disassembly of already assembled code. Inserting extra code and deleting it is a bit more complex. The manual is written in the symbols of the Timex tokens rather than in first letter commands as in Zeus.

Getting used to what some of these mean is the problem with Hot Z. For example, in Zeus, a number preceded by a # (symbol shifted 3, which appears on the screen as #) means the number is in hex while nonprefixed numbers are in decimal. In Hot Z,

which prefers hex, its the other way around, prefix with OR (symbol shifted U) for a decimal address. In other words, Zeus is a lot more user friendly.

There are a few extra commands that most assemblers use. They are:

ORG ##### Tells the assembler at what address you want the running code to start. Hot Z has no ORG command as you have to be at the right insertion address when you type in assembly code.

DISP ##### Displace the storage of the code by this amount from the ORG address and store it there as you assemble it. Displacements cannot be negative. Therefore, if ORG is set at 60000, a displacement of 40535 will store the code at 60000 + 40535 (less 65535) or at 35000.

Label EQU ##### Predefine a label which is located at the address given. Generally used to specify data storage areas. Can also be a pure number.

DEFB ###,### Tells the assembler that what follows is entered directly into RAM as numbers. Several can be used on the same line separated by commas. The numbers may be a precoded machine code as well as data. For Hot Z you have to move the cursor to the left position and enter in Hex.

DEFW #####,##### As with DEFB but two bytes long each, i.e., word long.

DEFS #### Define an open space of ### bytes in length. A way to reserve space for a future routine or data.

DEFM/string/ The string is loaded as ASCII a symbol at a time. A great help in entering print data strings. May have spaces inside the string.

ENT Start assembling with the next instruction. You don't have to assemble all your source code each time.

END #### Hot Z requires at least a temporary ending address which can be changed as desired before entering code. Zeus just compiles to the end of the assembly code and requires no end.

Comments can be added to the end of the mnemonic by entering a colon or semicolon and then adding the comment. Comments can be on any line by itself by just starting with the colon or semicolon (Check your manual to see which). No comments for Hot z.

Check your assembler for the proper RST number; it may require a hex or a numeric.

Most assemblers also have a limited length for labels. On many this is 6 symbols with the first one having to be alphabetic. Zeus allows up to 14 with a special directive. Also, as should be obvious, you cannot use any of the assembly codes as a label, i.e., LD will not work, but LDBC will--without a space between LD and BC.

The use of the $ symbol is quite universal in assemblers, e.g.,

DJNZ $ is equivalent to loop DJNZ loop

and,

DJNZ $+2 is equivalent to DJNZ 2,

while,

DJNZ $-2 is equivalent to DJNZ 254.

Note also that the DJNZ code does NOT use a comma between it and the following label whereas all other jump relatives do, i.e., JR NZ, loop.

Generally assemblers come with error codes for assembly. Hot Z gives them obviously for the last line entered as it immediately assembles it, Zeus gives the code and lists the line it occurred in. Both are easy to correct.

Using floating point: Two special codes in Hot Z allow you to assemble or disassemble code for the RST 48 routine and its special characteristics. Generally in disassembly you have to tell the program to shift to Floating Point Disassembly as the code doesn't make sense doing a regular disassembly. Try disassembling the floating point routines starting at ROM locations 15035 to 15461. With Zeus you have to use the DEFB and precode your floating point routine. You see, Escorts don't have power steering.

Also, you sometimes get into data tables. If it's ASCII, Hot Z has another directive to let you disassemble using that to try to make sense out of the code. In fact, try the ASCII disassembly first before trying the floating point disassembly as it is more likely to be data. Remember in disassembly, you are missing all the labels and comments that come with a full source code printout. Try ROM location 152 for a demonstration of ASCII disassembly as you are in the token spell table.

There is one situation which defies correct disassembly and that is a pixel table for printing either the ROM characters or UDC chararacters. ROM addresses 15616 to 16383 are a good example for demonstration. The same thing applies to jump tables found at various ROM addresses.

Hot Z will also allow the entry of break points into routines which allows one to run only particular sections of code from one breakpoint to another. Of course registers must be set up for partial running so it doesn't crash which also can be done.

Zeus has no debugging techniques. With HOT Z you can use the break points as discussed above or the one-stepper with a full display of all registers, including the primes, and all flags. You are even allowed to skip over instructions in case you are following a certain set of conditions or if those instructions cause a crash because they lead to a routine that isn't written as yet. It is time-consuming using it, but it gives you everything you could desire.

Name files can be saved in either version although it is not really necessary in Zeus as a reassembly gives you a new table in seconds. A large name file of the routines in ROM and the system variables is included with Hot Z if you wish to take up memory entering it.

Both also have a find function that allows one to look for each and every entry of a label or series of instructions for location purposes. You then can correct any misspellings or otherwise change the code, label, or comments (Zeus).

Hot Z also has a FILC command to fill empty chunks with any character you desire. Zeus has no such command and leaves the spaces with whatever was in them.

Hot Z also can relocate code. With Zeus you need to change the directive DISP and reassemble at the new displacement address.

Both assemblers also give a discription of several routines they use internally which you are also allowed to use in your routines if you like. Hot Z lists the calls for all floating point routines as well.

With tape or disk drives SAVES of code need the starting save address and length included with the name of the code. Reloading from tape or disk to the running address can be easily done by entering the correct running address and the length with the name of the program thus easily changing an object code to a running code in your Basic program.

Which assembler you use is of course your own personal decision be it a Corvette or an Escort. But do get familiar with one and stick to it. You can even mix if you don't get confused with the differences such as entering mnemonics with Zeus, saving the code and then debugging with Hot Z's one-stepper.

The unfortunate part of all assemblers and disassemblers is that they write out the code in Hex. I have yet to find one that doesn't. This isn't of concern because we can directly convert an object code to running code without worrying about the hex. It does, however, give one a headache of the excedrin number type in debugging especially when even the addresses are in hex which is standard technique as well. Be very careful converting hex addresses to decimal for your RAND USR calls and your saves

and all your setup pokes. This is always the first thing I check when a m/c program crashes. A small hand-held calculator does wonders and if it's solar powered you don't even have to ever worry about batteries or line cords. They are quite reasonable.

We always save two copies of everything on two separate tapes or disks so if one copy is bad we have another, don't we? And every once in a while we do our housekeeping and get rid of all the programs with errors but not before we have another program that replaces it with no errors or at least fewer errors. If you are like me with code, it starts simple with the main routine which I debug and save. Then I add what I call the bells and whistles--all the extra stuff. But, I don't mess with that main routine until each bell and whistle works and works in all its variations.

CHAPTER 2 SOME PRELIMINARIES

There are a few topics and loose ends from the Introduction to 2068 Machine Code that I would like to emphasize and clear up.

Intel Mnemonics

Since the Zilog Z80 CPU is an improvement on the Intel 8080, the Intel mnemonics came first and copy the format started with the 6500 series of chips. Since they are in existence, you may sometimes encounter them (as anyone with an Aerco disc drive system already knows). However, with a little thought these mnemonics can be read directly without having to translate back to Zilog type. Accomplishing this one also can then read most 6500 assembly as well.

Since the 8080 has no IX and IY registers and doesn't use the ED and CB extended codes the Intel mnemonics had to be extended to cover these extra instructions for the Z80 chip. Some copy the Z80 mnemonics, others don't. The result is a very inconsistant set, half Intel, half Zilog.

GENERAL SYMBOLS USED

We will use r for any single register
rr for a double register
n for 8 bit values
nn for 16 bit values
ii for Index registers IX and IY
d for displacements
b for bit number
c condition of a flag
s single register or (HL)(M for INTEL)
PC for program counter
SP for stack pointer
I interrupt register
R refresh register

The Intel set has these peculiarities:

1. The first letter of a register pair only is used.
   Thus B really = BC, D=DE, H=HL
   z will be used for this designation
2. M is used for (HL) which to them means memory.

LOAD

Whereas Zilog gets by with one LD instruction, Intel needs

different kinds. In the "Introduction to 2068 Machine Code" I told you that we don't differentiate between the various modes of loading a register or memory. Intel uses MOV to indicate loading one register with another and MVI, move immediate, to load a number directly into a register; STA to store a register number into memory; and LD, load direct, to get one from memory. Sometimes the L and D are separated by the registers.

INTEL	ZILOG
MOV r, r	LD r, r
MOV r, M	LD r, (HL)
MOV r, d(ii)	LD r, (ii+d)
MOV (HL), r	LD (HL), r
MOV d(ii),r	LD (ii+d), r
MVI r, n	LD r, n
MVI M, n	LD (HL), n
MVI d(ii), n	LD (ii+d), n
LDA nn	LD A, (nn)
STA nn	LD (nn), A
LDAX z	LD A, (rr)
STAX z	LD (rr), A
LDAI	LD A, I
LDAR	LD A, R
STAI	LD I, A
STAR	LD R, A
LXI rr, nn	LD rr, nn
LXI ii, nn	LD ii, nn
LBCD nn	LD BC, (nn)
LDED nn	LD DE, (nn)
LHLD nn	LD HL, (nn)
LIXD nn	LD IX, (nn)
LIYD nn	LD IY, (nn)
LSPD nn	LD SP, (nn)
SBCD nn	LD (nn), BC
SDED nn	LD (nn), DE
SHLD nn	LD (nn), HL
SIXD nn	LD (nn), IX
SIYD nn	LD (nn), IY
SSPD nn	LD (nn), SP
SPHL	LD SP, HL
SPIX	LD SP, IX
SPIY	LD SP, IY

PUSH/POP

Except for the single letter notation indicating a double register, PUSH and POP are the same.

e.g., PUSH B = PUSH BC

EXCHANGE

With a few exceptions, easily recognizable.

INTEL	ZILOG
XCHG	EX DE, HL
EXAF	EX AF, AF'
EXX	EXX
XTHL	EX (SP), HL
XTIX	EX (SP), IX
XTIY	EX (SP), IY

BLOCK MOVES AND COMPARES

LDI, LDIR, LDD, LDDR, INI, INIR, IND, INDR, OUTI, and OUTD are identical. Only changes are:

INTEL	ZILOG
OUTIR	OTIR
OUTDR	OTDR
CCI	CPI
CCIR	CPIR
CCD	CPD
CCDR	CPDR

JUMPS, RST, CALLS and RET

Easy if you keep in mind that the condition is put in the main name. Intel uses Overflow (O) and no overflow (NO) as well. Even the equivalents of JP (HL) are easy if you remember that you literally load the SP with HL. Also please keep in mind that a lower case c stands for "condition".

INTEL	ZILOG
JMP nn	JP nn
Jc nn	JP c, nn
JO nn	JP PO, nn
JNO nn	JP PE, nn
JMPR d	JR d
JRc d	JR c, d
JRO d	JR PO, d
JRNO d	JR PE, d
DJNZ d	DJNZ d
PCHL	JP (HL)
PCIX	JP (IX)
PCIY	JP (IY)
CALL nn	CALL nn
Cc nn	CALL c, nn
CO nn	CALL PO, nn
CNO nn	CALL PE, nn
RET	RET
Rc	RET c
RO	RET PO
RNO	RET PE

RETI	RETI
RETN	RETN
RST n	RST n

ARITHMETIC AND LOGIC

Again we have an immediate mode which makes the 3 letter mnemonic look different. Increment and decrement are abbreviated differently. D is prefixed to some double register arithmetic operations.

INTEL	ZILOG
ADD s	ADD A, s
ADD M	ADD A, (HL)
ADD d(ii)	ADD A, (ii+d)
ADI n	ADD A, n
ADC as for add	ADC
ACI n	ADC A, n
SUB s	SUB A, s
SUI n	SUB A, n
SBB s	SBC A, s
SBI n	SBC A, n
ANA s	AND A, s
ANI n	AND A, n
ORA s	OR A, s
ORI n	OR A, n
XRA s	XOR A, s
XRI n	XOR A, n
CMP s	CP A, s
CPI n	CP A, n
INR r	INC r
INR M	INC (HL)
INR d(ii)	INC (ii+d)
DCR r	DEC r
DCR M	DEC (HL)
DCR d(ii)	DEC (ii+d)
DAD rr	ADD HL, rr
DADC rr	ADC HL, rr
DSBC rr	SBC HL, rr
DADX z	ADD IX, rr
DADY z	ADD IY, rr
INX rr	INC rr
INX ii	INC ii
DCX rr	DEC rr
DCX ii	DEC ii

ROTATE AND SHIFT

Rotate/shift L/R circular and logic are the same but arithmetic is rotate/shift arithmetic L/R.

INTEL	ZILOG

RLC	RLCA
RAL	RLA
RRC	RRCA
RAR	RRA
RLCR r	RLC r
RLCR M	RLC (HL)
RLCR d(ii)	RLC (ii+d)
RALR s	RL s
RRCR s	RRC s
RARR s	RR s
SLAR s	SLA s
SRAR s	SRA s
SRLR s	SRL s
RLD	RLD
RRD	RRD
DAA	DAA

IN AND OUT

INTEL	ZILOG
IN n	IN A, (n)
INP r	IN r, (C)
OUT n	OUT A, (n)
OUTP r	OUT t, (C)

GENERAL PURPOSE and CONTROL

Some are the same, others quite different.

INTEL	ZILOG
CMA	CPL
CMC	CCF
STC	SCF
HLT	HALT
NEG	NEG
DI	DI
EI	EI
IM0	IM 0
IM1	IM 1
IM2	IM 2

BIT MANIPULATION

Except for the use of M for (HL) and d(ii) for (ii+d) they are identical with BIT, SET and RES.

6500 MNEMONICS

There you have the complete set. Use these few pages as you read some of the rp/m cp/m assembly and except for a few ringers you will learn them quite rapidly in about an hour or two. Then, if you want, try reading assembly for the Apple, Commadore or Atari

and you will be surprised how much you can already understand. Just remember that these machines do not have all the instructions available to the Z80 and as such have to do things in a round about way with resulting longer routines.

WHICH REGISTERS TO USE?

One of the big questions beginning students of m/c always ask is "How do you know what variable to put into what registers?" The answer to this question is, of course, "It depends on what you have to do with that variable." Even students who already know code can stand a review of this subject.

Even more so than in Basic, because you are limited in the number of things that you can have active at any one time as each must be assigned a register, knowing what has to come first is a must. Therefore, FLOW DIAGRAMMING YOUR PROGRAM IS A MUST.

One also has to consider the addition of another set of variables called memory pointers which we don't have to keep track of in Basic. For example, a very simple "print a word to the screen" routine requires several pointers: 1. A pointer to the memory location where you have stored the spelling of the word which is pointing to the present letter you are printing, 2. Calculating the code of this character into an address pointing to the 8 pixel bytes needed for this character located in the character pixel table, and 3. A pointer to the screen position you want to print to. With only 3 double registers, we already don't have room for a counter which is also a necessity. We can't use the A register for that as it will be necessary to transfer data from (DE) to (HL). Something has to be pushed to the stack to save it and provide room for more calculations. If it ever was true, the motto "PLAN AHEAD" is a must in m/c.

Now, don't be frustrated. Every beginner finds himself writing or calculating himself into a corner where he finds he needs something from the stack which he has already "buried". The beginner will also find his code much more lengthy and very unelegant. Don't let that bother you either. The proof of the value of code is that it runs, not how brief or elegant it is.

However, there are several considerations in picking a particular register or register pair that will help one avoid some of these pitfalls. I have already pointed out that all the codes that you may want are not available for all registers. It was pointed out also that the HL pair is the only double register you can add to or subtract from. This must be kept in mind if you have to make calculations on numbers bigger than 255. You will also find that HL is also the most versatile double register pair to get things to and from memory and even change memory when used as a pointer to memory. For example, there are only instructions like LD B,(HL) or AND A,(HL), OR A,(HL) and XOR A,(HL) along with SET BIT,(HL) or RESET BIT,(HL)

and of course the test, BIT (HL) and of course, INC (HL) and DEC (HL). These last several allow you to change memory without first loading the value to A, making the changes and then loading it back. Therefore, save HL for your most important pointer, the one you have to manipulate the most. And the one which can point at memory locations which require simple changes or tests.

DE is almost as versitle as HL because of the instruction EX DE, HL which allows you to flip-flop HL and DE into the HL bracket for the fancy manipulations allowable only in HL. Therefore, its the 2nd most important pointer location and the one where only occassional manupulations are necessary.

BC is the least valuable as a pointer but because of the instruction, DJNZ, B is the prime counter register for numbers less than 256. This makes the C register the least valuable unless you are inputting and outputting to ports when all the OUT (C) and IN (C) instructions save time and space. C is sometimes used to temporarilly store values for comparisons to A and other caluclations with A. When countdowns of bigger values than 255, the BC register pair is generally used.

One more thing about HL. There is a little known instruction which answers the question all of us have had from time to time, "How can I save HL to the stack and then POP off the now buried other value of HL I need." It is EX (SP), HL which means exchange the last value pushed to the stack with HL. This allows one to have two values of HL pointing at two different things as long as you only need to use one at a time. One is on the stack while you use the other. Then, with EX (SP), HL you save the one you are done with and now have the other ready to use.

Now, for the single registers. Your most important register is A. All math and logic including CP, NEG and a lot of special things can only occur in A. It is also the only register than can't be paired into a double register. It must be kept free of long term pointer values and other things because it also is used as the input register for the double egisters? It is not recommended that the beginner use them. First of all, if you are going to and from Basic make sure you save and retrieve the HL' register pair on the way in and out of your routine. Secondly, they can only be used to store a value so you might as well use the stack. To get at them you need to use EXX which not only exchanges BC with BC' but also DE with DE' and HL with HL'. Only EX AF,AF' works by itself. The prime registers are great if you need a second full set of registers or another completely separate set of calculations. You can't mix HL' and DE or things like that. Most calculations a beginner may be doing won't require their use.

IX and IY are always double registers and although very useful to the advanced student it must be remembered that IY must be reset before coming back to basic or you crash. The special

displacement indexing of the IX and IY makes tem ideal for looking at stored table values which is hardly a subject for beginning students.

So much for special registers. None of this is hard and fast LAW. You can use any register, except A, for any purpose you desire. Even BC can be used as a memory pointer because of the instructions LD A, (BC) and LD (BC), A.

Standard screen routines:

How are you doing on your book of standard routines you were to start? Every good machine programer has such a book so he doesn't have to continuously rewrite these standard routines. A good programmer also completely documents every program he writes in their final form. Because we all make a lot of mistakes writing code and have to constantly revise them we have a lot of copies lying around with earlier attempts that we really should get rid of. (We are not all geniuses like Mozart who is said to have written his music once in its final form as it never required revision.) It takes extra time to do it but if for some reason you have to ever come back and revise the program you know where you are. If you are like me you have 3 or 4 projects going at the same time and jump back and forth between them as the inspiration moves you so its even more important to know where you left off.

What routines should you have in your standard book? It depends upon your interests, of course, but some screen printing routines always come in handy. You should have at least:

CLEAR THE SCREEN:

ENTRY WITH A = ATTRIBUTE

```
210040  CLS         LD HL, #4000
010018              LD BC, #1800
75                  LD (HL),L: L = 0
54                  LD D, H
1E01                LD E, 1
EDB0                LDIR
77                  LD (HL), A
01FF02              LD BC, #02FF
EDB0                LDIR
C9                  RET
```

LOCATE ADDRESS IN DFILE

ENTRY: B = LINE, C = COLUMN
EXIT: HL = ADDRESS IN DFILE

```
78    DF-LOC        LD A, B
E6F8                AND 248: SAVE TOP 4 BITS ONLY
C640                ADD A, 64
```

```
67                      LD H, A
78                      LD A, B
E607                    AND 7
OF                      RRCA
OF                      RRCA
OF                      RRCA
81                      ADD A, C
6F                      LD L, A
C9                      RET
```

AND FIND ATTR:

ENTRY: B= LINE, C= COLUMN
EXIT: HL = ADDR IN ATTR FILE

```
78    ATTLOC            LD A, B
CB2F                    SRA A
CB2F                    SRA A
CB2F                    SRA A
C658                    ADD A, 88
67                      LD H, A
78                      LD A, B
E607                    AND 7
OF                      RRCA
OF                      RRCA
OF                      RRCA
81                      ADD A, C
6F                      LD L, A
C9                      RET
```

FIND ATTR ADDR FROM DFILE ADDR

ENTRY: HL = DFILE ADDR
EXIT: DE= ATTR ADDR

```
7C    DF-ATT            LD A,H
OF                      RRCA
OF                      RRCA
OF                      RRCA
E603                    AND 3
F658                    OR 88
57                      LD D, A
5D                      LD E, L
C9                      RET
```

And the opposite, FIND DFILE ADDR FROM ATTR ADDR

ENTRY: HL = ATTR ADDR
EXIT: DE= DFILE ADDR

```
7C    ATT-DF            LD A, H
E603                    AND 3
07                      RLCA
07                      RLCA
```

```
07          RLCA
F640        OR 64
57          LD D, A
5D          LD E, L
C9          RET
```

These are the absolute minimum. You should probably also have a scroll left and a scroll right routine as well. Later in the book we will give you the routine for scrolling the 64 column screen left and right. We leave it to the student to write his or her own scroll routines. They are quite simple if you remember you have to do it a pixel line at a time and you don't have to do them in the screen order.

Saving SCREEN$ Space

How many screens can you save in memory? Each screen requires, 6912 bytes and with 38657 bytes useable this makes for 5 screens worth with 4097 left for program. Even with only 4 screens you only have 11K for program. Can something be done to reduce this screen overhead? The answer of course depends upon what type of screen you are saving. IF it's a detailed graphic art you can't save much space but even a picture of the Mona Lisa has a lot of background of all the same type of character which can save space. Generally the background pixels are either all full, as in inverse printing, or all blank and the routine listed below is based on that fact.

The routine looks at each pixel on the screen in order. If it is a 0 or 255, a special routine is called to look at the next byte and if that is the same a count is accumulated until a different byte is found. At that point the count is placed in the byte following the 255 or the 0 in the compressed screen table. Therefore, each 255 or 0 pixel byte is followed by a count of how many consecutive bytes are the same. Only in the unlikely event of having a screen of alternately 255 and 0 would we get a screen save longer than 6912. An interesting point about the routine is that by using PRINT USR 65000 you get the first empty byte address and thus the length of your compressed screen save.

I loaded it at 65000 but it calls no routines so is immediately relocatable to where ever you wish.

```
65000 210040  COMPRESS   LD HL, 16384
65003 11XXXX             LD DE, SCRSAVE
65006 7E        LOOP     LD A, (HL)
65007 12                 LD (DE), A
65008 FEFF               CP 255
65010 280E               JR Z,FULL
65012 FE00               CP 0
65014 282D               JR Z, EMPTY
65016 13                 INC DE
65017 23                 INC HL
```

```
65018 7C                LD A, H: END OF FILE?
65019 FE5B              CP 91
65021 20EF              JR NZ, LOOP
65023 42        OUT1    LD B, D: LD FINLE COMP CODE BYTE TO BC
65024 4B                LD C, E
65025 C9                RET
65026 13        FULL    INC DE
65027 3E01              LD A, 1
65029 08        FLOOP   EX AF, AF': SAVE COUNT IN AF'
65030 23                INC HL
65031 7C                LD A, H: CHECK FOR END OF FILE
65032 FE5B              CP 91
65034 2814              JR Z, XOUT: DONE BUT DO END ROUTINE
65036 7E                LD A, (HL): CHECK NEXT BYTE = 255
65037 FEFF              CP 255
65039 200A              JR NZ, FOUT: NOT 255
65041 08                EX AF, AF': BUMP COUNT
65042 3C                INC A
65043 20F0              JR NZ, FLOOP: LOOP IF COUNT NOT TO 256
65045 3D                DEC A: A = 255
65046 12                LD (DE), A
65047 13                INC DE: START COUNT OVER
65048 12                LD (DE), A
65049 18E7              JR FULL
65051 08        FOUT    EX AF, AF'
65052 12                LD (DE), A
65053 13                INC DE
65054 18CE              JR LOOP: BACK TO MAIN LOOP
65056 08        XOUT    EX AF, AF'
65057 12                LD (DE), A
65058 13                INC DE
65059 18DA              JR OUT1: NOW EXIT
65061 13        EMPTY   INC DE
65062 3E01              LD A, 1: START COUNT
65064 08        ELOOP   EX AF, AF': SAVE IN AF'
65065 23                INC HL
65066 7C                LD A, H: END OF DFILE?
65067 FE5B              CP 91
65069 28F1              JR Z, XOUT
65071 7E                LD A, (HL): NEXT BYTE = 0?
65072 FE00              CP 0
65074 20E7              JR NZ, FOUT
65076 08                EX AF, AF': UPDATE COUNT
65077 3C                INC A
65078 20F0              JR NZ, ELOOP: CONTINUE IF NOT 256
65080 3D                DEC A: A = 255
65081 12                LD (DE), A
65082 13                INC DE
65083 3C                INC A: A=0
65084 12                LD (DE), A
65085 18,E6             JR EMPTY

65087 210040 LDSCREEN   LD HL, 16384: DFILE
65090 17XXXX            LD DE, SCRSAVE
```

```
65093 1A      LOOP1    LD A, (DE)
65094 FEFF             CP 255
65096 282E             JR Z, FILL
65098 FE00             CP 0
65100 2820             JR Z, MISS
65102 77               LD (HL), A
65103 23               INC HL
65104 13      RET0     INC DE
65105 7C               LD A, H: END OF SCREEN?
65106 FE58             CP 88
65108 38EF             JR C, LOOP1
65110 1A      LOOP2    LD A, (DE)
65111 FE00             CP 0: ATTR = 0?
65113 2809             JR Z, MISS1
65115 77               LD A, (HL)
65116 23               INC HL
65117 13      RET1     INC DE
65118 7C               LD A, H: END OF ATTR?
65119 FE5B             CP 91
65121 38F3             JR C, LOOP2
65123 C9               RET
65124 13      MISS1    INC DE
65125 1A               LD A, (DE)
65126 D5               PUSH DE
65127 1600             LD D, 0
65129 5F               LD E, A
65130 19               ADD HL, DE
65131 D1               POP DE
65132 18EF             JR RET1
65134 13      MISS     INC DE
65135 1A               LD A, (DE)
65136 D5               PUSH DE
65137 1600             LD DE, 0
65139 5F               LD E, A
65140 19               ADD HL, DE
65141 D1               POP DE
65142 18E5             JR RET1
65144 13      FILL     INC DE
65145 1A               LD A, (DE)
65146 47               LD B, A
65147 36FF    BACK     LD (HL), 255
65149 23               INC HL
65150 10FB             DJNZ BACK
65152 18DB             JR RET1
```

Please note the 2nd line of each routine has an XXXX in the code. This means that you must poke these value with the address of where you want the compressed code saved or where you have it saved when you expand it back into the DFile. Save the code as "scro.bin", 65000, 154. Those of you with disks will find it under that name for screen object code.

You can make saveral variations on these programs. If you don't have to save the attributes, all the 91's can be replaced with

the end of the pixels with an 88. Address 65110 must be poked with RET (201). If you only want to save the top 2/3rds change the 91's to 80, if only the top 1/3rd change them to 72. To start the save at the center third, start the routine with HL set to 18432 (4800 Hex). To compress only the last third of the screen load HL with 20480 (5000 Hex). Each one of these partial saves of course is shorter than a full screen save.

There is one inconsistency with the LDSCREEN program in that it is looking for a repeat attribute at address 65112. The compress routine doesn't do this so leaving the repeat address at 0--black paper and ink, an unlikely attribute, will mot crash it unless you use a 0 attribute.

The student will observe that these routines only reduce screen space for consecutive 255's and 0's. Now your screen may have a lot of consecutive pixel bytes other then these two in which case you should change the routine FULL and FILL to condense these bytes and expand them rather than 255. If you have another byte other than 0 you can change the MISS and EMPTY routines as well to accommodate that byte. And of course you need not stop at just 2 bytes but may add as many as you like by expanding the routines. Just remember what you compress in one must be expanded in the 2nd. There of course is one problem with there routines in that they don't let you design your own screen and you still need your favorite screen designer for that purpose.

The student will also realize that these routines can be modified for compressing not only a normal screen but also a high resolution graphic screen including all the attributes of DFILE2 by merely changing the address HL starts at.

There of course are other ways of looking at pixels other than their consecutive order in the DFile which could be written into a routine that can be used to compress a screen. This book is to give you ideas to write your own routines, not an exhaustive all-ready to use everything book.

If time is of the essence in your porgram, it still is faster loading a compressed screen already in memory than to load it directly full blown from a disk. How much can you save? A full detailed map of Europe, using literally dozens of unique graphics characters but never using a UDG function, covering 4 screens with overlap can be saved in 12800 bytes total space. Only one precaution. You can scroll a screen once it is expanded back out to the DFILE but you can't copy the edge of a compressed screen from its compressed state to the screen as the number of characters changes for each line or row. What you can do is fully expand a compressed screen somewhere else in memory and load characters from there to the screen as you scroll the DFILE screen.

A RELOCATABLE PRINT DRIVER

How can a program be relocatable and still have calls in it? Very simply. By having the program do a once-through routine that adjusts all calls to whatever starting address one loaded the program to.

But before you look at that code enter the following two statements in direct mode:

```
POKE 64000,201
PRINT USR 64000
```

You should get on the screen 64000. The PRINT USR function prints the final value of the BC register pair to the screen as it returns from machine code. Since all we did in machine code was tell it to return, this gives us an insight as to how the USR function works in that BC is loaded with the argument of USR, namely 64000. In other words, if for some reason we want to use that value of BC it is already loaded and we don't have to reset BC. That is the logic used in the following routine to relocate the code permanently.

This print driver is for an Aerco Printer Interface and it isn't going to do you any good unless you happen to have that print driver, unless you can modify it to work with your print driver.

```
64000 1806      ON          JR INSTALL
64002 1831      OFF         JR RESTORE
64004 3F                    DEFB LINE LENGTH-1
64005 0A                    DEFB LINE FEED CODE
64006 0000                  DEFB: SPACES USED BY PROGRAM
64008 21B302  INSTALL       LD HL, 691: START OF OFFSET TABLE
64011 09                    ADD HL, BC: NOTE THAT AT THIS POINT  WE
   HAVE NOT LOADED BC WITH ANY  SPECIFIC  VALUE.  HOWEVER  BC  =
   64000 IN THIS CASE SINCE ITS THE ARGUMENT OF USR.
64012 5E    LOOP            LD E, (HL):PUT OFFSET VALUE IN DE
64013 23                    INC HL
64014 56                    LD D, (HL)
64015 23                    INC HL
64016 7A                    LD A, D: IF DE=0 WE ARE DONE
64017 B3                    OR E
64018 2811                  JR Z, SETCHAN
64020 EB                    EX DE, HL: DE= POINTER/HL= OFFSET
64021 09                    ADD HL, BC: HL= ADDRESS TO CHANGE
64022 D5                    PUSH DE: SAVE OFFSET POINTER
64023 E5                    PUSH HL: SAVE CHANGE ADDRESS
64024 5E                    LD E (HL): LOAD DE WITH PRESENT OFFSET
                              FOUND IN (HL)
64025 23                    INC HL
64026 56                    LD D, (HL)
64027 EB                    EX DE, HL: PRESET  CHANGE  ADDR  OFFSET
                              TO HL
```

```
64028 09                    ADD HL, BC: NEW ADDRESS
64029 EB                    EX DE, HL: MOVE NEW ADDR TO DE
64030 E1                    POP HL:GET CHANGE ADDRESS BACK
64031 73                    LD (HL), E: CHANGE IT WITH DE VALUE
64032 23                    INC HL
64033 72                    LD (HL), D
64034 E1                    POP HL: GET OFFSET POINTER BACK
64035 18E7                  JR LOOP
64037 3EC3  SETCHAN         LD A, 195: CHANGE 64008-64010
64039 320800*               LD (64008),A: TO JP OFFSET + 78 THIS
   IS THE FIRST ADDRESS     THAT GETS CHANGED. ALL CHANGES ARE
   MARKED WITH A *
64042 213000*               LD HL, 64048: 2ND ADDR CHANGED.
64045 220900*               LD (64009), HL
64048 114E00*               LD DE, 64078: SET LPRINT CHANNEL
64051 180E                  JR CHANGE
64053 3A9233    RESTORE     LD A, (13202): RESTORE P CHANNEL
64056 A7                    AND A: ZERO?
64057 2805                  JR Z, NOTTV
64059 110005                LD DE 1280: SENDTV
64062 1803                  JR CHANGE
64064 11F409    NOTTV       LD DE, 2548
64067 2A4F5C    CHANGE      LD HL, (23631): CHANNELS
64070 010F00                LD BC, 15: 3RD CHANNEL
64073 09                    ADD HL, BC
64074 73                    LD (HL), E
64075 23                    INC HL
64076 72                    LD (HL), D
64077 C9                    RET

64078 FDCB0246 LPRINT       BIT 0,(IY+2) TVFLAG
64082 2807                  JR Z, UPPER
64084 FE0D                  CP 13  ENTER?
64086 282E                  JR Z, ENTER
64088 C3D000*               JP 64208: SENDCHAR
64091 47        UPPER       LD B, A
64092 3A0600*               LD A, (64006):COUNT
64095 FEFF                  CP 255: LINE FULL LEFT
64097 2808                  JR Z, SENDSP
64099 A7                    AND A: LINE FULL RIGHT
64100 281D                  JR Z, NEWLN
64102 3D                    DEC A: UPDATE COUNT
64103 320600*               LD (64006), A
64106 C9                    RET

64107 3E20     SENDSP       LD A, 32: SPACE
64109 C5                    PUSH BC
64110 CD3001*               CALL 64304: PRINTCH
64113 CD1C01*               CALL 64284: CHECKWD
64116 C1                    POP BC
64117 CD2801*               CALL 64296: UPDATEPOSN
64120 3A0700*               LD A, (64007) WIDTHLEFT
64123 B8                    CP B
64124 20ED                  JR NZ, SENDSP
```

```
64126 AF                    XOR A: A=0: RESET COUNT
64127 320600*               LD (64006), A
64130 C9                    RET

64131 78     NEWLN          LD A, B
64132 FE0D                  CP 13: ENTER?
64134 2865    ENTER         JR Z, ENTER2
64136 FE0C                  CP 12: DELETE?
64138 2004                  JR NZ, SKIPDEL
64140 3E7A                  LD A, 122: DELETE CODE
64142 184C                  JR PRTOKEN
64144 FE06   SKIPDEL        CP 6: PRINT COMMA?
64146 286A                  JR Z, COMMA
64148 FE10                  CP 16: INK CODE?
64150 D8                    RET C: UNPRINTABLE IF <16
64151 FE16                  CP 22: AT?
64153 2810                  JR Z, AT
64155 FE17                  CP 23: TAB?
64157 2808                  JR Z, TAB
64159 300E                  JR NC, CONTINUE
64161 3E01                  LD A, 1: UPDATE COUNTER
64163 320600*   RET1        LD (64006), A
64166 C9                    RET

64167 3EFF       TAB        LD A, 255
64169 18F8                  JR RET1
64171 3E02       AT         LD A, 2
64173 18F4                  JR RET1
64175 FE20     CONTINUE     CP 32
64177 D8                    RET C: RETURN IF <32 UNPRINTABLE
64178 FE7C                  CP 124: STICK?
64180 2826                  JR Z, PRTOKEN
64182 FE7E                  CP 126:FREE?
64184 2822                  JR Z, PRTOKEN
64186 FE7B                  CP 123: <123?
64188 380A                  JR C, GRAPHIC?
64190 FE80                  CP 128: GRAPHIC?
64192 3006                  JR NC, GRAPHIC?
64194 FDCB0166              BIT 4, (IY+1): FLAGS COMMAND MODE?
64198 2814                  JR Z, PRTOKEN
64200 FE80       GRAPHIC?   CP 128
64202 3804                  JR C, SENDCH
64204 FEA5                  CP 165: UDG?
64206 300A                  JR NC, UDG
64208 CD3001*   SENDCH      CALL PRINTIT
64211 CD1C01*               CALL CHECKWD
64214 CD2801*               CALL UPDPOS
64217 C9                    RET

64218 D6A5       UDG        SUB 165: OFFSET FOR UDG TOKEN
64220 F5      PRTOKEN       PUSH AF: SAVE CODE
64221 3A9233                LD A, (13202)
64224 A7                    AND A: ZERO?
64225 2805                  JR Z, KEYBOARD
```

```
64227 F1                POP AF: GET TOKEN BACK
64228 CD4507            CALL 1861: GET TOKEN
64231 C9                RET
64232 F1        KEYBOARD POP AF
64233 CD100C            CALL 3088: KEYBOARD INPUT
64236 C9                RET

64237 3E0D      NEWLN   LD A, 13: ENTER
64239 CD3001*           CALL PRINTIT: SEND ENTER
64242 3A0500*           LD A, (64007): UPD COUNT
64245 A7                AND A: ZERO?
64246 C43001*           CALL NZ, PRINTIT
64249 AF                XOR A: A =0
64250 320700*           LD (64007), A
64253 C9                RET

64254 3A0400*   COMMA   LD A, (64004): WIDTH - 1
64257 57                LD D, A
64258 CB2F              SRA A: WIDTH/2
64260 5F                LD E, A
64261 3E20      SPACE   LD A, SPACE
64263 D5                PUSH DE
64264 CD3001*           CALL PRINTIT
64267 CD1C01*           CALL CHECKWD
64270 D1                POP DE
64271 CD2801*           CALL UPDPOS
64274 3A0700*           LD A, (64007)
64277 BA                CP D: LESS THAN LINE?
64278 28D5              JR Z, NEWLN
64280 BB                CP E: LESS THAN 1/2 LINE?
64281 C8                RET Z
64282 18E9              JR SPACE

64284 3A0700*   CHECKWD LD A, (64007): POSITION
64287 57                LD D, A:STORE IN D
64288 3A0400*           LD A, (64004): WIDTH
64291 BA                CP D: >WIDTH?
64292 CCED00*           CALL Z, NEWLN
64295 C9                RET

64296 3A0700*   UPDPOS  LD A, (64007)
64299 3C                INC A
64300 320700*           LD (64007), A
64303 C9                RET

64304 F5        PRINTIT PUSH AF: SAVE REGISTERS
64305 C5                PUSH BC
64306 D5                PUSH DE
64307 E5                PUSH HL
64308 110000*           LD DE, 0
64311 4F                LD C, A: SAVE CODE IN C
64312 F5        NEXTCH  PUSH AF
64313 3A9233            LD A, (13202)
64316 A7                AND A: ZERO?
```

```
64317 2806              JR Z, SKIPBK
64319 F1                POP AF
64320 CD0920            CALL 8201: BREAK?
64323 1804              JR TESTBK
64325 F1        SKIPBK  POP AF
64326 CD541F            CALL 8020: SCROLL???
64329 3802      TESTBK  JR C, SKIPERR
64331 CF                RST 8: ERROR
64332 14                INVALID COLOR
64333 C5        SKIPERR PUSH BC
64334 DB7F      CKPRINT IN A, (127): PRINTER BUSY?
64336 4F                LD C, A: STORE IN C
64337 0650              LD B, 80: COUNT FOR WAIT
64339 DB7F      DEBOUNCE IN A, (127): CHECK AGAIN
64341 B9                CP C: STILL THE SAME?
64342 20F6              JR NZ, CKPRINT
64344 10F6              DJNZ DEBOUNCE
64346 C1                POP BC
64347 CB4F              BIT 1, A: PRINTER READY?
64349 2802              JR Z, READY
64351 CF                RST 8: ERROR
64352 A2                162 PRINTER NOT ON LINE
64353 CB67      READY   BIT 4, A: BREAK KEY PRESSED?
64355 280D              JR Z, SENDIT
64357 1B                DEC DE
64358 C5                PUSH BC: SAVE BC
64359 0660              LD B, 96
64361 10FE      WAIT    DJNZ WAIT
64363 C1                POP BC: GET BC BACK
64364 7A                LD A, D: DE = 0?
64365 B3                OR E
64366 20C8              JR NZ, NEXTCH
64368 CF                RST 8: ERROR
64369 BD                189 BREAK KEY PRESSED
64370 79        SENDIT  LD A, C
64371 F3                DI
64372 D37F              OUT (127),A
64374 E1                POP HL
64375 DB7F              IN A, (127)
64377 FB                EI
64378 D1                POP DE
64379 C1                POP BC
64380 F1                POP AF
64381 C9                RET
```

LPRINT and LLIST are quite easy and standard for most printers except when printing the 2068's UDG graphics. These codes have to be sent in graphic mode which means switching the printer to that mode as is done in COPY and then switching back. The 2040 printer which is always working in the graphic mode has no difficulties. COPY is the headache as each printer has a different set of codes to handle graphics. However, most printer do it in one of two different ways so that most of the time one or the other of the following routines can be used as long as

the customizing codes are sent first. Of course they still have to be handled through a printer interface which makes a difference as well.

We start with a load graphic code routine used by both copy routines. The codes given in the routine are those that would be poked there from the Basic for a Gemini 10X printer--the one I happen to have. Both copy routines call routines from the LPRINT routine.

```
64382 E5        SETUP1     PUSH HL
64383 218401*              LD HL, 64388
64386 1820                 JR SENDCODE
64388 031B3310  DATA1      DEFB: THE FIRST IS # TO SEND FOLLOWED
64392 00000000             DEFB: BY THE CODES--SPACE FOR 10
64396 000000               DEFB
64399 E5        SETUP2     PUSH HL
64400 219501*              LD HL, 64405
64403 180F                 JR SENDCODE
64405 041B4B0001 DATA2     DEFB: AS FOR DATA1
64410 000000000000         DEFB
64416 E5        SETUP3     PUSH HL
64417 21B801*              LD HL, 64440: DATA3
64420 C5        SENDCODE   PUSH BC
64421 CDED00*              CALL NEWLN
64424 7E                   LD A,(HL): GET COUNT
64425 47                   LD B,A
64426 FE00                 CP 0: IN ZERO RETURN
64428 2807                 JR Z, RET2
64430 23        LOOP2      INC HL
64431 7E                   LD A,(HL)
64432 CD3001*              CALL PRINTCH
64435 10F9                 DJNZ LOOP2
64437 C1        RET2       POP BC
64438 E1                   POP HL
64439 C9                   RET
64440 021B40    DATA3      DEFB: AS IN DATA1
64443 00000000000000       DEFB
```

COPY ROUTINE 1 FOR MOST PRINTERS

```
64450 DDE5      COPY1      PUSH IX: SAVE IX
64452 CD7E01*              CALL SETUP1
64455 0618                 LD B, 24: 24 LINES
64457 210040               LD HL, DFILE1
64460 CDD701*   COPYLOOP1  CALL COPYROUT
64463 10FB                 DJNZ COPYLOOP1
64465 CDA001*              CALL SETUP3: CALCEL GRAPHIC PRINT
64468 DDE1                 POP IX
64470 C9                   RET

64471 C5        COPYROUT   PUSH BC: SAVE LINE COUNT
64472 CD8F01*              CALL SETUP2
64475 0620                 LD B, 32: 32 CHAR SPACES/LINE
```

```
64477 1680      NEWMASK   LD D, 128: MASK FORDFILE PIXEL
     ADDR 64480 MAY CANGE FOR YOUR PRINTER
64479 1E01                LD E,1 :PIXEL TO PRINT
64481 E5                  PUSH HL
64482 1803                JR SKIPNL
64484 CD0702*   CALLNXL   CALL NEXTLN
64487 7E        SKIPNL    LD A,(HL)
64488 A2                  AND D
64489 2808                JR Z, MOVEP1L
   ADDR 64492, 64494 AND 64500 MAY CHANGE FOR YOUR PRINTER
64491 CB01                SLA E: SHIFT PIXEL LEFT
64493 CDC3                SET 0, E: ADD ANOTHER PIXEL TO STRING
64495 30F3                JR NC, CALLNXL: IF NO OVERFLOW
64497 1804                JR SENDPIXL
64499 CB23      MOVEP1L   SLA E: MOVE PIXEL 1 LEFT
64501 30ED                JR NC, CALLNXL
64503 7B        SENDPIXL  LD A, E
64504 CD3001*             CALL PRINTCH
64507 E1                  POP HL
64508 CB3A                SRL D: SHIFT MASK 1 RIGHT
64510 30DF                JR NC, CALLNXL
64512 CD1E02*             CALL NX3RD
64515 10D8                DJNZ NEWMASK
64517 C1                  POP BC
64518 C9                  RET

64519 C5        NEXTLN    PUSH BC
64520 7C                  LD A, H
64521 010001              LD BC, 256
64524 09                  ADD HL, BC
64525 E607                AND 7: SAVE 3 LOW BITS OF A
64527 FE07                CP 7: FULL?
64529 2006                JR NZ, RET4
64531 A7                  AND A: ZERO?
64532 01E007              LD BC, 2016: BYTES TO 1/3 OF SCREEN
64535 ED42                SBC HL, BC
64537 E5        RET4      PUSH HL: LD IX, HL
64538 DDE1                POP IX
64540 C1                  POP BC
64541 C9                  RET
64542 C5        NX3RD     PUSH BC
64543 7D                  LD A,L
64544 23                  INC HL
64545 FEFF                CP 255: END OF A THIRD?
64547 281C                JR Z, NEXT1/3
64549 E61F                AND 31
64551 FE1F                CP 31
64553 2014                JR NZ, RET3
64555 DDE5                PUSH IX: LD HL. IX
64557 E1                  POP HL
64558 7C                  LD A, H
64559 01E100              LD BC, 225
64562 09                  ADD HL, BC
64563 E607                AND 7
```

```
64565 FE07                CP 7: END OF A THIRD?
64567 2006                JR NZ, RET3
64569 A7                  AND A
64570 01E007              LD BC, 2016: LAST THIRD
64573 ED42                SBC HL, BC
64575 C1        RET3      POP BC
64576 C9                  RET
64577 010007  NEXT1/3     LD BC, 1792
64580 09                  ADD HL, BC
64581 18FB                JR RET3


START OF COPY 2: FOR OLIVETTE PR2300 TYPE PRINTERS

64583 151B301B2F00 DATA4  DEFB 21,27,48,27,47,0
64589 001B47313030        DEFB 0,27,71,49,48,48
64595 3B3033323B30        DEFB 59,48,51,50,60,48
64601 32341B5A            DEFB 50,52,27,90
64605 E5        SETUP5    PUSH HL
64606 214703*             LD HL, 64583
64609 C3A402*             JP SENDCODE

ENTRY POINT FOR COPY2

64612 E5        COPY2     PUSH HL: SAVE ALL REGISTERS
64613 F5                  PUSH AF
64614 C5                  PUSH BC
64615 D5                  PUSH DE
64616 CD5D02*             CALL SETUP5
64619 210040              LD HL, DFILE1
64622 CD8202*             CALL PRINTBLOCK
64625 21004B              LD HL, 18432: DFILE 2ND 3RD
64628 CD8202*             CALL PRINTBLOCK
64631 210050              LD HL, 20480: DFILE 3RD 3RD
64634 CD8202*             CALL PRINTBLOCK
64637 D1                  POP DE
64638 C1                  POP BC
64639 F1                  POP AF
64640 E1                  POP HL
64641 C9                  RET
64642 0E08   PRINTBLOCK   LD C, 8: PRINT LINES/BLOCK
64644 E5     SAVEPOINTER  PUSH HL: DFILE LINE POINTER
64645 110001              LD DE, 256: OFFSET TO NEXT PIX LINE
64648 0608                LD B, 8:PIXEL LINES/PRINT LINE
64650 E5        LOOP3     PUSH HL:PRESENT PIXEL
64651 C5                  PUSH BC: SAVE COUNTERS
64652 0620                LD B, 32: CHAR/LINE
64654 7E        NEXTPIX   LD A, (HL): GET NEXT PIXEL
64655 CDAA02*             CALL WAIT2
64658 CD3002*             CALL PRINTIT
64661 23                  INC HL
64662 10F6                DJNZ LOOP3
64664 C1                  POP BC: UPDATE OTHER COUNTERS
64665 E1                  POP HL
```

```
64666 A7                    AND A: RESET FLAGS
64667 ED5A                  ADC HL, DE: ADD OFFSET
64669 10EB                  DJNZ LOOP3
64671 E1                    POP HL: CHANGE PRINT LINES
64672 112000                LD DE, 32
64675 A7                    AND A: CLEAR FLAGS
64676 ED5A                  ADC HL, DE
64678 0D                    DEC C
64679 20DB                  JR NZ, SAVEPOINTER
64681 C9                    RET
64682 C5          WAIT2     PUSH BC
64683 0600                  LD B, 0
64685 00          LOOP4     NOP
64686 00                    NOP
64687 10FC                  DJNZ LOOP4
64689 C1                    POP BC
64690 C9                    RET
```

AND FINALLY THE ADDRESS OFFSET TABLE FOR RELOCATING THE CODE:

```
64691 2800                  DEFW 40
64693 2B00                  DEFW 43
64695 2E00                  DEFW 46
64697 3100                  DEFW 49
64699 5900                  DEFW 89
64701 5D00                  DEFW 93
64703 6800                  DEFW 104
64705 6F00                  DEFW 111
64707 7200                  DEFW 114
64709 7600                  DEFW 118
64711 7900                  DEFW 121
64713 8000                  DEFW 128
64715 A400                  DEFW 164
64717 D100                  DEFW 209
64719 D400                  DEFW 212
64721 D700                  DEFW 215
64723 F000                  DEFW 240
64725 F300                  DEFW 243
64727 F700                  DEFW 247
64729 FB00                  DEFW 251
64731 FF00                  DEFW 255
64733 0901                  DEFW 265
64735 0C01                  DEFW 268
64737 1001                  DEFW 272
64739 1301                  DEFW 275
64741 1D01                  DEFW 285
64743 2101                  DEFW 289
64745 2501                  DEFW 293
64747 2901                  DEFW 297
64749 2D01                  DEFW 301
64751 8001                  DEFW 384
64753 9101                  DEFW 401
64755 A201                  DEFW 418
64757 A601                  DEFW 422
```

```
64759 B101                DEFW 433
64761 C501                DEFW 453
64763 CD01                DEFW 461
64765 D201                DEFW 466
64767 D901                DEFW 473
64769 E501                DEFW 485
64769 F901                DEFW 505
64759 0102                DEFW 513
64761 5F02                DEFW 607
64763 6202                DEFW 610
64765 6902                DEFW 617
64767 6F02                DEFW 623
64769 7502                DEFW 629
64771 7B02                DEFW 635
64773 9002                DEFW 656
64775 9302                DEFW 659
END OF PROGRAM
```

EASY PRINT MESSAGE ROUTINE

We pointed out in the Introduction To 2068 Code the calling of several ROM routines that could be useful in your programs. One of these was the SCROLL THE SCREEN ONE LINE at 2361. Another simple but useful routine is PRINT A MESSAGE at 1855.

This routine uses RST 16 so before entering your machine code you must initialize the screen with a CLS and a PRINT AT 0,0;. Even if you never want to print at that address, it does no harm in messing up your screens. I have seen too many print routines using one-after-the-other sequences of LD A, Code # followed by RST 16. This routine does away with all those wasteful LD A and RST 16's. BUT you must follow the correct setup of registers and the messages before calling 1855. Even this is quite simple:

```
LD A, Message # - 1  (message 1 = 0)
LD DE, ADDR of Message Block -1
CALL 1855
RET if you like or continue with something else.
```

The protocol for the message block is that the first byte must be greater than 128. An artificial 128 or even using the 201 code of a RET statement from the preceding routine is satisfactory.

The 2nd requirement of the message block is that the last character of each messsage has its code 128 higher than normal. Suppose your message ends with a period (Code 46). Merely insert 174 instead.

The last character of message 1 signals the computer that it's at the end of that message and message 2 starts at the next byte, etc. Because of this arrangement, you obviously can't

encode tokens, standard graphics or UDG graphics inside your messages as they would automatically signal the end of a message as they are numbers over 127. They print as a "?" if unprintable or as a character after subtracting 128 from the code.

Now a bonus. You can even use all the print codes (Codes 16 to 23) like TAB, AT, INK, OVER, etc., at the start or in your messages to control your printing using the typical BASIC arguments to control them. The AT, for example, can be encoded with 22,5,5 to print the messsage at 5,5. OVER and INVERSE use the 1 and 0 as usual with INK and PAPER numbers from 1 to 8. Don't use the print codes for the argument numbers but the absolute values as in the example. If you go into printing one message over the top of the previous one, make sure that you enter enough blanks at the end to completely erase longer messages or you will get the tail of the longer message attached to the end of a shorter one just as you do in Basic.

CHAPTER 3. ACTIVATING AERCO BANKS FOR BASIC

This article first appeared in SMUG BYTES in April of 1987 but it is of so much importance that it must be included here just in case you missed it and do have Aerco disk drives. Even without spending the extra $50 for the extra 3 banks of memory, at least you can use Basic in the dock bank. It was written for nonmachine code people and is quite user friendly.

Here is the program those of us with Aerco Disk drives with extra banks of memory have been waiting for. Although we had the code for putting Basic into the dock bank since June (See FD-68 vol 1.2) albeit with an error, we were missing the necessary bank numbers for the other banks. Many thanks to David Hill for the 4 drive program modifications. I just put it all together.

The two port numbers to remember are 60639 for a 2 drive system and 62687 for the 4 drive system. The bank numbers for the 2 drive system are 128, 132, 136, and 140 AND 128, 129, 136 and 137 for the 4 drive system. You just have to remember the numbers that apply to your particular system.

To install the correct codes into the other banks you will have to type in two short m/c programs and a basic program. What is listed below is for the 2 DRIVE SYSTEM.

Program 1:

```
1 REM 2 DRIVES
5 FOR A = 24047 TO 24133
10 READ B: POKE A,B: NEXT A
20 DATA 243,62,3,211,244,33,0,0,17,0,128,1,0,64,237,176,62,3,
211,244,62,132,1,223,236,237,121,33,0,128,17,0,0,1,0,237,176,
62,136,1,223,236,237,121,33,0,128,17,0,0,1,0,64,237,176,62,140,
1,223,236,237,121,33,0,128,17,0,0,1,0,64,237,176,62,128,1,223,
236,237,121,62,1,211,244,251,201
```

Check the data line carefully and run the program then save this code to disk with MOVE "banks.bin",24047,87.

Program 2:

```
1 REM BASIC INSTALLER:
5 FOR A = 24320 TO 24516
10 READ B: POKE A,B: NEXT A
20 DATA 243,219,244,8,58,158,95,6,0,184,40,62,8,6,1,184,40,28,
33,0,92,17,0,128,1,203,0,237,176,62,1,211,244,33,0,91,17,0,92,1
30 DATA 203,0,237,176,251,201,33,0,92,17,0,91,1,203,0,237,176,
62,241,211,244,33,0,128,17,0,92,1,203,0,237,176,251,201,62,1,211
```

```
40 DATA 244,50,158,95,33,0,92,17,0,91,1,203,0,237,176,62,241,
211,244,33,204,128,54,128,35,54,13,35,54,128,33,89,92,17,75,92,1
50 DATA 83,92,62,204,54,205,18,2,35,19,3,62,128,119,18,2,24,
142,33,0,91,17,0,128,1,203,0,237,176,205,2,10,33,0,128,17,0,91,1
60 DATA 203,0,237,176,201,0,1,241,17,0,0,33,0,0,1,0,0,58,159,
95,211,244,126,8,58,160,95,211,244,8,18,19,35,11,120,177,32,235
70 DATA 58,160,95,211,244,201
```

Again, check the data lines carefully, run the program and save the code to disk with MOVE "dockbas.bin",24320,197.

Program 3:

```
1 REM 2 DRIVES
2 REM FOR 4 DRIVE SYSTEMS CHANGE POKE 60639 TO POKE 62687 USING
128, 129, 136 AND 137 FOR BANK NUMBERS. ALSO POKE 24071, 24089,
24107 AND 24125 WITH 244. THEN POKE 24068,129: POKE 24086,136:
POKE 24104,137
5 CAT "BANKS.BIN",
10 BORDER 1: PAPER 1: INK 5: CLS
15 OUT 244,1: OUT 60639,128: PRINT "NOW TRANSFERING OPERATING
SYSTEM TO SHADOW DOCK BANKS"''
20 RANDOMIZE USR 24320
25 FOR B = 128 TO 140 STEP 4:REM CHANGE FOR 4 DRIVE SYSTEM
30 OUT 60639,B:PRINT "OUT 60639, ";B:OUT 244,1:POKE 37,B: NEXT
B:REM INSTALL BANK NUMBERS
35 OUT 244,1
40 PRINT "ALL BANKS NOW HAVE AERCO DISK DRIVE INTERFACE
OPERATING SYSTEM WITH ID AT BYTE 37"
42 PRINT
45 CAT "DOCKBAS.BIN",
50 INPUT "WHICH BANKS DO YOU WISH BASIC SYSTEM INSTALLED? (0-3)
0=DOCK ";A
55 IF A <0 OR A >3 THEN GO TO 50
60 LET A = A*4: REM FOR 4 DRIVE SYSTEM USE--IF A >2 THEN LET A
= A + 6
65 LET A = A + 128
70 OUT 60639,A: RANDOMIZE USR 24320: OUT 244,1
75 PRINT "BASIC SYSTEM IN BANK ";A
80 INPUT "INSTALL BASIC SYSTEM IN ANOTHER BANK? (Y/N) ";Y$
85 IF Y$ = "Y" OR Y$ = "y" THEN POKE 24478,0: GO TO 50
90 CLS: PRINT TAB 8; FLASH 1; "READ CAREFULLY"; FLASH 0
100 PRINT ''"YOU ARE STILL IN THE HOME BANK BUT ARE USING CHUNK
0 OF THE LAST BANK YOU ACTIVATED WITH BASIC."
105 PRINT '"TO ACTIVATE THIS BANK MERELY DO:"; FLASH 1;
"RANDOMIE USR 24320: OUT 244,241"; FLASH 0
110 PRINT '"TO ACTIVATE BASIC IN DIFFERENT BANK DO: "; FLASH 1;
"OUT 24320,#: "; FLASH 0; "WHERE # IS ONE OF THE BANK NUMBERS
GIVEN. THEN FOLLOW WITH: RANDOMIZE USR 24320: OUT 244,241 ALL ON
THE SAME LINE."
120 PRINT "TO GET BACK TO HOME BANK DO: RANDOMIZE USR 24320 AND
OUT 244,1"
130 PRINT ''TAB 4; FLASH 1; "PRESS ANY KEY FOR MORE"; FLASH
0:PAUSE 0
```

```
140 CLS: PRINT "ANY BASIC PROGRAM CAN BE LOADED DIRECTLY INTO AN
ACTIVATED BANK BUT ""COPY"" COMMANDS MUST BE CHANGED TO
RANDOMIZE USR 24452"
150 PRINT "BINARY CODE PORTIONS OF PROGRAMS MUST BE FIRST LOADED
INTO THE HOME BANK AND TRANSFERRED TO THE DESIRED BANK WITH THE
CODE AT 24479 AS LISTED IN THIS PROGRAM. ONCE LOADED, A SAVE TO
THE DISK THEN ALLOWS DIRECT LOADING FROM BASIC USING THE NEW BIN
SAVE."
160 PRINT '"NEVER USE THE ""NEW"" COMMAND TO ERASE A PROGRAM AS
THE COMPLETE INSTALLATION ALSO IS DELETED. USE DELETE #,#
INSTEAD."
170 PRINT 'TAB 4; FLASH 1; "PRESS ANY KEY FOR MORE"; FLASH 0:
PAUSE 0
180 CLS: PRINT "THIS PROGRAM USES CODE AT 24047 TO 24135 AND
24320 TO 24516. THE BASIC PROGRAM MAY NOW BE ERASED. YOU MAY
WISH TO USE THE LINES AT 9000 FOR YOUR PROGRAM SETUPS AFTER
WHICH THEY TOO CAN GO."
190 PRINT ''''"TAB 7;"HAPPY COMPUTERING"
200 STOP
9000 REM XFER BIN
9005 INPUT "WHICH BANK? 128, 132, 136, 140? ";A: REM 128, 129,
136, 137 FOR 4 DRIVES
9010 POKE 60639,A: REM 62687 FOR 4 DRIVE
9015 INPUT "ENTER FILE NAME TO BE TRANSFERRED ";A$: LET B$ = A$
9020 LET A = (A - 128)/4 : REM "A = A - 128 : IF A > 2 THEN LET
A = A - 4" FOR 4 DRIVE SYSTEM
9025 IF LEN B$ > (10 - A) THEN LET B$ = B$( TO 10 - A)
9030 FOR X = 1 TO A: LET B$ = B$ + "*": NEXT X: REM NAME IS
SUFFIXED WITH 1 TO 4 * TO SPECIFY WHICH BANK
9035 LET A$ = A$ + ".BIN"","; CAT "A$",
9040 INPUT "STARTING ADDRESS ";A: LET S = A: GO SUB 9100
9045 INPUT "LENGTH OF CODE (MAX 32768) ";B: IF B > 32768 THEN
PRINT "CODE TOO LONG FOR BANK": STOP
9050 LET S = B: GO SUB 9110
9055 RANDOMIZE USR 24481
9060 LET B$ = B$ + ".BIN"","+ STR$ A + "," + STR$ B: MOVE "B$",
9065 OUT 244,1: STOP
9100 GO SUB 9200: POKE 24485,LO: POKE 24486,HI: POKE 24482,LO:
POKE 24483,HI: RETURN
9110 GO SUB 9200: POKE 24488,LO: POKE 24489,HI: RETURN
9200 LET HI = INT (S/256): LET LO = S - (HI*256): RETURN
```

Save this program to disk with MOVE "banks.bas", before running it.

Before running the program, let's explain what is going to happen. The first code actually takes chunks 0 and 1 and loads them in the home bank above 32768. It then activates all 4 banks in turn and writes the code to these banks in chunks 0 and 1 returning to the home bank. In the process it also writes the bank # into byte 37 of each bank. A peek of this byte will always tell you what bank is ready.

The code of the 2nd program sets up the Basic programer into

each bank you select. Certain things are necessary to run Basic in a bank. Since each program will be a different length a copy of the systems variables table must be preserved somewhere before leaving the home bank. This is done by storing it into the printer buffer for the 2040 printer. A new copy of the system variables is then set up with a program starting address of 32972 with the proper end markers inserted into the ram for the end of Basic, the end of vars and eline. The space between 32768 and 32972 stores the 204 bytes of the system variables table for that bank. Upon shifting banks the proper system variables are put into the normal systems variable area of chunk 2 which also contains the display file for the monitor/tv. Doing all the OUT 244,241's activates all but chunks 1, 2 and 3 of each bank. Never activate chunk 2 or you lose the display file and the systems variable table. Never activate chunk 3 or you lose the machine stack and the exrom switching code. Because the printer buffer is used, a special routine for the extra banks is used to copy out the printer buffer to another area so it can be used for the 2040 printer from the extra banks--that is why the Randomize Usr 24452.

Also note that the machine code stack is common to all banks. This gives us a headache in Basic. We can shift banks in the middle of a program as long as we are not in a subroutine when we do it--a subroutine pushes the return address unto the stack and it may get buried or be used at the wrong time if we are not careful.

The expert will note that only the basic 32 character per line screen can be used at this point. Further modifications are necessary before you activate other screen modes.

Also note at this time that there is no transferring of Basic variables from one program to another. Each bank has its own Basic variables table.

Now for a little demo. Run the program and install Basic in as many banks as you desire. Read all the help screens. When the program stops enter in command mode the RANDOMISE USR 24320 and Out 244,241. Do a CLS and try LIST---you get a 0/0 ok in the bottom corner but no program. Now, just for fun type in the line 1 PRINT PEEK 37 and RUN it just to make sure the bank Basic works. Okay? Let's go back to the home bank. RANDOMIZE USR 24320 and OUT 244,1. Doing a LIST should get back a listing of this program. For a different bank use OUT 60639,# where # is a different bank you have set up for Basic. Once again RANDOMISE USR 24320 and OUT 244,241 followed by CLS and LIST and once again no program. Enter something different for a one line program if you like and go back to the home bank again. Note that each use of RANDOMISE USR 24320 toggles you between the home bank and the activated bank. Don't forget the OUT 244 statements or your computer will get lost. Never use NEW or all this initialization you have been doing is lost.

What you do with all 5 banks in Basic mode only you and your wildest imagination can decide. Have fun.

The actual routines are:

Operating System:

```
24047 F3          DI
24048 3E03        LD A, 3
24050 D3F4        OUT (244),A
24052 210000      LD HL, 0
24055 110000      LD DE, 32768
24058 010040      LD BC, 16384
24061 EDB0        LDIR
24063 3E03        LD A, 3
24065 D3F4        OUT (244), A
24067 3E84        LD A, 132
24069 01DFEC      LD BC, 60639
24072 ED79        OUT (C), A
24074 210080      LD HL, 32768
24077 110000      LD DE, 0
24080 010040      LD BC, 16384
24083 EDB0        LDIR
24085 3E88        LD A, 136
24087 01DFEC      LD BC, 60639
24090 ED79        OUT (244), A
24092 210080      LD HL, 32768
24095 110000      LD DE, 0
24098 010040      LD BC, 16384
24101 EDB0        LDIR
24103 3E8C        LD A, 140
24105 01DFEC      LD BC, 60639
24108 ED79        OUT (244), A
24110 210080      LD HL, 32768
24113 110000      LD DE, 0
24116 010040      LD BC, 16384
24119 EDB0        LDIR
24121 3E80        LD A, 128
24123 01DFEC      LD BC, 60639
24126 ED79        OUT (C), A
24128 3E01        LD A, 1
24130 D3F4        OUT (244), A
24132 FB          EI
24133 C9          RET
```

BASIC INTO BANK

```
24320 F3          DI :DISABLE INTERRUPTS UNTIL DONE
24321 DBF4        IN A, (244) :GET CURRENT CHUNKS ENABLED
24323 08          EX AF, AF' : SAVE IN A'
24324 3A9E5F      LD A, (24478)
24327 0600        LD B, 0 :IF 244478=0 INITIALIZE AT 24394
24329 B8          CP B
24330 283E        JR Z, INIT
```

```
24332 06          EX AF,AF'
24333 0601        LD B, 1 :IF CURRENT BANK =1 THEN BANK
24335 B8          CP B
24336 281C        JR Z, BANK
24338 21005C HOME LD HL, 23552 :SAVE SYS VARS OF BANK
24341 110080      LD DE, 32768
24344 01CB00      LD BC, 203
24347 EDB0        LDIR
24349 3E01        LD A, 1: SHIFT BANKS
24351 D3F4        OUT (244), A
24353 21005B      LD HL, 23296: LOAD HOME SYS VARS
24356 11005C      LD DE, 23552
24359 01CB00      LD BC, 203
24362 EDB0        LDIR
24364 FB          EI
24365 C9          RET
24366 21005C BANK LD HL, 23552: SAVE HOME SYS VARS
24369 11005B      LD DE, 23396
24372 01CB00      LD BC, 203
24375 EDB0        LDIR
24377 3EF1        LD A, 241: CHANGE TO BANK
24379 D3F4        OUT (244), A
24381 210080      LD HL, 32768: LOAD BANK SYS VARS
24384 11005C      LD DE, 23552
24387 01CB00      LD BC, 203
24390 EDB0        LDIR
24392 FB          EI
24393 C9          RET
24394 3E01    INIT LD A, 1: SWICH TO HOME AND SET INIT BYTE
24396 D3F4        OUT (244), A
24398 329E5F      LD (24478), A
24401 21005C      LD HL, 23552: SAVE HOME SYS VARS
24404 11005B      LD DE, 23396
24407 01CB00      LD BC, 203
24410 EDB0        LDIR
24412 3EF1        LD A, 241: CHANGE TO BANK
24414 D3F4        OUT (244), A
24416 21CC80      LD HL, 32972: ADJUST PROG, VARS & ELINE
24419 3680        LD (HL), 128 :END MARKERS
24421 23          INC HL
24422 3680        LD (HL), 128
24424 23          INC HL
24425 3680        LD (HL), 128
24427 21595C      LD HL, 23641
24430 114B5C      LD DE, 23627
24433 01535C      LD BC, 23635
24436 3ECC        LD A, 204
24438 36CD        LD (HL), 205
24440 12          LD (DE), A
24441 02          LD (BC), A
24442 23          INC HL
24443 13          INC DE
24444 03          INC BC
24445 3E80        LD A, 128
```

```
24447 77              LD (HL), A
24448 12              LD (DE), A
24449 02              LD (BC), A
24450 18BE            JR HOME
24452 21005B COPY     LD HL, PRINT BUFFER
24455 110080          LD DE, BANK STORAGE
24458 01CB00          LD BC, 203
24461 EDB0            LDIR
24463 CD020A          CALL 2562 :COPY
24466 210080          LD HL, BANK STORAGE
24469 11005B          LD DE, PRINT BUFFER
24472 01CB00          LD BC, 203
24475 EDB0            LDIR
24477 C9              RET
24478 00              NOP : INIT FLAG
24479 01  XFER CODE   SOURCE BANK CHUNKS TO BE ENABLED
24480 F1              DESTINATION BANK CHUNKS TO BE ENABLED
24481 110000          LD DE, DESTINATION ADDRESS--SEE NOTES
24484 210000          LD HL, SOURCE ADDRESS
24487 010000          LD BC, LENGTH
24490 3A9F5F LOOP     LD A, (24479): LD SOURCE BANKS CHUNKS
24493 D3F4            OUT (244), A: ENABLE SOURCE BANKS
24495 7E              LD A, (HL) :LOAD BYTE FOR XFER
24496 08              EX AF,AF':STORE IT
24497 3AA05F          LD A, (24480): GET DEST CHUNKS
24500 D3F4            OUT (244), A:ENABLE DESTINATION
24502 08              EX AF,AF' :GET BYTE BACK
24503 12              LD (DE), A :PUT IN DEST
24504 13              INC DE :INCREMENT REGISTERS TO NEXT BYTE
24505 23              INC HL
24506 0B              DEC BC :REDUCE COUNTER AND CHECK FOR ZERO
24507 78              LD A, B
24508 B1              OR C
24509 20EB            JR NZ, LOOP
24511 3AA05C          LD A, (24480):ENABLE DESTINATION BANK
24514 D3F4            OUT (244), A
24516 C9              RET
```

NOTES: This program will only work between the home bank and another bank, not two different extra banks. DE must be the destination address in the extra bank and HL the source address in the home bank--generally they are the same address. BC is the number of bytes to transfer. Use normal lo/hi notation.

The Basic of program 3, lines 9000-9200 asks for the bin name without suffix, loads it into the home bank, transfers it to the enabled extra bank and then saves a copy from the extra bank to disk with 1 to 4 asterisks in the name to indicate which bank it was saved from. In other words, it's very user friendly.

This was the end of the article as originally presented. What follows is new.

ACTIVATING DUAL SCREEN MODES FOR THE HOME BANK AND OTHER BANKS

If you followed the directions above it was quite an experience in bank switching of a different nature and, thanks to Aerco, not that difficult an exercise. And, more importantly, done without invoking the bank switching routines. All we did is do a few INs and OUTs. And, since we also saved our programs, we also kept our disk drives active. If you wanted to you could have also saved your programs to tape. By invoking the SAVE command your computer does some more OUT commands to shift to the EXROM in bank 254 but when it comes back it sets all the home chunks on. This is why you have to do an OUT 244, 1 to get the jump changes in the Dock Chunk 0 to operate the disk drives again. This sort of romp to the EXROM bank and back was also not done with the bank switching routines. It's time to get fancy.

What do we have to change in the above program to activate both Display screens in one of the other 3 video modes? We must be extremely careful as Aerco has rewritten some of the ROM. Let's do some brainstorming. I call it thinking out loud but, since I have written it down for you to read I guess it's thinking out loud on paper.

We know we have to open Display File 2. Where is this located? If we don't know, where can we look to find out? In a Memory map. Where can we find that? If you have read the Intro book you would have known that anything that you want to know is found in the Appendixes of the User Manual as we practically lived out of it in that book. To the uninitiated it's on page 254. Dfile2, short for Display File 2, starts at 24576, the start of chunk 3. While we are at it notice that in addition to Dfile1, chunk 2 also contains the printer buffer where we were storing our home system variables when in another bank and the system variables. Since we wrote all the bank enabling code into and around the system variables table using areas that weren't being used, we serendipitously find we can leave it alone and don't have to change it.

What do we have to move out of the way in chunk 3 and where does it go? The machine stack and the ram-resident code, alias the function dispatcher and the bank switching routines, move to chunk 7 and the Machine code variables, ARSBUF and Channels move up a bit so that a basic program starts at 31488. This will all be done automatically for us by calling the Change Video routine in the EXROM.

Will this hurt us? No, but since the machine stack is used by all the banks, we can no longer invoke chunk 7 in our extended banks. Our good old OUT 244, 241 will now have to be OUT 244, 113. Our program length in these banks now drops to no longer than 24365. Well, as they say, you never get anything for nothing. What happens if a program runs beyond the 57344 mark of the start of chunk 7?

That last is an interesting question. I found the answer by accident several years ago when I was trying to find out what was written in the EXROM. I accidentally gave the computer a loop to transfer code from the EXROM to the upper portion of the home ROM that went at least 200 addresses beyond the 8171 address of the last byte in the EXROM. The m/c program didn't hang up or anything but merely started reading from the home ROM when it ran out of EXROM. Therefore, at any one time there can only be one chunk 0 active, one chunk 1, one chunk 2 etc. to 7 but it makes no difference to the computer in which bank each one is. Therefore if you did something crazy like OUT 244, 161 activating just chunks 0, 5 and 7 of the dock RAM and then loaded a long Basic program, it would start loading at 26710 in the home bank and fill the rest of chunk 3 and all of 4. As soon as it hit chunk 5, it would shift to the dock bank and start putting Basic there. After chunk 5 of the dock bank was full, the computer would jump to chunk 6 of the home bank for the rest of the program. And the interesting thing is that it would run in that crazy format and put its Basic variables partly in chunk 6 and when that was full, in chunk 7 of the dock bank. That is, it would run as long as you didn't change the format. Now, some crazy guy out there is going to have a light bulb turn on and say "Ah ha, if I...." but that is the fun in owning a 2068, isn't it? The answer to our question is that if we exceed address 57343 with our program in an extended bank and forgot to set RAMTOP we would find ourselves in the home bank.

How full will chunk 7 be after the transfer? Down to roughly F700 or 63232 if we include the room for the UDG's. That leaves us nearly 6K for any routines we want to have common to all banks. If you peeked ahead to the next chapter, you will find that that is more than enough room for all the 64 column routines and is exactly where we are going to put them. Its also a great spot for the relocatable Aerco print driver routine.

But what about the bank switching routines? They are moved up and down with errors in them and won't run if not corrected. Aerco wrote a routine upon bootup that corrects them but they don't stay correct once moved. Do we need to correct them? Are they ever used? Well, if you write your own code you can avoid using them. If you plan on using somebody else's code I doubt it. If you bought somebody else's program and they use it, the program should be friendly enough to make all the corrections before invoking the bank switching routines. Now, as far as I know, a lot has been written about these bank switching routines but nobody has yet to use them.

Do the Aerco drives use the bank switching routines? No, we have Mscript which uses Dfile2 and the disk drives without the routines. As you will find out in the next chapter, none of the double screen modes is supported in Basic. Neither is the function dispatcher nor the bank switching routine supported in its high location in chunk 7 so even if we wanted to use the bank switching routines we would have to write routines for

them.

Will the bank switching routines ever be used? I doubt it without rewriting some of the home ROM and unlocking some of the routines that are presently bypassed to error codes. Ever wonder why you got those "invalid color" errors? For more on the subject read the series of articles in Time Design by Wes Brozowski called "The Mystery of the Missing 253". It is not easy reading but will give you some idea of what Timex had in mind and then changed. One of these days someone will do exactly that but so far the best I've seen is correcting obvious errors, not unlocking the extra routines.

So, other than having to be careful not to invoke chunk 7 in extra banks, and limiting both program and code to addresses under 57344, we should be okay. BUT not okay for Mscript or any other program that tries to use all of the memory for data. Not without modification to reduce those memory requirements and to make sure the code isn't located beyond 57344. I have heard they are working on a version of Profile that uses 2 or more banks at the same time. It would be quite involved and would take the rewriting of the ROM routines that write to and retrieve from the Basic variables file at the very least.

Rather than give the routine for changing video here and repeat it in modified form in the next chapter along with some of the support routines necessary to use it, I leave it up to the student to modify the routine for his/her own purposes. By itself it is almost useless. The screen is interlaced 64 column across but you are only writing to or clearing Dfile1 from Basic.

CHAPTER 4 64 AND 80 COLUMN MODES

In the Introduction to 2068 Machine Code I gave you a routine to take the bank switching routines and the machine stack and move them up to chunk 7 of the home bank and put the user defined graphics below that and move other things out of the way to open the second display file. I also talked about what it would take to write to this double width screen either with standard width characters in the 64 column mode or 6 pixel wide characters of the 80 column mode. I also pointed out that both these modes were not supported from Basic and you had to write your own routines which I said were a bit beyond the ability of the beginning student so I gave no routines to write to the screen. This omission is corrected here.

Before giving the routines, let's review exactly what happens in the dual 64 normal characters across screen. The two display files are put on your TV or monitor interlaced. Every even character position belongs to display file 1 and every odd position to display file 2. In 64 mode, its easy as each screen position is still 8 pixels across but since the width of the screen is now 64 across, each character looks squeezed to half width on the screen. Since there are still only 24 lines on the full screen, the characters look the normal height. It's the same type of screen you see when you use Mscript.

64 COLUMN SCREEN

The 64 column mode is easiest to understand so let's tackle that first and then see how we have to modify it to take care of the more complex 80-85 column screen. Why do I say 80-85 columns? If you take a typical line of 64 characters you have 512 pixels. Divide that by 6 and you get 85.3 possible characters.

We have some help as to what to do as the 2068 Technical Manual gives us a set of routines to start interfacing to the 64 column mode in Appendix C-1. Unfortunately that part of your manual is almost impossible to read and even with the aid of a magnifying glass one can go blind and slightly insane trying to get it to run. Should you want to compare the routines listed below with those in the manual you will find that the manual does what is called a "non-assembled code". You just can't use a hexadecimal code loader program and take the code numbers on the left and type them in as is. They advise us to assemble it at 57344 which is E000 in hex. Certain lines in the assembly have quotes after them. That means that that line will change. The address given is typically listed as 000C which is the amount of OFFSET from the start of the program. If we were assembling at E000 that would be address E00C which has to be converted to low byte first format, eg., 0C0E.

The Technical Manual gives us several routines and and INPUT and ENTRY Table which allows interfacing to Basic. The other routines are:

WRCH	Write character whose code is in DATAB or the A register. UDG and Standard graphic characters will print.
WRSTR	Write string whose code is in DATAB or the A register. UDG and Std Graphics as above.
SETCU	Set cursor--equivalent to AT and TAB as ENTER, DELETE and the CURSOR keys do not work.
CLRSC	Clear screen-clears both screens
SETAT	Set attribute--input ink color to change
SETMS	Set mask--for over and inverse. Need SETCU to reset for over.
GTCHR	Get char at address specified
GETAT	Get the attribute
GETCU	Get the address of the cursor
SETMD	Set video mode--open/close D file 2 etc.
SCRL	Scroll

All these routines have both Basic and Machine code entry points and we will demonstrate the use of both. This is a nice start in supporting a 64 column screen but if you think any of the normal Basic print-to-the-screen commands will write to both screens you are badly mistaken. One must constantly poke value from Basic and use USR calls with these programs. AND what still is not supported is LIST, COPY, INPUT, INKEY$ and SCREEN$ which will only work in their normal 1 screen mode, inputing, listing, copying and saving only to or from the first data file. There is some hope for people with Aerco Disk drives to get around some of these difficulties which we will discuss later. But first the routines.

The extra variables to be set up for Basic and machine code are:

57344	DATAB	From Basic poke code of character to be printed or the name of the string to be printed.
57351	LINE	Basic Line being printed on (0 to 23)
57352	COLUMN	Basic column for next character (0 to 63).
57361	ATTCTL	Attribute. Entire screen is the same. Enter new ink here to change screen color.
57369/70	GETCTL	GET CONTROL. Internal format line and column of next print position. Note: Reverse of of basic: line 23 is top line, 63 left colm.

57380	VIMODE	Video mode 6=64 column: 0=normal.
57384/5	SCRCTL	Scroll control. Starting line and line count.
57389	BOTLN	Bottom line +1. Results in auto scroll.
57390	SCRLCT	Scroll count. Number of lines to scroll -1.
57391/2	CHTBL	Address of CHARACTER TABLE in use.
57395	LINLEN	Linelength = 64
57396/7	CURPOS	Cursor position. Next print position.
57398/9	DFADRS	Display File Address.
57400	MASKB	Mask byte. Bit 2= Inverse on, Bit 0 = Over.
57401	ATTBYT	Current attribute. Bright and Flash will not work in dual screen mode.
57402	GTINDX	128. Subtract this from code of last character of a string.
57403/4	STRGCT	String count. Holds remaining length of string yet to be printed.

Many of the routines have error trapping already built in to prevent trouble. For example, you cannot enter dual screen mode if you don't have enough free memory remaining. Also the print-to-the-screen routines return errors by number in the BC register so that if you say PRINT USR Number you automatically get a print return when debugging routines. (This latter messes up your routine as it prints 0 on screen 1 as you return to Basic printing right over the dual screen.)

All the vital variables have default values set by the SETMD routine which you can modify to suit once you have initialized the dual screen mode. The screen will be black on white and will start printing in the upper right corner of the screen with an automatic scroll of one line as you run out of screen.

So without further ado, the routines set up to be inserted by your favorite assembler. (Use : or ; as your assembler requires). I used Zeus so told it to ORG at 57344 (E000) and DISP 57344 and thus save the code at 49152 (C000). That gave me a nice even displacement to check addresses. Then I told it to MOVE "64o.bin",49152,1302 to save to disk. Or, if you are lazy, send for a disk or tape of the routines listed in this book all recorded error free and ready for use. See the Preface for details. Then just enter CAT "64.bas", or LOAD "64" and start typing. Use capshifted break to stop. Enter List and take a look at the other routines.

DEFINITIONS

```
SCRSZ    EQU 24: SCREEN SIZE IN LINES
SCINIT   EQU #1701: SCROLL CONTROL INIT
CLINIT   EQU #1800: CLEAR SCREEN INIT
CHRSET   EQU #3C00: ROM CHAR TABLE - 256
GRPHST   EQU ((GRPST)-#100): STND GRAPHICS TABLE
CHNGVID  EQU #0E8E: ROUTINE IN EXROM
RECLEN   EQU #1720: ROUTINE IN HOME ROM
HREXPT   EQU #FF: HOME ROM EXT SELECT PORT
DKHSPT   EQU #F4: HORIZONTAL SELECT REG PORT
VARS     EQU #5C4B: SYSTEM VARIABLES
STKEND   EQU #5C65
UDG      EQU #5C7B
COORDS   EQU #5C7D
RAMTOP   EQU #5CB2
PRAMT    EQU #5CB4
VIDMOD   EQU #5CC2
PARAMS   EQU #5CC3: (23747)
DRIVES   EQU #6840
INSERTSZ EQU #7B00 - DRIVES
MCVESZ   EQU DRIVES - #6000
DES7    EQU #FFFF - MOVESZ +#1
FIX      EQU DES7 - #6000
FIXTBL   EQU #1000
```

INPUTS AND ENTRIES

```
57344 20       DATAB     DEFB 32
57345 C33DE0   WRCHRB    JP WRCH0: ENTRY WRCHAR
57348 C3ECE0   WRSTRB    JP WRSTR0: ENTRY WRITE STRING
57351 0000     LINCOL    DEFW 0: SET CURSOR POSITION
57353 C335E1   SETCUB    JP SETCB0: ENTRY SET CURSOR POSN
57356 0018     CLRCTL    DEFW #1800: CLS CONTROL,
                            :CLRCTL = START LINE NUMBER
                            :CLRCTL + 1 = LINE CT
57358 C3F3E1   CLRSCB    JP CLRSB0 :ENTRY FOR CLEAR SCREEN
57361 38       ATTCTL    DEFB 56: BLACK ON WHITE
57362 C36EE2   SETATB    JP SETAB0: ENTRY TO SET ATTRIBUTE
57365 00       MSKCTL    DEFB 0:MASK CNTR BIT 2=INV, BIT  0=OVER
57366 C398E2   SETMSB    JP SETMB0: ENTRY TO SET INV OR OVER
57369 0000     GETCTL    DEFW 0:GET CONTR FROM BASIC
57371 C3A1E2   GTCHRB    JP GTCHB0: ENTRY FOR GET CHAR
57374 C321E3   GETATB    JP GETAB0: ENTRY FOR GET ATTR
57377 C328E3   GETCUB    JP GETCB0: ENTRY FOR GET CURSOR POSN
57380 00       VIMODE    DEFB 0:VIDMODE 0=NORMAL 6=64COL
57381 C345E3   SETMDB    JP STMDB0: ENTRY TO SET VIDMODE
57384 0117     SCRCTL    DEFW #1701:SCROLL CONTROL
                            :SCRCTL = STARTING LINE NUMBER
                            :SCRCTL + 1=LINE COUNT
57386 C345E1   SCRLB     JP SCRLB0: ENTRY TO SCROLL
```

OTHER VARIABLES

```
57389 17        BOTLN     DEFB 23: BOTTOM LINE +1 LINE  AT  WHICH
        :SCREEN IS CONSIDERED FULL. AUTO SCROLL IF SCRLCT <>0
57390 00        SCRLCT    DEFB 1: SCROLL CT-1 + # OF TIMES AUTO
57391 003C      CHTBL     DEFW CHRSET: ADDR OF CHAR TB  SCROLL
57393 95E3      GRTBL     DEFW GRPHST: ADDR OF UDG TABLE
57395 40        LINLEN    DEFB 64: LINE LENGTH
57396 4118      CURPOS    DEFW #1841: CURSOR POSN INTERNAL
57398 0040      DFADRS    DEFW #4000: CURRENT DISPAY ADDR
57400 00        MASKB     DEFB 0: MASK FOR CHAR
57401 38        ATTBYT    DEFB 56: CURRENT ATTR (BLACK ON WHITE)
57402 80        GTINDX    DEFB 128: INDEX ADJ CHCODE IN GETCHAR
57403 0000      STRGCT    DEFW 0: REMAINING CT FROM WRSTR WHEN
                             :SCREEN FULL

WRITE CHARACTER

:HERE FROM BASIC TO DISPLAY CHAR IN DATAB
57405 3A00E0    WRCH0     LD A,(DATAB)
:DISPLAY CHAR IN DATAB
57408 CD8DE4    WRCHAR    CALL LDPOSN: LD BC=CURPOS HL=DFADRS
57411 FE20                CP 32: TEST VALID RANGE
57413 DAC2E0              JP C, INVPAR
57416 FEA5                CP 165
57418 D2C2E0              JP NC, INVPAR
57421 C5                  PUSH BC: SAVE CURSOR POSN
57422 FE80                CP 128: ASCII PRINTABLE?
57424 3815                JR C, WRCHR12
57426 FE90                CP 144: STD UDG CHAR?
57428 3809                JR C, WRCHR11
57430 ED4B7B5C            LD BC, (UDG): DO UDG
57434 05                  DEC B
57435 D670                SUB 112: ADJUST FOR INDEX
57437 180C                JR WRCH13
57439 ED4B31E0  WRCH11    LD BC, (GRTBL): ADDR OF GRAPHICS TABLE
57443 D660                SUB 96: ADJUST FOR INDEX INTO TABLE
57445 1804                JR WRCH13
57447 ED482FE0  WRCH12    LD BC, (CHTBL): CHARACTER TABLE
57451 EB        WRCH13    EX DE, HL
57452 2600                LD H, 0
57454 6F                  LD L, A
57455 29                  ADD HL, HL
57456 29                  ADD HL, HL
57457 29                  ADD HL, HL
57458 09                  ADD HL, BC
57459 C1                  POP BC
57460 EB                  EX DE, HL
57461 79                  LD A, C: TEST IF END OF LINE
57462 3D                  DEC A
57463 3A33E0              LD A, (LINLEN)
57466 2005                JR NZ, WRCH14
57468 3C                  INC A: START OF NEW LINE
57469 05                  DEC B: ALSO BUMP B TO NEXT LINE
57470 4F                  LD C, A
57471 1801                JR WRCH15
```

```
57473 3C        WRCH14    INC A
57474 B9        WRCH15    CP C: TEST IF START OF LINE
57475 D5                  PUSH DE: TEST IF OFF SCREEN
57476 CCC6E0              CALL Z, TVFULO
57479 D1                  POP DE
        :HERE TO PUT CHAR IN DFILE
57480 C5        WRCH2     PUSH BC: SAVE CUR POSN
57481 E5                  PUSH HL: AND DFILE ADDR
57482 3A38E0              LD A, (MASKB)
57485 06FF                LD B, 255: ASSUME OVER ON
57487 1F                  RRA: TEST FOR OVER
57488 3801                JR C, WRCH3
57490 04                  INC B: TURN OVER OFF
57491 1F        WRCH3     RRA
57492 1F                  RRA: TEST FOR INV
57493 9F                  SBC A, A
57494 4F                  LD C, A: C=255 IF INV, ELSE 0
57495 3E08                LD A, 8: COUNTER
57497 A7                  AND A: CLEAR FLAGS
57498 EB                  EX DE, HL
57499 08        WRCH5     EX AF, AF': SAVE COUNTER
57500 1A                  LD A, (DE): GET OLD PIXEL ROW
57501 A0                  AND B: ADD OVER, A=255 IF OFF
57502 AE                  XOR (HL): ADD IN NEW PIXEL
57503 A9                  XOR C: INVERT IF ON
57504 12                  LD (DE), A: PRINT IT
57505 08                  EX AF, AF': GET COUNTER BACK
57506 14                  INC D: UPDATE POINTERS-NEXT PIXEL ROW
57507 23                  INC HL
57508 3D                  DEC A
57509 20F4                JR NZ, WRCH5
57511 E1        WRCH7     POP HL
57512 C1                  POP BC
57513 0D                  DEC C: ADJUST CUR POSN
57514 79                  LD A, C
57515 CB47                BIT 0, A: EVEN OR ODD?
57517 2006                JR NZ, CCLODD
57519 7C        CCLEVN    LD A, H
57520 F620                OR 32
57522 67                  LD H, A
57523 1805                JR WCHXT
57525 7C        CCLODD    LD A, H
57526 E6DF                AND 223: RESET BIT 5
57528 67                  LD H, A
57529 23                  INC HL
57530 CD85E4    WRCHXT    CALL STPOSN
57533 0E00      GOODRET   LD C, 0: RET BC=0
57535 0600      ERRRET    LD B, 0: ENTER HERE WITH C <>0
57537 C9                  RET
57538 0E01      INVPAR    LD C, 1:RET BC=1 FOR INVALID PARAM
57540 18F9                JR ERRRET
57542 3E18      TVFULO    LD A, SCRSZ
57544 90                  SUB B: A = LINE #
57545 57                  LD D, A: SAVE IN D
```

```
57546 3A2DE0            LD A, (BOTLN): GET BOTTOM LINE
57549 BA                CP D
57550 D256E4            JP NC, UPDPOSN: OK, RET TO WRITE CHAR
                          :VIA UPDATE AND STORE POSN
57553 3A2EE0            LD A, (SCRLCT): TEST SCROLL COUNT
57556 3D                DEC A
57557 200B              JR NZ, TVFUL1: AUTORSCROLL WITH COUNT
57559 3E01              LD A, 1
57561 322EE0            LD (SCRLCT), A: RESET SCROLL CT
57564 C1                POP BC: DISCARD RETURN TO WRITE CHAR
57565 C1                POP BC
57566 0E03              LD C, 3: RET BC=3 FOR SCREEN FULL
57568 18DD              JR ERRRET
57570 322EE0  TVFUL1    LD (SCRLCT), A
57573 ED4B28E0          LD BC, (SCRCTL): GET STARTING LINE/CT
57577 C35FE1            JP SCRLO

WRITE STRING

:HERE FROM BASIC ENTRY TO WRITE CHARACTER STRING TO SCREEN
:STRING IDENTIFIER (CH.CODE) IN SYS VAR PARAMS AT 5CC3

57580 3AC35C  WRSTR0    LD A, (PARAMS): GETSTRING ID
57583 E61F              AND 31: MASK OFF UPPER BITS
57585 F640              OR 64: STRING IDENTIFIER
57587 57                LD D, A: SAVE IN D
57588 2A4B5C            LD HL, (VARS): FIND STRING
57591 7E      WRSTR1    LD A, (HL)
57592 E67F              AND 127: TEST IF END OF VARS
57594 2835              JR Z, NOSTRG: NO FIND RET BC=2
57596 BA                CP D: MATCH?
57597 2808              JR Z, WRSTR2: FOUND
57599 D5                PUSH DE: SAVE STR ID
57600 CD2017            CALL RECLEN: RET ADDR NEXT VAR IN DE
57603 EB                EX DE, HL: ADDR TO HL
57604 D1                POP DE: RESTORE STR ID
57605 18F0              JR WRSTR1
57607 23      WRSTR2    INC HL: GET LEN
57608 4E                LD C, (HL)
57609 23                INC HL
57610 46                LD B, (HL)
57611 23                INC HL
57612 78                LD A, B: TEST IF NUL
57613 B1                OR C
57614 C8                RET Z: RET IF NULL
:ENTRY FROM M/C W ADDRESS OF CHAR.CODE STRING IN HL, LEN IN BC
57615 7E      WRSTRG    LD A, (HL): GET CODE
57616 E5                PUSH HL: SAVE ADDR
57617 C5                PUSH BC: SAVE LEN
57618 CD40E0            CALL WRCHAR
57621 79                LD A, B: TEST IF GOOD
57622 B0                OR C
57623 2009              JR NZ, WRSTR1: EXIT IF INVALID
57625 C1      WRSTR3    POP BC: COUNT
```

```
57626 E1                    POP HL: ADDR
57627 23                    INC HL: NEXT CHAR
57628 0B                    DEC BC: ADJ CT
57629 78                    LD A, B: DONE?
57630 B1                    OR C
57631 20EE                  JR NZ, WRSTRG: WRITE NEXT CHAR
57633 C9                    RET: BC=0
57634 79          WRSTX1    LD A, C: SAVE ERROR
57635 FE01                  CP 1: TEST IF UNRECOGNISABLE CODE
57637 28F2                  JR Z, WRSTR3
57639 E1                    POP HL: REMAINING COUNT TO HL
57640 223BE0                LD (STRGCT), HL: STORE
57643 E1                    POP HL: ADDR
57644 0E03                  LD C, 3: RET BC=3
57646 0600        WRSTX2    LD B, 0
57648 C9                    RET
57649 0E02        NOSTRG    LD C, 2
57651 18F9                  JR WRSTX2

POSITION CONTROL

:HERE FOR BASIC ENTRY. PARAMETERS IN LINCOL
57653 ED4B07E0 SETCBO       LD BC, (LINCOL)
:HERE IF LINCOL IN BC
57657 CD3AE4      SETCUR    CALL TSTPAR
57660 CD4BE4                CALL CONVFM: CONVERT INTERNAL FORMAT
57663 CD56E4                CALL UPDPOSN: UPDATE POSN
57666 C3BDE0                JP GOODRET

SCROLL

:HERE FROM BASIC TO SCROLL SCREEN
57669 ED4B28E0 SCRLBO       LD BC, (SCRCTL): GET CONTROL INFO
:HERE WITH CONTROL IN BC: B=# OF LINES, C=STARTING LINE #
57673 78          SCROLL    LD A, B: TEST VALIDITY
57674 A7                    AND A
57675 CAC2E0                JP Z, INVPAR: ERR IF COUNT = 0
57678 81                    ADD A, C
57679 FE01                  CP 1: ERR IF B+C=1
57681 DAC2E0                JP C, INVPAR
57684 FE19                  CP 25: ERR IF B+C>24
57686 D2C2E0                JP NC, INVPAR
57689 CD5FE1                CALL SCRLO: DO IT
57692 C3BDE0                JP GOODRET: RET BC=0
57695 C5          SCRLO     PUSH BC
57696 3E18                  LD A, 24: GET STARTING LINE INT FORMAT
57698 91                    SUB C
57699 47                    LD B, A
57700 3A33E0                LD A, (LINLEN)
57703 3C                    INC A: COL 0
57704 4F                    LD C, A
57705 CD74E4                CALL LNBO
57708 C1                    POP BC: ORIG BC
57709 C5                    PUSH BC: SAVE AGAIN
```

```
57710 7D        TESTAO    LD A, L: TEST IF START OF BLOCK
57711 A7                  AND A
57712 2851                JR Z,STBLK
57714 07                  RLCA
57715 07                  RLCA
57716 07                  RLCA
57717 4F                  LD C, A: GET # OF LINES
57718 3E08                LD A, 8
57720 91                  SUB C
57721 B8                  CP B: TEST AGAINST TOTAL LINES
57722 383F                JR C, GRBLK: TOTAL >BLOCK
57724 78        INSDBLK   LD A, B
57725 0600                LD B, 0: REMAINING COUNT
57727 C5                  PUSH BC
57728 47        LOOP      LD B, A: B=LINES THIS BLOCK
57729 0E08                LD C, 8: SCAN LINES
57731 78        LOOPO     LD A, B
57732 C5                  PUSH BC
57733 0F                  RRCA
57734 0F                  RRCA
57735 0F                  RRCA: CALC # OF BYTES
57736 4F                  LD C, A
57737 0600                LD B, 0: BC=32** OF LINES
57739 EB        LOOP1     EX DE, HL
57740 21E0FF              LD HL,#FFE0: GET ADDR OF PREV LINE
57743 19                  ADD HL, DE
57744 EB                  EX DE, HL: TO DE
57745 C5                  PUSH BC: SAVE COUNT
57746 E5                  PUSH HL: SAVE ADDR
57747 EDB0                LDIR: DO A LINE
57749 E1                  POP HL
57750 C1                  POP BC
57751 7C                  LD A, H
57752 CB6F                BIT 5, A: IS IT DF1?
57754 2005                JR NZ, NXTSC
57756 F620                OR 32
57758 67                  LD H, A: DO DF2
57759 18EA                JR LOOP1
57761 E6DF      NXTSC     AND 223: BACK TO DF1
57763 67                  LD H. A
57764 24                  INC H: NEXT SCAN LINE
57765 C1                  POP BC: SCAN COUNT
57766 0D                  DEC C
57767 20DA                JR NZ, LOOPO: MORE SCANS
57769 C1                  POP BC: RESIDUAL LINE COUNT
57770 78                  LD A, B
57771 A7                  AND A
57772 2804                JR Z, SCRLXT: DONE
57774 2E00                LD L, 0: START OF NEXT BLOCK
57776 18BC                JR TESTAO
57778 C1        SCRLXT    POP BC: ORIG BC
57779 79                  LD A, C: STARTING LINE
57780 80                  ADD A, B: ADD # OF LINES MOVED
57781 3D                  DEC A: LAST LINE SCROLLED
```

```
57782 4F                    LD C, A
57783 0601                  LD B, 1: CLEAR LINE 1
57785 1852                  JR CLRSCO: CLEAR VACATED LINE AND RET
57787 4F          GRBLK     LD C, A
57788 78                    LD A, B
57789 91                    SUB C: ADJ TOT LN CT BY # LNS THIS BLK
57790 47                    LD B, A
57791 C5                    PUSH BC
57792 79                    LD A, C: LOAD # OF LN THIS BLOCK
57793 18BD                  JR LOOP
57795 05          STBLK     DEC B: DEC LINE CT
57796 C5                    PUSH BC
57797 0E08                  LD C, 8
57799 C5          STBLK0    PUSH BC: SAVE COUNT
57800 EB          STBLK1    EX DE, HL
57801 21E0F8                LD HL, #E0F8
57804 19                    ADD HL, DE
57805 EB                    EX DE, HL: ADDRS OF  PREV LINE
57806 012000                LD BC, 32
57809 E5                    PUSH HL
57810 EDB0                  LDIR: DO MOVE
57812 E1                    POP HL
57813 7C                    LD A, H
57814 CB6F                  BIT 5, A: DF2?
57816 2005                  JR NZ, STBLK2
57818 F620                  OR 32
57820 67                    LD H, A
57821 18E9                  JR STBLK1
57823 E6DF        STBLK2    AND 223: BACK TO DF1
57825 67                    LD H, A
57826 24                    INC  H: NEXT SCAN LINE
57827 C1                    POP BC
57828 0D                    DEC C
57829 20E0                  JR NZ, STBLK0
57831 C1                    POP BC: REMAINING LINE COUNT
57832 78                    LD A, B
57833 A7                    AND A
57834 28C6                  JR Z, SCRLXT
57836 1120F8                LD DE, #F820
57839 19                    ADD HL, DE
57840 C36EE1                JP TESTA0: BC=LINE CT, HL=DF ADDR

CLEAR SCREEN

:HERE FROM BASIC ENTRY TO CLEAR SCREEN
57843 ED4B0CE0 CLRSB0     LD BC, (CLRCTL): GET CONTROL INFO
:HERE IF BC=LINE COUNT AND STARTING LINE #
57847 78          CLRSCN    LD A, B
57848 A7                    AND A: TEST VALIDITY
57849 CAC2E0                JP Z, INVPAR
57852 81                    ADD A, C
57853 FE01                  CP 1
57855 DAC2E0                JP C, INVPAR
57858 FE19                  CP 25
```

```
57860 D2C2E0            JR NC, INVPAR
57863 CD0DE2            CALL CLRSC0: DO IT
57866 C3BDE0            JP GOODRET: BC=0
57869 C5        CLRSC0  PUSH  BC
57870 3E18              LD A, 24
57872 91                SUB C
57873 47                LD B, A
57874 3A33E0            LD A, (LINLEN)
57877 3C                INC A: SET TO COL 0
57878 4F                LD C, A
57879 CD74E4            CALL LNBO
57882 50                LD D, B
57883 59                LD E, C: SAVE POSN
57884 C1                POP BC: ORIG BC
57885 D5                PUSH DE
57886 7D        CLRSC1  LD A, L: GET # OF LINES THIS BLOCK
57887 07                RLCA
57888 07                RLCA
57889 07                RLCA
57890 4F                LD C, A
57891 3E08              LD A, 8: COUNT
57893 91                SUB C
57894 B8                CP B
57895 383D              JR C, CLRSC7
57897 78        CLRSC2  LD A, B: # FO LINES
57898 0600              LD B, 0: REMAINING LINES
57900 C5                PUSH BC
57901 47        CLRSC3  LD B, A
57902 0E08              LD C, 8
57904 78        CLRSC4  LD A, B
57905 C5                PUSH BC
57906 E607              AND 7
57908 0F                RRCA
57909 0F                RRCA
57910 0F                RRCA
57911 4F                LD C, A: BC=32*# OF LINES
57912 0600              LD B, 0: C=0 IF 256 FOR 8 LINES
57914 0D                DEC C
57915 C5        CLRSC5  PUSH BC
57916 E5                PUSH HL
57917 54                LD D, H
57918 5D                LD E, L
57919 3600              LD (HL), 0: CLEAR DF1
57921 13                INC DE
57922 EDB0              LDIR
57924 E1                POP HL
57925 C1                POP BC
57926 7C                LD A, H
57927 CB6F              BIT 5, A
57929 2005              JR NZ, CLRSC6
57931 F620              OR 32: DO DF2
57933 67                LD H, A
57934 18EB              JR CLRSC5
57936 E6DF      CLRSC6  AND 223
```

```
57938 67                    LD H, A
57939 24                    INC H
57940 C1                    POP BC
57941 0D                    DEC C
57942 20D8                  JR NZ, CLRSC4
57944 C1                    POP BC: TOTAL LINE COUNT
57945 78                    LD A, B
57946 A7                    AND A
57947 2805                  JR Z, CLRXT
57949 2E00                  LD L, 0: HL=START OF NEXT BLOCK
57951 C31EE2                JP CLRSC1: B=REMAINING LINE COUNT
57954 C1        CLRSXT      POP BC: POSITION
57955 C356E4                JP UPDPOSN: RET VIA UPDATE & ST POSN
57958 4F        CLRSC7      LD C, A: SAVE # LINES THIS BLOCK
57959 78                    LD A, B
57960 91                    SUB C: ADJ LINE COUNT
57961 47                    LD B, A
57962 C5                    PUSH BC: REMAINING LINE COUNT
57963 79                    LD A, C: REMAINING LINE CT
57964 18BF                  JR CLRSC3

ATTRIBUTE CONTROL

:HERE FROM BASIC ENTRY TO SET ATTRIBUTE
57966 3A11E1    SETABO      LD A, (ATTCTL): GET CONTROL INFO
:HERE IF ATTR BYTE IN A
57969 E607      SETATT      AND 7: GET INK
57971 4F                    LD C, A: SAVE IN C
57972 3AC25C                LD A, (VIDMOD): TEST IF DF2 OPEN
57975 A7                    AND A
57976 CAC2E0                JP Z, INVPAR: INVALID IF DF2 NOT OPEN
57979 79                    LD A, C
57980 17                    RLA
57981 17                    RLA
57982 17                    RLA
57983 F5                    PUSH AF: SAVE ROTATED VALUE
57984 F606                  OR 6: SET HORIZ WRITE CODE
57986 47                    LD B, A: VALUE
57987 DBFF                  IN A, (HREXPT)
57989 E6C0                  AND 192: SAVE BITS 6 & 7
57991 B0                    OR B: SET VIDMODE
57992 D3FF                  OUT (HREXPT), A
57994 F640                  OR 64
57996 32C25C                LD (VIDMOD), A
57999 F1                    POP AF
58000 2F                    CPL: GET COMPLEMENTARY PAPER
58001 B1                    OR C: ADD INK
58002 3239E0                LD (ATTBYT), A: SAVE IT
58005 C3BDE0                JP GOODRET: BC=0

SET MASK
:HERE FROM BASIC ENTRY TO SET MASK
58008 3A15E0    SETMBO      LD A,(MSKCTL)
:ENTRY WITH MASK IN A
```

```
58011 3238E0    SETMSK    LD (MASKB),A
58014 C3BDE0              JP GOODRET

GET ROUTINES

:HERE FROM BASIC ENTRY TO GET CHARACTER
:(RETURNS CODE FOR CHARACTER AT POSITION IN GETCTL)
58017 ED4B19E0 GTCHBO     LD BC, (GETCTL): GET CONTROL INFO
:HERE IF CONTROL IN BC
58021 CD3AE4   GTCHAR     CALL TSTPAR: TEST PARAMETERS
58024 CD4BE4              CALL CONVFM: CONVERT TO INT FORMAT
58027 CD5BE4              CALL CALCPOS: GET DFILE ADDR
58030 ED5B2FE0            LD DE, (CHTBL): CHAR TABLE
58034 0660                LD B, 96: # OF PRINTABLE CHARS
58036 3E80                LD A, 128: SET ADJ INDEX
58038 323AE0   GTCH1      LD (GTINDX), A
58041 14                  INC D: ADJ TO START OF TABLE
58042 EB                  EX DE, HL: TO HL
58043 C5       GTCH2      PUSH BC: SAVE COUNT
58044 D5                  PUSH DE: SAVE DF ADDR
58045 E5                  PUSH HL: SAVE ADDR CHAR TABLE
58046 1A                  LD A, (DE): SCAN ROW FROM DF
58047 AE                  XOR (HL): TEST AGAINST CHAR SET
58048 2810                JR Z, GTCH3: MATCH
58050 F5                  PUSH AF
58051 3A3AE0              LD A, (GTINDX)
58054 FE90                CP 144: STANDARD GRAPHIC?
58056 2003                JR NZ, GTCH23
58058 F1                  POP AF
58059 1827                JR GTCH4: DON'T TEST INVERSE
58061 F1       GTCH23     POP AF
58062 3C                  INC A: TEST INVERSE
58063 2023                JR NZ, GTCH4: OFF
58065 3D                  DEC A: A =1 FOR INVERSE
58066 4F       GTCH3      LD C, A: C=0 FOR MATCH, -1 FOR INVERSE
58067 0607                LD B, 7
58069 14       GTCH31     INC D: NEXT SCAN IN DF
58070 23                  INC HL: NEXT CHAR BYTE
58071 1A                  LD A, (DE)
58072 AE                  XOR (HL)
58073 A9                  XOR C
58074 2018                JR NZ, GTCH4: NO MATCH
58076 10F7                DJNZ GTCH31: NEXT SCAN
:HERE IF MATCH WITH (SP)=CHAR IN CHAR SET
:(SP+2)=CHAR IN DF, (SP+5)=CHAR COUNT
58078 79                  LD A, C: A=-1 IF MATCH ON INVERSE
58079 C1                  POP BC
58080 C1                  POP BC
58081 C1                  POP BC: CLEAR STACK
58082 4F                  LD C, A: B=COUNT C=-1 IF MATCH ON INV
58083 3A3AE0              LD A, (GTINDX): ADJ CHAR CODE
58086 90                  SUB B: A=CHAR CODE
58087 FE20                CP 32: SPACE?
58089 2005                JR NZ, GTCH32
```

```
58091 0C                    INC C: INVERSE SPACE?
58092 2002                  JR NZ, GTCH32
58094 3E8F                  LD A, 143: SET CODE FOR GRAPHIC BLOCK
58096 4F        GTCH32      LD C, A
58097 0600                  LD B, 0
58099 C9                    RET: CHAR CODE IN A AND BC
58100 E1        GTCH4       POP HL: HERE WHEN NO MATCH
58101 110800                LD DE, 8: LOOK AT NEXT CHAR IN SET
58104 19                    ADD HL, DE
58105 D1                    POP DE: DFILE ADDR
58106 C1                    POP BC: CHAR COUNT
58107 10BE                  DJNZ GTCH2: LOOK AGAIN
:HERE TO TEST IF DONE
58109 EB                    EX DE, HL: DFILE ADDR TO HL
58110 3A3AE0                LD A, (GTINDX)
58113 FE80                  CP 128: STANDARD CHAR?
58115 200A                  JR NZ, GTCH5: TRY GRAPHICS
58117 ED5B31E0              LD DE, (GRTBL): STD GRAPHIC TABLE
58121 0610                  LD B, 16: # OF ENTRIES
58123 3E90                  LD A, 144: INDEX
58125 18A7                  JR GTCH1
58127 FE90      GTCH5       CP 144: STD GRAPHIC?
58129 200B                  JR NZ, GTCH6: DONE IF NOT
58131 ED5B7B5C              LD DE, (UDG): TRY UDG GRAPHIC
58135 15                    DEC D: ADJ START
58136 0615                  LD B, 21: # OF ENTRIES
58138 3EA5                  LD A, 165: INDEX
58140 1898                  JR GTCH1
58142 48        GTCH6       LD C, B: HERE WHEN NO MATCH ANYWHERE
58143 AF                    XOR A: SET BC & A=0
58144 C9                    RET

GET ATTR BYTE

:NOTE THAT IN 64 COL MODE ALL SCREEN POSNS HAVE SAME ATTR
:THERE NOT NECESSARY TO USE GETCTL POSITION PARAM
:HERE FROM BASIC ENTRY TO GET ATTR
58145 3A39E0    GETATTR     LD A, (ATTRBYTE): GET ATTR BYTE
58148 4F                    LD C, A: SAVE IN C
58149 0600                  LD B, 0
58151 C9                    RET

GET CURSOR POSITION

:HERE FROM BASIC ENTRY
58152 ED4B34E0 GETCB0       LD BC, (CURPOS): GET INTERNAL POSN
58156 CD4BE0                CALL CONVFM: CONVERT TO USER FORMAT
58159 3A33E0                LD A, (LINLEN)
58162 B9                    CP C: END OF LISTING?
58163 200B                  JR NZ, GETC1: NO
58165 0E00                  LD C, 0: START OF NEXT LINE
58167 78                    LD A, B
58168 3C                    INC A: BUMP TO NEXT LINE
58169 47                    LD B, A
```

```
58170 FE18              CP 24: OFF SCREEN?
58172 3802              JR C, GETC1: NO
58174 0617              LD B, 23: POSN ON BOTTOM LINE
58176 ED4307E0 GETC1    LD (LINCOL), BC: SAVE
58180 C9                RET: VALUE ALSO IN BC

VIDEO MODE CONTROL

:HERE FROM BASIC ENTRY WITH MODE IN VIMODE
58181 3A24E0   STMDB0   LD A, (VIMODE)
:HERE IF VIDMODE IN A
58184 A7       SETMODE  AND A
58185 284A              JR Z, SETMD1: TEST VALIDITY
58187 FE06              CP 6
58189 C2C2E0            JP NZ, INVPAR
58192 210118            LD HL, SCINIT: INITIALIZE PARAMETERS
58195 2228E0            LD (SCRCTL), HL: SCROLL CONTROL
58198 3E17              LD A, 23
58200 322DE0            LD (BOTLN), A: BOTTOM LINE
58203 3E01              LD A, 1
58205 322EE0            LD (SCRLCT), A: SCROLL COUNT
58208 3E40              LD A, 64
58210 3233E0            LD (LINLEN), A: LINE LENGTH
58213 211800            LD HL, CLINIT
58216 220CE0            LD (CLRCTL), HL: CLEAR SCREEN CONTROL
58219 AF                XOR A: SET A=0
58220 3238E0            LD (MASKB), A
58223 323BE0            LD (STRGCT), A: WRITE STRING REMAINING
58226 323CE0            LD (STRGCT+1), A: COUNT
58229 2100BC            LD HL, CHRSET: STD CHAR TABLE
58232 222FE0            LD (CHTBL), HL
58235 2195E3            LD HL, GRPHST: STD GRAPHIC SET
58238 2231E0            LD (GRTBL), HL
:INITIALIZE PARAMETERS
58241 3200E0            LD (DATAB), A: DATA BYTE
58244 3207E0            LD (LINCOL), A: COLUMN
58247 3208E0            LD (LINCOL+1) A: LINE
58250 3215E0            LD (MSKCTL), A: MASK CONTROL
58253 3219E0            LD (GETCTL), A: GET COLUMN
58256 321AE0            LD (GETCTL+1), A: GET LINE
58259 3E06              LD A, 6
58261 47       SETMD1   LD B, A
58262 3AC25C            LD A, (VIDMODE)
58265 A7                AND A: SET CURRENT MODE
58266 201D              JR NZ, SETMD2
58268 B0                OR B: CURRENT MODE 0 TEST IF NEW 0
58269 CABDE0            JP Z, GOODRET: RET BC=0
58272 21C012            LD HL, INSERTSZ
58275 114080            LD DE, MOVESZ
58278 19                ADD HL, DE: TOTAL ROOM NEEDED
58279 ED5B655C          LD DE, (STKEND)
58283 19                ADD HL, DE: ADD MEMORY IN USE
58284 DAF6E3            JP C, EXTNRM: NOT ENOUGH MEMORY
58287 ED5BB25C          LD DE, (RAMTOP)
```

```
58291 A7                    AND A
58292 ED52                  SBC HL, DE
58294 D2F6E3                JP NC, EXTNRM: NOT ENOUGH MEMORY
58297 CD21E4    SETMD2      CALL ENBLEXT: ENABLE ROM EXT
58300 78                    LD A, B: MODE TO A
58301 CDFBE3                CALL GETVAL: H/W VAL TO B, VIDMOD TO C
58304 C5                    PUSH BC: SAVE
58305 78                    LD A, B
58306 CD8EE0                CALL CHNGVID
58309 CD2EE4                CALL ENBLHOME
58312 C1                    POP BC
58313 79                    LD A, C
58314 32C25C                LD (VIDMODE), A: UPDATE VIDMODE
58317 CB77      UPDATE      BIT 6, A: UPDATE SYSVARS BASED ON MODE
58319 CABDE0                JP Z, GOODRET: DONE IF MODE 0
58322 3EC1      SXT4        LD A, 193: 64/80 COL MODE
58324 B0                    OR B: STORE ATTR BYTE BASED ON INK
58325 2F                    CPL
58326 CB38                  SRL B
58328 CB38                  SRL B
58330 CB38                  SRL B
58332 B0                    OR B
58333 3239E0                LD (ATTBYT), A: ATTRIBUTE BYTE
58336 3211E0                LD (ATTCTL), A: BASIC PARAMETER
58339 3A33E0    UPDT1       LD A, (LINLEN)
58342 3C                    INC A
58343 4F                    LD C, A: COLUMN LEN
58344 0618                  LD B, 24
58346 210000                LD HL, SCRSZ: BC= 0, HOME POSN
58349 227D5C                LD (COORDS), HL: INIT PLOT POSN
58352 CD56E4                CALL UPDPOSN: UPDATE & STORE HOME POSN
58355 C3BDE0                JP GOODRET
58358 0E02      EXTNRM      LD C, 2
58360 C3BFE0                JP ERRRET
```

INTERNAL SUBROUTINES

```
58363 4F        GETVAL      LD C, A
58364 CB47                  BIT 0, A
58366 2016                  JR NZ, TST01
58368 FE80                  CP 128
58370 281A                  JR Z, SET0
58372 CB7F                  BIT 7, A
58374 C0                    RET NZ
58375 47                    LD B, A
58376 FE02                  CP 2
58378 C8                    RET Z
58379 E6C6                  AND 198
58381 EE06                  XOR 6
58383 C0                    RET NZ
58384 3E40                  LD A, 64
58386 B1                    OR C
58387 4F                    LD C, A
58388 AF                    XOR A
```

```
58389 C9                    RET
58390 FE01      TST01       CP 1
58392 C0                    RET NZ
58393 0601                  LD B, 1
58395 0E81                  LD C, 129
58397 C9                    RET
58398 AF        SET0        XOR A
58399 47                    LD B, A
58400 C9                    RET
;ROUTINE TO ENABLE CHUNK 0 IN EXROM TO ACCESS CHNGVID ROUTINE
58401 F5        ENBLEXT     PUSH AF
58402 DBFF                  IN A, (HREXPT)
58404 F680                  OR 128
58406 D3FF                  OUT (HREXPT), A
58408 3E03                  LD A, 3
58410 D3F4                  OUT (DEMSPT), A
58412 F1                    POP AF
58413 C9                    RET

58414 F5        ENBLHOME    PUSH AF
58415 AF                    XOR A: SET A=0
58416 D3F4                  OUT (DEMSPT), A
58418 DBFF                  IN A, (HREXPT)
58420 E63F                  AND 63
58422 D3FF                  OUT (HREXPT), A
58424 F1                    POP AF
58425 C9                    RET

TEST PARAMETER VALIDITY

58426 3E17      TSTPAR      LD A, 23
58428 B8                    CP B
58429 3004                  JR NC, TSTPR1
58431 F1        PARERR      POP AF
58432 C3C2E0                JP INVPAR
58435 3A33E0    TSTPR1      LD A, (LINLEN)
58438 3C                    DEC A
58439 B9                    CP C: TEST COL >63
58440 38F5                  JR C, PARERR
58442 C9                    RET

CONVERT FORMAT

;CONVERT LINE/COL FROM USER TO INTERNAL FORMAT AND BACK
;USER FORMAT LINES 0-23; COL 0-63
;INTERNAL FORMAT LINES 24-1 COL 65-2 (COL 1 = END OF LINE)
58443 3A33E0    CONVFM      LD A, (LINLEN)
58446 3C                    INC A
58447 91                    SUB C
58448 4F                    LD C, A
58449 3E18                  LD A, 24
58451 90                    SUB B
58452 47                    LD B, A
58453 C9                    RET
```

UPDATE POSITION

```
:ENTER HERE WITH BC=LINE/COL IN INTERNAL FORMAT
58454 CD5BE4   UPDPOSN   CALL CALPOSN: CALCULATE POSITION
58457 182A               JR STPOSN: STORE UPDATE AND RET
```

CALCULATE POSITON

```
:ROUTINE TO CALCULATE POSITION IN DISPLAY FILE
:RETURNS ADDR IN HL: PRESERVES LINE/COL IN BC
58459 CD74E4   CALCPOS   CALL LNBO: GET DISPLAY FILE ADDR
58462 3A33E0             LD A, (LINLEN): TEST WHICH DFILE
58465 3C                 INC A
58466 91                 SUB C
58467 CB47               BIT 0, A
58469 2806               JR Z, CALCP1
58471 F5                 PUSH AF
58472 3E20               LD A, 32
58474 84                 OR H: IN DF2
58475 67                 LD H, A
58476 F1                 POP AF
58477 A7       CALCP1    AND A
58478 1F                 RRA: COL/2 IS POSN IN DFILE
58479 5F                 LD E, A
58480 1600               LD D, 0
58482 19                 ADD HL, DE
58483 C9                 RET
:GET DISPLAY FILE ADDR FOR START OF LINE IN B
58484 3E18     LNBO      LD A, SCRSZ
58486 90                 SUB B
58487 57                 LD D, A
58488 0F                 RRCA
58489 0F                 RRCA
58490 0F                 RRCA
58491 E6E0               AND 224
58493 6F                 LD L, A
58494 7A                 LD A, D
58495 E618               AND 24
58497 F640               OR 64
58499 67                 LD H, A
58500 C9                 RET
```

STORE CURSOR POSITION

```
58501 ED4334E0 STPOSN    LD (CURPOS), A
58505 2A36E0             LD (DFADRS), HL
58508 C9                 RET
```

LOAD CURSOR POSITION

```
58509 ED4B34E0 LDPOSN    LD BC, (CURPOS)
58513 2A36E0             LD HL, (DFADRS)
58516 C9                 RET
```

GRAPHICS CHARACTER SET

```
58517 00000000 GRPST     DEFB 0,0,0,0,0,0,0,0: CODE 128
58521 00000000
58525 0F0F0F0F00000000   DEFB 15,15,15,15,0,0,0,0: CODE 129
58533 F0F0F0F000000000   DEFB 240,240,240,240,0,0,0,0: CODE 130
58541 FFFFFFFF00000000   DEFB 255,255,255,255,0,0,0,0: CODE 131
58549 000000000F0F0F0F   DEFB 0,0,0,0,15,15,15,15: CODE 132
58557 0F0F0F0F0F0F0F0F   DEFB 15,15,15,15,15,15,15,15: CODE 133
58565 F0F0F0F00F0F0F0F   DEFB 240,240,240,240,15,15,15,15: 134
58573 FFFFFFFF0F0F0F0F   DEFB 255,255,255,255,15,15,15,15: 135
58581 00000000F0F0F0F0   DEFB 0,0,0,0,240,240,240,240: CODE 136
58589 0F0F0F0FF0F0F0F0   DEFB 15,15,15,15,240,240,240,240: 137
58597 F0F0F0F0F0F0F0F0   DEFB 240,240,240,240,240,240,240,240
58605 FFFFFFFFF0F0F0F0   DEFB 255,255,255,255,240,240,240,240
58613 00000000FFFFFFFF   DEFB 0,0,0,0,255,255,255,255: CODE 140
58621 0F0F0F0FFFFFFFFF   DEFB 15,15,15,15,255,255,255,255: 141
58629 F0F0F0F0FFFFFFFF   DEFB 240,240,240,240,255,255,255,255
58637 FFFFFFFFFFFFFFFF   DEFB 255,255,255,255,255,255,255,255

58644 LAST BYTE USED
```

Now that we have the code all we need is some Basic to enter and run it. This is just a demo type program and not to be interpreted as a final work of art or computering expertise.

```
    5 CLEAR 57343: CAT "64o.BIN", 57344,1302
   10 POKE 57380,6: RANDOMIZE USR 57381: REM SETMD
   20 FOR K = 1 TO 2040
   30 POKE 57344, PEEK 23560
   40 RANDOMIZE USR 57345: REM WRCH
   50 PAUSE 0
   60 NEXT K
   70 RANDOMIZE USR 57386: REM SCROLL
   80 GOTO 20
  100 RANDOMIZE USR 57358: REM CLRSC
  120 STOP
  130 LET A$ = "When in the course of human events it becomes
necessary..."
  135 POKE 23747, CODE "A"
  140 RANDOMIZE USR 57348: REM WRSTR
  145 STOP
  150 REM INVERSE
  155 POKE 57365,4: RANDOMIZE USR 57362: RANDOMIZE USR 57358:
GOTO 20
  157 REM TO TURN INVERSE OFF POKE 57365,0 IN LINE 155 AND
REPEAT
  499 STOP
  500 POKE 57380,0: RANDOMIZE USR 57381: REM BACK TO SINGLE
SCREEN
```

Save this program as "64". Running it will automatically set up the 64 column mode and put you into a type-to-the-screen loop which demonstrates the basic use of WRCH routine. It's a little

disconcerting not to have a cursor character to show you where you are on the screen.

Please note that only characters with codes in the range 32 to 122 will be printed. That means that the return key, the cursor keys and the delete key don't work. That hardly makes for a useable routine.

Students who have the Introduction to 2068 Machine Code will note a certain similarity between the WRCH code and that of the main print loop used as a demo of a screen writing routine and given in Appendix B. First there is a sort characters routine followed by a find character address routine and finally printing the character to the screen after checking for over and inverse. WRCH does not use the "design standard graphics from their code number" routine of the ROM but instead uses a table of graphic pixels. There is a good reason for this in that the GETCH routine, which compares pixel lines in the Dfiles to that in a table would need a graphic character pixel table for comparison purposes. Of course we don't have to worry about attributes as they are all the same. What is missing in WRCH is the activating of screen printing tokens that were added to the Intro routine.

Line 70 uses the SCROLL routine from Basic to scroll both screens up one line and as such is a bit more complex than the one screen routine of the ROM.

Line 100 uses the CLRSC routine from Basic to clear both screens.

Lines 130-140 demonstrates the use of WRSTR from Basic. The student will find this routine interesting in that it searches the Basic VARS file for the correct string and then uses WRCH to write it to the screen. A study of the first part of WRSTR will give a routine to search the VARS which may be useful for other programs.

Line 155 demonstrates how to use SETMS from Basic. Just set 57365 back to zero and rerun the line to turn it back off. AND set 57365 to 1 and rerun line 155 to turn over on. Of course you have to Poke 57351 with a line # and 57352 with a column # (0 to 63 in basic form like in the AT statement) and then do a RANDOMIZE USR 57353 to run SETCU and move the cursor back so you can overprint on top of something you already have on the screen. The student will realize that by just setting a new column number and then running SETCU consitutes a TAB, unfortunately without checking to see if a move to a new line is necessary which once more is the responsibility of the programer.

Line 500 takes one back to the single screen mode.

You will also note that in listing a program it only writes on

screen 1 which may be somewhat hard to read without first doing a CLRSC. INPUT promps also only work on screen 1.

The following have been checked out and work but are not demonstrated in the program as they are more diagnostic than useful:

GETCH: Set your position as in the AT function described above and run SETCU. Then run GETCH by using PRINT Randomize Usr. The code of the character at that address is returned on screen 1.

GETCU: With a Print Randomize Usr will give the present line and column number on screen 1.

A START ON A WORD PROCESSER

The 64 column mode is the best overall screen around which to build a word processor. Characters are much easier to read than on the 80 column screen. The only reason RPM uses an 80 column screen is to make it compatible with other computers which use that format. Writing to the 64 character screen and even scrolling it window fashion as Mscript does is quite easy to do.

All word processors require a data file to be kept so that one may again reproduce that portion of the document no longer on the screen. Since the main idea of a word processor is to eventually print out a hard copy, a file of the ASCII codes and attendant TAB, ENTER, and other hard copy printing functions is essential. This requires a write-to-a-file routine and a read from a certain spot in the file routine. Since our intent is not to write a word processor we will skip the file functions and instead concentrate on things that help us with the screen.

We will first want to print a cursur to the screen to find out where we are. As we type in a character it should appear on the screen where the cursor was and the cursor should move over one space. Let's also implement the ENTER key as a "start a new line" and the cursor keys to move the cursor around the screen. INK will be used for color change but PAPER won't work. OVER and INVERSE will have their usual functions. Instead of constantly moving back and forth from Basic to code we will call our routine once and stay in it until we do a STOP token.

We will also want to make it possible to use all the characters including the standard graphic and UDG graphics as well as the CAPS LOCK. These last 3 weren't possible from the above Basic program. We have a problem initializing some of the extended mode tokens as well.

Before going into the machine code, let's do a little Basic exploratory program and see exactly what is happening in our computer. Enter the following little program:

```
 5 FOR x = 1 TO 40
10 PRINT PEEK 23560;" ";CHR$ (PEEK 23560),
15 PAUSE 0
20 NEXT x
```

Run the program without the CAPS LOCK on and with your Basic User Manual open to Appendix B to check the codes. Before you ever get your finger off the enter key you get a 13 on the screen which if you check is the right code for ENTER. Hit another key and you will see that the printing skips a full line and a half--actually a line as the (comma) at the end of the line 10 in the program would give you the half shift. You will note that the Caps Shift works and so does the Symbol shift key but you can't get the caps lock to work. You only get a 6 when entering a caps shifted 2. According to the manual that is supposed to be a PRINT comma--shift half a line. The caps shifted 9 gives us a 15 but nothing afterwards is in standard or UDG graphics. And the Caps shift and symbol shift give us a 14 which is supposed to be a slug code but again we don't get the extended token codes. The 4 cursor keys, Caps shifted 5 to 8, give us the codes listed.

Now rerun the program in the CAPS LOCK mode--do a RUN, then a caps shifted 2 for a C cursor and then enter. Everything is in CAPS and nothing we can do shifts it out to lower case characters.

The next time run the program in the Graphic mode by getting the G cursor before you hit enter. Everything is in graphics and the alphabet gives you UDG graphics from A through U but what happens when you do a V, W, X, Y, or Z? Interesting that the computer gives you tokens for where V, W, X, Y and Z should be in an extension of the UDG table.

And once more run the program in the Extended token mode by getting the E cursor before you hit enter. Also use the symbol shift key with some of the other keys. Its very very interesting but we can get every token we want if we do the proper things.

What have we learned? If the computer is in the C mode everything is in CAPS, in the E mode everything is in Extended tokens including the symbol shifted tokens, if in G mode everything is in graphic codes. Only the K tokens are unavailable as we can't keep the computer in that mode.

But why can't we shift modes as we do in a Basic line entry? Or to put it another way, how does the computer keep track of what mode it is in? In the system variable table we have a variable at 23617 called MODE. This doesn't tell us a lot but if you have the ROM Manuscript or your Technical Manual you already know that setting no bits of this byte keeps the computer thinking it is in the K or L mode. Setting bit 0, doing a POKE 23617,1, turns everything in LastK to Extended token mode. Doing a POKE

23617,2, setting bit 2, turns everything to Graphics mode.

The CAPS LOCK flag is located in bit 3 of FLAGS2 address 23658.

So far so good. We are now ready to start our machine code program. When we want to do a CAPS LOCK we know we are going to get a 6 code which we can tell our program to switch on or off bit 3 of 23658. When we want graphics, the caps shifted 9 will give us a 15 code which is the signal to toggle bit 2 of 23617, and with the caps shift-symbol shift we will get a 14 which will be our signal to toggle on the extended mode, but this one for just one entry and then off again.

We will be using the underline (code 95) for our cursor. We print it in the next print space and then back up the pointer so that we erase it as we write the character we want. The rationale for delete is interesting in that first we have to delete the cursor, then back up the pointer and print the cursor there and in the process erase the written character.

Now the routine UPDPOS doesn't operate with numbers from 63 to zero but rather with column numbers from 65 to 2, and with line numbers from 24 to 1 kept in the variable called CURPOS. Therefore, the first position on the screen has the co-ordinates of 24,65, while the right upper corner is 24,2, and the lower left corner 1,65.

Now that we have done our planning, let's write our code. We will tack it on the back of our present code starting at address 58645.

DEFINITIONS:

```
SETMODE EQU 58184
WRCHAR  EQU 57408
CURPOS  EQU 57396
UPDPOS  EQU 58454
MODE    EQU 23617
FLAGS2  EQU 23658
SETATTR EQU 51969
MASKB   EQU 57400
LASTK   EQU 23560
ORG  58645
DISP 57344
ENT
```

```
58645 3E06 INITALIZE LD A,6: SET MODE
58647 CD48E3         CALL SETMODE
58650 3E5F   NEXTCH LD A, 95: PRINT CURSOR
58652 CD40E0         CALL WRCHAR
58655 ED4B34E0 MOVECB LD BC,(CURPOS): MOVE POINTER BACK
58659 0C             INC C
58660 79             LD A,C
58661 FE42           CP 66: LEFT END OF LINE?
58663 3809           JR C, SAMELN
```

```
58665 0E02           LD C, 2: SET UP TO RIGHT 1 LINE UP
58667 05             DEC B
58668 3E18           LD A, 24: OFF TOP OF SCREEN?
58670 B8             CP B
58671 3001           JR NC, SAMELN
58673 04             DEC B: CAN'T DO AS OFF SCREEN
58674 CD56E4 SAMELN  CALL UPDPOS
58677 CD36E6 GETCH   CALL GETCHAR: SEPARATE ROUTINE-NEED SEVERAL
58680 FE0D           CP 13: ENTER?                DIFFERENT TIMES
58682 200C           JR NZ, DELETE
58684 ED4B34E0       LD BC,(CURPOS)
58688 0E41           LD C, 65
58690 05             DEC B
58691 CD1CE6         CALL MOVECUR
58694 18D7           JR MOVECB
58696 FE0C   DELETE  CP 12
58698 201B           JR NZ, LEFT
58700 ED4B34E0       LD BC,(CURPOS)
58704 0C             INC C
58705 79             LD A, C: AT LEFT OF LINE?
58706 FE42           CP 66
58708 3803           JR C,SAMEL1
58710 04             INC C: NOTE: NO CHECK FOR OFF TOP OF SCREEN
58711 0E02           LD C,2
58713 C5             PUSH BC:SAVE POSITION
58714 CD1CE6 SAMEL1  CALL MOVE CUR: ERASE CURSOR
58717 AF             XOR A: A=0
58718 CD40E0         CALL WRCHAR: ERASE CHAR
58721 C1             POP BC: GET POSITION BACK
58722 CD56E4         CALL UPDPOS
58725 18B3   JUMP    JR NEXTCH: REWRITE CURSOR
58727 FE08   LEFT    CP 8
58729 2012           JR NZ, DOWN
58731 ED4B34E0       LD BC,(CURPOS)
58735 0C             INC C
58736 79             LD A, C: LEFT END OF LINE?
58737 FE42           CP 66
58739 3803           JR C, CURSOR
58741 0E02           LD C, 2: RIGHT OF I LINE UP
58743 04             INC B: NO CHECK OF OFF TOP OF SCREEN
58744 CD1CE6 CURSOR  CALL MOVECUR
58747 18A2           JR MOVECB
58749 FE0A     DOWN  CP 10
58751 2007           JR NZ, UP
58753 ED4B34E0       LD BC,(CURPOS)
58757 05             DEC B
58758 18F0           JR CURSOR
58760 FE0B       UP  CP 11
58762 200D           JR NZ, RIGHT
58764 ED4B34E0       LD BC,(CURPOS)
58768 04             INC B
58769 78             LD A,B
58770 FE18           CP 24
58772 38E2           JR C, CURSOR
```

```
58774 05              DEC B
58775 18DF            JR CURSOR
58777 FE09  RIGHT     CP 9
58779 200F            JR NZ, STOP
58781 ED4B34E0        LD BC,(CURPOS)
58785 0D              DEC C
58786 3E01            LD A,1
58788 B9              CP C
58789 38D1            JR C,CURSOR
58791 DE41            LD C, 65
58793 05              DEC B
58794 18CC            JR CURSOR
58796 FEE2    STOP    CP 226
58798 2001            JR NZ, CAPSLOCK
58800 C9              RET
58801 FE06 CAPSLOCK   CP 6
58803 200A            JR NZ, GRAPHICS
58805 3A6A5C          LD A,(FLAGS2)
58808 EE08            XOR 8
58810 326A5C          LD (FLAGS2),A
58813 18A6            JR JUMP
58815 FE0F GRAPHICS   CP 15
58817 200A            JR NZ, EMODE
58819 3A415C          LD A,(MODE)
58822 EE02            XOR 2
58824 32415C          LD (MODE),A
58827 1898     JUMP1  JR JUMP
58829 FE0E  EMODE     CP 14
58831 2046            JR NZ, PRINT
58833 3A415C          LD A,(MODE)
58836 EE01            XOR 1
58838 32415C          LD (MODE),A
58841 CD36E6          CALL GETCHAR
58844 FED9            CP 217: INK?
58846 2016            JR NZ, OVER
58848 AF              XOR A
58849 324115C         LD (MODE),A: MUST GO BACK TO L MODE FOR #
58852 CD36E6     X    CALL GETCHAR
58855 FE30            CP 48: IS CODE A #?
58857 38F9            JR C, X: TRY AGAIN
58869 FE3A            CP 58
58861 30F5            JR NC, X
58863 D630            SUB 48: REDUCE CODE TO #
58865 CD71E2          CALL SETATT
58868 181A            JR OUT1
58870 FEDE    OVER    CP 222
58872 200A            JR NZ, INVERSE
58874 3A38E0          LD A,(MASKB)
58877 EE01            XOR 1
58879 3238E0          LD (MASKB),A
58882 180C            JR OUT1
58884 FEDD  INVERSE   CP 221
58886 2008            JR NZ, OUT1
58888 3A38E0          LD A,(MASKB)
```

```
58891 EE04           XOR 4
58893 3238E0         LD (MASKB), A
58896 AF      OUT1   XOR A: A=0
58897 32415C         LD (MODE),A
58900 C335E5         JP GETCHAR
58903 CD40E0 PRINT   CALL WRCHAR
58906 18AF           JR JUMP1
58908 C5  MOVECUR    PUSH BC: SAVE POSITON
58909 3E01           LD A, 1: SET OVER ON
58911 3238E0         LD (MASKB), A
58914 3E5F           LD  A, 95: DELETE CURSOR
58916 CD40E0         CALL WRCHAR
58919 C1             POP BC: USE POSITION
58920 CD56E4         CALL UPDPOS
58923 3E5F           LD A, 95: WRITE CURSOR
58925 CD40E0         CALL WRCHAR
58928 3E00           LD A, 0
58930 3238E0         LD (MASKB),A
58933 C9             RET
58934 AF  GETCHAR    XOR A
58935 32085C         LD (LASTK),A
58938 010060         LD BC, 24576
58941 0B  DEBOUNCE   DEC BC
58942 78             LD A, B
58943 B1             OR C
58944 20FB           JR NZ, DEBOUNCE
58946 3A085C  NONE   LD A,(LASTK)
58949 A7             AND A: STILL ZERO?
58950 28FA           JR Z, NONE
58952 C9             RET
```

Please note that not all the checks for moving off the screen have been written into all the routines. In fact, these check routines to make sure parameters stay within limits is what takes up a lot of any machine code routine. Therefore, please don't check the cursor movements by deliberately running them off the screen as this program may crash. You will note several different methods of checking these limits are used to show the student that it is possible to write in different manners and still get it to work.

Save the code with "wpo.bin",50453,307. Load the "64o" code and then this code in back of it with "wpo.bin,58645,. Then simply do a RANDOMIZE USR 58645 and the program runs. Of course, doing a stop exits you back to Basic. You can get back in by again doing a RANDOMIZE USR 58645 with the cursor reset to the home position. POKE 57380,0 and RANDOMIZE USR 57381 gets you back to normal screen mode.

Strange effects will be noticed in the inverse and over mode. Both can be on at the same time. Inverse of course also inverts the cursor as well as the standard graphics. Using OVER of course leaves the cursor under the character as nothing in the character block is deleted as the 2nd character is printed over

the first and only pixels that are on in both characters are inverted. We would have to add a check in the present routine to detect when OVER is on which would jump to another routine to first delete the cursor and then overprint the character. However, 90 % of the time the only time one wants OVER is to underline so I left it as it is. You will also note that moving the cursor back to correct the spelling of a word moves the cursor over to the following character and deletes that causing you to retype to the end of the word.

You may also note that the keyboard is a bit sluggish if you are a speed typist. This can be adjusted by lowering the count in the debounce routine at 58941 as it counts down from 24576 to 1 before continuing to allow slow people to get their fingers off the keys. Also note that since the repeat function is in the ROM routines of finding LASTK it also works in our processor.

There is still a long way to go to have a full-fledged word processor as there is no word wrap, space insert, space delete, tab, print file, move and delete block, find word and all the write, edit and lprint to and from an ASCII file that must be added. On the other hand, our little addon 307 bytes of code did quite a bit. We leave it to the student to go on from this point. There is still 4K available in chunk 7. Most of the space below chunk 7 would be used for as long a storage file as possible for the ASCII codes as that determines how long a document you can handle. There are 3500 characters on a 55 line 64 column page if you count every byte as Tasword does. Less space is used by Mscript as only the "\" is used to indicate a blank line.

Scroll 64 Screen Left and Right

However, since they are short and may have uses in other than word processors, here are the two routines to scroll the entire 64 column screen left or right one character space leaving a trailing blank. I put them at the end of our present routine where those of you with the program disk may use them.

Scroll Left:

```
58953 F3    LEFT      DI: No interrupts until done
58954 210040          LD HL, 16384: DFILE1
58957 110060          LD DE, 24576: DEFLE2
58960 0EC0            LD C, 192: # of lines
58962 061F    NXTLN   LD B, 31: # double shifts/line
58964 1A      LOOP    LD A, (DE): DF2 to DF1
58965 77              LD (HL),A
58966 23              INC HL
58967 7E              LD A,(HL): DF1 to DF2
58968 12              LD (DE),A
58969 13              INC DE
58970 10F8            DJNZ LOOP
58972 1A              LD A,(DE): XFER LAST CHAR OF LINE
```

```
58973 77              LD (HL), A
58974 AF              XOR A: A=0 TO BLANK LAST CHAR
58975 12              LD (DE),A
58976 13              INC DE: TO NEXT LINE
58977 23              INC HL
58978 0D              DEC C
58979 20ED            JR NZ, NXTLN
58981 FB              EI
58982 C9              RET
```

Scroll Right:

```
58983 F3  RIGHT       DI: NO INTERRUPTS UNTIL DONE
58984 21FF57          LD HL, 22527: END OF DFILE1 PIXELS
58987 11FF77          LD DE, 30719: END OF DFILE2 PIXELS
58990 0EC0            LD C, 192
58992 061F  NXTLN     LD B, 31
58994 7E        LOOP  LD A,(HL)
58995 12              LD (DE), A
58996 1B              DEC DE
58997 1A              LD A,(DE)
58998 77              LD (HL),A
58999 2B              DEC HL
59000 10F8            DJNZ LOOP
59002 7E              LD A,(HL): XFER LAST CHAR
59003 12              LD (DE), A
59004 3600            LD (HL),0: BLANK LAST CHAR
59006 2B              DEC HL
59007 1B              DEC DE
59008 0D              DEC C
59009 20ED            JR NZ, NXTLN
59011 FB              EI
59012 C9              RET
```

You will note that since we are scrolling the entire screen we are not concerned with moving a full character at a time. We scroll the pixel lines in the sequence in which they are stored in the DFILES. All 24 lines of the screen scroll for you game fans out there. A major change would be needed not to scroll the bottom two lines as is done in most word processors. The first 2/3rds of the screen could be done as above but the last 1/3rd would require skipping the pixels of the last 2 lines. Use of DI and EI are recommended whenever we are using an extended routine that uses the lower half of the RAM so that we don't conflict with the refresh TV/monitor use of the data bus. Besides, it gives the viewer the impression of an instantaneous shift.

On to the 80 column mode. It uses the same 64 column screen we have just been using and only writes smaller characters to the sceen to cram 80 to 85 characters on a line rather than only 64. If you have the Aerco disk system, the RPM programs imported from other computers use this type of screen to use the print routines directly. One has to remember that most of these CPM or RPM programs were written quite a few years ago and usually

contain a minimum of screen routines which are not up to what we have been accustomed to.

80 COLUMN MODE

What do we have to do to modify the 64 column mode to 80 column printing using 5 bit wide characters and a 6th bit for spacing between characters? At this point load your source code back into Zeus or disassemble your 64 characacter code with Hot Z or do what ever you have to do to modify the 64 character code with your assembler. (Those of you with tapes or disks of the programs can forget doing the assembly as it's already done for you in "80.bin" or "80" CODE.) We will be assembling to the same starting address. The first thing we will need is a specially designed character table which we will enter later. But we used CHRSET to define this address so we have to change that EQU statement to read:

```
CHRSET EQU ((CHRST) - 256).
```

Next, as we pointed out earlier, we have 512 pixels per line which divided by 6 allows us an extra 32 pixels to play with so we can define where these pixels will go by adding them on to the end of the variable table at address 57405 as:

```
57405 01 MARGIN  DEFB 1: Margin adjust (0-2)
```

It can take on values zero to 2.

Because of the complex nature of at times writing character pixels to two different bytes in alternating display files we also need another variable called DFBIT at 57406:

```
57406 00 DFBIT: Bit posn in line (7,1,3,5).
```

It can have the values given.

We of course have to rewrite part of WRCH and all of GTCH and CALCPOS.

Rewriting WRCH: Adding the two extra variables of course pushes the start of our WRCH routine up two addresses. This will automatically be done by the assembler. The first part of WRCH is the sorting of the characters to either standard characters, standard graphics or UDC's. Jump down to two lines above the subroutine WRCH5 and delete from there to WRCHXT (addresses 57499 (E09B) to 57529 (E0B9) and replace by the following: (WRCH5 moves up):

```
57499 08        WRCH5     EX AF,AF': SAVE COUNT
57500 C5                  PUSH BC: SAVE MASK
57501 D5                  PUSH DE: SAVE CHAR TBL ADDR
57502 DD210000            LD IX, 0: SAVE STACK POINTER
```

```
57506 DD39              ADD IX, SP
57508 56                LD D, (HL): READ SCAN LINE FROM DFILE
57509 E5                PUSH HL: SAVE ADDR
57510 7C                LD A, H
57511 EE20              XOR 32 TOGGLE TO ALTERNATE DFILE
57513 67                LD H, A
57514 CB6F              BIT 5, A: WHICH FILE?
57516 2001              JR NZ, WRCH6: IN DF2
57518 23                INC HL NEXT BYTE
57519 5E        WRCH6   LD E, (HL): READ SCAN LINE FROM ALT DF
57520 EB                EX DE, HL: HL = SCAN, DE = LAST OF PNT
57521 78                LD A, B: OVER?
57522 A7                AND A
57523 2014              JR NZ, NOXOR
57525 3A3EE0            LD A, (DFBIT)
57528 D607              SUB 7
57530 ED44              NEG
57532 01FF03            LD BC, 1023: MASK
57535 2808              JR Z, NORR1: NO ADJ NEC START AT BIT 7
57537 37                SCF
57538 CB18      NEXT1   RR B: SHIFT MASK A TIMES
57540 CB19              RR C: 6 0'S = START AT BIT 1 OF B, 3
57542 3D                DEC A               OF B, OR 5 OF B
57543 20F9              JR NZ, NEXT1
57545 7C        NORR1   LD A, H: SCAN LINE FROM DF
57546 A0                AND B
57547 67                LD H, A: RETAIN ADJACENT CHAR PIXELS
57548 7D                LD A, L: CLEAR CURRENT CHAR PIXELS
57549 A1                AND C
57550 6F                LD L, A: HL = OLD BIT ROW
57551 3A3EE0    NOXOR   LD A, (DFBIT): GET STARTING BIT POSN
57554 D607              SUB 7
57556 ED44              NEG
57558 E5                PUSH HL: SAVE SCANS
57559 DD6E00            LD L, (IX+0) POINTER TO CHAR SET
57562 DD6601            LD H, (IX+1)
57565 46                LD B, (HL): SCAN LINE FROM CHAR SET TO
57566 0E00              LD C, 0:          BC
57568 F5                PUSH AF: SAVE A AND FLAGS
57569 78                LD A, B: PIXELS FROM CHAR SET
57570 E6FC              AND 252: ONLY PIXELS
57572 47                LD B, A
57573 F1                POP AF: RESTORE A AND FLAGS
57574 E1                POP HL: AND SCANS
57575 2808              JR Z, NORR2
57577 A7                AND A: CLEAR FLAGS
57578 CB18      NEXT2   RR B
57580 CB19              RR C
57582 3D                DEC A
57583 20F9              JR NZ, NEXT 2
57585 7C        NORR2   LD A, H: SCAN FROM DF
57586 A8                XOR B: INSERT/COMBINE NEW CHAR
57587 67                LD H, A
57588 7D                LD A, L
```

```
57589 A9                XOR C
57590 6F                LD L, A: DF XOR CHAR SET TO HL
57591 DD7E02            LD A ,(IX+2): INVERSE ON?
57594 A7                AND A
57595 281A              JR Z, NOINVERT
57597 3A3EE0            LD A, (DFBIT)
57600 D607              SUB 7
57602 ED44              NEG
57604 0100FC            LD BC, 64512: MASK TO INVERT
57607 2808              JR NZ, NORR3
57609 A7                AND A
57610 CB18      NEXT3   RR B: SHIFT MASK A TIMES
57612 CB19              RR C
57614 3D                DEC A
57615 20F9              JR NZ, NEXT3
57617 7C                LD A, H
57618 A8                XOR B: INVERT CHAR
57619 67                LD H, A
57620 7D                LD A,L
57621 A9                XOR C
57622 6F                LD L, A
57623 EB        NOINVERT EX DE, HL: HL = DF ADDR/DE = SCAN LINE
57624 73                LD (HL), E: SCAN LINE TO DF
57625 E1                POP HL: ALTERNATE DF
57626 72                LD (HL), D: SCAN LINE TO DF
57627 D1                POP DE: CHAR SET
57628 C1                POP BC: OVER/INVERSE INDICATORS
57629 08                EX AF, AF': RESTORE SCAN COUNT
57630 24                INC H: TO NEXT SCAN LINE
57631 13                INC DE: TO NEXT CHAR PIXEL
57632 3D                DEC A: MORE SCANS?
57633 C29BE0            JP NZ, WRCH5
57636 E1        WRCH7   POP HL: TOP SCAN IN DF
57637 C1                POP BC: CURSOR POSN
57638 0D                DEC C: ADJUST CURSOR POSN
57639 3A3EE0            LD A, (DFBIT): ADJUST BIT POSN
57642 D606              SUB 6
57644 323EE0            LD (DFBIT), A: STORE UPDATED POSN
57647 300E              JR NC, WRCHXT: NEXT CHAR IN SAME BYTE
57649 C608              ADD A, B: NEXT CHAR IN NEXT BYTE
57651 323EE0            LD (DFBIT), A: STORE ADJUSTED VALUE
57654 7C                LD A, H: ADJUST DF ADDR TO NEXT BYTE
57655 EE20              XOR 32: TOGGLE TO ALTERNATE DF
57657 67                LD H, A
57658 CB6F              BIT 5, A: WHICH DF?
57660 2001              JR NZ, WRCHXT: DONE IF IN DF2
57662 23                INC HL: NEXT BYTE IN DF1
57663           WRCHXT
```

The WRCH routine is now 132 bytes longer so all the following routines are displaced by 134 bytes. Once again the assembler will automatically adjust for you. We don't have to make any corrections to WRSTR, SETCUR, SCROLL, CLRSC, SETA or SETM. Since GTCH gets the code of a screen character, it is affected and

must be almost completely rewritten with some bit fiddling as we did in WRCH as follows:(It calls a routine called GTC2BC which we will add at the end of the code just before the character tables.)

```
58150 ED4B19E0 GTCHBO   LD BC, (GETCTL): GET CONTROL INFO
58154 CDEBE4   GTCHAR   CALL TSTPAR: TEST PARAMETERS
58157 3A3EE0            LD A, (DFBIT): CURRENT BIT POSN
58160 F5                PUSH AF
58161 CDFCE4            CALL CONVFM: CONVERT TO INT. FORMAT
58164 CD0CE5            CALL CALCPOS: GET DFILE ADDR
58167 ED5B2FE0          LD DE, (CHTBL)
58171 0660              LD B, 96: #OF CHARS
58173 3E80              LD A, 128
58175 323AE0   GTCH1    LD (GTINDX), A
58178 14                INC D: ADJ TO START OF TBL
58179 C5       GTCH2    PUSH BC: SAVE COUNT
58180 E5                PUSH HL: SAVE DF ADDR
58181 D5                PUSH DE: SAVE CHAR TABLE ADDR
58182 CD60E5   GTCH22   CALL GTC2BC
58185 78                LD A, B
58186 E6FC              AND 252: ONLY THE PIXEL BITS
58188 47                LD B, A
58189 EG                EX DE, HL: HL = CHAR SET POINTER
58190 7E                LD A, (HL): PIXEL ROW TO A
58191 E6FC              AND 252: MASK TO 6 BITS
58193 A8                XOR B: TEST AGAINST DF
58194 2810              JR Z, GTCH3: MATCH
58196 F5                PUSH AF
58197 3A3AE0            LD A, (GTINDX): STANDARD GRAPHIC?
58200 FE90              CP 144
58202 2003              JR NZ, GTCH23
58204 F1                POP AF
58205 183B              JR GTCH4: DON'T TEST INVERSE
58207 F1       GTCH23   POP AF
58208 FEFC              CP 252: TEST IF MATCH ON INVERSE
58210 2036              JR NZ, CTCH4: NO MATCH
58212 4F       GTCH3    LD C, A: C=0 FOR MATCH -4 FOR INVERSE
58213 0607              LD B, 7: SCAN COUNT
58215 EB       GTCH31   EX DE, HL: DF ADDR TO HL
58216 C5                PUSH BCZ SAVE MATCH AND SCAN COUNT
58217 24                INC H: NEXT SCAN IN DF
58218 13                INC DE: NEXT ROW IN CHAR TBL
58219 CD60E5            CALL GTC2BC: NEXT SCAN ROW TO BC
58222 78                LD A, B
58223 E6FC              AND 252: PIXELS ONLY
58225 47                LD B, A
58226 EB                EX DE, HL: HL=CHAR SET POINTER
58227 7E                LD A, (HL): CHAR SET ROW
58228 E6FC              AND 252: PIXELS ONLY
58230 A8                XOR B: MATCH TO THAT IN B
58231 C1                POP BC: GET PREV MATCH AND SCAN COUNT
58232 A9                XOR C
58233 201F              JR NZ, GTCH4: NO MATCH START AGAIN AT
```

```
58235 10EA              DJNZ GTCH31: DO NEXT SCAN          NEXT
58237 79       GTCH32   LD A, C: A=-4 IF INVERSE MATCH
58238 C603              ADD A, 3: SET TO -1 ON INVERSE
58240 C1                POP BC: HERE IF MATCH (SP)=PNT TO CHAR
58241 C1                POP BC: (SP+4)= CHAR COUNT
58242 C1                POP BC: B=COUNT C=-1 IF INVERSE MATCH
58243 4F                LD C, A: CONVERT MATCH TO CHAR CODE
58244 3A3AE0            LD A, (GTINDX)
58247 90                SUB B
58248 FE20              CP 32: SPACE?
58250 2005              JR NZ, GTCH32: NO
58252 0C                INC C: TEST IF MATCH ON INVERSE
58253 2002              JR NZ, GTCH32
58255 3E8F              LD A, 143: LD CODE FOR GRAPHIC BLOCK
58257 4F       GTCH32   LD C, A: RET IN C
58258 0600              LD B, 0: AND IN A
58260 F1                POP AF
58261 323EE0            LD (DFBIT), A: RESTORE DFBIT
58264 79                LD A, C: CHAR CODE TO A
58265 C9                RET
58266 E1       GTCH4    POP HL: POINTER TO CHAR SET
58267 110800            LD DE, 8: NEXT CHAR
58270 19                ADD HL, DE
58271 5D                LD E, L: XFER TO DE
58272 54                LD D, H
58273 E1                POP HL: DF ADDR
58274 C1                POP BC: CHAR COUNT
58275 109E              DJNZ GTCH2: LOOK AGAIN
58277 3A3AE0            LD A, (GTINDX): TEST IF DONE
58280 FE80              CP 128: STD CHAR SET?
58282 200A              JR NZ, GTCH5
58284 ED5B31E0          LD DE, (GRTBL)
58288 0610              LD B, 16: # OF ENTRIES
58290 3E90              LD A, 144
58292 1889              JR GTCH1
58294 FE90     GTCH5    CP 144
58296 200C              JR NZ, GTCH6
58298 ED5B7B5C          LD DE, (UDG)
58302 15                DEC D: ADJUST START
58303 0615              LD B, 21: # OF ENTRIES
58305 3EA5              LD A, 165: INDEX
58307 C33FE3            JP GTCH1
58310 F1                POP AF
58311 323EE0            LD (DFBIT), A: RESTORE DFBIT
58314 48       CTCH6    LD C, B
58315 AF                XOR A
58316 C9                RET
```

The GTCH routine also is longer thereby increasing addresses by 172 from here on out. GETA and GETC don't need changes. We have 1 change and 1 addition to STMD. We have to change the line length to 80:

```
58380 3E50              LD A, 80
```

Right above the line LD (LINLEN), A.

Right below this line insert the following two lines to set the margin:

```
58385 3E01          LD A, 1
58387 323dE0        LD (MARGIN), A
```

This makes our offset 177 addrdsses.

Since characters no longer occupy a full byte we have to redo CALCPOS as well:

```
58636 CD3FE5   CALCPOS   CALL LNBO: GET DFILE ADDR FOR LINE
58639 3A33E0             LD A, (LINLEN)
58642 3C                 INC A
58643 91                 SUB C
58644 F5                 PUSH AF: SAVE COL # (0-(LINLEN)-1))
58645 5F                 LD E, A
58646 87                 ADD A, A
58647 83                 ADD A, E: 3*(80 C POSN)/4=(64 C POSN)
58648 CB3F               SRL A
58650 CB3F               SRL A
58652 CB47               BIT 0, A
68654 2806               JR Z, CALCP1
68656 F5                 PUSH AF: ODD COLS ARE IN DF2
68657 3E20               LD A, 32
58659 B4                 OR H: SHIFT TO DF2
58660 67                 LD H, A
58661 F1                 POP AF
58662 A7       CALCP1    AND A: CLEAR CARRY
58663 1F                 RRA: COL/2
58664 5F                 LD E, A
58665 3A3DE0             LD A, (MARGIN): GET ADJUSTMENT VALUE
58668 83                 ADD A, E: AND ADD TO DE
58669 5f                 LD E, A
58670 1600               LD D, E
58672 19                 ADD HL, DE: DE=OFSET IN LINE, HL=DF
58673 1E07               LD E, 7                          ADDR
58675 F1                 POP AF: A=COL POSN FOR 80 COL
58676 E603               AND 3
58678 2802               JR Z, CALCP2: STARTS AT BIT 7
58680 5F                 LD E, A: STARTS AT BIT 1, 3 OR 5
58681 3D                 DEC A
58682 83       CALCP2    ADD A, E
58683 323EE0             LD (DFBIT), A
58686 C9                 RET
58687          LNBO      NO CHANGES IN THIS SUBROUTINE
```

STPOSN and LDPOSN need no changes. We now add GTC2BC:

```
58720 E5       GTC2BC    PUSH HL: SAVE DF ADDR
58721 46                 LD B, (HL): SCAN  LINE FROM DF
58722 7C                 LD A, H
```

```
58723 EE20              XOR 32
58725 67                LD H, A: TOGGLE TO ALTERNATE DFILE
58726 CB6F              BIT 5, A: WHICH DF?
58728 2001              JR NZ, GTC2B1
58730 23                INC HL
58731 4E        GTC2B1  LD C, (HL): BC=CHAR AT DFBIT
58732 3A3EE0            LD A, (DFBIT)
58735 D607              SUB 7
58737 ED44              NEG
58739 2807              JR Z, GTC2B2: A=# OF PIXELS FOR CHAR
58741 CB11      GTCSHFT RL C: TO B, BITS 7-2
58743 CB10              RL B
58745 3D                DEC A
58746 20F9              JR NZ, GTCSHFT
58748 E1        GTC2B2  POP HL
58749 C9                RET
```

And the compressed character set before the standard graphics table:

```
58750 00000000 CHRST    DEFB 0,0,0,0,0,0,0,0: CODE 32 BLANK
58754 00000000
58758 0010101010001000  DEFB 0,16,16,16,16,0,16,0: CODE 33 !
58766 0048480000000000  DEFB 0,72,72,0,0,0,0,0: CODE 34 "
58774 00287C28287C2800  DEFB 0,40,124,40,40,124,40,0: CODE 35
58782 00107C507C147C10  DEFB 0,16,124,80,124,20,124,16:CODE 36

58790 00646810102C4C00  DEFB 0,100,104,16,16,44,76,00: CODE 37
58798 0010281020483400  DEFB 0,16,40,16,44,72,52,0: CODE 38 &
58806 0010200000000000  DEFB 0,16,32,0,0,0,0,0: CODE 39 '
58814 0008101010100800  DEFB 0.8,16,16,16,16,8,0: CODE 40 (
58822 0020101010102000  DEFB 0,32,16,16,16,16,32,0: CODE 41 )

58830 000028107C102800  DEFB 0,0,40,16,124,16,40,0: CODE 42 *
58838 000010107C101000  DEFB 0,0,16,16,124,16,16,0: CODE 43 +
58846 0000000000101020  DEFB 0,0,0,0,0,16,16,32: CODE 44 ,
58854 00000000007C0000  DEFB 0,0,0,0,0,124,0,0: CODE 45 -
58862 0000000000303000  DEFB 0,0,0,0,0,48,48,0: CODE 46 .

58870 0000040810204000  DEFB 0,0,4,8,16,32,64,0: CODE 47 /
58878 00384C5454643800  DEFB 0.56,76,84,84,100,56,0: CODE 48 0
58886 0030501010107C00  DEFB 0,48,80,16,16,16,124,0: CODE 49 1
58894 0038440438407C00  DEFB 0,56,68,4,56,64,124,0: CODE 50 2
58902 0038441804443800  DEFB 0,56,68,4,56,64,124,0: CODE 51 3

58910 00081828487C0800  DEFB 0,8,24,40,72,124,8,0: CODE 52 4
58918 007C407804443800  DEFB 0,124,64,120,4,68,56,0: CODE 53 5
58926 0038407844443800  DEFB 0,56,64,120,68,68,56,0: CODE 54 6
58934 007C040810202000  DEFB 0,124,4,8,16,32,32,0: CODE 55 7
58942 0038443844443800  DEFB 0,56,68,56,68,68,56,0: CODE 56 8

58950 003844443C043800  DEFB 0,56,68,68,60,4,56,0: CODE 57 9
58958 0000001000001000  DEFB 0,0,0,16,0,0,16,0: CODE 58 :
58966 0000100000101020  DEFB 0,0,0,16,0,0,16,16,32: CODE 59 ;
```

```
58974 0000081020100800    DEFB 0,0,8,16,32,16,8,0: CODE 60 <
58982 0000007C007C0000    DEFB 0,0,0,124,0,124,0,0: CODE 61 =

58990 0000201008102000    DEFB 0,0,32,16,8,16,32,0: CODE 62 >
58998 0038440810001000    DEFB 0,56,64,8,16,0,16,0: CODE 63 ?
59006 0038445C5C403C00    DEFB 0,56,68,68,124,124,64,60: CODE 64
59014 003844447C444400    DEFB 0,56,68,68,124,68,68,0: CODE 65 A
59022 0078447844447800    DEFB 0,120,68,120,68,68,120,0: CODE 66

59030 0038444040443800    DEFB 0,56,68,64,64,68,56,0: CODE 67 C
59038 0070484444487000    DEFB 0,112,72,68,68,72,112,0: CODE 68
59046 007C407840407C00    DEFB 0,124,64,124,64,64,124,0: CODE 69
59054 007C407840404000    DEFB 0,124,64,124,64,64,64,0: CODE 70
59062 003844405C443800    DEFB 0,56,68,64,92,68,56,0: CODE 71 G

59070 0044447C44444400    DEFB 0,68,68,124,68,68,68,0: CODE 72 H
59078 007C101010107C00    DEFB 0,124,16,16,16,16,124,0: CODE 73
59086 0004040444443C00    DEFB 0,4,4,4,68,68,124,0: CODE 74 J
59094 0048506050484400    DEFB 0,72,80,96,80,72,68,0: CODE 75 K
59102 0040404040407C00    DEFB 0.64,64,64,64,64,124,0: CODE 76 L

59110 00444C5444444400    DEFB 0,68,76,84,68,68,68,0: CODE 77 M
59118 00446454544C4400    DEFB 0,68,100,84,84,76,68,0: CODE 78 N
59126 0038444444443800    DEFB 0,56,68,68,68,68,56,0: CODE 79 O
59134 0078444478404000    DEFB 0,120,68,68,120,64,64,0: CODE 80
59142 0038444444543800    DEFB 0,56,68,68,68,84,56,0: CODE 81 Q

59150 0078444478484400    DEFB 0,120,68,68,120,72,68,0: CODE 82
59158 0038403804443800    DEFB 0,56,64,56,4,68,56,0: CODE 83 S
59166 007C101010101000    DEFB 0,124,16,16,16,16,16,0: CODE 84 T
59174 0044444444445600    DEFB 0,68,68,68,68,68,56,0: CODE 85 U
59182 0044444444281000    DEFB 0,68,68,68,68,40,16,0: CODE 86 V

59190 0044444454542800    DEFB 0,68,68,68,84,84,40,0: CODE 87 W
59198 0044281010284400    DEFB 0,68,40,16,16,40,68,0: CODE 88 X
59206 0044442810101000    DEFB 0,68,68,40,16,1616,0: CODE 89 Y
59214 007C040810207C00    DEFB 0,124,4,8,16,32,124,0: CODE 90 Z
59222 0038202020203800    DEFB 0,56,32,32,32,32,56,0: CODE 91 [

59230 0000402010080400    DEFB 0,0,64,32,16,8,4,0: CODE 92 \
59238 0070101010107000    DEFB 0,112,16,16,16,16,112,0: CODE 93
59246 0010385410101000    DEFB 0,16,56,84,16,16,16,0: CODE 94 ^
59254 00000000000000FC    DEFB 0,0,0,0,0,0,0,252: CODE 95 _
59262 0018247020207C00    DEFB 0,24,36,112,32,32,124,0: CODE 96

59270 000038043C443C00    DEFB 0,0,56,4,40,68,60,0: CODE 97 a
59278 0020203824243800    DEFB 0,32,32,56,36,36,56,0: CODE 98 b
59286 00001C2020201C00    DEFB 0,0,28,32,32,32,28,0: CODE 99 c
59294 0008083848483800    DEFB 0,8,8,56,72,72,56,0: CODE 100 d
59302 0000384478403C00    DEFB 0,0,56,68,120,64,44,0: CODE 101 e

59310 0010287020202000    DEFB 0,16,40,120,32,32,32,0: CODE 102
59318 0000384848380830    DEFB 0,0,56,72,72,56,8,48,0: CODE 103
59326 0040407048484800    DEFB 0,64,64,112,72,72,72,0: CODE 104
```

```
59334 0010003010103800    DEFB 0,16,0,48,16,16,56,0: CODE 105 i
59342 0008000808084830    DEFB 0,8,0,8,8,8,72,48: CODE 106 j

59350 0020283030282400    DEFB 0,32,40,48,48,40,36,0: CODE 107 k
59358 0020202020201800    DEFB 0,32,32,32,32,32,24,0: CODE 108 l
59366 0000285454545400    DEFB 0,0,40,84,84,84,84,0: CODE 109 m
59374 0000704848484800    DEFB 0,0,120,72,72,72,72,0: CODE 110 n
59382 0000384444443800    DEFB 0,0,56,72,72,72,56,0: CODE 111 o

59390 0000704848704040    DEFB 0,0,112,72,72,112,64,64: CODE 112
59398 000038484838080C    DEFB 0,0,56,72,72,56,8,12: CODE 113 q
59406 0000182020202000    DEFB 0,0,24,32,32,32,32,0: CODE 114 r
59414 0000304030087000    DEFB 0,0,48,64,48,8,48,0: CODE 115 s
59422 0020702020201800    DEFB 0,32,112,32,32,32,24,0: CODE 116

59430 0000484848483000    DEFB 0,0,72,72,72,72,48,0: CODE 117 u
59438 0000444428281000    DEFB 0,0,72,72,40,40,16,0: CODE 118 v
59446 0000445454542800    DEFB 0,0,68,84,84,84,40,0: CODE 119 w
59454 0000442810284400    DEFB 0,0,68,40,16,40,68,0: CODE 120 x
59462 0000484848380870    DEFB 0,0,72,72,72,56,8,112: CODE 121 y

59470 00007C0810207C00    DEFB 0,0,124,8,16,32,124,0: CODE 122 z
59478 001C106010101C00    DEFB 0,28,16,96,16,16,28,0: CODE 123
59486 0010101010101000    DEFB 0,16,16,16,16,16,16,0: CODE 124
59494 0070100810107000    DEFB 0,112,16,8,16,16,112,0: CODE 125
59502 0028500000000000    DEFB 0,40,80,0,0,0,0,0,0: CODE 126

59510 788494A4A4948478    DEFB 120,132,148,164,164,148,120: 127
```

The standard graphics table must also be changed to 6 wide:

```
59518 00000000  GRTBL     DEFB 0,0,0,0,0,0,0,0: CODE 128
58522 00000000
59526 1C1C1C1C00000000    DEFB 28,28,28,28,0,0,0,0: CODE 129
59534 E0E0E0E000000000    DEFB 224,224,224,224,0,0,0,0: CODE 130
59542 FCFCFCFC00000000    DEFB 252,252,252,252,0,0,0,0: CODE 139
59550 000000001C1C1C1C    DEFB 0,0,0,0,28,28,28,28: CODE 140

59558 1C1C1C1C1C1C1C1C    DEFB 28,28,28,28,28,28,28,28: CODE 141
59566 E0E0E0E01C1C1C1C    DEFB 224,224,224,224,28,28,28,28: 142
59574 FCFCFCFC1C1C1C1C    DEFB 252,252,252,252,28,28,28,28: 143
59582 00000000E0E0E0E0    DEFB 0,0,0,0,224,224,224,224: CODE 144
59590 1C1C1C1CE0E0E0E0    DEFB 28,28,28,28,224,224,224,224: 145

59598 E0E0E0E0E0E0E0E0    DEFB 224,224,224,224,224,244,224,224:
59606 FCFCFCFCE0E0E0E0    DEFB 252,252,252,252,224,224,224,224:
59614 00000000FCFCFCFC    DEFB 0,0,0,0,252,252,252,252: CODE 148
59622 1C1C1C1CFCFCFCFC    DEFB 28,28,28,28,252,252,252,252: 149
59630 E0E0E0E0FCFCFCFC    DEFB 224,224,224,224,252,252,252,252:

59638 FCFCFCFCFCFCFCFC    DEFB 252,252,252,252,252,252,252,252:

59645 LAST BYTE USED      LENGTH OF CODE 2302
```

Assemble and save your code as CAT "80,bin",ADDR,2302 if to Aerco disk or as LOAD "80",CODE ADDR,2303. ADDR is the address of the start of your code depending upon what displacement you used. You will be using your code at 57344, so your Basic program will reload the code as CAT "80.bin",57344, 2302 or with LOAD "80" CODE 57344,2302 if from tape.

And to check your code for accuracy, or load directly if you don't happen to have an assembler:

```
57344 6BC33FE0C371E100
57352 00C3BAE10018C378
57360 E238C3F3E200C31D
57368 E30000C326E3C3CD
57376 E3C3D4E308C3F1E3

57384 0117C3CAE117017E
57392 E47EE75555180040
57400 0038800000000013A
57408 00E1CD58E5FE20DA
57416 47E1FEA5D247E1C5

57424 FE803815FE903809
57432 ED4B7B5C05D67018
57440 0CED4B31E0D66018
57448 04ED4B2FE0EB2600
57456 6F29292909C1EB79

57464 3D3A33E020053C05
57472 4F18013CB9D5CC4B
57480 E1D1C5E53A38E006
57488 FF1F3801041F1F9F
57496 4F3E0B08C5D5DD21

57504 0000DD3956E57CEE
57512 2067CB6F2001235E
57520 EB78A720143A3EE0
57528 D607ED4401FF0328
57536 0837CB18CB193D20

57544 F97CA0677DA16F3A
57552 3EE0D607ED44E5DD
57560 6E00DD6601460E00
57568 F578E6FC47F1E128
57576 08A7CB18CB193D20

57584 F97CAB677DA96FDD
57592 7E02A7281A3A3EE0
57600 D607ED440100FC28
57608 08A7CB18CB193D20
57816 F97CA8677DA96FEB

57624 73E172D1C1082413
57632 3DC29BE0E1C10D3A
```

```
57640 3EE0D606323EE030
57648 0EC608323EE07CEE
57656 2067CB6F200123CD

57664 50E50E000600C90E
57672 0118F93E1890573A
57680 2DE0BAD207E53A2E
57688 E03D200B3E01322D
57696 E0C1C10E0318DD32

57704 2EE0ED4B2EE0C3E4
57712 E13AC35CE61FF640
57720 572A4B5C7EE67F28
57728 35BA2808D5CD2017
57736 EBD118F0234E2346

57744 2378B1C87EE5C5CD
57752 3FE079B02009C1E1
57760 230B78B120EEC979
57768 FE0128F2E1223BE0
57776 E10E0E0600C90E02

57784 18F9ED4B28E0CDEB
57792 E4CDFCE4CD07E5C3
57800 42E1ED4B28E078A7
57808 CA47E181FE01DA47
57816 E1FE19D247E1CDE4

57824 E1C342E1C53E1891
57832 473A33E03C4FCD3F
57840 E5C1C57DA7285107
57848 07074F3E0891B838
57856 3F780600C5470E08

57864 78C50F0F0F4F0600
57872 EB210FFF19EBC5E5
57880 EDB0E1C17CCB6F20
57888 05F6206718EAE6DF
57896 6724C10D20DAC178

57904 A728042E0018BCC1
57912 79803D4F06011852
57920 4F789147C57918BD
57928 05C50E08C5EB21E0
57936 F819EB012000E5ED

57944 B0E17CCB6F2005F6
57952 206718E9E6DF6724
57960 C10D20E0C178A728
57968 C61120FB19C3F3E1
57976 ED4B0CE078A7CA47

57984 E181FE01DA47E1FE
57992 19D247E1CD92E2C3
```

```
58000 42E1C53E1891473A
58008 33E03C4FCD3FE550
58016 59C1D57D0707074F

58024 3E0891B8383D7806
58032 00C5470E0878C5E6
58040 070F0F0F4F06000D
58048 C5E5545D360013ED
58056 B0E1C17CCB6F2005

58064 F6206718EBE6DF67
58072 24C10D20D8C178A7
58080 28052E00C3A3E2C1
58088 C307E54F789147C5
58096 7918BF3A11E0E607

58104 4F3AC25CA7CA47E1
58112 79171717F5F60647
58120 DBFFE6C0B0D3FFF6
58128 4032C25CF12FB132
58136 39E0C342E13A15E0

58144 3238E0C342E1ED4B
58152 19E0CDEBE43A3EE0
58160 F5CDFCE4CD0CE5ED
58168 5B2FE006603E8032
58176 3AE014C5E5D5CD60

58184 E578E6FC47EB7EE6
58192 FCA82810F53A3AE0
58200 FE902003F1183BF1
58208 FEFC20364F0607EB
58216 C52413CD60E578E6

58224 FC47EB7EE6FCA8C1
58232 A9201F10EA79C603
58240 C1C1C14F3A3AE090
58248 FE2020050C20023E
58256 8F4F0600F1323EE0

58264 79C9E1110800195D
58272 54E1C1109E3A3AE0
58280 FE80200AED5B31E0
58288 06103E901889FE90
58296 200CED5B7B5C1506

58304 153EA5C33FE3F132
58312 3EE048AFC93A39E0
58320 4F060009ED4B34E0
58328 CDFCE43A33E0B920
58336 0B0E00783C47FE18

58344 38020617ED4307E0
58352 C93A24E0A7284FFE
```

```
58360 08C247E121011722
58368 28E03E17322DE03E
58376 01322EE03E503233

58384 E03E01323DE02100
58392 18220CE0AF3238E0
58400 323BE0323CE0217E
58408 E4222FE0217EE722
58416 31E03200E03207E0

58424 3208E03215E03219
58432 E0321AE03E06473A
58440 C25CA7201DB0CA42
58448 E121C01211400819
58456 ED5B655C19DAA7E4

58464 ED5BB25CA7ED52D2
58472 A7E4CDD2E478CDAC
58480 E4C578CD8E0ECDDF
58488 E4C17932C25CCB77
58496 CA42E13EC1B02FCB

58504 38CB38CB38B03239
58512 E03211E03A33E03C
58520 4F0618210000227D
58528 5CCD07E5C342E10E
58536 02C344E14FCB4720

58544 16FE80281ACB7FC0
58552 47FE02C8E6C6EE06
58560 C03E40B14FAFC9FE
58568 01C006010E81C9AF
58576 47C9F5DBFFF680D3

58584 FF3E03D3F4F1C9F5
58592 AFD3F4DBFFE63FD3
58600 FFF1C93E17B83004
58608 F1C347E3A33E03D
58616 B938F5C93A33E03C

58624 914F3E189047C9CD
58632 0CE51844CD3FE53A
58640 33E03C91F55F8783
58648 CB3FCB3FCB3F2806
58656 F53E20B467F1A71F

58664 5F3A3DE0835F1600
58672 191E07F1E6032802
58680 5F3D83323EE0C93E
58688 1890570F0F0FE6E0
58696 6F7AE618F64067C9

58704 ED4334E02236E0C9
58712 ED4B34E02A36E0C9
```

```
58720 E5467CEE2067CB6F
58728 2001234E3A3EE0D6
58736 07ED442807CB11CB

58744 103D20F9E1C9
START OF CHARACTER TABLE
```

Of course we need a Basic program to run our code. How about the same Basic program we used for 64 column mode? You see, with that setup at the beginning of the code, the same addresses which call these setup routines call them for the 80 column mode. All we have to do is change the first line of the program to load our new code program and POKE 57380 with 8 instead of 6 in line 10. We won't repeat that demo program here.

Our word processing code, however, will not work without some modifications of all the routines that use a length-of-line check like ENTER for a new line, DELETE, the cursor keys, etc. All must be revised from 64 to 80 columns and since they don't come out to an even character space the UPDPOS routine must be called a lot more often to update not only CURPOS but also DFBIT which depend upon the value in MARGIN. In writing the characters to the screen these special variables are also used automatically by WRCH. We leave it for the student to revise the word processor code for 80 column mode. It's about time you wrote a routine on your own. If you never write a program how are you ever going to learn about all the pitfalls that I can't possibly tell you about but have to learn on your own? People's minds work differently and some forget to do one thing while others forget to do other things so each person has to learn what his or her tendencies are. In this case you are just revising a source program so it should be easier. Work slowly and carefully and think through everything that you have to do and in what order. Once you write your own code you gain an appreciation for what other people have to go through to write code. That first program is always the hardest to give birth to. There are people around who can think in code just as you do in Basic.

CHAPTER 5 TWO SCREEN MODE

We now proceed to the two screen mode. These are two normal 32 column wide, 24 line deep screens with one attribute per character space that can be alternately displayed on the TV or monitor. According to your manual this mode is supposed to be used when doing rapid animation. One can write to either screen at anytime no matter what screen is being displayed at the time.

A word of caution to the game writer. The fastest you can change screens is of course 60 times per second which is faster than the human eyes can detect. Remember that motion pictures flick by the eye at 24/sec. so slow down your screen changes. That is not the caution. The caution is that if you are moving sprites across the screen and shooting at them, this requires detecting when a sprite has been touched, and this must be done on both screens not just one. Sometimes it easier to work in the single screen mode.

Like the double screen mode, the 2 screen mode is not supported from Basic so once again we have to write some routines, again courtesy of the Timex Technical Manual. The same general format will be used with an entry system from Basic. They are for the most part the same ones that we used in the 64 and 80 column modes with an additional routine included to transfer data from one screen to the other.

VIMODE is used to select which screen is being displayed. A value of 0 is of course the regular 1 screen mode. We have seen that 6 and 8 give the double screen 64 and 80 column mode. A value of 2 will give you the high resolution graphics mode. Using 128 will open both display files and display DFILE 1 on the monitor. 129 will keep both files open but display DFILE 2. A 4 will allow work on the contents of DFILE 1 and a 5 on DFILE 2. An analysis of this bitwise is very interesting. Bit 7 "on" turns out to be the 2 screen mode. Bit 0 "off" gives, by default, DFILE 1. Bit 0 "on" gives DFILE 2. Bit 3 (value of 4) activates writing or changing DFILE 1 or 2 depending upon bit 0 being on or off.

As with the previous routines, since Chunk 7 is used to store the Bank Switching routines to make room for opening DFILE 2, we will be writing to Chunk 6 at address 57344.

DEFINITIONS

```
SCRSZ    EQU 24: SCREEN SIZE IN LINES
SCINIT   EQU #1701: SCROLL CONTROL INIT
CLINIT   EQU #1800: CLEAR SCREEN INIT
CHRSET   EQU #3C00: ROM CHAR TABLE - 256
GRPHST   EQU ((GRPST)-#100): STND GRAPHICS TABLE
CHNGVID  EQU #0E8E: ROUTINE IN EXROM
RECLEN   EQU #1720: ROUTINE IN HOME ROM
ININT    EQU #30F9: ROUTINE IN HOME ROM
FP2A     EQU #3193: ROUTINE IN HOME ROM
HREXPT   EQU #FF: HOME ROM EXT SELECT PORT
DKHSPT   EQU #F4: HORIZONTAL SELECT REG PORT
VARS     EQU #5C4B: SYSTEM VARIABLES
CHRADD   EQU #5C5D
STKEND   EQU #5C65
UDG      EQU #5C7B
COORDS   EQU #5C7D
ATTRP    EQU #5C8D
MASKP    EQU #5C8E
PFLAG    EQU #5C91
RAMTOP   EQU #5CB2
PRAMT    EQU #5CB4
VIDMOD   EQU #5CC2
PARAMS   EQU #5CC3: (23747)
DRIVES   EQU #6840
INSERTSZ EQU #7B00 - DRIVES
MCVESZ   EQU DRIVES - #6000
DES7    EQU #FFFF - MOVESZ +#1
FIX      EQU DES7 - #6000
FIXTBL   EQU #1000
```

INPUTS AND ENTRIES

```
57344 20         DATAB     DEFB 32: DATA BYTE FROM BASIC
57345 C346E0     WRCHRB    JP WRCH0: ENTRY WRCHAR
57348 C3EEE0     WRSTRB    JP WRSTR0: ENTRY WRITE STRING
57351 0000       LINCOL    DEFW 0: SET CURSOR POSITION
  :LINCOL = START LINE #, LINCOL +1 = COLUMN #
57353 C317E1     SETCUB    JP SETCB0: ENTRY SET CURSOR POSN
57356 0018       CLRCTL    DEFW #1800: CLS CONTROL,
  :CLRCTL = START LINE #, CLRCTL + 1 = LINE CT
57358 C3D4E1     CLRSCB    JP CLRSB0 :ENTRY FOR CLEAR SCREEN
57361 00000000   SPARE0    DEFW 0,0: NOT USED (ALIGNMENT ONLY)
57365 00000000   SPARE1    DEFW 0,0:
57369 0000       GETCTL    DEFW 0:GET CONTR FROM BASIC
57371 C35CE2     GTCHRB    JP GTCHB0: ENTRY FOR GET CHAR
57374 C3DCE2     GETATB    JP GETAB0: ENTRY FOR GET ATTR
57377 C3F1E2     GETCUB    JP GETCB0: ENTRY FOR GET CURSOR POSN
57380 00         VIMODE    DEFB 0:VIDMODE 0=NOR, 128=DF1, 1=DF2
  :5= PRINT OPERATION TO DF2, 4= PRINT OPERATION TO DF1
57381 C30EE3     SETMDB    JP STMDB0: ENTRY TO SET VIDMODE
57384 0117       SCRCTL    DEFW #1701:SCROLL CONTL
```

```
  ;SCRCTL = STARTING LINE #, SCRCTL + 1=LINE COUNT
57386 C327E1    SCRLB     JP SCRLB0: ENTRY TO SCROLL

OTHER VARIABLES

57389 17        BOTLN     DEFB 23: BOTTOM LINE +1 LINE  AT  WHICH
  ;SCREEN IS CONSIDERED FULL. AUTO SCROLL IF SCRLCT <>0
57390 00        SCRLCT    DEFB 1: SCROLL CT-1 + # OF TIMES AUTO
57391 003C      CHTBL     DEFW CHRSET: ADDR OF CHAR TB  SCROLL
57393 E0E5      GRTBL     DEFW GRPHST: ADDR OF UDG TABLE
57395 20        LINLEN    DEFB 32: LINE LENGTH
57396 2118      CURPOS    DEFW #1821: CURSOR POSN INTERNAL
57398 0040      DFADRS    DEFW #4000: CURRENT DISPAY ADDR
57400 00        MASKB     DEFB 0: MASK FOR CHAR
57401 00        ATTBYT    DEFB 0: CURRENT ATTR (BLACK ON WHITE)
57402 80        GTINDX    DEFB 128: INDEX ADJ CHCODE IN GETCHAR
57403 0000      STRGCT    DEFW 0: REMAINING CT FROM WRSTR WHEN
  ;SCREEN FULL
57405 0000      SPARE2    DEFW 0
57407 00        ATTMSK    DEFB 0
57408 C3DAE3    COPYSB    JP COPYS0: ENTRY TO COPY SCREEN
57411 C3F5E4    EXCHSB    JP EXCHB0: ENTRY TO  EXCHANGE SCREENS

WRITE CHARACTER

;HERE FROM BASIC TO DISPLAY CHAR IN DATAB
57414 3A00E0    WRCH0     LD A,(DATAB)
;DISPLAY CHAR IN DATAB
57417 CDA2E4    WRCHAR    CALL LDATTR: GET ATTR AND MASK
57420 CD5DE6    WRCH01    CALL LDPOSN: LD BC=CURPOS HL=DFADRS
57423 FE20                CP 32: TEST VALID RANGE
57425 DAC4E0              JP C, INVPAR
57428 FEA5                CP 165
57430 D2C4E0              JP NC, INVPAR
57433 C5                  PUSH BC: SAVE CURSOR POSN
57434 FE80                CP 128: ASCII PRINTABLE?
57436 3815                JR C, WRCHR12
57438 FE90                CP 144: STD UDG CHAR?
57440 3809                JR C, WRCHR11
57442 ED4B7B5C            LD BC, (UDG): DO UDG
57446 05                  DEC B
57447 D670                SUB 112: ADJUST FOR INDEX
57449 180C                JR WRCH13
57451 ED4B31E0  WRCH11    LD BC, (GRTBL): ADDR OF GRAPHICS TABLE
57455 D660                SUB 96: ADJUST FOR INDEX INTO TABLE
57457 1804                JR WRCH13
57459 ED4B2FE0  WRCH12    LD BC, (CHTLB): CHARACTER TABLE
57463 EB        WRCH13    EX DE, HL
57464 2600                LD H, 0
57466 6F                  LD L, A
57467 29                  ADD HL, HL
57468 29                  ADD HL, HL
57469 29                  ADD HL, HL
57470 09                  ADD HL, BC
```

```
57471 C1                POP BC
57472 EB                EX DE, HL
57473 79                LD A, C: TEST IF END OF LINE
57474 3D                DEC A
57475 3A33E0            LD A, (LINLEN)
57478 2005              JR NZ, WRCH14
57480 3C                INC A: START OF NEW LINE
57481 05                DEC B: ALSO BUMP B TO NEXT LINE
57482 4F                LD C, A
57483 1801              JR WRCH15
57485 3C        WRCH14  INC A
57486 B9        WRCH15  CP C: TEST IF START OF LINE
57487 D5                PUSH DE: TEST IF OFF SCREEN
57488 CCC8E0            CALL Z, TVFULO
57491 D1                POP DE
:HERE TO PUT CHAR IN DFILE
57492 C5        WRCH2   PUSH BC: SAVE CUR POSN
57493 E5                PUSH HL: AND DFILE ADDR
57494 3A38E0            LD A, (MASKB)
57497 06FF              LD B, 255: ASSUME OVER ON
57499 1F                RRA: TEST FOR OVER
57500 3801              JR C, WRCH3
57502 04                INC B: TURN OVER OFF
57503 1F        WRCH3   RRA
57504 1F                RRA: TEST FOR INV
57505 9F                SBC A, A
57506 4F                LD C, A: C=255 IF INV, ELSE 0
57507 3E08              LD A, 8: COUNTER
57509 A7                AND A: CLEAR FLAGS
57510 EB                EX DE, HL
57511 08        WRCH5   EX AF, AF': SAVE COUNTER
57512 1A                LD A, (DE): GET OLD PIXEL ROW
57513 A0                AND B: ADD OVER, A=255 IF OFF
57514 AE                XOR (HL): ADD IN NEW PIXEL
57515 A9                XOR C: INVERT IF ON
57516 12                LD (DE), A: PRINT IT
57517 08                EX AF, AF': GET COUNTER BACK
57518 14                INC D: UPDATE POINTERS-NEXT PIXEL ROW
57519 23                INC HL
57520 3D                DEC A
57521 20F4              JR NZ, WRCH5
57523 15        WRCH7   DEC D: ADJUST DFILE ADDR
57524 EB                EX DE, HL: DFILE ADDR TO HL
57525 CD65E6            CALL UPDATT: UPDATE ATTR FILE
57528 E1                POP HL: STARTING ADDR
57529 C1                POP BC: CURSOR POSN
57530 0D                DEC C: ADJUST CUR POSN
57531 23                INC HL
57532 CD55E6    WRCHXT  CALL STPOSN
57535 0E00      GOODRET LD C, 0: RET BC=0
57537 0600      ERRRET  LD B, 0: ENTER HERE WITH C <>0
57539 C9                RET
57540 0E01      INVPAR  LD C, 1:RET BC=1 FOR INVALID PARAM
57542 18F9              JR ERRRET
```

```
57544 3E18      TVFUL0    LD A, SCRSZ
57546 90                  SUB B: A = LINE #
57547 57                  LD D, A: SAVE IN D
57548 3A2DE0              LD A, (BOTLN): GET BOTTOM LINE
57551 BA                  CP D
57552 D228E6              JP NC, UPDPOSN: OK, RET TO WRITE CHAR
  :VIA UPDATE AND STORE POSN
57555 3A2EE0              LD A, (SCRLCT): TEST SCROLL COUNT
57558 3D                  DEC A
57559 200B                JR NZ, TVFUL1: AUTOSCROLL WITH COUNT
57561 3E01                LD A, 1
57563 322EE0              LD (SCRLCT), A: RESET SCROLL CT
57566 C1                  POP BC: DISCARD RETURN TO WRITE CHAR
57567 C1                  POP BC
57568 0E03                LD C, 3: RET BC=3 FOR SCREEN FULL
57570 18DD                JR ERRRET
57572 322EE0    TVFUL1    LD (SCRLCT), A
57575 ED4B28E0            LD BC, (SCRCTL): GET STARTING LINE/CT
57579 C341E1              JP SCRL0

WRITE STRING

:HERE FROM BASIC ENTRY TO WRITE CHARACTER STRING TO SCREEN
:STRING IDENTIFIER (CH.CODE) IN SYS VAR PARAMS AT 5CC3

57582 CDB8E6    WRSTR0    CALL GETSTRG
57585 C8                  RET Z: NULL OR STRING NOT FOUND
:ENTRY FROM M/C W ADDRESS OF CHAR.CODE STRING IN HL, LEN IN BC
57586 CDA2E6    WRSTRG    CALL LDATTR: GET ATTR & MASK CTLS
57589 7E        WRSTLP    LD A, (HL): GET CODE
57590 E5                  PUSH HL: SAVE ADDR
57591 C5                  PUSH BC: SAVE LEN
57592 CD4CE0              CALL WRCH01
57595 78                  LD A, B: TEST IF GOOD
57596 B1                  OR C
57597 2009                JR NZ, WRSTX1: EXIT IF INVALID
57599 C1        WRSTR3    POP BC: COUNT
57600 E1                  POP HL: ADDR
57601 23                  INC HL: NEXT CHAR
57602 0B                  DEC BC: ADJ CT
57603 78                  LD A, B: DONE?
57604 B1                  OR C
57605 20EE                JR NZ, WRSTLP: WRITE NEXT CHAR
57607 C9                  RET: BC=0
57608 79        WRSTX1    LD A, C: SAVE ERROR
57609 FE01                CP 1: TEST IF UNRECOGNIZABLE CODE
57611 28F2                JR Z, WRSTR3
57613 E1                  POP HL: REMAINING COUNT TO HL
57614 223BE0              LD (STRGCT), HL: STORE
57617 E1                  POP HL: ADDR
57618 0E03                LD C, 3: RET BC=3
57620 0600      WRSTX2    LD B, 0
57622 C9                  RET
```

POSITION CONTROL

```
:HERE FOR BASIC ENTRY. PARAMETERS IN LINCOL
57623 ED4B07E0 SETCBO    LD BC, (LINCOL)
:HERE IF LINCOL IN BC
57627 CD0CE6   SETCUR    CALL TSTPAR
57630 CD1DE6             CALL CONVFM: CONVERT INTERNAL FORMAT
57633 CD28E6             CALL UPDPOSN: UPDATE POSN
57636 C3BFE0             JP GOODRET
```

SCROLL

```
:HERE FROM BASIC TO SCROLL SCREEN
57639 ED4B28E0 SCRLBO    LD BC, (SCRCTL): GET CONTROL INFO
:HERE WITH CONTROL IN BC: B=# OF LINES, C=STARTING LINE #
57643 78       SCROLL    LD A, B: TEST VALIDITY
57644 A7                 AND A
57645 CAC4E0             JP Z, INVPAR: ERR IF COUNT = 0
57648 81                 ADD A, C
57649 FE01               CP 1: ERR IF B+C=1
57651 DAC4E0             JP C, INVPAR
57654 FE19               CP 25: ERR IF B+C>24
57656 D2C4E0             JP NC, INVPAR
57659 CD41E1             CALL SCRLO: DO IT
57662 C3BFE0             JP GOODRET: RET BC=0
57665 C5       SCRLO     PUSH BC
57666 3E18               LD A, 24: GET STARTING LINE INT FORMAT
57668 91                 SUB C
57669 47                 LD B, A
57670 3A33E0             LD A, (LINLEN)
57673 3C                 INC A: COL 0
57674 4F                 LD C, A
57675 CD3AE6             CALL LNBO
57678 C1                 POP BC: ORIG BC
58679 E5                 PUSH HL
58680 C5                 PUSH BC: SAVE AGAIN
57681 7D       TESTAO    LD A, L: TEST IF START OF BLOCK
57682 A7                 AND A
57683 285C               JR Z,STBLK
57685 07                 RLCA
57686 07                 RLCA
57687 07                 RLCA
57688 4F                 LD C, A: GET # OF LINES
57689 3E08               LD A, 8
57691 91                 SUB C
57692 B8                 CP B: TEST AGAINST TOTAL LINES
57693 384A               JR C, GRBLK: TOTAL >BLOCK
57695 78       INSDBLK   LD A, B
57696 0600               LD B, 0: REMAINING COUNT
57698 C5                 PUSH BC
57699 47       LOOP      LD B, A: B=LINES THIS BLOCK
57700 0E08               LD C, 8: SCAN LINES
57702 78       LOOPO     LD A, B
57703 C5                 PUSH BC
```

```
57704 OF                RRCA
57705 OF                RRCA
57706 OF                RRCA: CALC # OF BYTES
57707 4F                LD C, A
57708 0600              LD B, 0: BC=32** OF LINES
57710 EB        LOOP1   EX DE, HL
57711 21E0FF            LD HL,#FFE0: GET ADDR OF PREV LINE
57714 19                ADD HL, DE
57715 EB                EX DE, HL: TO DE
57716 E5                PUSH HL: SAVE ADDR
57717 EDB0              LDIR: DO A LINE
57719 E1                POP HL
57720 24                INC H
57721 C1                POP BC
57722 0D                DEC C
57723 20E9              JR NZ, LOOP0: MORE SCANS
57725 C1                POP BC: RESIDUAL LINE COUNT
57726 78                LD A, B
57727 A7                AND A
57728 2804              JR Z, SCRLXT: DONE
57730 2E00              LD L, 0: START OF NEXT BLOCK
57732 18CB              JR TESTA0
57734 C1        SCRLXT  POP BC: ORIG BC
57735 D1                POP DE: STARTING ADDR
57736 C5                PUSH BC: SAVE ORIG BC
57737 68                LD L, B: # OF LINES SCROLLED
57738 2600              LD H, 0
57740 29                ADD HL, HL: 32** OF LINES
57741 29                ADD HL, HL
57742 29                ADD HL, HL
57743 29                ADD HL, HL
57744 29                ADD HL, HL
57745 44                LD B, H: COUNT TO BC
57746 4D                LD C, L
57747 62                LD H, D: ADDR TO HL
57748 6B                LD L, E
57749 CD92E6            CALL CALCATT: ATTR ADDR TO HL
57752 EB                EX DE, HL
57753 21E0FF            LD HL, 65504: -32
57756 18                ADD HL, DE
57757 EB                EX DE, HL
57758 EDB0              LDIR: SCROLL ATTR
57760 C1                POP BC: ORIG BC
57761 79                LD A, C: STARTING LINE
57762 80                ADD A, B: ADD # OF LINES MOVED
57763 3D                DEC A: LAST LINE SCROLLED
57764 4F                LD C, A
57765 0601              LD B, 1: CLEAR LINE 1
57767 1845              JR CLRSC0: CLEAR VACATED LINE AND RET
57769 4F        GRBLK   LD C, A
57770 78                LD A, B
57771 91                SUB C: ADJ TOT LN CT BY # LNS THIS BLK
57772 47                LD B, A
57773 C5                PUSH BC
```

```
57774 79                    LD A, C: LOAD # OF LN THIS BLOCK
57775 18B2                  JR LOOP
57777 05        STBLK       DEC B: DEC LINE CT
57778 C5                    PUSH BC
57779 0E08                  LD C, 8
57781 C5        STBLK0      PUSH BC: SAVE COUNT
57782 EB        STBLK1      EX DE, HL
57783 21E0F8                LD HL, #F8E0:0729H
57786 19                    ADD HL, DE
57787 EB                    EX DE, HL: ADDRS OF  PREV LINE
57788 012000                LD BC, 32
57791 E5                    PUSH HL
57792 EDB0                  LDIR: DO MOVE
57794 E1                    POP HL
57795 24                    INC H: NEXT SCAN  LINE
57796 C1                    POP BC
57797 0D                    DEC C
57798 20ED                  JR NZ, STBLK0
57800 C1                    POP BC: REMAINING LINE COUNT
57801 78                    LD A, B
57802 A7                    AND A
57803 28B9                  JR Z, SCRLXT
57805 1120F8                LD DE, #F820
57808 19                    ADD HL, DE
57809 C351E1                JP TESTA0: BC=LINE CT, HL=DF ADDR

CLEAR SCREEN

:HERE FROM BASIC ENTRY TO CLEAR SCREEN
57812 ED4B0CE0 CLRSB0     LD BC, (CLRCTL): GET CONTROL INFO
:HERE IF BC=LINE COUNT AND STARTING LINE #
57816 78        CLRSCN      LD A, B
57817 A7                    AND A: TEST VALIDITY
57818 CAC4                  JP Z, INVPAR
57821 81                    ADD A, C
57822 FE01                  CP 1
57824 DAC4E0                JP C, INVPAR
57827 FE19                  CP 25
57829 D2C4E0                JP NC, INVPAR
57832 CDEEE1                CALL CLRSC0: DO IT
57835 C3BFE0                JP GOODRET: BC=0
57838 CDA2E6    CLRSC0      CALL LDATTR: GET ATTR CTL
57841 C5                    PUSH BC
57842 3E18                  LD A, 24
57844 91                    SUB C
57845 47                    LD B, A
57846 3A33E0                LD A, (LINLEN)
57849 3C                    INC A: SET TO COL 0
57850 4F                    LD C, A
57851 CD3AE6                CALL LNB0
57854 50                    LD D, B
57855 59                    LD E, C: SAVE POSN
57856 C1                    POP BC: ORIG BC
57857 D5                    PUSH DE
```

```
57858 7D       CLRSC1   LD A,L: GET # OF LINES THIS BLOCK
57859 07                RLCA
57860 07                RLCA
57861 07                RLCA
57862 4F                LD C, A
57863 3E08              LD A, 8: COUNT
57865 91                SUB C
57866 B8                CP B
57867 3847              JR C, CLRSC7
57869 78       CLRSC2   LD A, B: # FO LINES
57870 0600              LD B, 0: REMAINING LINES
57872 C5                PUSH BC
57873 47       CLRSC3   LD B, A
57874 0E08              LD C, 8
57876 78       CLRSC4   LD A, B
57877 C5                PUSH BC
57878 E607              AND 7
57880 0F                RRCA
57881 0F                RRCA
57882 0F                RRCA
57883 4F                LD C, A: BC=32** OF LINES
57884 0600              LD B, 0: C=0 IF 256 FOR 8 LINES
57886 0D                DEC C
57887 E5       CLRSC5   PUSH HL
57888 54                LD D, H
57889 5D                LD E, L
57890 3600              LD (HL), 0: CLEAR DF1
57892 13                INC DE
57893 EDB0              LDIR
57895 E1                POP HL
57896 24                INC H: NEXT SCANLINE
57897 C1                POP BC
57898 0D                DEC C
57899 20E7              JR NZ, CLRSC4
57901 E5                PUSH HL: SAVE NEXT ADDR
57902 25                DEC H: LAST SCAN ROW
57903 CD92E6            CALL CALCATT: GET ADDR IN ATTR FILE
57906 78                LD A, B: # LINES THIS BLOCK
57907 E607              AND 7
57909 0F                RRCA
57910 0F                RRCA
57911 0F                RRCA
57912 4F                LD C, A: BC =32** OF LINES
57913 0600              LD B, 0
57915 0D                DEC C: ADJUST
57916 54                LD D, H
57917 5D                LD E, L
57918 3A39E0            LD A, (ATTBYT): STORE VALUE
57921 77                LD (HL), A: UPDATE ATTR FILE
57922 13                INC DE
57923 EDB0              LDIR
57925 E1                POP HL
57926 C1                POP BC: TOTAL LINE COUNT
57927 78                LD A, B
```

```
57928 A7                    AND A
57929 2805                  JR Z, CLRSXT
57931 2E00                  LD L, 0: HL=START OF NEXT BLOCK
57933 C302E2                JP CLRSC1: B=REMAINING LINE COUNT
57936 C1        CLRSXT      POP BC: POSITION
57937 C328E6                JP UPDPOSN: RET VIA UPDATE & ST POSN
57940 4F        CLRSC7      LD C, A: SAVE # LINES THIS BLOCK
57941 78                    LD A, B
57942 91                    SUB C: ADJ LINE COUNT
57943 47                    LD B, A
57944 C5                    PUSH BC: REMAINING LINE COUNT
57945 79                    LD A, C: REMAINING LINE CT
57946 18B5                  JR CLRSC3

GET ROUTINES

:HERE FROM BASIC ENTRY TO GET CHARACTER
:(RETURNS CODE FOR CHARACTER AT POSITION IN GETCTL)
57948 ED4B19E0 GTCHBO       LD BC, (GETCTL): GET CONTROL INFO
:HERE IF CONTROL IN BC
57952 CD0CE6    GTCHAR      CALL TSTPAR: TEST PARAMETERS
57955 CD1DE6                CALL CONVFM: CONVERT TO INT FORMAT
57958 CD2DE6                CALL CALCPOS: GET DFILE ADDR
57961 ED5B2FE0              LD DE, (CHTBL): CHAR TABLE
57965 0660                  LD B, 96: # OF PRINTABLE CHARS
57967 3E80                  LD A, 128: SET ADJ INDEX
57969 323AE0    GTCH1       LD (GTINDX), A
57972 14                    INC D: ADJ TO START OF TABLE
57973 EB                    EX DE, HL: TO HL
57974 C5        GTCH2       PUSH BC: SAVE COUNT
57975 D5                    PUSH DE: SAVE DF ADDR
57976 E5                    PUSH HL: SAVE ADDR CHAR TABLE
57977 1A                    LD A, (DE): SCAN ROW FROM DF
57978 AE                    XOR (HL): TEST AGAINST CHAR SET
57979 2810                  JR Z, GTCH3: MATCH
57981 F5                    PUSH AF
57982 3A3AE0                LD A, (GTINDX)
57985 FE90                  CP 144: STANDARD GRAPHIC?
57987 2003                  JR NZ, GTCH23
57989 F1                    POP AF
57990 1827                  JR GTCH4: DON'T TEST INVERSE
57992 F1        GTCH23      POP AF
57993 3C                    INC A: TEST INVERSE
57994 2023                  JR NZ, GTCH4: OFF
57996 3D                    DEC A: A =1 FOR INVERSE
57997 4F        GTCH3       LD C, A: C=0 FOR MATCH, -1 FOR INVERSE
57998 0607                  LD B, 7
58000 14        GTCH31      INC D: NEXT SCAN IN DF
58001 23                    INC HL: NEXT CHAR BYTE
58002 1A                    LD A, (DE)
58003 AE                    XOR (HL)
58004 A9                    XOR C
58005 2018                  JR NZ, GTCH4: NO MATCH
58007 10F7                  DJNZ GTCH31: NEXT SCAN
```

```
:HERE IF MATCH WITH (SP)=CHAR IN CHAR SET
:(SP+2)=CHAR IN DF, (SP+5)=CHAR COUNT
58009 79                    LD A, C: A=-1 IF MATCH ON INVERSE
58010 C1                    POP BC
58011 C1                    POP BC
58012 C1                    POP BC: CLEAR STACK
58013 4F                    LD C, A: B=COUNT C=-1 IF MATCH ON INV
58014 3A3AE0                LD A, (GTINDX): ADJ CHAR CODE
58017 90                    SUB B: A=CHAR CODE
58018 FE20                  CP 32: SPACE?
58020 2005                  JR NZ, GTCH32
58022 0C                    INC C: INVERSE SPACE?
58023 2002                  JR NZ, GTCH32
58025 3E8F                  LD A, 143: SET CODE FOR GRAPHIC BLOCK
58027 4F        GTCH32      LD C, A
58028 0600                  LD B, 0
58030 C9                    RET: CHAR CODE IN A AND BC
58031 E1        GTCH4       POP HL: HERE WHEN NO MATCH
58032 110800                LD DE, 8: LOOK AT NEXT CHAR IN SET
58035 19                    ADD HL, DE
58036 D1                    POP DE: DFILE ADDR
58037 C1                    POP BC: CHAR COUNT
58038 10BE                  DJNZ GTCH2: LOOK AGAIN
:HERE TO TEST IF DONE
58040 EB                    EX DE, HL: DFILE ADDR TO HL
58041 3A3AE0                LD A, (GTINDX)
58044 FE80                  CP 128: STANDARD CHAR?
58046 200A                  JR NZ, GTCH5: TRY GRAPHICS
58048 ED5B31E0              LD DE, (GRTBL): STD GRAPHIC TABLE
58052 0610                  LD B, 16: # OF ENTRIES
58054 3E90                  LD A, 144: INDEX
58056 18A7                  JR GTCH1
58058 FE90      GTCH5       CP 144: STD GRAPHIC?
58060 200B                  JR NZ, GTCH6: 'DONE IF NOT
58062 ED5B7B5C              LD DE, (UDG): TRY UDG GRAPHIC
58066 15                    DEC D: ADJ START
58067 0615                  LD B, 21: # OF ENTRIES
58069 3EA5                  LD A, 165: INDEX
58071 1898                  JR GTCH1
58073 48        GTCH6       LD C, B: HERE WHEN NO MATCH ANYWHERE
58074 AF                    XOR A: SET BC & A=0
58075 C9                    RET

GET ATTR BYTE

:HERE FROM BASIC ENTRY TO GET ATTR
58076 ED4B19E0 GETABO       LD BC, (GETCTL)
:HERE WITH LINCOL IN BC
58080 CD0CE6    GETATTR     CALL TSTPAR: TEST PARAMETER
58083 CD1DE6                CALL CONVFM: CONVERT TO INT FORMAT
58086 CD2DE6                CALL CALCPOS
58089 CD92E6                CALL CALCATT
58092 7E                    LD A, (HL): ATTR IN A
58093 4F                    LD C, A: SAVE IN C
```

```
58094 0600                LD B, 0
58096 C9                  RET

GET CURSOR POSITION

:HERE FROM BASIC ENTRY
58097 ED4B34E0 GETCB0     LD BC, (CURPOS): GET INTERNAL POSN
58101 CD1DE6              CALL CONVFM: CONVERT TO USER FORMAT
58104 3A33E0              LD A, (LINLEN)
58107 B9                  CP C: END OF LISTING?
58108 200B                JR NZ, GETC1: NO
58110 0E00                LD C, 0: START OF NEXT LINE
58112 78                  LD A, B
58113 3C                  INC A: BUMP TO NEXT LINE
58114 47                  LD B, A
58115 FE18                CP 24:              RET

GET CURSOR POSITION

:HERE FROM BASIC ENTRY
58097 ED4B34E0 GETCB0     LD BC, (CURPOS): GET INTERNAL POSN
58101 CD1DE6              CALL CONVFM: CONVERT TO USER FORMAT
58104 3A33E0              LD A, (LINLEN)
58107 B9                  CP C: END OF LISTING?
58108 200B                JR NZ, GETC1: NO
58110 0E00                LD C, 0: START OF NEXT LINE
58112 78                  LD A, B
58113 3C                  INC A: BUMP TO NEXT LINE
58114 47                  LD B, A
58115 FE18                CP 24: OFF SCREEN?
58117 3802                JR C, GETC1: NO
58119 0617                LD B, 23: POSN ON BOTTOM LINE
58121 ED4307E0 GETC1      LD (LINCOL), BC: SAVE
58125 C9                  RET: VALUE ALSO IN BC

VIDEO MODE CONTROL

:HERE FROM BASIC ENTRY WITH MODE IN VIMODE
58126 3A24E0   STMDB0     LD A, (VIMODE)
:HERE IF VIDMODE IN A
58129 47       SETMODE    LD B, A: STORE IN B
58130 A7                  AND A
58131 2815                JR Z, SETMD0: TEST VALIDITY
58133 FE80                CP 128
58135 2811                JR Z, SETMD0
58137 FE01                CP 1
58139 280D                JR Z, SETMD0
58141 FE04                CP 4: HANDLE 4 AND 5
58143 CABAE3              JP Z, FRFV
58146 FE05                CP 5
58148 CABAE3              JP Z, FRFV
58151 C3C4E0              JP INVPAR
58154 3AC25C   SETMD0     LD A, (VIDMOD)
58157 A7                  AND A
```

```
58158 205A                JR NZ, SETMD2: DF2 OPEN
58160 B0                  OR B
58161 CABFE0              JP Z, GOODRET
:INITIAL OPENING OF 2ND FILE
58164 210117    SETMD1    LD HL, SCINIT:INITIALIZE PARAMETERS
58167 2228E0              LD (SCRCTL), HL: SCROLL CONTROL
58170 3E17                LD A, 23
58172 322DE0              LD (BOTLN), A: BOTTOM LINE
58175 3E01                LD A, 1
58177 322EE0              LD (SCRLCT), A: SCROLL COUNT
58180 3E20                LD A, 32
58182 3233E0              LD (LINLEN), A: LINE LENGTH
58185 210018              LD HL, CLINIT
58188 220CE0              LD (CLRCTL), HL: CLEAR SCREEN CONTROL
58191 AF                  XOR A: SET A=0
58192 323BE0              LD (STRGCT), A: WRITE STRING REMAINING
58195 323CE0              LD (STRGCT+1), A: COUNT
58198 21003C              LD HL, CHRSET: STD CHAR TABLE
58201 222FE0              LD (CHTBL), HL
58204 21E0E5              LD HL, GRPHST: STD GRAPHIC SET
58207 2231E0              LD (GRTBL), HL
:INITIALIZE PARAMETERS
58210 3200E0              LD (DATAB), A: DATA BYTE
58213 3207E0              LD (LINCOL), A: COLLUMN
58216 3208E0              LD (LINCOL+1) A: LINE
58219 3219E0              LD (GETCTL), A: GET COLUMN
58222 321AE0              LD (GETCTL+1), A: GET LINE
58225 21C012              LD HL, INSERTSZ
58228 114008              LD DE, MOVESZ
58231 19                  ADD HL, DE: TOTAL ROOM NEEDED
58232 ED5B655C            LD DE, (STKEND)
58236 19                  ADD HL, DE: ADD MEMORY IN USE
58237 DAB5E3              JP C, EXTNRM: NOT ENOUGH MEMORY
58240 ED5BB25C            LD DE, (RAMTOP)
58244 A7                  AND A
58245 ED52                SBC HL, DE
58247 DAB5E3              JP NC, EXTNRM: NOT ENOUGH MEMORY
58250 78        SETMD2    LD A, B: MODE TO A
58251 CDF3E5              CALL ENBLEXT: ENABLE ROM EXT
58254 CDCDE5              CALL GETVAL: H/W VAL TO B, VIDMOD TO C
58257 C5                  PUSH BC: SAVE
58258 78                  LD A, B
58259 A7                  AND A: ZERO?
58260 2001                JR NZ, SETM21
58262 79                  LD A, C
58263 CD8EE0    SETM21    CALL CHNGVID
58266 CD00E6              CALL ENBLHOME
58269 C1                  POP BC
58270 79                  LD A, C
58271 32C25C    SETMD3    LD (VIDMOD), A: UPDATE VIDMODE
58274 3A33E0    UPDATE    LD A, (LINLEN)
58277 3C                  INC A
58278 4F                  LD C, A: COLUMN LEN
58279 0618                LD B, 24
```

```
58281 210000              LD HL, 0: BC= 0, HOME POSN
58284 227D5C              LD (COORDS), HL: INIT PLOT POSN
58287 CD28E6              CALL UPDPOSN: UPDATE & STORE HOME POSN
58290 C3BFE0              JP GOODRET
58293 0E02      EXTNRM    LD C, 2
58295 C3C1E0              JP ERRRET
58298 3AC25C    FRFV      LD A, (VIDMOD): ONLY VALID IF 80,81,84
58301 4F                  LD C, A: SAVE IN C
58302 CB7F                BIT 7, A
58304 CAC4E0              JP Z, INVPAR
58307 78                  LD A, B: MODE 4 OR 5 TO A
58308 FE04                CP 4
58310 200A                JR NZ, DTADF2: DATA TO DF2
58312 CB41      DTADF1    BIT 0, C: ALREADY DESTINATION?
58314 CABFE0              JP Z, GOODRET: NO ACTION NECESSARY
58317 A9                  XOR C: SET VIDMODE TO 80 OR 84
58318 E6FE                AND 254: BIT 0=0
58320 18CD                JR SETMD3: UPDATE VARIABLES
58322 CB41      DTADF2    BIT 0, C: DF2 OPEN?
58324 C2BFE0              JP NZ, GOODRET
58327 A9                  XOR C: VIDMODE TO 81 OR 85
58328 18C5                JR SETMD3: UPDATE VARIABLES
```

COPY SCREEN ROUTINE

```
:COPY LINES WITHIN OR BETWEEN DFILES
:HERE FORM BASIC ENTRY
58330 2A5D5C    COPYS0    LD HL, (CHRADD): SAVE CURRENT POSN
58333 E5                  PUSH HL
58334 CDB8E6              CALL GETSTRG: GET STRING OF PARAMS
58337 2009                JR NZ, COPY01: FOUND
58339 E1                  POP HL: RETURN AS NOT FOUND
58340 225D5C              LD (CHRADD), HL
58343 C9                  RET
58344 00000000  COPYWK    DEFS 4: WORK AREA FOR BASIC PARAMS
58348 225D5C    COPY01    LD (CHRADD), HL: STR ADDR TO CH-ADD
58351 CD18E4              CALL GETNO
58354 32E9E3              LD (COPYWK+1), A: SOURCE LINE #
58357 CD18E4              CALL GETNO
58360 32E8E3              LD (COPYWK), A: DEST LINE #
58363 CD18E4              CALL GETNO
58366 32EBE3              LD (COPYWK+3), A: SOURCE DF
58369 CD18E4              CALL GETNO
58372 32EAE3              LD (COPYWK+2), A: DEST DF
58375 CD18E4              CALL GETNO
58378 E1                  POP HL
58379 225D5C              LD (CH_ADD), HL: RESTORE CH_ADD
58382 ED4BE8E3            LD BC, (COPYWK)
58386 ED5BEAE3            LD DE, (COPYWK+2)
58390 1825                JR COPYSC
58392 2A5D5C    GETNO     LD HL, (CHRADD): LOAD ADDR
58395 7E                  LD A, (HL): GET CHAR FROM STR
58396 FE2C                CP 44: COMMA?
58398 280F                JR Z, GETN2
```

```
58400 FE30              CP 48: DIGIT?
58402 3811              JR C, COPERR: INVALID
58404 FE3A              CP 58
58406 300D              JR NC, COPERR: END OF STR
58408 CDF930   GETN1    CALL ININT
58411 CD9331            CALL FP2A
58414 D0                RET NC
58415 23       GETN2    INC HL
58416 225D5C            LD (CH_ADD), HL
58419 18E3              JR GETN0
58421 E1       COPERR   POP HL: DISCARD RET
58422 E1       COPER1   POP HL: RESTORE CH_ADD
58423 225D5C            LD (CH_ADD), HL
58426 C3C4E0            JP INVPAR
:ENTRY WITH PARAM IN REGISTERS
58429 A7       COPYSC   AND A
58430 CAC4E0            JP Z, INVPAR: TEST PARAM
58433 FE19              CP 25: ERR OF LINE CT=0
58435 D2C4E0            JP NC, INVPAR
58438 F5                PUSH AF
58439 CDABE5            CALL GTADRS: GET DF ADDR FOR SOURCE
58442 DAC4E0            JP C, INVPAR
58445 E5                PUSH HL: SAVE SOURCE ADDR
58446 41                LD B, C: DEST LINE
58447 53                LD D, E: DEST DF
58448 CDA8E5            CALL GTADRS: GET DF ADDR FOR DEST
58451 DAC4E0            JP C, INVPAR
58454 D1                POP DE: SOURCE ADDR
58455 EB                EX DE, HL: DE=DEST, HL=SOURCE
58456 F1                POP AF: GET COUNT
58457 47                LD B, A
58458 E5                PUSH HL: SAVE STARTING ADDR
58459 D5                PUSH DE
58460 C5                PUSH BC:AND ORIG LINE CT
58461 7D       COPYS1   LD A, L: CALC # OF LINES SOURCE BLOCK
58462 07                RLCA
58463 07                RLCA
58464 07                RLCA
58465 4F                LD C, A
58466 3E08              LD A, 8
58468 91                SUB C
58469 4F                LD C, A: # LINES THIS BLOCK
58470 C5                PUSH BC: SAVE # LINES,# LINES SOURCE B
58471 B8                CP B: COMPARE TO TOTAL
58472 3871              JR C, COPYS5: TOTAL >THIS BLOCK
58474 7B                LD A, E: CALC # LINES DEST BLOCK
58475 07                RLCA
58476 07                RLCA
58477 07                RLCA
58478 4F                LD C, A
58479 3E08              LD A, 8
58481 91                SUB C: A=# LINES DEST BLOCK
58482 C1                POP BC: B=TOTAL LINES, C=# LINES DEST
58483 4F                LD C, A
```

```
58483 B8                CP B
58484 3879              JR C, COPYS7: USE DEST LINE CT
58487 78        COPYS2  LD A, B: COPY WITHIN BOTH S & D BLOCKS
58488 0600              LD B, 0: REMAINING LINES
58490 C5                PUSH BC: SAVE
58491 47        COPYS3  LD B, A: # OF LINES
58492 0E08              LD C, 8: SCAN COUNT
58494 78        COPYS4  LD A. B: # OF LINES
58495 C5                PUSH BC: SAVE
58496 E607              AND 7
58498 0F                RRCA
58499 0F                RRCA
58500 0F                RRCA
58501 4F                LD C, A: BC=32*(# LINES)
58502 0600              LD B, 0
58504 A7                AND A: ZERO?
58505 2002              JR NZ,COPY41
58507 0601              LD B,1
58509 E5        COPY41  PUSH HL: SAVE ADDR
58510 D5                PUSH DE
58511 EDB0              LDIR: COPY BC BYTES
58513 D1                POP DE: RESTORE ADDRS
58514 E1                POP HL
58515 14                INC D: NEXT SCAN LINE
58516 24                INC H
58517 C1                POP BC: LINE CT
58518 0D                DEC C
58519 20E5              JR NZ,COPYS4: NEXT SCAN ROW
58521 78                LD A,B: SAVE # JUST MOVED
58522 C1                POP BC: LINE CT LEFT
58523 4F                LD C,A
58524 78                LD A,B
58525 A7                AND A: ZERO?
58526 2822              JR Z,COPYXT: DONE
58528 C5                PUSH BC: LINE CT LEFT
58529 79                LD A,C
58530 E607              AND 7
58532 0F                RRCA
58533 0F                RRCA
58534 0F                RRCA
58535 4F                LD C,A
58536 0600              LD B,0
58538 A7                AND A
58539 2002              JR NZ,COPY42
58541 0601              LD B,1
58543 25        COPY42  DEC H: ADJ ADDRS TO START
58544 09                ADD HL,BC
58545 3E78              LD A,120
58547 A4                AND H
58548 67                LD H,A
58549 E5                PUSH HL: SAVE ADJ SOURCE ADDR
58550 15                DEC D
58551 EB                EX DE,HL: DEST ADDR TO HL
58552 09                ADD HL,BC
```

```
58553 3E78              LD A,120
58555 A4                AND H
58556 67                LD H,A
58557 D1                POP DE: SCORCE ADDR TO DE
58558 EB                EX DE,HL: HL=SORCE, DE= DEST
58559 C1                POP BC
58560 189B              JR COPYS1: B= LINE CT LEFT
58562 C1        COPYXT  POP BC: ORIG LINE CT
58563 68                LD L,B: CALC # OF BYTES
58564 2600              LD H,0
58566 29                ADD HL,HL: *32
58567 29                ADD HL,HL
58568 29                ADD HL,HL
58569 29                ADD HL,HL
58570 29                ADD HL,HL
58571 44                LD B,H
58572 4D                LD C,L
58573 E1                POP HL: DEST ADDR TO HL
58574 CD92E6            CALL CALCATT: ADDR OF ATTR FILE
58577 EB                EX DE,HL: DE = DEST ADDR
58578 E1                POP HL: SOURCE ADDR
58579 CD92E6            CALL CALCATT: S ADDR OF ATTR FILE
58582 EDB0              LDIR
58584 C3BFE0            JP GOODRET
58587 41        COPYS5  LD B,C: # LINES SOURCE BLOCK TO B
58588 7B                LD A,E: CALC # LINES DEST BLOCK
58589 07                RLCA
58590 07                RLCA
58591 07                RLCA
58592 4F                LD C,A
58593 3E08              LD A,8
58595 91                SUB C: A= # LINES DEST BLOCK
58596 4F                LD C,A: TO C
58597 B8                CP B: COMPARE TO SOURCE BLOCK
58598 3808              JR C,COPYS7: USE DEST LINE CT
58600 C1                POP BC: B= TOTAL , C= DEST LINE CT
58601 78        COPYS6  LD A,B
58602 91                SUB C
58603 47                LD B,A: LINE CT LEFT
58604 C5                PUSH BC
58605 79                LD A,C: # OF LINES
58606 188B              JR COPYS3: COPY A LINES
58608 79        COPYS7  LD A,C: # LINES DEST BLOCK
58609 C1                POP BC: B= TOTAL, C= DEST LINE CT
58610 4F                LD C,A
58611 18F4              JR COPYS6
```

EXCHANGE SCREENS

```
:EXCHANGE LINES WITHIN OR BETWEEN SCREENS
:HERE FROM BASIC TO GET PARAMS FROM STR INTO REGISTERS
58613 2A5D5C    EXCHB0  LD HL,(CH_ADD): SAVE CURRENT POSN
58616 E5                PUSH HL
58617 CDB8E6            CALL GETSTRG: GET STR OF PARAMETERS
```

```
58620 2005              JR NZ,EXCH01: FOUND
58622 E1                POP HL: EXIT
58623 225D5C            LD (CHRADD),HL: RESTORE POSN
58626 C9                RET
58627 225D5C   EXCH01   LD (CHRADD),HL: ADDR OF STRING
58630 CD18E4            CALL GETNO: GET PARAMETERS
58633 32E8E3            LD (COPYWK),A: SOURCE LINE #
58636 CD18E4            CALL GETNO
58639 32EAE3            LD (COPYWK+2),A: DEST LINE #
58642 CD18E4            CALL GETNO
58645 32E9E3            LD (COPYWK+1),A: SOURCE DF
58648 CD18E4            CALL GETNO
58651 32EBE3            LD (COPYWK+3),A: DEST DF
58654 CD18E4            CALL GETNO: # OF LINES
58657 E1                POP HL: RESTORE CH_ADD
58658 225D5C            LD (CHRADD),HL
58661 ED4BE8E3          LD BC,(COPYWK): LOAD REGISTERS
58665 ED5BEAE3          LD DE,(COPYWK +2)
:B= DF OF SOURCE C= LINE # FOR SOURCE A= # OF LINES TO XFER
:D= DF OF DEST D= LINE # FOR DEST
58669 A7       EXCHSC   AND A: ZERO?
58670 CAC4E0            JP Z,INVPAR
58673 FE19              CP 25: TOO MANY?
58675 D2C4E0            JP NC,INVPAR
58678 F5                PUSH AF: SAVE LINE CT
58679 D5                PUSH DE: SAVE SOURCE 2 INFO
58680 50                LD D, B: DF
58681 41                LD B, C: LINE #
58682 CDA8E5            CALL GTADRS: GET DF ADDR OF SORCE 1
58685 DAC4E0            JP C,INVPAR
58688 D1                POP DE: SORCE 2 INFP
58689 E5                PUSH HL: SAVE SOURCE 1 ADRS
58690 43                LD B,E
58691 CDA8E5            CALL GTADRS: GET DF ADDR OF SORCE 2
58694 DAC4E0            JP C,INVPAR
58697 D1                POP DE: ADDRS IN DE AND HL
58698 F1                POP AF: A= LINE CT
58699 E5                PUSH HL: SOURCE 2 START ADDR FOR EXIT
58700 D5                PUSH DE: SOURCE 1 START ADDR FOR EXIT
58701 F5                PUSH AF: ORIG LINE CT
58702 F5                PUSH AF: WORKING LINE CT
58703 0608     EXCH0    LD B,8: SCAN CT
58705 E5       EXCH1    PUSH HL: SAVE ADDR & LINE CT
58706 D5                PUSH DE
58707 C5                PUSH BC
58708 012000            LD BC, 32: BYTES/LINE
58711 CD93E5            CALL EXCHLP: EXCHANGE BYTES
58714 C1                POP BC: SCAN CT IN B
58715 D1                POP DE: ADDRESSES
58716 E1                POP HL
58717 14                INC D: ADJUST TO NEXT SCAN LINE
58718 24                INC H
58719 10F0              DJNZ EXCH1: DO NEXT ROW
58721 F1                POP AF: LINE CT
```

```
58722 15                DEC D: ADJUST BACK 1 LINE
58723 25                DEC H
58724 3D                DEC A
58725 2812              JR Z, EXCHXT: DONE
58727 F5                PUSH AF: REMAINING LINE CT
58728 012000            LD BC, 32
58731 09                ADD HL, BC
58732 3E78              LD A, 120
58734 A4                AND H
58735 67                LD H, A
58736 EB                EX DE, HL
58737 09                ADD HL, BC
58738 3E78              LD A, 120
58740 A4                AND H
58741 67                LD H, A
58742 EB                EX DE, HL
58743 18D6              JR EXCHO: DO NEXT LINE
58745 F1        EXCHXT  POP AF: ORIG LINE CT
58746 6F                LD L, A
58747 2600              LD H, 0
58749 29                ADD HL, HL: X32
58750 29                ADD HL, HL
58751 29                ADD HL, HL
58752 29                ADD HL, HL
58753 29                ADD HL, HL
58754 44                LD B, H
58755 4D                LD C, L
58756 E1                POP HL: SOURCE 1 ADDR
58757 CD92E6            CALL CALCATT: GET ADDR OF ATTR FILE
58760 EB                EX DE, HL: TO DE
58761 E1                POP HL: SOURCE 2 ADDR
58762 CD92E6            CALL CALCATT: HL= ATT ADDR SOURCE 2
58765 CD93E5            CALL EXCHLP: EXCHANGE ATTR
58768 C3BFE0            JP GOODRET
58771 C5        EXCHLP  PUSH BC: SAVE CT
58772 7E                LD A, (HL): FROM SOURCE 2
58773 4F                LD C, A: TO WORKING REG
58774 1A                LD A, (DE): FROM SOURCE 1
58775 A9                XOR C: XCHANGE VIA XOR
58776 F5                PUSH AF
58777 A9                XOR C: A= DATA DOR SOURCE 2
58778 4F                LD C, A: C= DATA FOR SOURCE 2
58779 F1                POP AF
58780 A9                XOR C: INTERMEDIATE RESULT
58781 12                LD (DE), A
58782 71                LD (HL), C
58783 13                INC DE: ADJUST NEXT BYTE
58784 23                INC HL
58785 C1                POP BC
58786 0B                DEC BC
58787 78                LD A, B
58788 B1                OR C
58789 20EC              JR NZ, EXCHLP
58791 C9                RET
```

```
:SUBROUTINE TO GET DF ADRS OF LINE IN B FOR DF IN D CARRY= ERR
58792 78       GTADRS    LD A, B
58793 FE18               CP 24: VALID?
58795 301E               JR NC, GTERR
58797 0F                 RRCA
58798 0F                 RRCA
58799 0F                 RRCA
58800 E6E0               AND 224
58802 6F                 LD L, A
58803 78                 LD A, B
58804 E618               AND 24
58806 F640               OR 64
58808 67                 LD H, A
58809 7A                 LD A, D
58810 A7                 AND A
58811 280E               JR Z, GTERR: ERR IF NOT 1 OR 2
58813 FE03               CP 3
58815 300A               JR NC, GTERR
58817 CB47               BIT 0, A: WHICH DF?
58819 2004               JR NZ, GTAD1: DF1
58821 7C                 LD  A, H
58822 F620               OR 32: DF2
58824 67                 LD H, A
58825 A7       GTAD1     AND A: RET NO CORRY
58826 C9                 RET
58827 37       GTERR     SCF: SET CARRY FOR ERROR
58828 C9                 RET

INTERNAL SUBROUTINES

58829 4F       GETVAL    LD C, A
58830 CB47               BIT 0, A
58832 2016               JR NZ, TST01
58834 FE80               CP 128
58836 281A               JR Z, SET0
58838 CB7F               BIT 7, A
58840 C0                 RET NZ
58841 47                 LD B, A
58842 FE02               CP 2
58844 C8                 RET Z
58845 E6C6               AND 198
58847 EE06               XOR 6
58849 C0                 RET NZ
58850 3E40               LD A, 64
58852 B1                 OR C
58853 4F                 LD C, A
58854 AF                 XOR A
58855 C9                 RET
58856 FE01     TST01     CP 1
58858 C0                 RET NZ
58859 0601               LD B, 1
58861 0E81               LD C, 129
58863 C9                 RET
58864 AF       SET0      XOR A
```

```
58865 47                    LD B, A
58866 C9                    RET
:ROUTINE TO ENABLE CHUNK 0 IN EXROM TO ACCESS CHNGVID ROUTINE
58867 F5        ENBLEXT     PUSH AF
58868 DBFF                  IN A, (HREXPT)
58870 F680                  OR 128
58872 D3FF                  OUT (HREXPT), A
58874 3E03                  LD A, 3
58876 D3F4                  OUT (DKHSPT), A
58878 F1                    POP AF
58879 C9                    RET

58880 F5        ENBLHOME    PUSH AF
58881 AF                    XOR A: SET A=0
58882 D3F4                  OUT (DKHSPT), A
58884 DBFF                  IN A, (HREXPT)
58886 E63F                  AND 63
58888 D3FF                  OUT (HREXPT), A
58890 F1                    POP AF
58891 C9                    RET

TEST PARAMETER VALIDITY

58892 3E17      TSTPAR      LD A, 23
58894 B8                    CP B
58895 3004                  JR NC, TSTPR1
58897 F1        PARERR      POP AF
58898 C3C4E0                JP INVPAR
58901 3A33E0    TSTPR1      LD A, (LINLEN)
58904 3D                    DEC A
58905 B9                    CP C: TEST COL >63
58906 38F5                  JR C, PARERR
58908 C9                    RET

CONVERT FORMAT

:CONVERT LINE/COL FROM USER TO INTERNAL FORMAT AND BACK
:USER FORMAT LINES 0-23; COL 0-63
:INTERNAL FORMAT LINES 24-1 COL 65-2 (COL 1 = END OF LINE)
58909 3A33E0    CONVFM      LD A,(LINLEN)
58912 3C                    INC A
58913 91                    SUB C
58914 4F                    LD C,A
58915 3E18                  LD A,24
58917 90                    SUB B
58918 47                    LD B,A
58919 C9                    RET

UPDATE POSITION

:ENTER HERE WITH BC=LINE/COL IN INTERNAL FORMAT
58920 CD2DE6    UPDPOSN     CALL CALCPOS: CALCULATE POSITION
58923 1828                  JR STPOSN: STORE UPDATE AND RET
```

CALCULATE POSITON

```
:CALCULATE POSITION IN DFILE
:RETURNS ADDR IN HL: PRESERVES LINE/COL IN BC
58925 CD3AE6   CALCPOS   CALL LNBO: GET DFILE ADDR
58928 3A33E0             LD A,(LINLEN): WHICH DFILE
58931 3C                 INC A
58932 91                 SUB C
58933 5F                 LD E,A
58934 1600               LD D,0
58936 19                 ADD HL,DE
58937 C9                 RET
:GET DFILE ADDR FOR START OF LINE IN B
58938 3E18     LNBO      LD A,SCRSZ
58940 90                 SUB B
58941 57                 LD D,A
58942 0F                 RRCA
58943 0F                 RRCA
58944 0F                 RRCA
58945 E6E0               AND 224
58947 6F                 LD L,A
58948 7A                 LD A,D
58949 E618               AND 24
58951 F640               OR 64
58953 67                 LD H,A
58954 3AC25C             LD A,(VIDMOD)
58957 CB47               BIT 0,A: WHICH DF?
58959 C8                 RET Z
58960 3E20               LD A,32: SET TO DF2
58962 B4                 OR H
58963 67                 LD H,A
58964 C9                 RET
```

STORE CURSOR POSITION

```
58965 ED4334E0 STPOSN    LD (CURPOS),BC
58969 2236E0             LD (DFADRS),HL
58972 C9                 RET
```

LOAD COURSOR POSITION

```
58973 ED4B34E0 LDPOSN    LD BC,(CURPOS)
58977 2A36E0             LD HL,(DFADRS)
58980 C9                 RET
```

UPDATT

```
:UPDATE ATTR BYTE FOR CHAR JUST PRINTED
:HL = ANY SCAN OF CHAR IN DF
58981 CD92E6   UPDATT    CALL CALCATT: ADDR OF ATTR TO HL
58984 3A39E0             LD A,(ATTBYT)
58987 5F                 LD E,A: CURRENT ATTR VALUE TO E
58988 3A3FE0             LD A,(ATTMSK): MASK TO D
58991 57                 LD D,A
```

```
58992 3A38E0            LD A,(MASKB):FLAGS TO B
58995 47                LD B,A
58996 7E                LD A,(HL): BYTE FROM ATTR FILE
58997 AB                XOR E
58998 A2                AND D
58999 AB                XOR E: NEW ATTR BYTE
59000 CB60              BIT 4,B: INK COMP. OF PAPER?
59002 2808              JR Z,UPDAT1
59004 E6F8              AND 248: INK BLACK
59006 CB6F              BIT 5,A: PAPER 4-7?
59008 2002              JR NZ,UPDAT1: USE BLACK INK
59010 F607              OR 7: USE WHITE INK
59012 CB70      UPDAT1  BIT 6,B: SET PAPER COMP. INK?
59014 2808              JR Z,UPDAT2
59016 E6C7              AND 199: SET PAPER TO BLACK
59018 CB57              BIT 2,A: INK 4-7?
59020 2002              JR NZ,UPDAT2: USE BLACK PAPER
59022 F638              OR 56: USE WHITE PAPER
59024 77        UPDAT2  LD (HL),A
59025 C9                RET
59026 7C        CALCATT LD A,H
59027 0F                RRCA
59028 0F                RRCA
59029 0F                RRCA
59030 E603              AND 3
59032 F658              OR 88
59034 CB6C              BIT 5,H: WHICH DF?
59036 2802              JR Z,CALCA1
59038 F620              OR 32: DF2
59040 67        CALCA1  LD H,A
59041 C9                RET
59042 F5        LDATTR  PUSH AF
59043 3A8D5C            LD A,(ATTRP)
59046 3239E0            LD (ATTBYT),A
59049 3A8E5C            LD A,(MASKP)
59052 323FE0            LD (ATTMSK),A
59055 3A915C            LD A,(PFLAG)
59058 0F                RRCA
59059 323AE0            LD (MASKB),A
59062 F1                POP AF
59063 C9                RET
59064 3AC35C    GETSTRG LD A,(PARAMS)
59067 E61F              AND 31: TOP 3 BITS
59069 F640              OR 64
59071 57                LD D,A
59072 2A4B5C            LD HL,(VARS)
59075 7E        GTSTR1  LD A,(HL)
59076 E67F              AND 127
59078 2813              JR Z,NOSTRG
59080 BA                CP D
59081 2808              JR Z,GTSTR2
59083 D5                PUSH DE
59084 CD2017            CALL RECLEN
59087 EB                EX DE, HL
```

```
59088 D1                POP DE
59089 1BF0              JR GTSTR1
59091 23       GTSTR2   INC HL
59092 4E                LD C, (HL)
59093 23                INC HL
59094 46                LD B, (HL)
59095 23                INC HL
59096 78                LD A, B
59097 B1                OR C
59098 C9                RET
59099 0E02     NOSTRG   LD C, 2
59101 0600              LD B, 0
59103 C9                RET
59104 00000000 GRPST    DEF B 0,0,0,0,0,0,0,0: GRAPH PIX TABLE
59108 00000000
59112 0F0F0F0F00000000  DEF B 15,15,15,15,0,0,0,0
59120 F0F0F0F000000000  DEF B 240,240,240,240,0,0,0,0
59128 FFFFFFFF00000000  DEF B 255,255,255,255,0,0,0,0
59136 000000000F0F0F0F  DEF B 0,0,0,0,15,15,15,15
59144 0F0F0F0F0F0F0F0F  DEF B 15,15,15,15,15,15,15,15
59152 F0F0F0F00F0F0F0F  DEF B 240,240,240,240,15,15,15,15
59160 FFFFFFFF0F0F0F0F  DEF B 255,255,255,255,15,15,15,15
59168 00000000F0F0F0F0  DEF B 0,0,0,0,240,240,240,240
59176 0F0F0F0FF0F0F0F0  DEF B 15,15,15,15,240,240,240,240
59184 F0F0F0F0F0F0F0F0  DEF B 240,240,240,240,240,240,240,240
59192 FFFFFFFFF0F0F0F0  DEF B 255,255,255,255,240,240,240,240
59200 00000000FFFFFFFF  DEF B 0,0,0,0,255,255,255,255
59208 0F0F0F0FFFFFFFFF  DEF B 15,15,15,15,255,255,255,255
59216 F0F0F0F0FFFFFFFF  DEF B 240,240,240,240,255,555,255,255
59224 FFFFFFFFFFFFFFFF  DEF B 255,255,255,255,255,255,255,255
LAST BYTE USED 59231
```

This code is saved on disk as "2scr.bin",57344,1889.

Basic for Using Dual Screen Mode:

The astute student will immediately recognize that the Basic interface routine at the beginning of the program uses exactly the same addresses and techniques we had for the 64 column and 80 column modes. Therefore, our Basic program is going to look quite similar except this time around we have to provide for writing to 2 different screens. However, an amazing thing does take place. All the Basic commands you are used to using still work with DF1 no matter what screen is showing on your TV or monitor. In addition, the Basic commands of INK, PAPER, BORDER, BRIGHT, FLASH, OVER, and INVERSE work from the Basic System Variable storage of these flags so will work on either screen. Thus only PRINT, AT, TAB, ATTR, CLS, LIST, DRAW, POINT, PLOT, SCREEN$, CIRCLE, SCROOL and EDIT won't work on DF2. This last is highly important as INPUTS and ERROR MESSAGES won't print on the bottom 2 lines of DF2. Should you get stuck with DF2 active on the monitor and for some reason your basic program stops, there will be no message printed to tell you what is wrong and in

addition you have to type in commands blind to get back to DF1 active or restart your program. Don't just hit keys randomly should this happen to you or you fill up the edit space with garbage which will result in your having to pull the plug to get control back.

Let's present the program and then discuss it. It's on the disk as 2SCR.BAS for those of you who have the disk.

```
  5 CLEAR 57343: CAT "2scr.bin",57344,1889
 10 INVERSE 1: PAPER 6: POKE 57380,128: RANDOMIZE USR 57381: REM
GOTO 2 SCREEN MODE WITH DF1 ACTIVE
 15 PRINT "Screen 1 works from Basic"
 19 REM As a word entry processor (no tokens please)
 20 FOR k=1 to 600: IF PEEK 23560>143 THEN STOP
 25 POKE 57344,PEEK 23560: RANDOMIZE USR 57345: PAUSE 0: NEXT k
 30 REM CLS demo wil clear only a portion of the screen if you
like. Make sure you have something on the screen to clear.
 31 POKE 57356,0: POKE 57357,2: REM 2 lines starting with line
0. Reset to 0 and 23 to clear whole screen.
 32 RANDOMIZE USR 57358: POKE 57357,24: REM reset to full
screen
 35 STOP
 37 REM SCROOL Demo 57384 with # of lines to scroll, 57385 with
starting line # (default is 1 and 23 for whole screen)
 40 RANDOMIZE USR 57386
 45 STOP
 50 LET A$="PRINTING TO SCREEN 2 while showing SCREEN 1"
 55 POKE 57380,128: RANDOMIZE USR 57381: POKE 57380,5: RANDOMIZE
USR 57381: POKE 57356,0: POKE 57357,24: RANDOMIZE USR 57358:
POKE 23747, CODE "A": RANDOMIZE USR 57348: PRINT AT 10,0;A$'
"Press any key to show screen 2": PAUSE 0: POKE 57380,1:
RANDOMIZE USR 57381: PAUSE 0: GOTO 890
 70 POKE 57380,128: RANDOMIZE USR 57381: PRINT AT 5,0;
"Transfering something to Screen 2 from Screen 1 but only these
2 lines";AT 9,0;"Press any key to show screen 2": PAUSE 0
 75 LET A$="5,2,1,2,2": POKE 23747, CODE "A": RANDOMIZE USR
57408: POKE 57380,1: RANDOMIZE USR 57381: PAUSE 0: GOTO 890
 80 PRINT AT 8,0;"Exchange Screen 1 with Screen 2 Press any key
to continue": LET A$="0,0,1,2,22": POKE 23474, CODE "A": PAUSE
0:
 85 RANDOMIZE USR 57411: PAUSE 0: POKE 57380,1: RANDOMIZE USR
57381: PAUSE 0: GOTO 890
150 POKE 57380,1: RANDOMIZE USR 57381: REM Goto 2 screen mode
with DF2 active
155 PRINT "SCREEN 2 doesn't work from Basic as this prints to
DF1"
160 LET A$="SCREEN 2 Open for your input. Type a message. Quit
with STOP."
165 POKE 23747, CODE "A": RANDOMIZE USR 57348
167 POKE 57351,0: POKE 57352,3: RANDOMIZE USR 57353: REM AT 3,0
Set cursor to 3,0
170 FOR k=1 TO 600: IF PEEK 23560>142 THEN GOTO 890
175 POKE 57344, PEEK 23560: RANDOMIZE USR 57345: PAUSE 0: NEXT
```

```
k
499 STOP
505 POKE 57380,0: RANDOMIZE USR 57381: REM go back to  1  screen
mode
889 STOP
890 POKE 57380,128: RANDOMIZE USR 57381: REM DF1 active to  show
EDIT lines
899 STOP
900 OUT 244,1: MOVE "2scr.bas",5
```

At first glance this looks like a lot of pokes and randomize usr calls which can be quite confusing but it really is quite simple.

Line 5 lowers RAMTOP and loads the code which shouldn't give any one a headache. Line 900 resaves the basic in case you make changes to this program. Line 500 merely takes us back to the single screen normal mode.

Every time we exit a program that ends with DF2 being displayed and we want to exit with an edit line we go to line 890 to get us back into DF1 active on the TV or monitor so we can read what we type in.

Line 10 demonstrates the use of INVERSE and PAPER so we have some color and then opens both files and displays DF1 on the TV or monitor.

In code we would put the mode in register A and call SETMODE.

Line 15 prints a message to the screen to demonstrate that Basic commands do work.

Lines 20 and 25 converts our computer to a word input processor using a routine called WRCHR recycling every time a key is pressed until you enter a token. The pause is just to wait for the next input. Note where the input typing goes to the screen--at the very top overwriting the first message. The reason for this is that we haven't set the cursor as we do in line 167 when on screen 2. You may wish to add this line. I left it this way to show that Basic uses a different variable to keep track of the screen position than the machine code routine.

In code we would use the A register to hold the character codes. Note that we use a different entry point for all calls from machine code than we do from Basic. We can use any codes up to and including the user defined graphics but no tokens.

Lines 31 and 32 demo the CLS function clearing only the top two lines of the screen. By the way, the CLS of the 64 and 80 column modes can be used in exactly the same way. We end CLS with resetting to a full screen mode.

In code we would put the line count in B and the starting line in C.

Line 37 and 40 do a full screen scroll of 1 line. In code the line count would be put in B and the starting line in C.

NOTE that the same thing would happen to DF2 if we merely did a POKE of 57380 with 1 and a RANDOMIZE USER 57381. We should also do a GOTO 890 so we don't hang up in screen 2.

There are also functions to find out what character is printed at what space and what the attribute for a space is but we leave these exotica to the student to explore on his/her own.

However, we can get a bit fancy. To demonstrate the real power, let's print to screen 2 while displaying screen 1.

Line 50 sets up a string which we will print to both screens only in different locations.

Line 50 starts by making sure we are displaying DF1. Then we go to MODE 5 telling the computer to do print operations to DF2. Next we CLS screen 2. Finally we tell it to print A$ to DF2. Then in Basic and without changing modes, we tell it to print A$ to DF1 with the addition of a direct string. We stop with a PAUSE to view it. Next we go to MODE 1 to display DF2 with a PAUSE for us to view it. Finally we end with a GOTO 890 to get back DF1 and our edit line.

There is also MODE 4 to use in code. It is not much value in Basic since any simple PRINT command will go to DF1 no matter what file is being displayed. MODE 4 tells the computer to use DF1 for the following print operations.

Line 70 and 75 show how to transfer something written to screen 1 to screen 2. Line 70 merely goes to screen 1 and writes 2 messages and waits for us to press a key. In line 75 we set up A$ with the following sequence of parameters: (the letters in back of each statement are the register we would use if we were writing a program to do the transfer in code.)

1. Source starting line.	B
2. Destination starting line.	C
3. Source file.	D
4. Destination file.	E
5. # of lines to transfer.	A

We tell the computer about A$ by poking the code to PARAMETERS at 23474 and calling COPY. We then change modes to show DF2 and that the lines were transferred. PAUSE is used to hold this screen until another key hit turns us back to DF1 by going to line 890 and an edit line.

Note that only full lines can be transferred, not just a part of

a line. Also note that we can transfer lines from one part of a screen to another part of the same screen if we wish. There is nothing that says source and destination must be different display files.

Lines 80 and 85 demo the Exchange of material between the 2 display files. Again A$ is set up with the parameters and/or registers:

1.	Source 1 DF	B
2.	Source 1 starting line	C
3.	Source 2 DF	D
4.	Source 2 starting line	E
5.	Line count	A

The difference is that what is at one position in one file gets exchanged with the other position in the other (or the same) file. Again only full lines can be exchanged.

Line 150 displays DF2. If we haven't used DF2 as yet it will be a black screen as all the attributes are still 0 until we start writing. ATTR P will be used in the usual definition of bytes in that attribute.

Line 155 merely demonstrates that Basic will not work to write to DF2. This also limits us to named strings rather than undefined strings in PRINT statements.

Lines 160 and 165 use WRSTR to write A$ to DF2 and display it. In code we wouldn't use a variable but only a series of addresses containing the message. HL would be pointed to the first address of the message with BC containing the length.

Line 167 resets the cursor to 3,0 with SETCUR. In code, register B would be the line # and C the column #.

Lines 170 and 175 uses WRCHR to turn the screen into a word processor. Entering the STOP token quits the program as we did for DF1.

Other things could be added to the demonstration to show more of the routines written into the code. There are routines to query the 2 screens for what character is in a certain space or what the attribute of a certain space is which may have applications in some games. However, I think that what I have demonstrated is enough to whet your appetite.

The student will have observed through all of this that if one makes DF2 the file displayed, one can write to DF1 using Basic to prepare it for display. Then at the proper time use COPY to transfer the whole file or just the part that a rewritten to DF2 for display.

What you want to use the Dual screen mode for is up to you. If

Basic is too slow for your screen writing routines, you can always switch your sluggish routines to code.

CHAPTER 6 HI RES GRAPHICS MODE

The Hi Rez Graphics Mode has been thought of by some people to be the mode to paint your masterpiece, your Mona Lisa. You are going to be disappointed. DF1 is used to display the pixels in a normal 24 line 32 column screen. DF2 holds one attribute per line of 8 screen pixels, in other words 8 attributes per character space. Nothing is held in the attribute portion of either file. This is neither high resolution pixel counts or high resolution color. Each attribute is still limited to only two of the eight Basic colors of paper and ink and though you may use inverse to invert them you can't have 3 colors in a horizontal line shorter then 9 pixels across. Vertically they can change every pixel. This sometimes leads to the necessity of mildly distorting a picture at least in the horizontal direction. Yet some quite fantastic screens have been produced. HI RES GRAPHICS is not for running sprites.

There are quite a few paint programs already on the market which will have some quite interesting features to help you draw and design your screen. Some of the better ones even give you a blown up view of the portion of the screen you are working on. Let's admit that the Basic way of designing UDG graphic characters is too slow and cumbersome and too memory hungry for most applications. Faster and shorter methods are necessary. Running these programs is learning all the commands involved with these programs. Generally one does not want to change a screen once it is completely designed so most programs merely open DF2, load the 2 files and display them while loading the rest of the program and then drop back to normal screen mode. This wastes all the space of DF2. But a word of caution, some beautiful title screens are NOT HiRes mode but just ingenious and imagnative designs in normal screen mode. Also another caution, an unused DF2 is not a good place to run machine code routines from as it is in the lower half of RAM and can cause interferences with the data and address buses upon screen refresh. We will talk more about this interference in the next chapter.

This time around our Technical Manual is no help as there are no routines already written.

First we will need a change mode routine to open DF2 and set the MODE to 2--the HiRes mode. At least our manual tells us this much. Since you will also need this routine to display your picture in your program, the change video routine we used in the 64 column, the 80 column and the dual screen modes is a bit long. The routine we used in Introduction To 2068 Machine Code is shorter. We also want to save as much space as necessary so

we will be writing to chunk 7. The UDG, the machine stack and the Ram Resident code take up the top 2280 bytes. So we can only go up to 63254 in memory.

```
63217 F3  CHGVID        DI: NO INTERRUPTS
63218 3E01              LD A,1 :CALL THE CHGVIDEO ROUTINE IN
63220 DEF4              OUT (244),A: EXROM
63222 DBFF              IN A,(255)
63224 CBFF              SET 7,A
63226 D3FF              OUT (255),A
63228 3E06              LD A,6
63230 F5                PUSH AF
63231 FB                EI
63232 CD8E0E            CALL 3726: CHANGE VIDEO
63235 F3                DI
63236 DBFF              IN A,(255): ALTHOUGH DF2 IS OPEN GO
63238 CBBF              RES 7,A     BACK TO SINGLE MODE AND
63240 D3FF              OUT (255),A GO BACK TO HOME BANK
63242 AF                XOR A
63243 D3F4              OUT (244),A
63245 F1                POP AF
63246 FE80              CP 128
63248 2002              JR NZ,END
63250 3206FF            LD (65286),A
63253 FB        END     EI
63254 C9                RET
```

The reason why we go back to the single mode screen but leave DF2 open is that the routine sets all the attributes in DF2 to 0 and since an attribute of 0 means black paper and ink, it means that you would see absolutely nothing until you also run the following routine which we call FILL and load immediately below the above routine:

```
63200 3E38      FILL    LD A,56: Attribute, white ink, black
63202 210060            LD HL, 24576: DF2               paper
63205 0618              LD B,24
63207 0E00      LOOP    LD C,0: ACTUALLY 256 THE WAY WE WILL
63209 77        LOOP1   LD (HL),A:  BE USING IT
63210 23                INC HL
63211 0D                DEC C
63212 20FB              JR NZ,LOOP1
63214 10F7              DJNZ LOOP
63216 C9                RET
```

This routine as written will only fill the top 3rd of the screen with the attribute of white ink, black paper. To fill the middle third we must poke 63203-4 with 26112 (6800H) and to fill the bottom third we poke the same address with 28160 (7000H). We had better at least fill the bottom third of DF2 with this routine or we will have to type in our next direct command blindly.

Neither of these programs is on the disk so you will have to type them in yourself.

Now to switch back into Hires mode all we have to do is OUT 255,2 from Basic. OUT 255,0 switches us back to single screen mode. Both without having to invoke a RANDOMIZE USR statement.

Okay, how do we do our picture? Simple enough. As in the Dual Screen mode, all print commands still work on DF1 so we can draw a black and white picture without color using the standardized Basic routines and then proceed to color the picture.

How do we know what address we are at in DF1? We already have a routine for that way back in Chapter 2 with a program called DF-LOC. But obviously, we cannot use the normal screen ATTR-LOC routine from Chapter 2. But don't dismay, all is not lost. If we have the DF1 address it is quite simple to calculate the corresponding attribute address in DF2. The two files are exactly 8192 bytes apart. If we have the address of the DF1 file in HL all we have to do is turn on bit 5 of H and we have the attribute address. How about the simple command:

```
CBEC    SET 6,H.
```

With this hint, the astute student will realize that we can also use:

```
CBB4    RES 6,H
```

to go from attribute address back to the character address merely by turning off bit 5 of the high register. Of course we don't necessarily have to use HL to hold the address as there are similar commands for B and D as well. We leave it the the student to write this simple adaptation for locating attribute addresses.

The real problem with Hires is that the PAPER pattern color changes don't match square character spaces or are too coarse for what you have planned. For example a circle in red ink on yellow paper is easy to draw. Now, filling the inside of the circle with a paper color different than red or yellow is going to run into a problem by having some of the inside color spill out of the circle. The solution to this problem is that you have to use thicker pixel lines for the red circle and this may spoil your idea.

As you start experimenting with HiRes you will find that you can draw a picture, although a bit rough with squared edges, by just using the attribute file. You will also discover that by alternating using attributes of contrasting colors you can get colors other then the basic 8 colors of the 2068. This can be further augmented by using alternate pixels on and off with a different ink color as well.

EXTENDED PAINT--A HiRes Mode Program

A good paint program will have a routine to blow up a section of

the picture so that you can temporarily adjust pixels and attributes as necessary, see how they look and then cancel them if it is not what is desired or make them permanent. Good programs also have a means of putting patterns into a program as well. One such program, which incidentaly was written by a fellow SMUG member, is: Extended Paint by David Franson. It is available through SMUG for $19.95 and comes with an extensive 20 page users manual.

Extended Paint is quite long containing a Basic section of 14200 bytes and a code section of almost 16000 which means the program is too long to include in your program. It only uses code to do those things that can't be done readily in Basic. An examination of the Basic shows a lot of pokes to make various code routine do double or triple duty. Most of the code section is reserved for holding copies of DF1 and DF2 which allows one to paint all 24 lines of the screen and also to make temporary changes and rejecting them by going back to the last permanent change of the picture. It omits most of the design of your pixel screen which the author assumes you can do using your favorite Script drawing program and then working on a saved screen of that program.

Between a Script writing program using a font of your choice, Extended Paint and the two routines given earlier in this chapter you have all that you really need to design and display your own masterpieces. You don't need the Script writing program if you don't want fancy lettering.

To show you what you all get with Extended Paint I will describe it in detail. Some features are quite unique. Dave discussed some of these with me and used some of my suggestions in developing the program.

There are three direct commands:

Space Bar to Toggle on and off the Menu. Another problem that Dave has solved is where do you put a Menu on the screen. The bottom 2 lines of the screen are used for that but because of the saving files feature can be rewitten as needed by rewriting the picture from the storage files. Pressing the Space bar once brings up the Main Menu. Pressing the space bar again returns the bottom two lines to the picture mode.

The following commands need lower case letters. Dave doesn't tell you this but his Basic program doesn't allow for upper case commands.

M to toggle the cursor move function on and off. This function allows moving around the picture without making changes. Extended Paint uses a joystick to move around the cursor. The fire button on the joystick signals the computer that you want a color change to which you type in the color number from the keyboard. Every color change is followed by the option to turn on flash and then with a second option for using bright.

Z togles on and off the zoom command which blows up a 4 by 3 section of the screen for resolving any pixel or color problems across the bottom of the screen. Each pixel is expanded to a full character space in its INK (if on) or PAPER (if off) color. This routine uses code so you have to toggle out of it before breaking the program.

S cancels or aborts anything that was going on and returns you to the appropriate menu.

To get everything into 2 lines of Menu space, abbreviations of the Commands were necessary. The Main Menu provides the following first letter commands:

Csr (Cursor color) allows you to change the cursor colors of INK, PAPER, BRIGHT and FLASH. Actually Dave's cursor move routine just changes the attribute without ever changing the Ink pixels by temporarily storing the attribute of the space it occupies and replacing it with the cursor attribute. The stored attribute is put back into its place under the move command as the cursor moves on the the next space.

Velcty (Cursor Velocity). Allows a change of cursor move speed from 1 to 9 with 1 being fast and 9 slow.

Fill allows one to fill the entire screen with an attribute of your choice. Pattern and Hue are used for smaller portions of the screen.

Border allows one to change border colors.

Pttrn (Pattern) is used to paint a particular 1/3rd of the screen with a particular attribute. Specifying 1 at the first prompt of FROM and 2 at the second prompt of TO paints the top third. 2 and 3 the mid third, 3 and 4 the bottom third, 1-3 the top two thirds, etc. You get three more prompts of Offset, Step and the Attribute Paper, Ink, Br(ight) and Fl(ash). To paint the entire section a solid color enter 0 to Offset and Step and enter the colors you want for the attribute.

Now, for a pattern, repeat the Pttrn operation with a different attribute but this time enter a number other then zero at Offset and Step. By playing around with various Offsets and Steps you get patterns generated on the picture without the use of ink pixels but merely by changing attributes in your specified set change rates. You can always cancel the whole effect by using a zero offset and Step.

Hue is used to replace all the occurrences of a particular ink or paper color in a particular third of the screen with another paper or ink color. Obviously Hue should be used

early in the design to help set the background colors of your solids and patterns.

Refer toggles on and off referring to the normal screen attributes file to help you in desiging colors for the HiRes mode if you loaded a screen that already had color applied to it from the normal screen mode.

Grab applies the same colors from a normal screen with color to the high resolution mode if you happen to like that color scheme and merely want to use Extended Paint to resolve color clashes.

Screens brings up the screen menu all commands of which deal with saving and getting copies of DF1 and DF2 to or from high memory or to make changes to the Ink pixel screen and consists of the following commands:

Bars puts a series of vertical bars on the pixel screen in Ink color.

Cls(BW) is something you don't want to unless you want to start over as it is the good old Basic CLS applied to the pixel screen only of course.

Retrn returns you to the Main Menu.

1PutBW stores a copy of the pixel screen to high memory where it can always be restored with 0GetBW. It is also used to permanently store a new copy of the pixel screen after you have made changes that you are happy with. NOTE: Any cassette operations done through the Tape Menu automatically do a PutBW operation before returning to the Main Menu so that you already have a copy there. You get a --PutBW OK-- to tell you of the successful completion of the command.

0GetBW Puts a copy of the pixel screen stored in high memory back into DF1. You always get a --GetBW OK-- prompt to tell you of the successful completion of the command.

2PutCLR (Put Color Screen) is the same "save to high memory a copy of DF2", the attribute screen, as PutBW does for the pixel screen. Again you get a --PutCLR OK-- to show successful completion.

9GetCLR (Get Color Screen) is the same restore from high memory of the last PutCLR attribute save back to DF2 as GetBW is for the pixel screen. You get the --GetCLS OK-- when done.

Tape brings up the Tape Menu which deals with saving and loading things to tape. If you are like me and like things on disk drive, this section takes a lot of changes to convert to disk drive operation. It starts at line 5000 in the Basic program and jumps to several subroutines. Also don't forget to change the loader program commands and line 1 and delete the VERIFY options which also don't work with disks. With Aerco Drives don't forget the OUT 244,1 statements necessary to activate the operating system.

SavPROG saves a copy of the Basic and CODE to cassette with option to verify. It automatically does a 1PutBW and 2PutCLR as well.

Basic returns you to Basic. Use GOTO 5000 to reenter.

Return returns to the Main Menu.

1LdBW loads a pixel screen from cassette. Low resolution colors are also loaded and can be reviewed with the Main Menu command REFER.

0SvBW saves the pixel screen to cassette again with a VERIFY option.

2LdCLR loads a high resolution attribute screen from cassette to DF2.

9SvCLR saves the attribute screen to cassette again with a VERIFY option.

Output gives you the Output Menu for sending your masterpiece to either the 2040 printer, dot matrix printer or a Color Jet printer in a pixel by pixel mode or an expanded 4 page wide by 3 page long picture. This feature is not included in many paint programs. Care should be taken in using the expanded modes as they take considerable time, cause printers to get quite hot and can wear out a ribbon in a hurry. The gray scale drivers try to reproduce a gray tone representation of your picture by using a low density of dots for light colors and a high density of dots for dark colors. Therefore, pictures with a lot of black and blue are harder on your printer and ribbon than light colored pictures. You may wish to put your printer off line for a time to let it cool down during printing. Don't worry, it starts up exactly where it left off without problem when put back on line. In all cases "s" aborts the printing.

324CLR sends a gray scale print of the attribute screen only to the TS2040 printer.

424GRY sends a gray scale combined print of both pixels and attribute files to the TS2040 printer in eight sections and requires 12 minutes. Crop marks allow you

to paste together the sections into an expanded oversized picture.

Rtrn returns you to the Main Menu.

5FsCLR is like 324CLR in that it sends only the attribute file to a full sized Epson-compatible printer.

6FsCLR is like 424GRY in combining both pixel and attribute files to send 4 wide by 3 high pages with crop marks for assembly to an Epson-compatible printer. It requires 35 minutes to print. The picture is roughly 3 ft by 2 ft when assembled.

7CanonSM sends a full color print of 2.25 by 3 inches to a Canon PJ-1080A, IBM Color Jet Printer, Quadram Quadjet or Tandy CGP 220 color inkjet printer in about 2 minutes.

8CanonLRG sends the same pictures expanded to a 4.5 by 6 inch format to the same set of printers as the 7CanonSM command. It takes about 4 1/2 minutes and is really great.

This completes the various Menu commands of Extended Paint. I think it shows how much time and energy goes into writing an extensive program such as this. Dave's manual of instructions goes on to explain exactly how he created the introductory screen for Extended Paint. Until you actually work with the program or try to do it from Basic you don't realize how much time it can save you.

One last comment. In HiRes Mode the usual K, L, C, E, or G cursor doesn't blink as it does in normal mode. It gives you an indication of which mode you are in although it makes it a bit difficult to find should you decide to List the Program. Just remember that you have to save the DF1 file first before listing the program or the file is forever gone.

For those of you who would like to write your own paint program, in addition to finding the attribute routine from the pixel address you also have to write a cursor moving routine that moves a cursor around the screen. Make sure that you also include some way of saving the attribute of the screen your cursor is in so you can rewrite it when you move the cursor off that character space and on to the next. Special situtations exist at the 1/3rd of a screen boundaries which you have to take into account.

Extended Paint's cursor routines wrap around so that running off the bottom of the screen causes the cursor to reappear at the top of the screen. Other direction runoff produce similar opposite effects. Cursor moving routines have been published elsewhere.

After you have your cursor routines working, you can then work on a zoom routine to expand several character spaces into character-wide pixels. Such routines have been published elsewhere and maybe you can adapt one of those to your use. Once expanded and changed you have to compress it back down and load it into the back spot in the picture again.

Making copies of the two files to high memory and then bringing them back down to DF1 or DF2 should prove no problem as all it requires is a short LDIR routine.

Saving and loading files to and from cassette or disk is easy but just remember that you can't save DF2 with a SCREEN$ command but require a CODE notation for cassette or a .bin extension for disks.

Printer driver routines that first expand and combine pixel and attribute densities and then send them to the printer have not been published but sending them sideways to a printer has been published.

Sorry I can't give you all these routines as Dave uses them but his program is copyrighted. These last few paragraphs outlining what you would have to do also shows what an author has to go through in order to get a working program. Like writing books it takes hundreds of hours of time.

For those of you who have a copy of Extended Paint the following partial routine listing should help:

47353 Hue exchange
47425 Epson gray scale dump & Jet Print dump
47436 Gray scale dump to 2040 printer
48220 Buffer fill
49905 Color only gray scale dump
50310 Screen Menu routine
50346 Storage of DF1 in high memory
56490 Storage of DF2 in high memory
62635 LDIR transfer routine
62650 ZOOM
62780 Cursor right
62893 Cursor left
63004 Cursor down
63098 Cursor up
63198 Fill
63217 Change Video

Happy Painting.

CHAPTER 7. PLAYING WITH BORDERS AND SPRITES
OR
EVERYTHING FOR THE SERIOUS GAME DESIGNER

This means smoothly moving horizons that can move up and down the screen a pixel line at a time rather than a character space at a time. It also means smoothly moving sprites in any direction one pixel line at a time and at any speed from as fast as laser gun fire to as slow as the proverbial turtle. These sprites should also be able to move over each other.

Although the 2068 Technical Manual gives sprite support routines they are not the smoothly moving ones we are after. So we abandon those routines for some that will work.

To do what we want we are now going to embark on some very fancy routines that do some crazy things to the standard mode screen. Things that I don't believe even the writers of the ROM ever intended it to do. Most of these routines were published for the Sinclair Spectrum machine that works on British TV's and had to be adapted to work with American TVs. They were originally published under the title "Advanced Code for the Sinclair Spectrum" by David Webb, Melbourne House Publishers. Sorry, but the book is no longer in print.

To do these things we are going to mess with the Z80's Interrupt modes. This was a topic that was a bit too advanced a subject to cover in detail in "Introduction to 2068 Machine Code" but here goes.

Everything You Ever Wanted to Know About Interrupts

But Were Afraid to Ask?

The Z80 has 3 maskable interrupt modes labeled as IM0, IM1 and IM2. IM0 is a bit redundant in that it expects an instruction from a peripheral to begin making its way along the data bus during the interrupt acknowledge cycle. Unfortunately, the Spectrum or the 2068 usually holds FFH on the data bus during an interrupt which would mean RST 56 which the Z80 duly executes--jumping to Address 56 and doing that before returning. Unfortunately IM1 does the same thing. That is why IM0 is redundant. IM1 is the standard interrupt mode of the 2068 ROM. Address 56 contains the Maskable Interrupt Routine.

As the name implies, the maskable interrupts can be masked which is another word for saying it can be defeated. Students will remember the instruction DI (Disable interrupts) and the stress

put on using this instruction before executing any long running routine like an LDIR or LDDR routine. By long we mean anything that could take longer than 1/60 of a second which is the time interval between successive interrupt signals sent to the CPU. The 2068 uses this signal to execute its Maskable Interrupt Routine at address 56 to scan the keyboard and increment the Frames Counter. It also signals the SCLD chip to begin a TV refresh cycle. We were also told to never exit machine code without doing EI to make sure we didn't have a dead keyboard which is equivalent to a crash as the only thing to do is pull the plug as the computer is hung up.

The TS1000 had something called a SLOW and a FAST mode. Fast was in reality nothing much more then a DI statement while SLOW was little more than an EI statement to reenable Screen refreshes.

We were also warned never to use the instruction HALT as it was said to stop execution of the program at that statement and wait for an interrupt. With DI in force an interrupt would never come.

What really happens when an interrupt is received is that the CPU finishes the instruction it is on, pushes the program counter address of the next instruction on the stack, saves all the registers and jumps to address 56 and does the routine there until it meets a RET at which point it restores all the registers and the program counter and continues as if nothing happened.

On the other hand, the Non-Maskable Interrupt is a hardware or peripheral interrupt. A line inside the computer called NMI normally has 5 volts on it. If for any reason, it temporarily goes low, like being grounded, a NMI has occurred. The registers are again all saved along with the status of the MI as the machine jumps to address 102(66H). On the ZX81/TS1000 machine it checks for SLOW and continues without any provision for adding another NMI routine. On the 2068 we have a problem as it checks for an address in the System Variables located at address 23728-23729. If you check your User's Manual you will find the statement that the addresses are not used. This is due to a bug in the next instruction of the ROM. Whether this bug is deliberate or not is open to some debate. The instruction in effect reads as a JR NZ, 1 and effectively goes right back to the routine it just came from if it finds something other than 0 there. This should be a warning to all programmers not to indiscriminately use so called "not used" addresses in the System Variable table. This means all NMI won't work.

This is NOT the way a NMI should work. The errant instruction at address 109 should be JR Z. This change would cause the computer to return immediately if the NMI interrupt address at 23728/9 was 0000 or jump to the address contained there and handle the interrupt with a routine you could write, finally returning to the program when finished. Unfortunately, the error is in ROM

and nothing short of burning a new ROM on an EPROM will correct it. Some so called "corrected ROMs" have done exactly that so beware. Unless...

Unless you are NOT in the home bank. If you are in the dock bank with an LROS program, your technical manual states that you MUST write both a MI and NMI routine to handle interrupts at the addresses 56 and 102 respectively. If you are lucky enough to have an Aerco disk drive interface you have 64k of RAM in the dock bank at all times. Everytime you use the drives you have to make sure you have Chunk 0 of the dock bank activated and that makes the interrupt routines written by Aerco active. It thus would be possible to correct the NMI handling routine since the Aerco program first starts at address 256 with only the LROS identification, the MI and NMI routines below address 256. There is only one thing wrong with this and that is that we are in an LROS program and can't run Basic from there. Aerco solves this problem by switching back to Chunk 0 of the Home bank most of the time and only using the Dock bank to access the disk drive programs.

Why use the NMI? Certain peripheral devices, noteably the keyboard, can give a NMI to the CPU indicating that they need attention. Most computers use the NMI to indicate input coming from the keyboard. As we saw above, Sinclair computers use the MI to scan the keyboard, not the NMI. But that is no reason to disable the NMI. Other devices such as a printer or a modem could be made to give a NMI when they need attention as well.

For example, most printers have a 2k to 8k buffer which is loaded and then printed out. When empty, it is reloaded with the next batch, etc. All the time that the printer is emptying its buffer it is sending back a signal saying "I'm busy, don't send more yet." The way it stands at present, the Print Driver program monitors this signal in a wait loop and is really wasting time. Instead the "busy" signal could be put on the NMI line thus freeing the CPU to do something else and return for another buffer reload only when needed.

Or, for another example, your modem is monitoring the phone for calls. Nothing might happen for hours, but when it does the modem needs attention immediately. Or yet another, your computer is set up to play watchdog for your house yet you would like to do an iterative program that takes hours to run overnight and have the answer in the morning. You could do both with a working NMI. Sinclair had the right idea with the NMI routine address at 23768. This routine with the same error is also in the Spectrum ROM. Why did they chicken out?

Enough talk about bugs in the ROM and NMI and as far as that goes IM0 and IM1. They won't get us anywhere. What about IM2?

When an interrupt occurs, the Z80 takes the byte currently on the data bus as the LOW byte and the contents of the interrupt

"I" register as the high byte to form an address. It looks at this address, supposedly in a table, to get an address stored low byte first to load into the program counter and goes to that routine for execution. Suppose the I register held FE and the data bus held 40. This would mean looking at FE40 for an address. If FE40 contained 38H and FE41 contained 00, the jump would be to 0038H.

In actual fact, since the data bus USUALLY holds FFH during an interrupt all our "vector addresses" would probably end in FFH. A problem of 'snow' or picture breakup is experienced when running code below 32768, due to the fact that code running there runs about 20% slower because the ULA (Uncommitted Logic Array) which is responsible for producing the TV signal competes with the CPU for access to the data bus and the ULA always wins. This would mess up all the precise timeing that we need for control. Thus, we prefer the address of the jump to be somewhere ABOVE 32678 in RAM. This restricts us to 80H to FFH for our I register value.

Inside the Z80 are two special registers called interrupt flip-flops named IFF1 and IFF2. During an NMI, IFF2 stores the previous value of IFF1 while IFF1 is reset for the duration of the interrupt. The function of IFF1 is to tell the CPU whether maskable interrupts are allowed. If it were set while doing a NMI another MI could occur which would be disaster. Hence IFF1 is off during a NMI. Thinking back to DI and EI, its IFF1 that gets SET for DI and RESET for EI. Thus only after the execution of an EI are MI's detected.

Whenever an interrupt is accepted, the IFFs are reset automatically. It is the programmer's responsibility to re-enable them before returning from the interrupt routine with RETI. It could cause untold problems if an interrupt were to occur between enabling them and returning from the last one. Thus we have the delayed action of the EI until AFTER the next statement has been executed. With a sequence of:

```
EI
RETI
```

an interrupt can't occur until after the RETI is done. The use of RETI and the I register are seldom discussed in beginning books on machine code either. Now you know how to use them.

Another instruction often overlooked is LD A,R. When the instruction is executed, the P/V flag is set to the contents of IFF2. Thus we can use the instruction to see if DI is in effect or not. A set P/V flag means it is NOT. Suppose we want to preserve the contents of the IFF2 while we disable the interrupts to do something and then we want to restore it to its former value. We can do it with:

```
LD A,R
```

```
    PUSH AF    Save the P/V flag
    DI         disable
  Insert what we want to do here.
    POP AF
    JP PO, NOT ON
    EI
```

LD A,I also has the same effects on the P/V flag.

Several paragraphs back we said that the data bus usually held FFH on an interrupt. This may not be true for certain hardware "add ons" that do not decode signals on the IOREQ and READ lines of the Z80 correctly. As a result, the data bus may not be FFH. When starting to work on these programs I would always crash with the Aerco Disk drives activated, i.e., OUT 244,1 active. When I went to OUT 244,0 no crashes occurred.

How can we get around the problem of not being certain that the data bus will hold an FFH during an interrupt? Quite simply by having every byte in a string of 256 bytes hold the same value. Since we want to leave the UDG where they are, the highest value for the I register will be FEH. Then we have to write addresses from FE00 to FEFF with the same value. How about FD?

That would make the address to jump to FDFD--3 bytes below FE00 and just enough room for a JP instruction. FDFD is 65021. Rather than give you the routine for initializing IM 2 at this point, let it suffice to say that it is called INI1 in the programs listed later on.

This concludes our discussion of interrupts and maybe you are saying to yourself, "Why did I have to ask?" Well, this is a textbook and you may want to refer back to it sometime in the future when you are working on something else. Maybe a re-reading is in order. But don't give up, it's going to be worth it in the end.

AMERICAN vs BRITISH TV

The reason for the interrupt discussion was mentioned but only in passing. To re-emphasize directly, we want to control the output to the screen. To control that, we must know when the output starts and then break into it at the right moment and rewrite the output. This will be done by using a HALT statement in our program and force it to then jump to an Interrupt Handler routine where things are timed exactly to several millionths of a second. This timing is what we are discussing here.

When I got to this section, I had to get the help of some TV and Ham operator friends who could tell me a bit about how a TV signal is rastered.

One thing that has always bothered me is that the ROM still

contains the 50 cycle codes from British alternating current signals to wait for a second. We in America use 60 cycle/second alternating current. So why didn't they change them to 60's? I rationalized it away by saying it didn't have to be that exact and that 50/60th of a second was close enough. It turns out that a correction WAS made when it comes to TV signals. How?

Read a Spectrum book and then read the TS2068 Technical Manual. The Spectrum contains a crystal creating a clock signal at 3.500 megaHertz, the TS2068 a clock signal of 3.528 megaHertz. Doesn't look like much of a difference but it does when generating TV signals.

The British TV generates 625 scan lines 50 times a second. The American TV only 525 scan lines 60 times a second. Don't bother multiplying the numbers together as the number of scan lines generated per second is not what we want. What we want is the number of T states per TV scan line. For British TV that is:

3,500,000/(625*50)= 112 T states.

For the American TS2068 it's:

3,528,000/(525*60)= 112 T states.

What do you know, the exact same timing. The ULA runs faster in America because of a slightly faster clock to exactly compensate for what is needed. I wouldn't want to put it to a test but the 2068 should be slightly faster than the Spectrum in program execution as well. So far so good.

Now we read in the 2068 Technical Manual that 2 scan lines are used per pixel height on the screen be it British or American. It therefore takes 24x8x2 or 384 scan lines to write the text portion of the screen. The rest of the scan lines appear above and below the text lines as Border and as scan lines not shown on the screen by being off the top and bottom of the CRT mask. In America these off screen lines are used for Closed Caption data.

How many scan lines before the text and how many after? This is going to vary between British and American Sets. First the British: At 625 scan lines per frame and 384 used for text that leaves 241 lines for top and bottom borders and off screen. The poor Americans on the other hand only have 525 lines per frame and if 384 are used in the text portion that leaves a skimpy 141 scan lines for top and bottom margins and off screen. Remembering that each scan line is exactly 112 T states long, the amount of things that could be done before processing of the picture had to start was considerably shorter for the American programs. Although not as grand as the British programs, they still work and still do wonders. Keep these numbers in mind as we continue.

What we want to do is create a straight line of color delineation right across the screen from the left edge of the CRT to the other edge, right through the border and text areas. The bottom part of the CRT has one background color and the top another such as we would observe as a horizon in an airlane or on the sea. Next we want to be able to smoothly move this horizon up and down a pixel line at a time, not a character space at a time, so as to simulate waves or dives and climbs. We also want to leave some things in permanent positions on the screen to simulate cockpit windows or the crosshairs of a periscope.

The Interrupt Handler

This means the critical line is that horizon and if we are going to change background colors somewhere through a character space we not only have to change the border color at the right time but also flip-flop the paper colors of the entire character line as well. Our Interrupt handler should be able to handle all these different applications.

Since all the emphasis is going to have to depend upon speed, we are going to have to do as much processing of the data as possible beforehand and then put it into the 'print buffer' of the Interrupt Handler. Then, on every interrupt, the print processor 'empties' the buffer, one entry at a time, and puts the corresponding character and its attributes in the correct place in the display and attribute files respectively.

We are constantly talking about T states. To the uninitiated that means the number of clock cycles, at 3,528,000/second, that it takes the CPU to process an instruction. Zilog has published a list of these times. A listing is given in Appendix A of "Introduction to 2068 Machine Code". The shortest instruction takes 4 T states and the longest 23 T states. Since all these routines are full of timing loops which you may have to change to get these programs adjusted to your particular TV or monitor, I suggest that you get hold of a copy of the Timing Table for the Z80.

We shall label the start of this print buffer as BUFFER. Each entry is six bytes long and the data is formatted as follows:

1. Attribute byte
2. Attribute address low
3. Attribute address high
4. Display file address high
5. Character data address low
6. Character data address high

Note the absence of the Display file address low. If you don't know by now, the attribute address low and the display file address low are ALWAYS the same number. The routine saves a bit of time by taking advantage of that fact.

We need room for a flag bit in these 6 bytes of data. What do we give up? About the only spot is a bit in the attribute byte. We need the 3 paper and the 3 ink bits so it's either the Flash bit or the Bright bit. Bright would give us more colors so let's keep that and give up Flash. What we have decided to do is shift the 7 low bits left one byte and use the 0 bit for this flag. This flag is used to allow us to signal either OR-printing or OVER-printing. When the flag is SET it will tell the print processor to OR-print the 2nd sprite over the first making it sort of transparent. If the flag is off, the 2nd sprite covers up the 1st in all the spaces it uses.

In order to generate a stable horizon, it is imperative that the routine executed between the interrupt and the horizon generator always use exactly the same amount of time. We use this time to print out the print buffer so as a result we always have to fill up the entire buffer with padded fake characters so it always takes the same amount of processing time. But because American TV pictures have less scan lines than the British (remember those extra scan lines for the top and bottom border and off the screen?), we have had to cut the buffer length from the British 40 characters to 24 and still keep the horizon at the top of the first print line. The routine actually reserves all 40 spaces so IF you do not HAVE the horizon sitting right at the top of the first line but down somewhere closer to the center of the screen which would give you more time, you can expand it out to its full 40 or even further (but not without reassembly!!). In fact we expand the buffer out to 40 in the sprite demo routine and then restore it.

Various other tricks have been resorted to, to make the routines as fast as possible. With a 40 character buffer (40x6=240 bytes) if we start our buffer at an address of XX10H we can use

```
            INC L
as opposed to INC HL
```

to step through the buffer as the buffer doesn't go over a page boundary and thus save 2 T states every time we use it. It has the further advantage that we can test for the end of the buffer merely by testing the ZERO flag.

If you are not going to use the top area of the screen for animation or horizon or if you don't mind flickering in that area, then you can expand the number of characters the print processor can handle. The top limit is 42 without going over a page boundary of memory (a change in the high byte of an address). However, beyond this limit, some alteration of the correct INC L to INC HL is necessary to step across the page boundaries. As an exampe, let's assume our buffer always ends at FCFFH. With 43 entries of 6 bytes each, the buffer would start at FBFE. The first INC L is okay but the second would have to be changed to INC HL to get over FBFF. Similarly, for an 86 character buffer the start address would be FAFCH requiring the

4th INC L to also be changed.

We are all used to using a counter and a loop for repetitive events but this wastes time. Even the fastest loop, DJNZ, takes 13 T states to recycle without doing useful work so many of the routines have all the repeats written out. In the print buffer routine alone it saves 13x40 = 530 T states which is enough to add another character to the buffer all by itself.

We are now ready for the Interrupt Handler. There will be liberal notes explaining it as we go along. Again it is set up to be assembled using Zeus. If you do so, omit all the comments and you will be able to get everything in this chapter in with only 2 different assemblies.

It is this routine that we will want to jump to upon receiving an interrupt so we will have to do a:

```
POKE 65021, 195
POKE 65022, 112
POKE 65023, 232
```

in Basic to complete the setup. You will see them in the Basic portion of the program when we present it.

```
            ORG 59500
            DISP 54036 : Store at 48000
            ENT
            BUFFER EQU FF70: Our interrupt jump routine uses
:the FE00-FEFE locations so if you expand over 42 characters
:you have to find another place for it.
59500 70FF  BUFFPT      DEFW BUFFER: USE ONLY 24 CHARACTERS
59502 0000  CHSTRE      DEFB 0
59503 00    BORD        DEFB 0: STORE FOR TOP BORDER COLOR
59504 F5    INTERP      PUSH AF: SAVE ALL REGISTERS
59505 C5                PUSH BC
59506 D5                PUSH DE
59507 E5                PUSH HL
59508 216FEB            LD HL, BORD :SET TOP BORDER
59511 7E                LD A,(HL)
59512 E610              AND 16
59514 EE05  TOPBRD      XOR 5
59516 D3FE              OUT (254),A: BORDER COLOR SET
59518 77                LD (HL),A :SAVE BORDER COLOR
59519 2170FF            LD HL, BUFFER : START WORKING THROUGH
59522 7E    NXTCH       LD A,(HL): GET ATTRIBUTE BYTE   BUFFER
59523 A7                AND A: ZERO = FAKE CHARACTER
59524 286C              JR Z,FAKE
59526 2C                INC L:ATTR ADDR TO DE
59527 5E                LD E,(HL)
59528 2C                INC L
59529 56                LD D,(HL)
59530 2C                INC L:TO CHAR ADDRESS
59531 1F                RRA: FLAG TO CARRY
```

```
59532 12                  LD (DE),A: PRINT ATTRIBUTE
59533 56                  LD D,(HL): DE IS NOW CHAR ADDR
59534 2C                  INC L:CHAR DATA ADDR TO BC
59535 4E                  LD C,(HL)
59536 2C                  INC L
59537 46                  LD B,(HL)
59538 302F                JR NC,NTOR:EVERYTHING WE HAVE DONE TO
:THIS POINT HAS NOT CHANGED THE CARRY FLAG SO IT STILL HOLDS
:WHAT WE PUT THERE WITH RRA, NOT SET MEANS OVERPRINT
59540 EB                  EX DE,HL: HL NOW SCREEN POINTER
59541 0A        POR       LD A,(BC): GET PIXEL LINE
59542 B6                  OR (HL): ADD SCREEN PIXELS
59543 77                  LD (HL),A:PRINT PIXEL LINE
59544 03                  INC BC: NXT PIXEL LINE
59545 24                  INC H:NEXT SCREEN PIXEL LINE
59546 0A                  LD A,(BC):AS AT POR, INSTEAD OF A LOOP
:WE ARE DOING 7 MORE TIMES AS WE SAID WE WOULD TO SAVE TIME
59547 B6                  OR (HL)
59548 77                  LD (HL),A
59549 03                  INC BC
59550 24                  INC H
59551 0A                  LD A,(BC)
59552 B6                  OR (HL)
59553 77                  LD (HL),A
59554 03                  INC BC
59555 24                  INC H
59556 0A                  LD A,(BC)
59557 B6                  OR (HL)
59558 77                  LD (HL),A
59559 03                  INC BC
59560 24                  INC H
59561 0A                  LD A,(BC)
59562 B6                  OR (HL)
59563 77                  LD (HL),A
59564 03                  INC BC
59565 24                  INC H
59566 0A                  LD A,(BC)
59567 B6                  OR (HL)
59568 77                  LD (HL),A
59569 03                  INC BC
59570 24                  INC H
59571 0A                  LD A,(BC)
59572 B6                  OR (HL)
59573 77                  LD (HL),A
59574 03                  INC BC
59575 24                  INC H
59576 0A                  LD A,(BC)
59577 B6                  OR (HL)
59578 77                  LD (HL),A
59579 EB                  EX DE,HL: GET BACK BUFFER POINTER
59580 2C                  INC L
59581 C282EB              JP NZ, NXTCH: LOOP BACK UNTIL END OF
:BUFFER AS WAS DISCUSSED, IF L =0 HAVE CROSSED PAGE BOUNDARY
59584 C304E9              JP ROWS: CONTINUE WITH HORIZON GEN
```

```
59587 0A        NTOR    LD A,(BC): PRINT ON TOP OF OLD CHAR
59588 12                LD (DE),A: PRINT NEW PIXEL LINE
59589 12                LD (DE),A: BALANCE INSTRUCTION TO MAKE
:NTOR TAKE AS MUCH TIME AS FOR
59590 03                INC BC
59591 14                INC D
59592 0A                LD A,(BC): AGAIN FOR 7 MORE TIMES
59593 12                LD (DE),A
59594 12                LD (DE),A
59595 03                INC BC
59596 14                INC D
59597 0A                LD A,(BC)
59598 12                LD (DE),A
59599 12                LD (DE),A
59600 03                INC BC
59601 14                INC D
59602 0A                LD A,(BC)
59603 32                LD (DE),A
59604 12                LD (DE),A
59605 03                INC BC
59606 14                INC D
59607 0A                LD A,(BC)
59608 12                LD (DE),A
59609 12                LD (DE),A
59610 03                INC BC
59611 14                INC D
59612 0A                LD A,(BC)
59613 12                LD (DE),A
59614 12                LD (DE),A
59615 03                INC BC
59616 14                INC D
59617 0A                LD A,(BC)
59618 12                LD (DE),A
59619 12                LD (DE),A
59620 03                INC BC
59621 14                INC D
59622 0A                LD A,(BC)
59623 12                LD (DE),A
59624 C3EBE8            JP ZZ: BALANCING TIMING
59627 2C        ZZ      INC L: LOOP BACK UNTIL END OF BUFFER
59628 C282E8            JP NZ, NXTCH: L=0 ONLY IF BUFFER END
59631 C304E9            JP ROWS: CONTINUE WITH HORIZON GEN
59634 3E05      FAKE    LD A,5: SKIP TO NEXT CHAR IN BUFFER
59636 85                ADD A,L
59637 6F                LD L,A
59638 11FF50            LD DE, #50FF: NOW WASTE TIME
59641 1A                LD A,(DE)
59642 01003D            LD BC,#3D00
59645 EB                EX DE,HL
59646 00                NOP
59647 C302E9            JP Y
59650 1891              JR POR: CONTINUE
```

ROWS will be the first line of the next routine which is the

Horizon Handler. But first some more theory.

The Horizon Handler

Although extending the horizon to the end of the picture tube doesn't allow us to write pixels on the border area, it does get rid of the confines of the border and makes the picture look bigger. The reason we can't write to the border is that it gets printed quite rapidly and there isn't enough time to switch it fast enough with OUT statements as we would have to do for pixels.

Keep in mind that the Interrupt Handler is running in the CPU while the ULA is writing the lines of the screen off the top of the picture tube and the top border lines down to the first print line. Thus if we want our border at this point we could jump right into the border change routine. If we want the border lower down, we have the option of doing two things:

1. Do nothing but time how much further down the screen we want to go. This allows us to run the border up to the top print line at any time.

2. Lengthen the limit of the print buffer thus allowing it to run longer so that we can get more things to the screen. This compromises our ability to run the border to the top of the screen at any time. The border change routine will still work below this lowered limit.

Which we select will of course depend upon our priorities. Let's look at option 1 first.

The first thing we will do is determine how much time we have to wait to change our border color once it gets down to the first row of text. For each pixel line down we will wait the required 224 T states. This little routine of waits looks like this:

```
Load A = # of scan lines to wait

SCAN1   LD B,15     7 T STATES
  LN    DJNZ LN     14*13+8= 190 T STATES
        AND 255     7 T STATES
        INC HL      6 T STATES
        DEC A       4 T STATES
        JP NZ, SCAN1  10 T STATES
```

For a total wait of 224 T states. This wait loop will be used in the Horizon Handler. Then all we have to do is an OUT (254), A statement with A = border color as shifted and the border color is changed. Both the left border and the right will be on the same screen line if we time this change at the 'flyback time' of the picture.

The TV or monitor picture is written by tracing the electron

beam(s) across the screen from left to right. At the right the beam(s) must be moved back to the left to start the next line. A special circuit blanks out this trace so it doesn't show on the screen. This is called 'flyback' and takes roughly 30% of the 112 T states needed to print a screen line. Remember a pixel print line is 2 screen lines so a single screen line print is 112 T states of which roughly 33 T states are used as flyback time. If we miss this time slot and send the OUT command too late, the left border screen line will still be the top border color while the right border screen line will be the new bottom color.

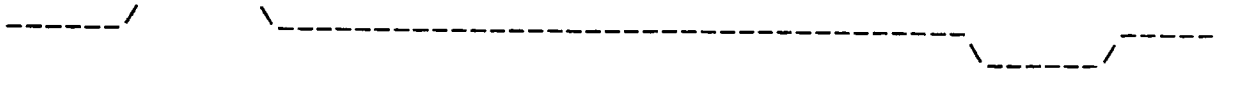
If the horizon is on a border between character lines this would be all we would have to do. We would merely have all the text attributes above the change use the top paper color and all the text attributes below the change use the bottom paper color. These colors can be any of the 8 standard colors. For purposes of the discussion to follow we will be using cyan paper for a sky color and blue paper for a sea color.

If, however, our horizon number (rows) is not divisible by eight we will have to have preset the character line involved with cyan paper. Then as the ULA is done using them for the last time swiftly start switching them to blue paper. As the beam passes the right border we have to stop to do an OUT (254),A to switch the border color and then finish the switch of the rest of the character attributes to blue. In this way we maximize the amount of time we have available for switching the attributes. We will need all the time we can get because the fastest we can do this is with the two instructions:

```
LD (HL), A      7 T states
INC L           4 T states
```

Since we have 32 attributes to change, simple arithmetic says it is going to need 352 T states which is too slow. Since we have to write these attributes to low memory we are further slowed down with interrupts by the ULA using the data bus.

After experimenting, the most attributes that we can hope to switch in the allotted time for a continuous level horizon is about 22. Since most arcade games tend to have all the action in the center of the screen we will put these 22 attributes there. This would make our horizon look like:



where the height of the hill and depth of the valley are really only one pixel line off from completely level.

This is somewhat less then perfection. There, however, are several solutions to this problem. The best solution is to fill

the five columns on the left with one row of pixels immediately above the horizon using cyan ink and do the same just below the horizon on the 5 right columns with blue ink.

The sequence of operations then becomes:

From the previous interrupt the attributes of the line concerned have been filled with cyan paper using any ink desired. We will use White ink.

1. Wait for the exact time.
2. Fill right 5 attributes with cyan paper, blue ink.
3. Fill left 5 attributes with blue paper, cyan ink.
4. Fill middle 22 attributes with blue paper, white ink.
5. Fill left 5 attributes with blue paper, white ink.
6. Fill right 5 attributes with blue paper, white ink.
7. Wait until TV has finished the line but use this time to prepare the print buffer for the next interrupt.
8. Fill all 32 attributes with cyan paper, white ink ready for the next interrupt.

At a certain critical point during step 4, the the electron beam(s) will reach the right-hand side of the screen having generated the cyan row immediately above the horizon. While the beam(s) are in flyback to the left side of the screen, we must take a break from filling the attributes to and output the new border value which we will store at (BOTBRD+1). We can even add 16 to the border value to get an audible click from the speaker. This combined with the value in (TOPBRD+1) allows us a choice of no sound or a 60 to 120 Hz sound. The click will change in pitch depending upon how much time there is between the top click and the bottom click.

There is one disadvantage doing things this way. Up to two rows of any sprite printed on the horizon in the left and right five columns will be cyan and blue respectively but will change to the desired colors as it moves out of these two columns.

The technique will work for anything down to 2 text rows above the horizon. The number of rows is stored in (ROWS+1). Zero text rows pose no problem, as the whole screen will be sea, and we just jump to NOWAIT where the border color is changed. However, in the very unlikely event that you may require just one text row above the horizon, then then you will have to revert to the old technique of filling it with ink and using cyan ink and blue paper. The reason for this is that there is not enough time to manipulate the attributes in the manner required by the new technique. In this case the interrupt handler jumps to the label WT1LN and waits for the TV to reach row one before changing the border. In the demo program you will see the hill and valley as you move the horizon up to the very top line.

Scattered throughout the listing you will find labels such as HCOL1 to HCOL4. These are used in the next section to change the

sky and sea colors. HRZN3 will be used by the horizon movement routines. It holds the address of the 26th horizon attribute and will be set to 001B (in the ROM) whenever no attribute work is required. We start by setting (ROWS+1) to 96 which is about halfway down the screen.

After the horizon has been generated, the interrupt handler has two more tasks. First, it must "tidy up" the print buffer by deleting all the entries just printed, interting zero attribute bytes to signify 24 null or fake entries. Part of this is also to reset BUFFPT to the first free entry which is now at BUFFER, and sets the number of entries, CHSTRE, to zero. Secondly, it retrieves all the registers stored at the beginning of the interrupt and terminates.

Here then is the second part of the interrupt handler which generates the border. We also add on the initialization routine at the end.

```
59652 3E60      ROWS      LD A,96: # OF ROWS TO WAIT
59654 D602                SUB 2 :USING 2 HERE AVOIDS 1 LINE
59656 DACCE9              JP C, NOWAIT              PROBLEM
59659 CAC2E9              JP Z,WTILN: IF A=2 WAIT 1 LINE
59662 3D                  DEC A
59663 CA1DE9              JP Z,GO4IT: START HORIZON GENERATION
59666 060F      SCAN1     LD B,15 : DELAY 224 T STATES
59668 10FE      LN        DJNZ, LN
59670 E6FF                AND 255
59672 23                  INC HL
59673 3D                  DEC A
59674 C212E9              JR NZ, SCAN1
59677 060A      GO4IT     LD B,9 :TIMING BALANCER--MAY HAVE TO
:BE ADJUSTED FOR YOUR TV OR MONITOR BY CHANGING THIS LOOP AND
:THE TWO INSTRUCTIONS FOLLOWING IT
59679 10FE      SELF5     DJNZ, SELF5
59681 7E                  LD A,(HL): TIMING INSTRUCTION
59682 7E                  LD A,(HL): TIMING INSTRUCTION
59683 211B00    HRZN3     LD HL, 27: SETUP TO PRINT TO ROM
59686 7D                  LD A, L: FIND 26TH ATTRIBUTE OF LINE
59687 E6E0                AND 224: SAVE ONLY TOP 3 BITS
59689 47                  LD B, A
59690 116FE8              LD DE, BORD:FIND BOTTOM BORDER COLOR
59693 1A                  LD A,(DE)
59694 E610                AND 16: SAVE ONLY BEEP BIT
59696 EE01      BOTBRD    XOR 1: LOAD 59697 WITH BORDER COLOR
59698 12                  LD (DE),A: SAVE BORDER COLOR
59699 1E29      HCOL1     LD E, 41: CYAN PAPER BLUE INK
59701 0E0D      HCOL2     LD C, 13: BLUE PAPER CYAN INK
59703 73                  LD (HL),E: FILL RIGHT 5 ATTR WITH CYAN
:PAPER, BLUE INK
59704 2C                  INC L
59705 73                  LD (HL),E
59706 2C                  INC L
59707 73                  LD (HL),E
```

```
59708 2C                LD L
59709 73                LD (HL),E
59710 2C                INC L
59711 73                LD (HL),E
59712 68                LD L, B
59713 1E0F      FCOL3   LD E, 15: BLUE PAPER, WHITE INK
59715 71                LD (HL),C: FILL 5 LEFT ATTR WITH BLUE
:PAPER, CYAN INK
59716 2C                INC L
59717 71                LD (HL),C
59718 2C                INC L
59719 71                LD (HL),C
59720 2C                INC L
59721 71                LD (HL),C
59722 2C                INC L
59723 71                LD (HL),C
59724 2C                INC L: START ON 22 MID ATTR WITH BLUE
:PAPER, WHITE INK
59725 73                LD (HL),E
59726 2C                INC L
59727 73                LD (HL),E
59728 2C                INC L
59729 73                LD (HL),E
59730 2C                INC L
59731 73                LD (HL),E
59732 2C                INC L
59733 73                LD (HL),E
59734 2C                INC L
59735 73                LD (HL),E
59736 2C                INC L
59737 73                LD (HL),E
59738 2C                INC L
59739 73                LD (HL),E
59740 2C                INC L
59741 73                LD (HL),E
59742 2C                INC L
59743 73                LD (HL),E
59744 2C                INC L
59745 73                LD (HL),E
59746 2C                INC L
59747 73                LD (HL),E
59748 2C                INC L
59749 73                LD (HL),E
59750 2C                INC L
59751 D3FE              OUT (254),A: BEAM IN FLYBACK SO  CHANGE
:BORDER COLOR
59753 73                LD (HL),E
59754 2C                INC L
59755 73                LD (HL),E
59756 2C                INC L
59757 73                LD (HL),E
59758 2C                INC L
59759 73                LD (HL),E
59760 2C                INC L
```

```
59761 73                LD (HL),E
59762 2C                INC L
59763 73                LD (HL),E
59764 2C                INC L
59765 73                LD (HL),E
59766 2C                INC L
59767 73                LD (HL),E
59768 2C                INC L
59769 73                LD (HL),E
59770 7D                LD A, L: NOW 5 LEFT ATTR WITH BLUE
:PAPER, WHITE INK
59771 68                LD L, B
59772 73                LD (HL),E
59773 2C                INC L
59774 73                LD (HL),E
59775 2C                INC L
59776 73                LD (HL),E
59777 2C                INC L
59778 73                LD (HL),E
59779 2C                INC L
59780 73                LD (HL),E
59781 6F                LD L, A: FINALLY 5 RIGHT ATTR WITH
:BLUE PAPER, WHITE INK
59782 2C                INC L
59783 73                LD (HL),E
59784 2C                INC L
59785 73                LD (HL),E
59786 2C                INC L
59787 73                LD (HL),E
59788 2C                INC L
59789 73                LD (HL),E
59790 2C                INC L
59791 73                LD (HL),E
59792 68                LD L, B: STORE START OF ATTR LINE
59793 E5        INIT3   PUSH HL
59794 216EE8            LD HL, CHSTRE: ENSURE THAT THE BUFFER
:IS FILLED WITH FAKE CHARACTERS (ZERO ATTRIBUTE SPACE)
59797 7E                LD A,(HL)
59798 A7                AND A
59799 280F              JR Z, END
59801 110600    INIT2   LD DE,6
59804 72                LD (HL),D : D=0 SO RESETS CHSTRE
59805 2170FF            LD HL, BUFFER
59808 226CEB            LD (BUFFPT),HL
59811 47                LD B, A
59812 72        NXTFL   LD (HL),D : D=0 SO RESETS ATTR IN
:BUFFER
59813 19                ADD HL,DE: TO NEXT ATTR IN BUFFER
59814 10FC              DJNZ NXTFL
59816 E1        END     POP HL
59817 7C                LD A, H: H=0 MEANS NO ATTR TO FILL
59818 A7                AND A: TEST FOR ZERO
59819 CABBE9            JP Z, NOPLG
59822 064D              LD B, 77 : WAIT UNTIL TV HAS FINISHED
```

```
:WITH THIS ATTR LINE. NOTE: IF FLICKERING OCCURS  THEN  INCREASE
:THIS DELAY FOR YOUR TV.
59824 10FE      SELF4      DJNZ SELF4
59826 0E1F                 LD C, 31: FILL ATTR LINE WITH CYAN
:PAPER, WHITE INK TO PREPARE FOR NEXT INTERRUPT
59828 54                   LD D, H
59829 5D                   LD E, L
59830 1C                   INC E
59831 362F      HCOL4      LD (HL),47 :CYAN PAPER, WHITE INK
59833 EDB0                 LDIR
59835 E1        NOPLG      POP HL: RETRIEVE REGISTERS
59836 D1                   POP DE
59837 C1                   POP BC
59838 F1                   POP AF
59839 FB                   EI
59840 ED4D                 RETI: RETURN FROM INTERRUPT

59842 060F      WT1LN      LD B, 15
59844 10FE      SELF11     DJNZ SELF11
59846 E6FF                 AND 255
59848 23                   INC HL
59849 3D                   DEC A
59850 2B                   DEC HL
59851 3C                   INC A
59852 2A30E9    NOWAIT     LD HL,(BOTBRD): ENTER HERE FOR ROW
:ZERO HORIZON. FIND NEW BORDER COLOR
59855 116FE8               LD DE, BORD
59858 1A                   LD A,(DE)
59859 E610                 AND 16: SAVE ONLY BEEP
59861 AC                   XOR H
59862 12                   LD (DE),A: SAVE IN
59863 D3FE                 OUT (254),A: OUTPUT IT
59865 2600                 LD H, 0: SET FLAG FOR NO ATTR TO FILL
59867 C391E9               JP INIT3: JUMP BACK TO MAIN ROUTINE
```

The Initialization Routine

We will, of course, be using interrupt mode IM 2, and you will see we have elected to use a 257 byte vector table pointing to a jump instruction at FDFDH (65021) to the interrupt handler. By putting the BUFFER at FF10H we have neatly used up the last 1/2K of RAM, wasting only 15 bytes. The following routine sets up the vector table at FE00, selects IM 2 and then jumps into the interrupt handler in order to ensure that the buffer is clear.

```
59870 F3        INT1       DI: WE CAN'T BE INTERRUPTED UNTIL DONE
59871 F5                   PUSH AF: SAVE ALL REGISTERS SINCE WE
:WILL BE EXITING VIA THE INTERRUPT HANDLER WHICH RETRIEVES THEM
59872 C5                   PUSH BC
59873 D5                   PUSH DE
59874 E5                   PUSH HL
59875 3EFE                 LD A, 254: SET I REGISTER
59877 ED47                 LD I, A
```

```
59879 2100FE                LD HL, #FE00: SET UP VECTOR TABLE
:STARTING AT 65024 WITH 257 CONSECUTIVE BYTES OF FDH
59882 45                    LD B, L
59883 3D                    DEC A
59884 77        TBLP        LD (HL),A
59885 23                    INC HL
59886 10FC                  DJNZ TBLP:TABLE LOOP
59888 77                    LD (HL), A :THE 257TH BYTE
59889 3E18                  LD A, 24: SET A FOR # OF CHARACTERS
:FOR CHSTRE
59891 ED5E                  IM 2 :RESET INTERRUPT MODE
59893 2600                  LD H, 0
59895 E5                    PUSH HL
59896 216EE8                LD HL, CHASTRE
59899 C399E9                JP INIT2: TO INTERRUPT HANDLER TO
:CLEAR THE PRINT BUFFER
```

The DISABLE INTERRUPT ROUTINE

We of course need a routine to get back to the normal interrupt mode.

```
59902 3E3E      DISINT      LD A, 63 : TO RESET THE I REGISTER
59904 ED56                  IM 1
59906 ED47                  LD I, A
59908 C9                    RET
```

THE HORIZON SET ROUTINE--HRZST1

So far we have the routines to get into and write a horizon but we have nothing to control the up and down movement of the horizon. It will just sit at whatever we have preset (ROWS+1) to.

Although the CPU always receives an interrupt signal at exactly the same time of the TV Frame, it will never react to it until it has finished processing the current instruction. This can lead to a variation of as much as 23 T states (as in the case of the instruction which takes the longest time to complete EX (SP), IX.). In human terms this isn't very long but it is long enough to process about 1/10th of the way through a row thereby affecting our artificial horizon shifting the central level portion momentarily to the left or right. In fact, the horizon generator can tolerate about a variation of 4 T states either more or less with no ill effect; as long as we get to the HALT instruction (which causes the CPU to execute NOP's each of which takes 4 T states until an interrupt is received) before an interrupt occurs we will have no problem.

If you wish to execute some other routine while expecting the

interrupt to take you back to the interrupt handler you may get some flickering on the screen. This can be corrected by making the "ink rows", those 5 columns we put on each side, 6 or 7 wide until they completely cover the flickering area. Naturallly to do this will require changing the horizon routines as they calculate the correct addresses correctly and correctly manipulate the new attributes.

We now begin the development of HRZST1. Its function will be to delete the last ink rows inserted on the screen and calculate the addresses of the new ones, after we have moved the horizon. The address of the left 5 ink rows will be stored in HRZN1, and that of the right 5 rows in HRZN2. When no ink rows are required as is the case when the horizon is between character lines, then the high byte of these two addresses will be zero to signify a flag of no work necessary.

The address of the current attribute line will always be stored in HRZN4 and will also be put in the interrupt handler itself at (HRZN3+1). Again the high byte will be set to zero to indicate no operations necessary and of course pointing the interrupt handler to print into ROM which we know has no permanent results. This of course is to assure the right timing of the border change.

```
59909 0000      HRZN1     DEFW 0: LEFT 5 INK COLUMN ADDR STORE
59911 0000      HRZN2     DEFW 0: RIGHT 6 INK COLUMN ADDR STORE
59913 0000      HRZN4     DEFW 0: CURRENT ATTR ADDRESS LINE
:HIGH BYTE OF THESE ADDRESSES = 0 INDICATES NO WORK NECESSARY
59915 2A05EA    HRZST1    LD HL,(HRZN1): DELETE OLD INK ROWS AND
:SET UP NEW VALUES FOR LOCATIONS OF INK ROWS AND ATTRIBUTES
:ENTRY WITH  C= # OF TEXT ROWS ABOVE THE HORIZON
59918 AF                  XOR A: A =0, CLEAR 5 LEFT INK ROWS
59919 77                  LD (HL),A
59920 2C                  INC L
59921 77                  LD (HL),A
59922 2C                  INC L
59923 77                  LD (HL),A
59924 2C                  INC L
59925 77                  LD (HL),A
59926 2C                  INC L
59927 77                  LD (HL),A
59928 2A07EA              LD HL,(HRZN2): NOW CLEAR 5 RIGHT INK
:ROWS
59931 77                  LD (HL),A
59932 2C                  INC L
59933 77                  LD (HL),A
59934 2C                  INC L
59935 77                  LD (HL),A
59936 2C                  INC L
59937 77                  LD (HL),A
59938 2C                  INC L
59939 77                  LD (HL),A
59940 79        HRZST2    LD A, C: GET ROWS COUNTER
```

```
59941 3205E9                LD (ROWS+1),A: SETUP CORRECT VALUE IN
:INTERRUPT HANDLER
59944 FEC1                  CP 192: IS HORIZON BELOW TEXT AREA?
59946 3031                  JR NC, NOWRK: NO INK WORK NECESSARY
59948 07                    RRCA: LOCATE ATTR LINE
59949 07                    RRCA
59950 E603                  AND 3: SAVE 2 BOTTOM BITS
59952 5658                  OR 88: ADD OFFSET
59954 67                    LD H,A: HIGH BYTE SET
59955 79                    LD A, C:CALCULATE LOW BYTE
59956 87                    ADD A, A: 2X
59957 87                    ADD A, A: NOW 4X
59958 E6E0                  AND 224: SAVE ONLY 3 HIGH BITS
59960 6F                    LD L,A
59961 2209EA                LD (HRZN4),HL: STORE IT
59964 79                    LD A, C
59965 E607                  AND 7: IF HORIZON BETWEEN CHARACTER
:LINES NO INK WORK NECESSARY
59967 281C                  JR Z, NOWRK
59969 7D                    LD A, L: LOCATE 28TH ATTRIBUTE
59970 F61B                  OR 27
59972 45                    LD B, L
59973 6F                    LD L, A
59974 2224E9                LD (HRZN3+1),HL: PUT IN INTERRUPT
:HANDLER
59977 79                    LD A, C: LOCATE 28TH BYTE OF DISPLAY
:FILE ROW BELOW HORIZON
59978 1F                    RRA: DIVIDE BY 2
59979 37                    SCF
59980 1F                    RRA: DIVIDE BY 2 AND ADD 128
59981 A7                    AND A: CLEAR CARRY FLAG
59982 1F                    RRA : DIVIDE BY 2
59983 A9                    XOR C: FLIP BYTE ACCORDING TO ORIGINAL
:NUMBER OF LINES
59984 E6F8                  AND 248: SAVE 5 HIGH BITS
59986 A9                    XOR C: FLIP BITS ACCORDING TO C AGAIN
59987 67                    LD H, A
59988 2207EA                LD (HRZN2), HL: NOW FIRST BYTE OF ROW
:ABOVE HORIZON
59991 25                    DEC H
59992 68                    LD L, B
59993 2205EA                LD (HRZN1),HL
59996 C9                    RET
59997 AF        NOWRK       XOR A: A=0 PUT IN ALL HIGH ADDRESSES
59998 3206EA                LD (HRZN1+1),A
60001 3208EA                LD (HRZN2+1),A
60004 3225E9                LD (HRZN3+1),A
60007 C9                    RET
```

The Horizon Set routine will be called by the Horizon Move routine. If you just want to reset a horizon level without constantly moving it, HRZST1 can be called directly merely by loading C with the number of rows above the horizon and calling the HRZST1 and then filling in the ink rows and attribute as

required. If the horizon level is being set for the first time, then you should skip the section that "blanks out" the old ink rows by entering at HRZST2 instead.

THE HORIZON MOVE ROUTINE--HRZMV1

We need two more variables for the move routine:

HRZSPD to hold the amount of the move from 1 to 8 lines.
CNTRL to hold the direction 4 = down and 8 = up.

Although the routine calls for keyboard control, a joystick read could be installed by reading the up and down bits and setting them in CNTRL.

Due to a problem of a horizon on row one described earlier, movement has been limited to levels below 1. Although there are roughly 236 lines from the top of the text area to the bottom of a British screen, there are only about 222 on an American set. We thus limit our program at address 60033 to 222. You may need to adjust this level higher or lower for your set. It's this value that lets you run the horizon right down off the bottom of your screen. Don't go overboard with too high a value as, if you remember our discussion, the further down the screen the border is set, the more time the Interrupt Handler has to run, the less time you have to do other things before the CPU gets another interrupt.

The main function of the move routine is to take care of the attributes as the horizon moves. If, for example, we have just moved the horizon up to row 0 of the current line or into the line above it, then we will need to change a line of attributes from sky to sea. Similarly, if we have just moved down into a new character line then we must fill its attributes with sky, ready for the next interrupt.

HRZMV1 will be called by the forthcoming master horizon routine, HRZNMK. When the routine HRZMV1 is completely satisfied with the attributes, it makes a jump to HRZST1 in order to set up the new values for HRZN1 to HRNZ4 and blank out any old ink rows.

Presently two unused labels are used in the listing, HCOL5 and HCOL6. These will be used by a later routine which will set up new sky and sea colors.

```
60008 00        HRZSPD    DEFB 0 :HORIZON SPEED 0 TO 8
60009 00        CNTRL     DEFB 0 :HORIZON DIRECTION 4= DOWN 8=UP
60010 3A68EA    HRZMV1    LD A,(HRZSPD)
60013 47                  LD B, A: STORE SPEED IN B
60014 3A69EA              LD A,(CNTRL)
60017 110000              LD DE, 0
60020 CB5F                BIT 3, A: UP?
60022 C2A5EA              JP NZ, UP2
```

```
60025 CB57                  BIT 2, A: DOWN?
60027 C8                    RET Z: NO CHANGE
60028 3A05E9                LD A, (ROW+1): CHANGE ROWS ABOVE
:HORIZON BY HRZSPD AND CONTINUE WITH DOWN
60031 80                    ADD A, B
60032 FEDE                  CP 222: SAFETY CHECK FOR MINIMUM
:HORIZON LEVEL 1. YOU MAY HAVE TO CHANGE FOR YOUR SET
60034 D0                    RET NC: TOO LOW SO NO HORIZON CHANGE
:NECESSARY
60035 FEC1                  CP 192: IF ROWS >192 SKIP TO HRZST1
60037 4F                    LD C, A: ROWS TO C FOR CALL OF HRZST1
60038 D20BEA                JP NC, HRZST1
60041 E607                  AND 7: ON ROW 0 OF A LINE?
60043 2007                  JR NZ, NROWZ1
60045 CB58                  BIT 3, B: IS SPEED 8?
60047 CA0BEA                JP Z, HRZST1: IF NOT JUMP TO HRZST1
60050 1808                  JR ROWZ1
60052 B8        NROWSZ1     CP B: ARE WE MOVING FROM ROW 0 OF A
:LINE?
60053 2805                  JR Z, ROWZ1
60055 D20BEA                JP NC, HRZST1: IF STILL ON SAME LINE
:JUMP TO HRZST1 AS WELL
60058 1E20                  LD E, 32: COUNT FOR 1 LINE
60060 064D                  LD B, 77: DELAY TO ENSURE TV IS DONE
:WITH THE NEW LINE
60062 10FE      SELF42      DJNZ SELF42
60064 062F      HCOL5       LD B, 47 :CYAN PAPER, WHITE INK
60066 C3C9EA                JP UPINIT
60069 3A05E9    UP2         LD A,(ROWS+1)
60072 90                    SUB B: DECREASE ROWS ABOVE HORIZON BY
:HRZSPD
60073 D8                    RET C: RETURN IF NEGATIVE
60074 FE02                  CP 2: ALSO RETURN IF <2
60076 D8                    RET C
60077 FEB9                  CP 185: IF ROWS >184 SKIP TO HRZST1
60079 4F                    LD C, A: PUT IN C FOR HRZST1
60080 D20BEA                JP NC, HRZST1
60083 ED44                  NEG
60085 E607                  AND 7
60087 110000                LD DE, 0
60090 2007                  JR NZ, NROWZ2: JUMPIF NOT ON ROW 0
:OF A LINE
60092 CB58                  BIT 3,B:  ELSE  FILL  IN  CURRENT  LINE
WITH
:BLUE PAPER, WHITE INK
60094 2807                  JR Z, HCOL6
60096 11E0FF                LD DE, #FFE0: -32
60099 B8        NROWZ2      CP B
60100 D20BEA                JP NC, HRZST1: OTHERWISE FILL OLD LINE
:WITH BLUE PAPER, WHITE INK
60103 060F      HCOL6       LD B,15 :ATTRIBUTE
60105 2A09EA    UPINIT      LD HL,(HRZN4)
60108 19                    ADD HL,DE: THE ALL PURPOSE FILLER
60109 54                    LD D,H
```

```
60110 5D                LD E,L
60111 1C                INC E
60112 70                LD (HL),B
60113 79                LD A,C
60114 011F00            LD BC, 31
60117 EDB0              LDIR
60119 4F                LD C,A
60120 C30BEA            JP HRZST1: AS ALWAYS JUMP TO HRZST1
```

There are quite a few jumps to HRZST1 from various places, 5 to be exact depending what you have been doing. We now have all the code necessary to set and move the horizon level. HRZMV1 handles all the work on the attributes as the horizon moves, while HRZST1 makes sure we know where those attributes are, deletes the old ink rows and calculates the new ones.

THE HRZNMK ROUTINE

All that remains is to actually insert those ink rows into the display file before every interrupt (they may have been over-printed by sprites since the last one). The master routine for this is HRZNMK (MK for MaKe) will do this after having called HRZMV1, which makes any necessary changes to the attributes and variables. Since HRZNMK calls HRZMV1 which calls HRZST1 the only program we really have to call directly after each interrupt is this one.

Of course we have to set up calling this routine by first setting CNTRL and HRZSPD of HRZMV1 before doing so. The next program after this is the first of 3 demo programs which shows how it all comes together.

```
60123 CD6AEA    HRZNMK  CALL HRZMV1: START WITH A CALL
60126 2A05EA            LD HL, (HRZN1): RETURN IF NO INK ROWS
:NEEDED
60129 24                INC H: TEST ZERO BY INC AND DEC H
:REMEMBER IF HIGH BYTE IS ZERO NO INK NEEDED SIGNAL
60130 25                DEC H
60131 C8                RET Z
60132 3EFF              LD A, 255: A FULL 8 PIXELS
60134 77                LD (HL),A : FIRST THE LEFT 5
60135 2C                INC L
60136 77                LD (HL),A
60137 2C                INC L
60138 77                LD (HL),A
60139 2C                INC L
60140 77                LD (HL),A
60141 2C                INC L
60142 77                LD (HL),A
60143 2A07EA            LD HL,(HRZN2): NO NEED TO TEST AGAIN
:AS IF WE HAVE TO DO THE LEFT SIDE WE ALSO HAVE TO DO THE RIGHT
```

```
60146 77              LD (HL),A
60147 2C              INC L
60148 77              LD (HL),A
60149 2C              INC L
60150 77              LD (HL),A
60151 2C              INC L
60152 77              LD (HL),A
60153 2C              INC L
60154 77              LD (HL),A
60155 C9              RET
```

Not much of a routine but let's say it makes it easier to operate this way then including it in HRZMV1.

Don't you think its about time to see what we have done so far works? The DEMO program below gives a hint at what we are trying to achieve but is only the first part. It, however, illustrates what we are trying to achieve and will use all the routines we have written to this point.

THE DEMO ROUTINE

The routine gives you direct control over the horizon. Pressing key 1 to 5 will cause the horizon to move upwards, while pressing CAPS SHIFT to V downwards. Keys 8,9 and 0 are used to control the speed of the horizon. Think of these keys as a 3 bit binary number. 0 being the low bit and 8 the high. Pressing all 3 gives the fastest speed, 0 by itself the lowest. Most of this routine is just a plain reading of a keyboard. It could be easily rewritten to read a joystick as well.

```
60156 CDDEE9    DEMO    CALL INTI: SET UP IM 2 MODE
60159 0E54              LD C,84 : SET INITIAL HORIZON
60161 CD24EA            CALL HRZST2
60164 76        DMLP    HALT: WAIT FOR INTERRUPT
60165 0E00              LD C, 0: C = DIRECTION
60167 3EFE              LD A, 254
60169 DBFE              IN A,(FE): READ KEYBOARD CS TO V
60171 2F                CPL: FLIP DIGITS
60172 E61F              AND 31: SAVE 5 LOW BITS ONLY
60174 2802              JR Z, ND1: NOT PRESSED, GOTO NEXT TEST
60176 CBD1              SET 2, C: C= 4 = DOWN
60178 3EF7      ND1     LD A, 247: TEST 1-5
60180 DBFE              IN A,(254)
60182 2F                CPL
60183 E61F              AND 31
60185 2802              JR Z, NU1: NOT UP, TEST SPEED
60187 CBD9              SET 4, C: STORE DIRECTION
60189 79        NU1     LD A,C
60190 3269EA            LD (CNTRL),A
60193 3EEF              LD A, 239: TEST 8-0
60195 DBFE              IN A,(254)
```

```
60197 2F          CPL
60198 E607        AND 7: SAVE RIGHT 3 BITS 0-7
60200 3C          INC A: ADD 1  1-8
60201 3268EA      LD (HRZSPD),A: STORE
60204 CDDBEA      CALL HRZNMK: DO SETUP
60207 3E7F        LD A, 127: TEST FOR BREAK
60209 DBFE        IN A,(254)
60211 1F          RRA: ROTATE LOW BIT TO CARRY
60212 38CE        JR C, DMLP: NO INPUT SO CONTINUE
60214 CDFEE9      CALL DISINT: BACK TO IM 1
60217 C9          RET
```

At this point we can test our suite of horizon routines as it does not use the additional routine we would need if we want to change our sky and sea attributes from machine code. At this time we assemble our program and save both a copy of the assembly and source code. We will be recalling the source code and continuing the listing to use the rest of memory at the next routine. For owners of disks merely load the program called "MASTDEMO". This program will cover everything in this chapter so at this point don't cheat but just run item 2 of the menu "Horizon Demo". Cheating will just spoil the rest of the demo for you.

For people who are assembling their own code, save your assembled code under the name of "spr.bin",48000,717. Also use the T command to find out the length of your source code and save that as well.

We need some Basic to go along with this so here is the start of the program "MASTDEMO". Use the same line numbers as we will be adding to this program as well.

Part 1. MASTDEMO

```
  1 CLEAR 57999: CAT "SPR.BIN",59500: REM load code
  2 OUT 244,0 :REM Switch back to home bank to avoid crashs.
 10 CLS: PRINT AT 3,2; "Select your choice:"; AT 5,4; "1. MOIRE"
; TAB 9;"Hit Any Key To Continue"; TAB 4;"2. Horizon Demo"; TAB
10; "Z=down, 1=up"; TAB 4;"3. Horizon with Target"; TAB 10;"Z  =
down, n = up"; TAB 4; "4. SPRITE Demo"
 13 PRINT TAB 4;"8. END"
 15 IF INKEY$ ="" THEN GO TO 15
 20 IF INKEY$ <"1" OR INKEY$ >"8" THEN GO TO 15
 25 LET A = VAL INKEY$
 30 GOSUB 40+10*A
 40 GO TO 10
 52 RETURN
 60 GOSUB 200: RANDOMIZE USR 59422: PAUSE 0: RETURN
 70 RETURN
 80 RETURN
```

```
 90 RETURN
100 RETURN
110 RETURN
120 STOP
200 POKE 65021,195: POKE 65022,112: POKE 65023:232: REM THIS SETS UP THE JUMP FOR OUR INTERRUPT HANDLER
```

We also need some code to set up our RANDOMIZE USR numbers. We will give you the whole thing although we won't need most of it until we have assembled more code. Disassembling this code will show the student that we always save HL and sometimes IX' before calling the program and then restore them before returning to basic. Doing some of these routines with a bare call caused crashes.

```
202 RESTORE 200: FOR X = 59400 TO 59427: READ Y: POKE X,Y: NEXT X
205 DATA 229,253,229,205,78,241,253,225,225,201,229,205,206,235,225,201,229,205,252,234,225,201,229,205,59,238,225,201
206 RETURN
299 STOP
```

There is more to this program and we will be changing and adding to it at a later point. Save the program before running it.

How was the demo? I hope it was worth it. But, as they say "You ain't seen anything yet!"

Back to work. We finish the horizon routines with one last one called:

THE HORIZON COLOR ROUTINE

The main function of this routine is to set up the other routines for any combination of sea and sky colors along with ink colors for above and below the horizon. If you are moving sprites through the horizon you should set these two ink colors to the same value or the sprite will change color on you as it goes through the horizon.

You, of course, have the option of causing the interrupt handler to generate a background "motor" sound, either at 60 Hz or 120 Hz by adding 16 to one or both of the paper values respectively. The registers should be set up as:

H = sea paper value (+16 for sound)
L = sky paper value (+16 for 120 Hz sound)

B = sea ink value.
C = sky ink value.

before calling HRZCOL.

For those of you doing your own assembly on Zeus, relaod your source code and continue.

```
ENTRY H = SEA PAPER, L = SKY PAPER, B = SEA INK, C = SKY INK
ADD 16 TO H OR L OR BOTH FOR MOTOR SOUND

60218 7C        HRZCOL    LD A, H
60219 3231E9              LD (BOTBRD+1), A
B0222 E607                AND 7: LEAVE SEA PAPER IN H
60224 67                  LD H, A : BUT WITHOUT SOUND
60225 7D                  LD A, L
60226 327BE8              LD (TOPBRD+1), A
60229 E607                AND 7: STRIP SOUND FROM SKY PAPER
60231 6F                  LD L, A: RESTORE IN L
60232 07                  RLCA :HCOL1 NEEDS SKY-PAPER AS PAPER
:AND SEA-PAPER AS INK
60233 07                  RLCA
60234 07                  RLCA
60235 5F                  LD E, A: TEMP STORE OF SKY-PAPER AS
:PAPER
60236 B4                  OR H: ADD SEA PAPER AS INK
60237 3234E9              LD (HCOL+1), A
60240 7C                  LD A, H: HCOL2 NEEDS SEA-PAPER AS PAPER
:AND SKY-PAPER AS INK
60241 07                  RLCA
60242 07                  RLCA
60243 07                  RLCA
60244 57                  LD D, A: TEMP STORE OF SEA-PAPER AS
:PAPER
60245 B5                  OR L: ADD SKY-PAPER AS INK
60246 3236E9              LD (HCOL2+1),A
60249 7A                  LD A, D: HCOL3 NEEDS SEA-PAPER AS PAPER
:AND SEA-INK AS INK
60250 B0                  OR B: ADD SEA-INK AS INK
60251 3242E9              LD (HCOL3+1), A
60254 32C8EA              LD (HCOL6+1), A: AS DOES HCOL6
60257 7B                  LD A, E: HCOL4 NEEDS SKY-PAPER AS
PAPER
:AND SKY-INK AS INK
60258 B1                  OR C: ADD SKY-INK AS INK
60259 32BBE9              LD (HCOL4+1),A
60262 32A1EA              LD (HCOL5+1),A: AS DOES HCOL5
60265 C9                  RET
```

This will get all the various colors in the right places. As you can see, it's a little bit too much to remember exactly what has to go where.

USING THE PRINT BUFFER FOR STATIONARY ITEMS

Up to this point we really haven't used that print buffer as we haven't printed anything to the screen. We have merely played around with the border and the attributes to get that horizon straight across the screen. We now go on to a set of routines that will show you how we can use that print buffer to send things to the screen each and every interrupt. These should only be things that may be overwritten by moving sprites as you remember we have only 24 spaces if we are not going to compromise running the border all the way up to the top line of the text area. Later on we will develop sprite routines. If you don't move the sprites every frame, which is generally too fast for most items anyway, you can flip-flop between printing sprites and printing stationary items on alternating interrupts.

This stationary printing becomes useful as for example in imposing a set of gunsights onto the center of the screen as the enemy ship or boat, in the form of a sprite, moves in back of the sight. This would need continual OR-printing on every interrupt. Since time is so short between interrupts (especially if you are operating the horizon generator at low screen level), it would be helpful NOT to have to load all the data for such a target sight into the print buffer after every interrupt.

To do this, we devise a system of routines which use a sub-section of the print buffer which we shall call "RO-Buffer" for 'Read Only Buffer'. Unlike normal entries in the print buffer, those in the RO-Buffer will not be erased by the interrupt handler after they have been printed. This does save time. Thus all we have to do before each interrupt is to make sure the correct attributes for each cell concerned have been inserted in the RO-Buffer. If the attribute mask is the zero byte, then we won't even have to do this since the character will always be printed with the same attributes regardless of the "old" attributes for that cell.

THE ALTER BUFFER ROUTINE

Consecutive entries into the buffer normally grow upward from the start of the print buffer. To keep them separate, we will make the RO-Buffer grow from the end of the print buffer down-ward. The two regions should never overlap. We have to define two new variables called the ROLNTH to store the number of entries in the RO-Buffer and ROBFPT the low address start of the RO-Buffer. Making ROLNTH = 0 means no RO-Buffer.

We first need a routine to set up this RO-Buffer. What this routine actually does is change the value of ROLNTH by the value we have in C, which may be positive or negative. Having adjusted ROLNTH, the routine fills all the entries between the end of the "normal" entries, i.e., BUFFPT and the beginning of the RO-Buffer, with "null" characters, to prevent any garbage from

being printed. Because of this, ALTRBF should always be called with the interrupts disabled.

```
ENTRY: C= ALTERATION OF LENGTH
EXIT: HL= START OF RO-BUFFER, B = 0, A = # OF UNUSED ENTRIES IN
    PRINT BUFFER

60266 00        ROLNTH    DEFB 0: RO-BUFFER LENGTH
60267 0000      ROBFPT    DEFW 0: STARTING ADDRESS OF RO-BUFFER
60269 216AEB    ALTRBF    LD HL, ROLNTH: ALTER ROLNTH
60272 7E                  LD A, (HL)
60273 81                  ADD A, C
60274 77                  LD (HL), A
60275 3A6EE8              LD A,(CHSTRE): FIND # OF UNUSED ENTRIES
:IN PRINT BUFFER (>= 0)
60278 47                  LD B, A
60279 3E18                LD A, 24: THIS NUMBER MUST BE CHANGED
:IF YOU CHANGE THE LENGTH OF THE PRINT BUFFER
60281 90                  SUB B
60282 96                  SUB (HL): ALSO SUBTRACT ROLNTH
60283 47                  LD B, A: STORE IN B
60284 2A6CE8              LD HL, (BUFFPT): FILL WITH NULL CHAR
60287 2807                JR Z, HOPFL
60289 110600              LD DE, 6: EACH ENTRY 6 LONG
60292 72        BLNK      LD (HL), D: D=0
60293 19                  ADD HL, DE: NEXT ENTRY ADDRESS
60294 10FC                DJNZ BLNK
60296 226BE8              LD (ROBFPT), HL: STORE IT
60299 C9                  RET
```

Obviously any routine that outputs characters to the print buffer can be easily adapted to use the RO-Buffer instead. The RO-Buffer is only a subsection of the print buffer. Before showing its use we need a routine that will refresh the RO-Buffer.

The Refresh of the RO-Buffer

This is a SeRVice Routine, hence the label SRVR1. It takes the attribute address from an entry, finds the attribute from the file, masks it with our standard variable MASK, which as usual is placed directly after ATT, the attribute of our characters. The completed attribute byte is then rotated left by one bit (remember that bit 0 is the OR flag to the interrupt handler) and inserted into the RO-Buffer. If we wish to select OR-printing then we set bit 7 of ATT (bit 7 of MASK should always be 0), which will then be rotated to bit 0 before inserting into the buffer. ATT and MASK will be defined later.

We again use the zero flag to detect the end of the RO-Buffer, i.e., when L becomes zero at the page boundary.

```
EXIT: B=MASK, C=ATTRIBUTE, A=0, HL= BYTE AFTER PRINT BUFFER

60300 2A6BEB   SRVR1    LD HL,(ROBFPT)
60303 ED4B92ED          LD BC,(ATT):B=MASK, C=ATTRIBUTE
60307 2C       NXSRV1   INC L: TAKE ATTR ADDRESS
60308 5E                LD E,(HL)
60309 2C                INC L
60310 56                LD D,(HL)
60311 2D                DEC L
60312 2D                DEC L: BACK TO ATTR
60313 1A                LD A,(DE): GET CURRENT ATTR
60314 A9                XOR C:CREAT NEW ONE
60315 A0                AND B
60316 A9                XOR C
60317 07                RLCA :ROTATE LEFT TO PRESERVE OR FLAG
60318 77                LD (HL),A ZSTORE NEW ATTR IN BUFFER
60319 7D                LD A, L: TO NEXT ENTRY
60320 C606              ADD A,6
60322 6F                LD L, A
60323 20EE              JR NZ, NXSRV1
60325 C9                RET
```

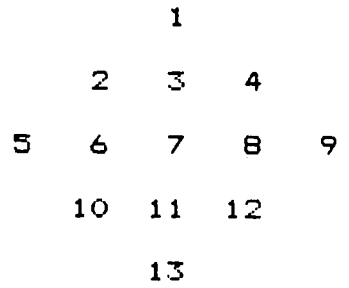
SRVR1 should be called whenever there is a chance that the attributes of the cell have been changed, e.g., by moving a sprite over them or changing the horizon. The routine also gives you the opportunity to vary the color of the permanent characters on your screen by altering ATT. However, since it deals with the entire RO-Buffer every entry in it will also use this same ATT value. If this is not desired, then the easiest modification is to change addresses 60314 etc.

```
From   XOR C      To    LD C,(HL)
       AND B            RRC C
       XOR C            XOR C
                        AND B
                        XOR C
```

thereby effectively using the original value of ATT with which the entry was inserted (by a modified HIPRNT routine which we still have to discuss). Of course this modification doesn't fit so the program would have to be reassembled.

The Send Data to RO-Buffer Routine--SRVR2

Now for the routine to send pixels to the RO-Buffer. It can be either standardized 2068 characters or UDG graphics. For our example we will be sending the pixels of our gunsight. It will be 5 character spaces high and 5 character spaces wide in the shape of a diamond. Viz.,



Two tables will be required. The first will hold the desired position on the screen, while the second will be the actual pixels for the image stored in the usual 8 across 8 down format. The position table and the pixel table must have the UDG's in the same order and without gaps but can be located anywhere in memory.

We will be using a special numbering of character spaces on the screen. Since each line across the screen is 32 characters wide, each row will start with a multiple of 32 starting with 0, i.e., 0, 32, 64, 96, 128, etc. For a change, it is easier to count this out in hexadecimal rather than decimal. The numbers then become, 00, 20, 40, 60, 80, A0, C0, E0, 100, 120, 140, etc. ending at 2E0. To this number all we have to add is the column number we want, a number from 0 to 31 or 00 to 1F in hex. We will still need 2 bytes for each position. The advantage of this is that the low byte will be identical to the low byte of the attribute address AND the low byte of the Display File address as well. If you prefer a coordinate system, SRVR2 can easily be rewritten for that as well.

The tasks of SRVR2 are to take the position of a character, calculate the attribute address, display file address, and the character data address and store them all in the correct order in RO-Buffer.

```
ENTRY: HL = START OF POSITION DATA
       DE = START OF RO-BUFFER AREA
       BC = ADDRESS OF CHARACTER TABLE
        A = # OF CHARACTERS

60326 C5        SRVR2     PUSH BC: USE BC AS A CONSTANT
60327 010800              LD BC, 8
60330 08        NXCHR9    EX AF, AF':SAVE COUNT
60331 1C                  INC E:TO ATTR ADDR LOW
60332 7E                  LD A,(HL):XFER LOW BYTE OF ATTR ADDR
60333 12                  LD (DE),A
60334 1C                  INC E: HI BYTE OF ATTR ADDR
60335 23                  INC HL :TO ATTR HI STORE
60336 7E                  LD A,(HL): CALC HI ATTR ADDR
60337 F658                OR 88: ADD 88
60339 12                  LD (DE), A: TO RO-BUFF
60340 1C                  INC E: TO DFILE HI ADDR
60341 E603                AND 3: CALC HI DFILE ADDR
```

```
60343 07                RLCA
60344 07                RLCA
60345 07                RLCA
60346 F640              OR 64: ADD OFFSET
60348 12                LD (DE),A :STORE IT
60349 1C                INC E: TO CHAR LO
60350 23                INC HL: TO CHAR ADDR STORE
60351 E3                EX (SP),HL: RETRIEVE CHAR PIXEL ADDR
:SAVE POSITION ADDRESS
60352 EB                EX DE,HL
60353 73                LD (HL), E: STORE ADDR IN BUFFER
60354 2C                INC L
60355 72                LD (HL),D
60356 2C                INC L
60357 EB                EX DE, HL
60358 09                ADD HL, BC: ADD 8 TO GET START OF NEXT
:PIXEL DATA ADDRESS
60359 E3                EX (SP),HL :SAVE NEW CHAR DATA ADDR,
:RETRIEVE POSITION DATA ADDR
60360 08                EX AF, AF': RETRIEVE COUNTER
60361 3D                DEC A
60362 20DE              JR NZ, NXCHAR9
60364 E1                POP HL :CLEAR STACK
60365 C9                RET
```

I now have to apologize to my readers. In the above set of routines we use ATT and MASK which we still havn't defined. Next in the memory is a program called TARGET which in reality combines the printing of a Target with our moving horizon for our next demo program. Unfortunately, we can't run this program without also using the 2 programs that follow. This doesn't bother the readers with copies of the programs and code, but to get the disassembly right bear with me until I tell you that you should again assemble the program, add to MASTDEMO and finally run the second demo.

Adding the Gunsight to The Moving Horizon.

We want the gunsight to be centered on the screen. Referring to the diagram a few pages back and the discussion of how we were going to number character spaces, the position for the top character is 14FH. We also note that character 2 is one row down and one space left or 1FH more at 16EH, followed by character 3 at 16FH and character 4 at 170H. The 3rd row starting at character 5 starts at 18D. These positions and the rest will be put into the data base TRGPOS for target position. That of the character pixel data will go into TRGDAT in the same order.

The first task of TARGET is to call INT1 to set up the interrupt mode IM 2 and clear the buffer. Next we set up the Horizon starting level and finally RO-buffer. For demo purposes, SRVR1

is put in the interrupt loop so that it is called at every interrupt but it need not necessarily be in your program if you don't move the horizon all the time. The two missing variables MASK is loaded with 127 (full, as bit 7 should alway be reset) and OR-print by setting ATT 128.

```
60366 CDDEE9   TARGET    CALL INT1
60369 F3                 DI :NO INTERRUPTS UNTIL DONE
60370 0E54               LD C,84: SET HORIZON LEVEL
60372 CD24EA             CALL HRZST2: AND SET IT UP, WE DON'T
:HAVE TO CHANGE ANY COLORS
60375 0E0D               LD C, 13: SET UP RO-BUFF, 13 SPACES
60377 CD6DEB             CALL ALTRBF
60380 EB                 EX DE,HL :SETS DE TO (ROBFPT), GET
:READY TO XFER DATA TO RO-BUFF
60381 212FEC             LD HL, TRGPOS
60384 0149EC             LD BC, TRGDAT
60387 3E0D               LD A, 13
60389 CDA6EB             CALL SRVR2: DO IT
60392 21807F             LD HL, #7F80: SELECT FULL MASK AND OR-
:PRINTING BY SETTING BIT 7 OF ATT
60395 2292ED             LD (ATT),HL
60398 FB                 EI :READY--ENABLE INTERRUPTS
60399 CD8CEB   TSLP      CALL SRVR1
60402 76                 HALT: WAIT
60403 3EFE               LD A, 254: TEST KEYBOARD FOR HORIZ MV
60405 DBFE               IN A,(254): FIRST CS TO V
60407 2F                 CPL
60408 E61F               AND 31
60410 2802               JR Z, ND2: NOT DOWN
60412 CBD1               SET 2, C
60414 3E7F     ND2       LD A, 127
```

Just for fun 127 is going to give you B to break for up. 247 returns it to 1-5.

```
60416 DBFE               IN A,(254)
60418 2F                 CPL
60419 E61F               AND 31
60421 2802               JR Z, NU2 :NOT UP
60423 CBD9               SET 3, C
60425 79       NU2       LD A, C
60426 3269EA             LD (CNTRL),A
60429 3EEF               LD A, 239: CHECK FOR SPEED CHANGE
60431 DBFE               IN A,(254)
60433 2F                 CPL
60434 E607               AND 7
60436 3C                 INC A
60437 3268EA             LD (HRZSPD),A
60440 CDDBEA             CALL HRZNMK
60443 3E7F               LD A,127
60445 DBFE               IN A,(254)
60447 1F                 RRA
60448 38CD               JR C, TSLP: DO IT AGAIN
```

```
60450 CDFEE9              CALL DISINT
60453 3A6AEB              LD A,(ROLNTH): RESET RO-BUFF
60456 ED44                NEG
60458 4F                  LD C,A
60459 CD6DEB              CALL ALTRBF
60462 C9                  RET

60463 4F016E01 TRGPOS     DEFW #014F,#016E
60467 6F0170018D01        DEFW #016F,#0170,#018D
60473 8E018F019001        DEFW #018E,#018F,#0190
60479 9101AE01AF01        DEFW #0191,#01AE,#01AF
60485 B001CF01            DEFW #01B0,#01CF
```

The pixel data is arranged in 4 byte chunks, every 2 lines are thus the pixels for a character space.

```
60489 00181010 TRGDAT     DEFB 0,24,16,16
60493 18101018            DEFB 24,16,16,24
60497 00000003            DEFB 0,0,0,3
60501 0E181030            DEFB 14,24,16,48
60505 1010FE93            DEFB 16,16,254,147
60509 1018107C            DEFB 16,24,16,124
60513 00000080            DEFB 0,0,0,128
60517 E0301018            DEFB 224,48,16,24
60521 00000092            DEFB 0,0,0,146
60525 FF000000            DEFB 255,0,0,0
60529 2161434A            DEFB 33,97,67,74
60533 FF424361            DEFB 255,66,67,97
60537 D7111100            DEFB 215,17,17,0
60541 D7001111            DEFB 215,0,17,17
60545 080C84A4            DEFB 8,12,132,164
60549 FF84840C            DEFB 255,132,132,12
60553 00000092            DEFB 0,0,0,146
60557 FE000000            DEFB 254,0,0,0
60561 21301018            DEFB 33,48,16,24
60565 0E030000            DEFB 14,3,0,0
60569 D77C1018            DEFB 215,124,16,24
60573 1093FE10            DEFB 16,147,254,16
60577 08181030            DEFB 8,24,16,48
60581 E0800000            DEFB 224,128,0,0
60585 10181010            DEFB 16,24,16,16
60589 18101018            DEFB 24,16,16,24
```

We made provision in the print processor part of the interrupt handler for an "OR printing" function that merges a new character with the current contents of a screen cell in the display file but gave no support for it. Here is that routine.

The OR-printing Routines

Whenever we come to print a character on the screen, we will

need to know whether the contents of the destination cell are to be preserved by OR-printing, or destroyed by over-printing. For example, when two character in a game move into the same cell we will probably want to merge them together, while if we are moving a character from one cell to the next, trailing a blank behind it to automatically delete the old image, then we certainly won't want to OR-print the space with the old image.

For these purposes we need a map in memory called an OR-map to keep track of which cells are "occupied". Only one bit per cell is required. A "1" will indicate "cell occupied" and a "0" will indicate "cell empty". At 768 cells/screen, we can do the whole thing in 96 bytes.

We will need two routines. A clear the OR-map and a check of the OR-map.

```
60593           ORMAP     DEFS 96: THE MAP ITSELF

60689 21B1EC    CLOR      LD HL,ORMAP:CLEAR ORMAP
60692 015F00              LD BC, 95
60695 70                  LD (HL),B :B=0
60696 54                  LD D,H
60697 5D                  LD E,L
60698 13                  INC DE
60699 EDB0                LDIR
60701 C9                  RET
```

Everytime we are preparing to send a character to the print buffer and wish it to be considered for OR-printing with existing and future characters, we should access the bit corresponding to the destination cell in the OR-map.

If that bit is set, then we know there is already something in that cell, and we select OR-printing by setting bit 7 of the attribute byte, ATT. If the bit is zero then as far as we are concerned the cell is empty. We then set the proper bit of the OR-map to signal that the cell is now full and reset bit 7 of ATT to select OVER-printing. The character is then sent to the print buffer using HIPRINT in the usual manner.

The following routine, ORCHK, carries out these steps using the pointer ATCC, which holds the location of the current attribute byte, as a means of locating the correct OR-map bit.

```
60702 2A4BED    ORCHK     LD HL,(ATCC)
60705 7D                  LD A,L :DIVIDE 10 LOWEST BITS OF ATTR
:ADDR BY 8
60706 CB1C                RR H
60708 1F                  RRA
60709 CB1C                RR H
60711 1F                  RRA
60712 CB3F                SRL A
60714 5F                  LD E, A: STORE IN DE
```

```
60715 1600                LD D,0
60717 7D                  LD A, L
60718 21B1EC              LD HL,ORMAP
60721 19                  ADD HL, DE: ADD OFFSET
60722 E607                AND 7: ROTATE A MASK UNTIL "1" IS  OVER
:THE REQUIRED BIT
60724 47                  LD B,A
60725 04                  INC B
60726 3E01                LD A, 1
60728 0F        NXTROT    RRCA
60729 10FD                DJNZ NXTROT
60731 4F                  LD C, A: STORE MASK IN C
60732 A6                  AND (HL) :CELL OCCUPIED
60733 1192ED              LD DE, ATT
60736 1A                  LD A,(DE)
60737 CBBF                RES 7, A
60739 2802                JR Z, NOTOR: IF NOT THEN OVER-PRINT
60741 CBFF                SET 7, A: ELSE OR-PRINT
60743 12        NOTOR     LD (DE),A
60744 79                  LD A,C :NOW SET BIT IN OR-MAP = USED
60745 B6                  OR (HL)
60746 77                  LD (HL),A
60747 C9                  RET
```

The variable ATCC called in this routine is also used by HIPRINT which follows.

The Print to the Normal Buffer Routine

Here finally is the routine to set up the regular print buffer. We again use the attribute file address to keep track of where we are in the display file rather than the way Sinclair Basic operates. The variable ATCC set to 5800H sets our marker to the top left home position of the screen.

Remembering that CHSTRE hold the number of used entries in the print buffer and BUFFPT point at the next free entry we are ready to begin. Note that A must be loaded with character codes of either characters below code 128 or UDG character codes.

```
60748 0058      ATCC      DEFW #5800
60750 003C      CHARS     DEFW #3C00

ENTRY: A=CHARACTER CODE
EXIT: BC= ADDRESS OF CHARACTER DATA
      DE= ATCC (AS DEFINED ABOVE)
      HL= NEXT BUFFER ENTRY
      A = HI BYTE OF DFILE ADDRESS
60752 6F        HIPRINT   LD L, A: MULTIPLY CODE BY 8
60753 2600                LD H, 0
60755 29                  ADD HL, HL
60756 29                  ADD HL, HL
```

```
60757 29                    ADD HL, HL
60758 ED5B4EED              LD DE,(CHARS)
60762 19                     ADD  HL,DE:  ADD  OFFSET-NOW  AT  CHAR
:PIXEL ADDR
60763 44                    LD B, H: STORE IN BC
60764 4D                    LD C, L
60765 216EE8    PLACE       LD HL,CHSTRE
60768 7E        PWAIT       LD A,(HL)
60769 FE18                  CP 24 :MUST  BE  POKED  WITH  DIFFERENT
:BUFFER LENGTH IF CHANGED
60771 DA69ED                JP C, GO
60774 FB                    EI
60775 18F7                  JR PWAIT
60777 F3        GO          DI
60778 34                    INC (HL) :UPDATE BUFFER CHAR COUNT
60779 ED5B4CED              LD DE,(ATCC)
60783 2A92ED                LD HL,(ATT) :H=MASK, L=ATT
60786 1A                    LD A,(DE) :MAKE NEW ATTR
60787 AD                    XOR L
60788 A4                    AND H
60789 AD                    XOR L
60790 2A6CE8                LD HL,(BUFFPT) :HL= FIRST  FREE  BUFFER
:LOCATION, ROTATE AND STORE ATTR
60793 07                    RLCA
60794 77                    LD (HL),A
60795 2C                    INC L :STORE ATTR ADDR
60796 73                    LD (HL), E
60797 23                    INC HL
60798 72                    LD (HL), D
60799 2C                    INC L
60800 7A                    LD A, D :FIND AND STORE HI BYTE OF
:DFILE ADDR
60801 E603                  AND 3
60803 07                    RLCA
60804 07                    RLCA
60805 07                    RLCA
60806 F640                  OR 64 :ADD OFFSET
60808 77                    LD (HL), A
60809 23                    INC HL
60810 71                    LD (HL), C :STORE CHAR DATA ADDR
60811 2C                    INC L
60812 70                    LD (HL), B
60813 23                    INC HL :SET BUFFER TO NEXT FREE ENTRY
60814 226CE8                LD (BUFFPT),HL
60817 C9                    RET
```

And the last 2 bytes we need:

```
60818 38        ATT         DEFB 56
60819 00        MASK        DEFB 0
```

It's time to assemble the code again and save it. We can use the same title and starting address as previously. The length now

becomes 1319. We also should save the source code and we may as well use the old name for that as well. Use the Zeus T command to find out how much longer the source program has grown since last time.

This also means that we are finally ready for our second demo. We, of course, could test the printing of the target by itself but it is a lot more impressive to do both the border movement and the target together. We do have to change our Basic Program MASTDEMO a bit.

All we have to do is change line 70 to read:

```
70 GOSUB 200: RANDOMIZE USR 59416: RETURN
```

We have already put the call at 59416 with the subroutine at line 200. Save your new copy of "MASTDEMO" and run the program using option 3 on the menu. Note how a different line of keys has to be used for "up" this time.

We are finally getting somewhere, aren't we. All we really need are some sprite targets to shoot at.

PLOTTING and DRAWING

There are times when the fastest way to put a point on the screen is with PLOT or a fast machine code equivalent of it. Straight lines also are easy with DRAW. With INVERSE or OVER active, it can be a new point or line or erasing the old one. These routines are faster than the Basic ones mainly becaose a lot of the error checking of the basic routines has been removed. We do need a routine to draw to all 24 lines of the screen.

The PLOT Routine

The PLOT routine calculates the attribute address from the DFILE address. This is easier and faster than going back to the original coordinates each time. We also use our own PFLAG so it won't interfere with the Basic System Variable at 23697. OVER 1 is bit 1 and INVERSE 1 is bit 3 which is not the same two bits as in Basic where it is 2 and 0 respectively.

We also employ a coordinate system with (0,0) at the top left corner of the screen rather than the bottom left. Actually the Basic routine converts X and Y to top right coordinates before running so why not start with that designation in the first place. The bottom right corner becomes (255,191) as we go to all 24 lines of the screen rather than only the top 22 as in Basic.

PLOT also uses the two data bytes ATT and MASK which we ended the last routine with.

```
ENTRY: H=X  L=Y (TOP LEFT CORNER IS (0,0)
EXIT: DE=ADDRESS OF PIXEL IN DISPLAY FILE
      A= (DE), C= (PFLAG)  BIT 1 (PFLALG)= OVER, BIT 3= INVERSE

60820 00        PFLAG     DEFB 0
60821 7D        PLOT      LD A, L: FIND ADDRESS OF REGUIRED BYTE
:IN DFILE
60822 E6C0                AND 192: KEEP WITHIN RANGE
60824 1F                  RRA
60825 37                  SCF
60826 1F                  RRA
60827 0F                  RRCA
60828 AD                  XOR L
60829 E6F8                AND 248
60831 AD                  XOR L
60832 57                  LD D, A
60833 7C                  LD A, H
60834 07                  RLCA
60835 07                  RLCA
60836 07                  RLCA
60837 AD                  XOR L
60838 E6C7                AND 199
60840 AD                  XOR L
60841 07                  RLCA
60842 07                  RLCA
60843 5F                  LD E, A
60844 D5                  PUSH DE :ADDRESS STORED IN DE
60845 7A                  LD A, D: FIND ATTR ADDR
60846 0F                  RRCA
60847 0F                  RRCA
60848 0F                  RRCA
60849 E603                AND 3
60851 F658                OR 88: ADD OFFSET
60853 57                  LD D, A
60854 ED4B92ED            LD BC, (ATT)
60858 1A                  LD A,(DE): CHANGE ATTR
60859 A9                  XOR C
60860 A0                  AND B
60861 A9                  XOR C
60862 12                  LD (DE), A
60863 D1                  POP DE :RETRIEVE DFILE ADDR
60864 7C                  LD A, H
60865 E607                AND 7
60867 47                  LD B, A
60868 04                  INC B :B HOLDS (TARGET BIT #)+1
60869 3EFE                LD A, FE
60871 0F        PLOOP     RRCA :ROTATE A WINDOW TO TARGET BIT
:MISSING BIT IS THE WINDOW
60872 10FD                DJNZ PLOOP
60874 47                  LD B, A
60875 3A94ED              LD A, (PFLAG) :PUT (PFLAG) IN C
```

```
60878 4F                LD C, A
60879 1A                LD A, (DE)
60880 CB49              BIT 1, C: OVER?
60882 2001              JR NZ, OVER1
60884 A0                AND B
60885 CB59      OVER1   BIT 3, C :INVERSE?
60887 2002              JR NZ, INV1
60889 A8                XOR B
60890 2F                CPL :INVERT WINDOW GIVING BIT
60891 12        INV1    LD (DE), A
60892 C9                RET
```

The final part of the routine that does the actual plotting merits some explanation if you already are not familiar with OVER and INVERSE printing. Ignoring what happens to the other 7 bits in our byte, since they are always unchanged in the end, let us just examine the behavior of the "target" bit.

```
AND B
```

makes the target zero if OVER 0 is selected. OVER 1 cause the instruction to be skipped.

```
BIT 3, C
JR NZ, INV1
```

checks for inverse and jumps to the end if INVERSE 1, leaving the bit as it was if OVER 1 was selected, or zero (PAPER) if OVER 0 was selected.

Finally, having "narrowed down" our selections to INVERSE 0,

```
XOR B
CPL
```

leaves the bit complemented in the case of OVER 1, or set in the case of OVER 0. The byte is then replaced in the DFILE.

The DRAW Routine

Lest ye be dismayed, shouldn't every good book on screen printing contain an explanation of the draw routine? This one follows the same algorithm of the ROM of the 2068 but maximizes the speed of the routine. It gets quite involved.

We will be using the same coordinates as the PLOT routine just given by setting the upper left corner to (0,0). We also assume we are drawing from (X1, Y1) to (X2, Y2) both inclusive. We start by plotting (X1, Y1).

Next we have to decide in what direction to draw the line. IF

(X2 - X1) is positive, we will be drawing to the right. If it is negative, we draw to the left. Similarly, if (Y2 - Y1) is positive, we will be drawing down, otherwise it's up.

The routine loads the D and E registers with the unit changes in X and Y respectively as related to the direction along each axis, i.e., D = 1 means right while E = -1, actually 255, means up.

The rough part comes in deciding exactly how a line can be formed by making incremental unit jumps either horizontally, vertically or diagonally between points on a grid. Straight horizontally or straight vertically are easy and even a perfect 45 degree diagonal isn't bad. Its the other lines between these simple cases that have to consist of combinations of straight and diagonal mixtures in the correct proportion.

We define B as ABS (X2 - X1) and C as ABS (Y2 - Y1). If B is greater than C, then we will need a mixture of HORIZONTAL and diagonal steps. If B is less then C we need a mixture of VERTICAL and diagonal steps. Obviously, if B = C we only need diagonal steps.

The routine decides whether we need vertical or horizontal steps, and stores the required value of DE, as explained earilier, in the variable VHSTP. The direction of the diagonal steps is stored in DIASTP.

We can readilly determine how many steps of horizontal or vertical and diagonal we need. If B - C is positive we need Horizontal steps equal to B - C along with C diagonal steps. If, on the other hand, B - C is negative we need ABS (B - C) vertical steps with B diagonal steps. However, we always select B to handle the greater number of steps be they vertical, horizontal or diagional and C to handle the other, whichever it is. We now only have to ensure that the straight and diagonal steps are evenly distributed. The following procedure is used.

B is copied to H, and then halved and copied to L. Now, entering the loop, C is added to L, and if the result equals or exceeds B, then it is reduced by B and a diagonal step is taken. Otherwise, a straight step is taken. A point is plotted, the counter in H is decremented and the loop is repeated until the line is complete.

You can also think of the loop as continually adding C to itself and taking a diagonal step until a new multiple of B is passed. The reason L is initialized to B/2 is simply to assure that the line is straight at the beginning.

DRAW exits with coordinates of the end point of the line in HL. Since HL also holds the coordinates of the first point of a line on entry to the routine, we can therefore use HL unchanged to continue drawing the next line starting from this same end

point.

```
ENTRY: H = X1, L = Y1, B = X2, C = Y2
  DRAWS FROM (H, L) TO (B, C) INCLUSIVE
EXIT: DE = ADDR OF LAST PIXEL PLOTTED.
   NOTE: B IS THE GREATER, AND C IS THE LESSER OF:
            ABS (X2 - X1) AND ABS (Y2 - Y1)

60893 0100      DIASTP    DEFW 1
60895 0100      VHSTP     DEFW 1
60897 C5        DRAW      PUSH BC :SAVE WHILE WE CALL PLOT
60898 CD95ED              CALL PLOT :PLOT FIRST POINT
60901 C1                  POP BC :RETRIEVE END POINT
60902 110101              LD DE, #0101 :INITIALIZE DE WITH  RIGHT
:AND DOWN
60905 78                  LD A, B
60906 94                  SUB H :LEFT = -1, RIGHT = +1
60907 3004                JR NC, X2X1 :RIGHT
60909 15                  DEC D :MAKE D =255 FOR LEFT
60910 15                  DEC D
60911 ED44                NEG
60913 47        X2X1      LD B, A :X2 - X1
60914 79                  LD A, C
60915 95                  SUB L : UP OR DOWN?
60916 3004                JR NC, Y2Y1
60918 1D                  DEC E :MAKE E = 255 FOR UP
60919 1D                  DEC E
60920 ED44                NEG
60922 4F        Y2Y1      LD C, A :Y2 - Y1
60923 B0                  OR B :CHECK THAT LINE ISN'T A POINT
:B + C = 0 IS A POINT = EQUALS NO CHANGE FROM HL
60924 C8                  RET Z : DONE IF POINT
60925 79                  LD A, C
60926 B8                  CP B: WHICH IS BIGGER B OR C?
60927 E5                  PUSH HL
60928 62                  LD H, D :STORE DIRECTION OF DIAGONAL
60929 6B                  LD L, E
60930 22DDED              LD (DIASTP),HL
60933 2E00                LD L, 0 :DECIDE BETWEEN  VERT  &  HORIZ
:STEPS DEPENDING ON WHICH IS BIGGER C OR B
60935 3804                JR C, BBC
60937 65                  LD H, L :SWICH D AND E
60938 6B                  LD L, E
60939 48                  LD C, B :ALSO SWITCH B AND C
60940 47                  LD B, A
60941 22DFED    BBC       LD (VHSTP),HL :STORE V/H STEP
:B IS NOW >= C. THE ROUTINE TAKES B - C STR. AND C DIAG STEPS
60944 60                  LD H, B
60945 78                  LD A, B
60946 CB3F                SRL A :DIVIDE BY 2
60948 6F                  LD L, A
60949 7D        NXTSTP    LD A, L
60950 81                  ADD A, C
60951 3803                JR C, DIAG :DECIDE ON DIAG OR STR  STEP
```

```
: THIS TIME
60953 B8                    CP B
60954 3808                  JR C, VERHOR
60956 90          DIAG      SUB B
60957 6F                    LD L, A
60958 ED5BDDED              LD DE, (DIASTP)
60962 1805                  JR STEP
60964 6F          VERHOR    LD L, A
60965 ED5BDDED              LD DE, (VHSTP)
60969 E3          STEP      EX (SP), HL
60970 7C                    LD A, H :MAKE THE STEP ALONG X
60971 82                    ADD A, D
60972 67                    LD H, A
60973 7D                    LD A, L :MAKE THE STEP ALONG Y
60974 83                    ADD A, E
60975 6F                    LD L, A
60976 C5                    PUSH BC :THE ACTUAL PLOT
60977 CD95ED                CALL PLOT
60980 C1                    POP BC
60981 E3                    EX (SP), HL :GET COUNTER BACK
60982 25                    DEC H
60983 20DC                  JR NZ, NXTSTP
60985 E1                    POP HL
60986 C9                    RET
```

How about a demo of some interference graphics plotting using our routines? This particular demonstration does not need the IM 2 Interrupt Handler that we needed for the horizon generation. Nothing really fancy here and really nothing you all haven't seen before except that we have an explanation of the code that produces it. We also need a CLS routine. I call this routine,

The MOIRE Program

because although the whole thing is plotted in blue and yellow we get another color on the screen because of the rapidly changing of Paper and Ink Pixels in the pattern.

```
60987 3E0E      MOIRE     LD A,14: BLUE PAPER, YELLOW INK
60989 CD83EE              CALL CLS
60992 3E02                LD A, 2: TURN OVER ON
60994 3294ED              LD (PFLAG), A
60997 3606                LD A, 6: BORDER TO YELLOW
60999 D3FE                OUT (254), A
61001 210E00              LD HL, 14 :SET ATTR =14 AND MASK = 0
61004 2292ED              LD (ATT), HL
61007 2100FF              LD HL, #FF00 :X1 = 255, Y1 = 0, X2 =
:0, Y2 = 0 AS BC = 0 EXITING CLS. THIS THEN DRAWS A LINE FROM
:THE TOP RIGHT CORNER TO THE TOP LEFT CORNER.
61010 CDE1ED              CALL DRAW
61013 2C                  INC L : HL= 0, X1 = 0, Y1 = 0
```

```
61014 01BF00              LD BC, #00BF :X2 = 0, Y2 = 192, SO DO
:LEFT BORDER
61017 CDE1ED              CALL DRAW
61020 24                  INC H: X1 = 1. WE NOW CREATE PATTERN IN
:THE REMAINING PIXELS
61021 E5        NXTDR1    PUSH HL
61022 016080              LD BC, #8060 :X2 = 128, Y2 = 96. DRAWS
:LINE FROM LOWER LEFT TO CENTER OF SCREEN
61025 CDE1ED              CALL DRAW
61028 C1                  POP BC :RESTORE BC WITH 0,192
61029 C5                  PUSH BC
61030 06FF                LD B, 255 :X2 = 255, SO DRAW FROM
:CENTER TO LOWER RIGHT CORNER
61032 CDE1ED              CALL DRAW
61035 E1                  POP HL
61036 2D                  DEC L: MOVE ONE SPACE UP SCREEN
61037 20EE                JR NZ, NXTDR1 :CONTINUE IN LOOP UNTIL
:L = 0 WHICH MEANS AT THIS POINT THE LEFT AND RIGHT SIDES OF
:THE SCREEN ARE FULL.
61039 2C                  INC L: L = 1 TO STAY OFF THE BORDER
61040 E5        NXTDR2    PUSH HL
61041 016080              LD BC, #8060 :RESET BC TO CENTER. DRAW
:FROM TOP EDGE TO CENTER
61044 CDE1ED              CALL DRAW
61047 C1                  POP BC
61048 C5                  PUSH BC
61049 0EBF                LD C, 191 :RESET TO BOTTOM EDGE
61051 CDE1ED              CALL DRAW
61054 E1                  POP HL
61055 24                  INC H: MOVE A SPACE OVER RIGHTWARDS
61056 20EE                JR NZ, NXTDR2
61058 C9                  RET
```

In case you haven't followed the routine completely we have drawn every diagonal line possible across the screen going through the center point. However, because over is on, whenever a conflicting pixel of one line draws over a pixel already drawn by another line, the pixel is turned off. This happens quite a bit in the center of the screen where all the lines cross. Further out there are not as many conflicts but still a few. Everything that is blue indicates an even number of prints of that pixel while yellow indicates an odd (including the first) number of prints of that pixel. Things get confusing where the interference colors come in as the eye cannot detect those individual pixels. Obviously different color combinations can be tried by poking address 60988 with different values of paper and ink. This routine also points out how important it is to know which registers have to be loaded with what variable to run a routine but also what values are in the various registers upon exiting a routine as well even if they are nonsense.

Before assembling we need the CLS routine called by MOIRE.

```
ENTRY: A = ATTR
PRESERVED A
EXIT: BC= 0, DE = #5B00, HL = #5AFF

61059 210040   CLS      LD HL, #4000
61062 010018            LD BC, #1800
61065 75                LD (HL), L :L= 0
61066 54                LD D, H
61067 1E01              LD E, 1
61069 EDB0              LDIR
61071 77                LD (HL), A
61072 01FF02            LD BC, #02FF
61075 EDB0              LDIR
61077 C9                RET
```

If speed is of the essence then I should warn you that the DRAW routine should only be used if "general" lines from non-specific points are required. It is almost always quicker to use a customized routine, perhaps employing a lookup table of plotting coordinates, if specific lines are being drawn.

For example, if you frequently need to draw a line right across row 0 of the screen, it is far quicker to load the first 32 bytes of the DFILE with 255 than it is to draw from (0,0) to (255,0).

We now assemble the source code and save both the source code and the object code as previously outlined. We load "MASTDEMO" and change the following line to read:

```
 50 GOSUB 200: RANDOMIZE USR 59422: PAUSE 0: RETURN
```

We use the PAUSE 0 to hold all 24 lines on the screen. As soon as you hit a key the 2 bottom lines of the pattern are destroyed by the program end notation. In our case its the reprinting of the menu.

As I said, you have seen it before but now at least you understand what was done to do it.

SPRITES

We are now ready to begin development of sprite generation, printing and control routines for the production of smoothly moving flickerless pixel graphics in conjunction with the Interrupt driven print processor.

We define a sprite to be some image contained in a movable object block of adjacent characters on the screen. This block will always be either square or rectangular. The image may be anything from 1X1 on up in size.

The most obvious approach to moving a sprite from one position to another is to "blank out" the "old" image by printing spaces over it and then printing the new image in the new position. If the two images do not overlap, this is a perfectly acceptable technique.

If, however, as is usually the case, we have only moved the sprite a few pixels, then there seems little point in printing a space in a cell common to both images, and then have to replace it with part of the new image instantly.

Since you have concluded by now, time is at a premium, we will use a more unique approach. Each shape will be surrounded by a narrow region of blank pixels, so that as we move from one position to the next, these trailing blanks will wipe out any part of the old image not already obliterated by the printing of the new image.

In this way we can achieve animation in just one print operation rather than the two required by the former technique. Additionally, the number of characters we need to print will be halved. This is a decided advantage when you bear in mind that our standard interrupt handler can only print 24 characters without having to compromise our horizon routine. We already used 13 of the 24 for our gunsight which leaves us only 9 for anything else.

To move sprites smoothly we have the capability of moving them one pixel at a time. This will be the standard step in what follows. We should also have the option of moving more than one pixel at a time or not moving every TV frame refresh.

If our shape is contained in a space of M character spaces wide (i.e., 8xM pixels wide) and we want to move a pixel at a time, then the most character spaces our sprite can occupy is M+1 character spaces.

If in addition, we specify that the shape can never occupy more than M+1 different character spaces wide in moving from the old to the new position, then we see that BOTH the old and new images can be contained in an area M+1 spaces wide.

For example, if a 1x1 block occupied pixel spaces 7 and 8 of space 14 and spaces 1 to 6 in space 15, we would be restricted in only moving it 2 pixel spaces right to totally occupy space 15 only in the new position. Position 14 would then be printed with a blank. By our restriction, we could not move 3 pixel spaces right this move. There would be other times when a 3 pixel right move would not be forbidden.

This restriction is, in practice, quite easy to apply, and leads to the result that we animate any shape contained in (m x n) characters by continually printing a set of sprite images of fixed dimensions (m+1) by (n+1). Thus to move a 1 character cell

requires a 2x2 cell, whereas a 2x2 sprite, requires a 3x3 cell.

If you can imagine our m x n shape floating around within its (m+1) by (n+1) sprite, we find that by virtue of 8 shifts up and 8 shifts right that we can have 64 different images of our shape.

If the shape is movable at one pixel at a time in each of the X and Y directions, then each of these 64 patterns would at some time need to be printed.

Before proceeding further, let's define two variables. We will call the horizontal distance (in pixels) of the shape from the left hand edge of its sprite XP, and the vertical distance of the shape from the bottom edge of the sprite YP.

We now have a direct choice over the method of generation of the sprite patterns. We may either store just one image in memory and manipulate the shape within it bit-wise to obtain the required image for printing, or we may store the images in a somewhat larger table in RAM, using an indexing technique to "select" the required image without further manipulation.

The first technique doesn't require much space but is very time consuming performing the XP horizontal shifts and YP vertical shifts on each of the (M+1)xNx8 bytes twice.

Although the sprite may be one of 64 possible patterns, the situation is not as serious as it seems. We do not need to store 64 different images in memory as it is possible to produce the eight images corresponding to the different values of YP from the one image corresponding to a given XP. Thus we only need to store 8 images of the sprite, one for each value of XP.

We only have to "load" one image of our sprite and then have the computer generate the other 7 images from this one. We store them in the following character space order for a 3 x 4 sprite:

1	4	7	10	XX
2	5	8	11	XX
3	6	9	12	XX

Each sprite will be stored as though YP = 0, that is, the sprite will be on the bottom edge of the block space with a BLANK top line. Then, starting at the upper left corner of the sprite we enter the 8 pixels of character space "1" from top to bottom and continue with character space "2" doing the whole left side of the block space before starting the second set of character spaces. For many sprites you may have a completely blank character space because the block MUST be retangular (or square). We call this initial block image "0".

We will be writing a routine to have the computer design the other 7 sets of character spaces from this original set. The

left edge of the 2nd set will have a line of blank pixels as each pixel moves left one pixel positon. The character spaces marked "XX" in the above diagram handle the overflow from character spaces 10, 11 and 12. These spaces are the +1 in that (m+1) in our formula.

To have the image move right we simply start with image 1 and cycle through the various images one at a time and since our sprite is going to print in (m+1) spaces wide, we merely have to print the cycle of images over each other in the same character spaces for the first 8 moves. By this time the originaly blank "XX" spaces will have moved to the left of the image. We then just move right a character space and repeat. Our sprite position will be defined by specifying the TOP LEFT character space of the block on the screen. These character space coordinates are called XC and YC respectively.

To move left we merely cycle through the cycle of images backwards.

The various images corresponding to each value of YP will be produced by pointing the sprite generator at th YPth row of the XPth image counting downward from the top row "0". By pointing at the 4th row we effectively move the sprite "up" 4 rows.

This is for single pixel movement. For it we need 8 images. If we opt for 2 pixel at a time movement, we can cut our memory requirement in half because then we only need 4 images as we can omit the 1st, 3rd, 5th and 7th image. If we go to 1/2 character space at a time movement, we only need 2 images, the 0 and 4th. This turns out to be guite jerky but also very fast.

With this explaination, we can now calculate the amount of space we need to store the images. Take an (m x n) character shape, enclosed in a (m+1) by (n+1) sprite space and producing "a" images, and bearing in mind that each cell requires 8 bytes and each set of images requires a preceding blank column, we have:

Memory needed = 8x(a)x(m+1)x(n+1)+8x(n+1)

Thus for a 3 by 2 shape moving 2 pixels at a time requires:

8x4x(3+1)x(2+1)+8x(2+1) = 408 bytes

In addition to this, and assuming that all the images for the sprites currently in use are stored consecutively in memory, we must include zero bytes after the last image of the last sprite, to allow for the memory "take up" when YP = 8 and the last image is being used. In this case, those 8 bytes will represent the bottom right corner blanking of the sprite.

To control and keep track of a sprite we will be using a Sprite Motion Table of 17 bytes for each sprite. This table will tell our sprite animation routine at what speed to move the sprite

along X and Y, the whereabouts of the sprite at any time, the location of the image data, the dimensions of the sprite, the color it is to be printed in, etc.

We will use the IY register (another no-no of beginning machine code violated) to point at the beginning of a Sprite Motion Table whenever we want to move a sprite. Before proceeding to a complete breakdown of this table, we REDEFINE XP to be the number of the image currently being used by the sprite generator. Thus if there are four images, XP will now cycle continuously through the values (0,1,2,3) as the sprite moves across the screen. When there are 8 images, XP will have the same value as before, that is the number of pixels from the shape to the left hand edge of the sprite. Otherwise, you will need to multiply XP by the step between images to find this distance. This conversion must be borne in mind when you are writing collision detection routines.

Here then is a list of a typical motion data table:

IY+0 XP = Current image number (<8)

IY+1 VX= rate of change of XP (+ or -)

IY+2 N = number of images = (max value of XP)+1

IY+3 XC = leftmost sprite column

IY+4 YP (0-7)

IY+5 VT = rate of change of YP (+ or -)

IY+6 YC = Position of uppermost sprite line

IY+7 LO address
of row 0, cell 0 of image 0
IY+8 HI address

IY+9 Cycle count= number of repeats of movement

IY+10 Cycle period= wait this many frames before making next move.

IY+11 Width of expanded sprite

IY+12 Length of expanded sprite

IY+13 LO
length of one image = width x depth x 8
IY+14 HI

IY+15 Attribute byte and flag for OR printing

IY+16 Attribute mask

The "cycle count" will be decremented every time the routine is used. If not zero the sprite makes its trip again.

The "cycle period" tells the sprite move routine how many TV frames to wait before moving the sprite another increment. This together with the motion per move controls your speed across the screen.

As I said, we don't want to figure out all these positions of our sprite and type them in. We use two routines to expand each sprite. The first is called PADOUT and will copy the dormant shape from its storage area to the "sprite image area", adding a column of blanks to the right, a line of blanks above it, and the statutory preceding blank column to image "0". The second routine, SPREX, will manipulate a copy of image 0, row by row, using shifting and rotating operations to generate the other images.

Because we are allowing our sprite to move at various rates of pixels per printing to the screen we must allow for enough "margin" so we get image blanking in both horizontal and vertical directions. Without going into a lot of detail, suffice it to be stated that the number of blanking pixels in the horizontal direction is given by:

$$Ph = D(VX + XP) - 8$$

and that in the vertical direction by:

$$Py = VY + YP - 8$$

where XP and YP are the maximun rate we will use (0 to 7).

For example, suppose we wish to design, within a 4 x 3 sprite and hence a 3 x 2 shape, a fighter plane capable of moving at up to four pixels per movement in the X direction and up to 3 pixels in the Y direction with a maximum rate of XP = 7 and YP = 7. Our formulas demand 3 pixels must be left on the right margin of our 24 pixel wide sprite and 2 rows must be left blank on the top of our 24 high pixel row sprite.

Padout Routine

Padout uses several "tables" of data which we will only define later in a demo program. Just to list them, one will be the actual image table of pixel design, the 2nd will be the 17 byte Motion Table, and the 3rd will be the Expanded Image Space for the various images.

For those of you doing your own assembly, reload the previous source code and continue at the end. This code just fits the

remaining space.

OBJECT: TO "PADOUT" BARE SPRITE DATA AND PRODUCE ENTRIES IN THE SPRITE MOTION TABLE.

ENTRY: DE = SPRITE DATA ADDRESS
HL = IMAGE STORAGE AREA
B = COLUMN WIDTH OF STANDARD SPRITE
C = LINE DEPTH OF STANDARD SPRITE
IY = SPRITE MOTION DATA ADDR
EXIT: HL = ADDRESS OF IMAGE 0
B = COLUMN WIDTH OF EXPANDED SPRITE
C = ROW DEPTH OF EXPANDED SPRITE
NOTE: HL & BC WILL BE USED BY SPREX
AF' DESTROYED

```
61078 0C        PADOUT    INC C: CALC # OF ROWS OF EXPANDED
:SPRITE AND STORE IT
61079 FD710C              LD (IY+12),C
61082 CB21                SLA C
61084 CB21                SLA C
61086 CB21                SLA C :X8
61088 78                  LD A, B :STORE BC FOR SPREX
61089 04                  INC B
61090 FD700B               LD  (IY+11),B  :LOAD  EXPANDED  SPRITE
:WIDTH
61093 C5                  PUSH BC
61094 08                  EX AF, AF' :SAVE COLUMNS COUNTER
61095 D5                  PUSH DE :SAVE DATA ADDR
61096 0600                LD B, 0 :START WITH COLUMN OF BLANKS
61098 C5                  PUSH BC
61099 70                  LD (HL), B
61100 0D                  DEC C
61101 CDEEEE              CALL CL :CLEARING SUBROUTINE
61104 C1                  POP BC
61105 79                  LD A, C :UPDATE C
61106 D608                SUB 8
61108 4F                  LC C, A :BC = # OF ROWS IN STD COLUMN
61109 E1                  POP HL :DATA ADDRESS
61110 D5                  PUSH DE
61111 C5                  PUSH BC
61112 08                  EX AF, AF' :GET COUNTER BACK
61113 E5        NXSCOL    PUSH HL :INSERT A SPACE ON TOP LINE
61114 EB                  ES DE, HL
61115 3600                LD (HL), 0
61117 0E07                LD C, 7 :COUNT FOR REST OF SPACE
61119 CDEEEE              CALL CL
61122 E1                  POP HL
61123 C1                  POP BC :FILL REST OF COLUMN WITH
:SPRITE DATA
61124 C5                  PUSH BC :SAVE COUNTER FIRST
61125 EDB0                LDIR
61127 3D                  DEC A
61128 20EF                JR NZ, NXSCOL
```

```
61130 C1                POP BC :EXPAND WITH A  RIGHTMOST  BLANK
:COLUMN
61131 79                LD A, C
61132 C607              ADD A, 7
61134 4F                LD C, A
61135 EB                EX DE, HL
61136 70                LD (HL), B
61137 CDEEEE            CALL CL
61140 E1                POP HL :GET BACK ADDR OF IMAGE 0
61141 D1                POP DE :AND VALUE IN DE, FOR SPREX
61142 D5                PUSH DE
61143 E5                PUSH HL
61144 60                LD H, B :CALCULATE # OF  BYTES  IN  ONE
:IMAGE & STORE IT IN SPRITE MOTION DATA
61145 68                LD L, B
61146 42                LD B, D
61147 54                LD D, H
61148 19        MUL1    ADD HL, DE
61149 10FD              DJNZ MUL1
61151 FD750D            LD (IY+13),L
61154 FD740E            LD (IY+14),H
61157 E1                POP HL
61158 C1                POP BC
61159 FD7507            LD (IY+7),L :PUT IMAGE  0  LOCATION  IN
:SPRITE MOTION DATA
61162 FD7408            LD (IY+8),H
61165 C9                RET

61166 54        CL      LD D, H :CLEARING ROUTINE OF PADOUT
61167 5D                LD E, L
61168 13                INC DE
61169 EDB0              LDIR
61171 C9                RET
```

As you can see, we have already loaded 6 of our 17 sprite motion values into the table.

The Sprite Expand Routine, SPREX

Now that we have taken our bare, unexpanded sprite data from memory and created image 0 from it, we need to generate the other sprites. As explained, each successive image is formed by shifting the rows of the previous one by one or more bits to the right. SPREX does this by taking each row of image 0 in turn, copying it into an area of "workspace" and then repeatedly shifting it and copying it into the appropriate position for each of the other images. We will label the start of the workspace as WKSPC. Since we only need to place one row of a sprite in it at a time, twenty bytes should be ample--that is enough to do a sprite 19 columns wide!

SPREX also needs IY pointing at the start of the motion data for the sprite being worked on. This pointer and the entry values of HL and BC are already set up for you by calling PADOUT, so the only parameter you have to set after calling PADOUT is the step (in pixels) between images stored in D.

However, beware. Nearly all of the alternate register set is used by the routine so if you are intending to come back to Basic after using SPREX, be sure to preserve HL' before entering this routine.

OBJECT: TO FORM THE "SHIFTED" IMAGES OF EXPANDED SPRITE DATA AS PRODUCED BY "PADOUT"

ENTRY: HL = ADDRESS OF IMAGE 0
D = STEP BETWEEN IMAGES
B = WIDTH OF EXPANDED SPRITE
C = DEPTH OF EXPANDED SPRITE

PRESERVED: BC
DESTROYED: B',C',D',E',H',L'

EXIT: DE' = ENTRY VALUE OF DE, BC' = 0, L = 0

```
61172 0000000000 WKSPC   DEFS 20 :WORKSPACE NEEDED
61178 0000000000
61182 0000000000
61185 0000000000
61192 3E08       SPREX   LD A, 8
61194 1EFF               LD E, 255
61196 92         SUBDIV  SUB D
61197 1C                 INC E
61198 30FC               JR NC, SUBDIV
61200 FD7302             LD (IY+2), E
61203 1D                 DEC E
61204 D5                 PUSH DE
61205 C5                 PUSH BC
61206 0600               LD B, 0 :BC = LEN OF 1 COLUMN IN BYTES
61208 11F4EE             LD DE, WKSPC
61211 D9                 EXX
61212 E1                 POP HL :H' = WIDTH, L' = # OF ROWS
61213 D1                 POP DE :D' = IMAGE STEP, E' = # OF
:IMAGES - 1 GENERATE ONE ROW OF EACH IMAGE
61214 D5         NXROW9  PUSH DE
61215 44                 LD B, H
61216 D9                 EXX : BACK TO NORMAL REGS
61217 E5                 PUSH HL :STORE ADDR OF ROW 0 OF IMAGE
0
61218 11F4EE             LD DE, WKSPC
61221 D9                 EXX
61222 D9         NXBYT3  EXX :BUILD THAT ROW OF SPRITE IN WORK
:SPACE
61223 7E                 LD A, (HL)
```

```
61224 09                ADD HL, BC
61225 12                LD (DE), A
61226 13                INC DE
61227 D9                EXX
61228 10F8              DJNZ NXBYT3
61230 D9        NXPOS   EXX
61231 EB                EX DE, HL
61232 D9                EXX
61233 D5                PUSH DE :SHIFT ROW BY D' PIXELS
61234 4A                LD C, D
61235 7C        NXSFT   LD A, H :ONE PIXEL AT A TIME
61236 D9                EXX
61237 21F4EE            LD HL, WRSPC
61240 A7                AND A :CLEAR CARRY FOR ROTATION
61241 CB1E      NXBYT   RR (HL)
61243 2C                INC L
61244 3D                DEC A
61245 20FA              JR NZ, NXBYT
61247 D9                EXX :NEXT SHIFT
61248 0D                DEC C
61249 20F0              JR NZ, NXSHF
61251 D9                EXX :GET ADDR OF NEXT IMAGE ROW TO HL
61252 EB                EX DE, HL
61253 11F4EE            LD DE, WKSPC
61256 D9                EXX
61257 44                LD B, H :XFER ROW OF H' COLUMNS TO
:IMAGE AREA
61258 D9        NXBYT2  EXX
61259 1A                LD A, (DE)
61260 77                LD (HL), A
61261 09                ADD HL, BC
61262 13                INC DE
61263 D9                EXX
61264 10F8              DJNZ NXBYT2 :LOOP BACK TO GENERATE THE
:SAME ROW OF THE OTHER IMAGES
61266 D1                POP DE
61267 1D                DEC E
61268 20D8              JR NZ, NXPOS
61270 D9                EXX
61271 E1                POP HL :FIND NEXT ROW OF IMAGE 0
61272 23                INC HL
61273 D9                EXX :RETRIEVE DE' AND REPEAT FOR NEXT
:ROW
61274 D1                POP DE
61275 2D                DEC L
61276 20C0              JR NZ, NXROW9
61278 D9                EXX :RETURN WITH CORRECT REGISTER SET
61279 C9                RET
```

The SPRINT Routine

We now have to have a routine to send our data to the interrupt

buffer. That is the function of the SPRINT routine. This particular routine was designed with speed and versatility in mind. This is essential if we want a horizon and the move of sprites every TV frame. Time should take priority over compactness of code at all times. Although SPRINT does the work, you will not normally call it directly, as it will be subsidiary to a more general routine which follows it called SPRMV (SPRite MoVe), which will perform various manipulations of the motion data before jumping to SPRINT.

The routine allows us to check for OR printing using the OR Map system described earlier. A slightly modified version of ORCHK has been built into SPRINT for speed but as usual will only be selected if Bit 7 of the sprite's attribute, stored in (IY+15), is set. Resetting the bit ignores the OR map and replaces whatever is in that character space with the new sprite image.

We also need the routine ATTLOC which we outlined earlier in this book.

As with SPREX, the prime registers are used again. SPRINT also assumes there is enough room in the interrupt buffer to hold all the characters and should thus not be called until an interrupt clears the buffer if there is not room. Sending the data to the buffer again uses single register increments and if you have expanded the interrupt buffer beyond the 42 character mark you again have to make corrections to these increments to get over the intervening page boundaries as was discussed earlier. Remember that CHSTRE holds the number of used entries in the buffer, and BUFFPT points at the next free entry. Both are adjusted accordingly the routine.

Since it is not desirable to be interrupted when only half a sprite has been sent to the buffer, the interrupts should be disabled while sending the characters. You cannot do this IF you are running the horizon generation.

SPRINT copes beautifuly with sprites that "spill off" the edges of the screen, or even those that are not even on the screen. Only that part of the sprite in the text area of the screen will print. However, the width of this "sprite window" can be changed by changing the values of the instructions labeled LFTLM1, LFTLM2, RGTLM1 and RGTLM2, obviously standing for LeFT LiMit and RiGhT LiMit respectively.

```
OBJECT: TO SEND SPRITE DATA TO THE INTERRUPT BUFFER.

ENTRY: B =  XP, C = YP, D = YC, E = XC
       HL = ADDR OF IMAGE 0
ALL AS SETUP BY SPRMV
EXIT: DE = 0
DESTROYS: A'F'B'C'D'E'H'L'

61280 D5  SPRINT          PUSH DE
```

```
61281 78                LD A, B :IF XP = 0 LEAVE HL AT IMAGE 0
61282 A7                AND A
61283 2809              JR Z, POS0
61285 FD5E0D            LD E,(IY+13) :FIND CORRECT IMAGE
61288 FD560E            LD D,(IY+14)
61291 19        NXA     ADD HL, DE
61292 10FD              DJNZ NXA
61294 09        POS0    ADD HL, BC :FIND CORRECT VERT POSITION
61295 C1                POP BC :FIND LOCATION OF TOP-LEFT  ATTR
:OF SPRITE
61296 E5                PUSH HL
61297 CD19F1            CALL ATTLOC
61300 FD5E0B            LD E,(IY+11) :E COUNTS THE COLUMNS LEFT
61303 D9                EXX :DECIDE ON OR OR OVER PRINTING
61304 FD6E0F            LD L, (IY+15)
61307 FD6610            LD H, (IY+16)
61310 CB7D              BIT 7, L
61312 CBBD              RES 7, L
61314 D9                EXX
61315 CA3AF0            JP Z, SPRTN0
61318 78                LD A, B :OR PRINT SO FIND ADDR IN ORMAP
61319 87                ADD A, A
61320 87                ADD A, A
61321 47                LD B, A
61322 79                LD A, C
61323 CB2F              SRA A
61325 CB2F              SRA A
61327 CB2F              SRA A
61329 80                ADD A, B
61330 D9                EXX :ADD BASE ADDR OF ORMAP
61331 EB                EX DE, HL
61332 4F                LD C, A
61333 0600              LD B, 0
61335 21B1EC            LD HL, ORMAP
61338 09                ADD HL, BC
61339 3A6EE8            LD A, (CHSTRE)
61342 47                LD B, A
61343 D9                EXX :HL' HOLDS  LOCATION  IN  ORMAP  SO
:ROTATE MASK OVER CORRECT CELL-BIT IN ORMAP
61344 79                LD A, C
61345 E607              AND 7
61347 47                LD B, A
61348 3E80              LD A, 128
61350 2803              JR Z, NROT1
61352 0F        NXTRT   RRCA
61353 10FD              DJNZ NXTRT
61355 D9        NROT1   EXX :STORE MASK IN C'
61356 4F                LD C, A
61357 D9                EXX
61358 79                LD A, C
61359 C1                POP BC :POINT BC AT IMAGE DATA
61360 E5        NXTX1   PUSH HL
61361 D9                EXX :STORE ORMAP ADDR
61362 E5                PUSH HL
```

```
61363 D9                    EXX
61364 FD560C                LD D, (IY+12) :LET D = DEPTH IN LINES
61367 FE20      RGTLM1      CP 32 :IF PRINT POSN IS OUT OF  X-RANGE
:THEN SKIP COLUMN
61369 3071                  JR NC, HOPCL1
61371 FE00      LFTLM1      CP 0 :SAME FOR TOO FAR LEFT
61373 386D                  JR C, HOPCL1
61375 08                    EX AF, AF' :STORE COLUMN POSN IN A'
61376 7C        NXTY1       LD A, H :IF PRINT POSN BELOW TEXT  THEN
:END
61377 FE5B                  CP 91
61379 3068                  JR NC, OUT81
61381 FE58                  CP 88 :IF PRINT POSN  ABOVE  TEXT  AREA
:THEN SKIP THIS LINE OF SPRITE
61383 D5                    PUSH DE
61384 EB                    EX DE, HL
61385 382F                  JR C, NPR1
61387 D9                    EXX :DECIDE WHETHER "OR" PRINT ON  THIS
:CELL IS NEEDED
61388 79                    LD A, C
61389 A6                    AND (HL)
61390 CBBB                  RES 7, E
61392 2802                  JR Z, NOTOR3
61394 CBFB                  SET 7, E :IF CELL OCCUPIED SET FLAG  TO
:OR PRINT
61396 79        NOTOR3      LD A, C
61397 B6                    OR (HL)
61398 77                    LD (HL), A
61399 D9                    EXX
61400 1A                    LD A, (HL) :SEND CHAR TO BUFFER
61401 D9                    EXX
61402 AB                    XOR E
61403 A2                    AND D
61404 AB                    XOR E
61405 04                    INC B
61406 D9                    EXX
61407 2A6CE8                LD HL, (BUFFPT)
61410 07                    RLCA
61411 77                    LD (HL), A
61412 2C                    INC L
61413 73                    LD (HL), E
61414 2C                    INC L
61415 72                    LD (HL), D
61416 2C                    INC L
61417 7A                    LD A, D
61418 E603                  AND 3
61420 07                    RLCA
61421 07                    RLCA
61422 07                    RLCA
61423 F640                  OR 64 :ADD OFFSET
61425 77                    LD (HL), A
61426 2C                    INC L
61427 71                    LD (HL), C
61428 2C                    INC L
```

```
61429 70                LD (HL), B
61430 2C                INC L
61431 226CE8            LD (BUFFPT), HL
61434 210800    NPR1    LD HL, 8 :INC DATA POINTER TO NEXT CELL
61437 09                ADD HL, BC
61438 44                LD B, H
61439 4D                LD C, L
61440 EB                EX DE, HL
61441 D1                POP DE
61442 15                DEC D :IF LINE COUNT 0 THEN NEXT COLUMN
61443 2810              JR Z, IN16
61445 D9                EXX :ELSE MOVE ORMAP POINT TO NEXT LINE
61446 7D                LD A, L
61447 C604              ADD A, 4
61449 6F                LD L, A
61450 D9                EXX :AND MOVE ATTR POINT TO NEXT LINE
61451 7D                LD A, L
61452 C620              ADD A, 32
61454 6F                LD L, A
61455 30AF              JR NC, NXTY1
61457 24                INC H
61458 C3C0EF            JP NXTY1 :LOOP BACK FOR NEXT LINE OF
:SPRITE
61461 08        IN16    EX AF, AF' :INCREASE COLUMN POSN
61462 3C                INC A
61463 D9                EXX :GET ORMAP ADDR AND MOVE MASK TO
:INC POINTER IF NECESSARY
61464 E1                POP HL
61465 CB09              RRC C
61467 3001              JR NC, NINC2
61469 2C                INC L
61470 D9        NINC2   EXX
61471 E1                POP HL :POINT HL AT 1ST ATTR OF NEXT
:COLUMN
61472 2C                INC L
61473 1D                DEC E :LOOP BACK FOR NEXT COLUMN
61474 C2B0EF            JP NZ, NXTX1
61477 D9                EXX :SET NEW VALUE OF CHSTRE
61478 78                LD A, B
61479 326EE8            LD (CHSTRE), A
61482 D9                EXX
61483 C9                RET
:JUMPS TO HERE TO OMIT ALL OR PART OF A COLUMN
61484 08        HOPCL1  EX AF, AF'
61485 60        OUT81   LD H, B :MOVE IMAGE POINTER TO NEXT
:SPRITE POSN
61486 69                LD L, C
61487 010800            LD BC, 8
61490 09        NXT81   ADD HL, BC
61491 15                DEC D
61492 20FC              JR NZ, NXT81
61494 44                LD B, H
61495 4D                LD C, L
61496 18DB              JR IN16
```

```
61498 79        SPRTNO    LD A, C :THE SHORTER  AND  FASTER  OVER
:PRINTING ROUTINE
61499 C1                  POP BC
61500 D9                  EXX : HOLD CHSTRE IN E
61501 ED5B6EE8            LD DE, (CHSTRE)
61505 D9                  EXX
61506 E5        NXTX2     PUSH HL
61507 FD560C              LD D,(IY+12) :LET D = DEPTH IN LINES
61510 FE20      RGTLM2    CP  32 :IF PRINT POSN  OUT  OF  X-RANGE
:THEN SKIP COLUMN
61512 3056                JR NC, HOPCL2
61514 FE00      LFTLM2    CP 0 : SAME FOR LEFT SIDE
61516 3852                JR C, HOPCL2
61518 08                  EX AF, AF' :STORE IN A
61519 7C        NXTY2     LD A, H :IF PRINT POSN BELOW TEXT  AREA
:THEN END
61520 FE5B                CP 91
61522 304D                JR NC, OUT82
61524 FE58                CP 88 :IF PRINT POSN  ABOVE  TEXT  AREA
:THEN MISS THIS LINE
61526 D5                  PUSH DE
61527 EB                  EX DE, HL
61528 3822                JR C, NPR2
61530 1A                  LD A, (DE) :SEND CHARACTER TO BUFFER
61531 D9                  EXX
61532 AD                  XOR L
61533 A4                  AND H
61534 AD                  XOR L
61535 1C                  INC E
61536 D9                  EXX
61537 2A6CE8              LD HL, (BUFFPT)
61540 07                  RLCA
61541 77                  LD (HL),A
61542 2C                  INC L
61543 73                  LD (HL),E
61544 2C                  INC L
61545 72                  LD (HL), D
61546 2C                  INC L
61547 7A                  LD A, D
61548 E603                AND 3
61550 07                  RLCA
61551 07                  RLCA
61552 07                  RLCA
61553 F640                OR 64: ADD OFFSET
61555 77                  LD (HL), A
61556 2C                  INC L
61557 71                  LD (HL), C
61558 2C                  INC L
61559 70                  LD (HL), B
61560 2C                  INC L
61561 226CE8              LD (BUFFPT), HL
61564 210800              LD HL, 8: INC DATA POINTER TO NEXT
:CELL OF IMAGE
61567 09                  ADD HL, BC
```

```
61568 44                LD B, H
61569 4D                LD C, L
61570 EB                EX DE, HL
61571 D1                POP DE
61572 15                DEC D :IF LINE COUNT IS 0 THEN NEXT
:COLUMN
61573 280A              JR Z, IN17
61575 7D                LD A, L
61576 C620              ADD A, 32
61578 6F                LD L, A
61579 30C2              JR NC, NXTY2
61581 24                INC H
61582 C34FF0            JP NXTY2
61585 08        IN17    EX AF, AF' :INCREASE COLUMN POSN
61586 3C                INC A
61587 E1                POP HL
61588 2C                INC L
61589 1D                DEC E :LOOP BACK FOR NEXT COLUMN
61590 C242F0            JP NZ, NXTX2
61593 D9                EXX
61594 7B                LD A, E
61595 326EE8            LD (CHSTRE), A
61598 D9                EXX
61599 C9                RET
:JUMPS HERE TO OMIT ALL OR PART OF COLUMN
61600 08        HOPCL2  EX AF, AF'
61601 60        OUT82   LD H, B :MOVE IMAGE POINTER TO NEXT
:SPRITE COLUMN
61602 69                LD L, C
61603 010800            LD BC, 8
61606 09        NXT82   ADD HL, BC
61607 15                DEC D
61608 20FC              JR NZ, NXT82
61610 44                LD B, H
61611 4D                LD C, L
61612 18E3              JR IN17 :JUMP BACK INTO MAIN ROUTINE
```

The Sprite Move Routine

We now have routines to load a srite, expand it and send it to the interrupt buffer. These have been all chained together so all we have to do is call SPRINT which will be done with this routine which updates the values of XP, XC, YC and YP according to VX and VY which as you will recall are other variables in the movement table. It also sets up all the values needed by SPRINT and the other routines. The only parameter required by SPRMV is the address of the motion table in IY.

```
OBJECT: GENERAL PURPOSE SPRITE CONTROLLER
ENTRY: IY = MOTION DATA TABLE ADDR
NOTE: B'C'D'E'H'L'A'F' DESTROYED
```

IY PRESERVED

```
61614 FD3509    SPRMV     DEC (IY+9) :DEC CYCLE COUNT
61617 C0                  RET NZ
61618 FD7E0A              LD A, (IY+10)
61621 FD7709              LD (IY+9), A
61624 FD6608              LD H, (IY+8) :LET HL = ADDR OF IMAGE O
61627 FD6E07              LD L, (IY+7)
61630 FD7E00              LD A, (IY+0) :ADD STEP TO XP
61633 FD8601              ADD A, (IY+1)
61636 F2D3F0              JP P, NNEG1
61639 FD8602              ADD A, (IY+2) :IF RESULT NEG  THEN  LET
:XP = XP + XMAX AND LET XC = XC - 1
61642 FD3503              DEC (IY+3)
61645 FD5E03              LD E, (IY+3)
61648 C3EDF0              JP XDN :JUMP TO DEAL WITH Y
61651 FD4602    NNEG1     LD B, (IY+2)
61654 B8                  CP B IF XP<XMAX THEN GO FOR Y
61655 FD5E03              LD E, (IY+3)
61658 3811                JR C, XDN
61660 FD3403              INC (IY+3) :ELSE INC XC, LET XP = XP  -
:XMAX
61663 90                  SUB B
61664 08                  EX AF, AF'  :DEC  HL  BY  1  COLUMN  TO
:ALLOW FOR BLANK
61665 FD7E0C              LD A, (IY+12)
61668 01F8FF              LD BC, #FFF8 : MINUS 8
61671 09        NXUB      ADD HL, BC
61672 3D                  DEC A
61673 C2E7F0              JP NZ, NXUB
61676 08                  E AF, AF'
61677 FD7700    XDN       LD (IY+0), A :STORE NEW VALUE OF XP
61680 47                  LD B, A
61681 FD7E04              LD A, (IY+4) :ADD STEP TO YP
61684 FD8605              ADD A, (IY+5)
61687 F206F1              JP P, NNEG2
61690 E607                AND 7 :IF RESULT NEG THEN LET YP  =  YP
:MOD 8
61692 4F                  LD C, A
61693 FD3406              INC (IY+6) :AND INC YC
61696 FD5606              LD D, (IY+6)
61699 C313F1              JP YDN :IF YP>7 THEN LET YP =  YP  -  8
:AND DEC YC
61702 FE08      NNEG2     CP 8
61704 FD5606              LD D, (IY+6)
61707 4F                  LD C, A
61708 3805                JR C, YDN
61710 E607                AND 7
61712 FD3506              DEC (IY+6)
61715 FD7704    YDN       LD (IY+4), A
61718 C360EF              JP SPRINT
```

We also need the ATTLOC routine called for in SPRINT

```
61721 78       ATTLOC   LD A, B
61722 CB2F              SRA A
61724 CB2F              SRA A
61726 CB2F              SRA A
61728 C658              ADD A, 88
61730 67                LD H, A
61731 78                LD A, B
61732 E607              AND 7
61734 0F                RRCA
61735 0F                RRCA
61736 0F                RRCA
61737 81                ADD A, C
61738 6F                LD L, A
61739 C9                RET
```

To summarize: 7 bytes of the 17 in the motion table are initalized by the routines PADOUT and SPREX. The variables we have to initalize ourselves are XP, XC, YP and YC. Remember that XP is measured to the right and YP upwards from the bottom left corner of the sprite, while XC is measured to the right of and YC downwards from, the top left corner of the screen. XC and YC also are the coordinates of the top left corner of the sprite.

Speeds VX and VY are measured in the same directions as XP and YP, and may be greater than, less than or equal to 0. If VX>0 then movement is to the right, while if VX<0 then it is to the left. Similarly VY>0 is upward movement, while VY< 0 is downward. Thus having movement in both XP and YP causes diagonal motion. If the absolute values of VX and VY are not equal the sprite will move in off diagonal directions. Thus you can move sprites any which way you like at any speed you like.

The cycle count is provided as a means for regulating the frequency at which the sprites move, and also whether two or more sprites move in phase with each other.

For example, suppose that we have two sprites, with motion data at labels MDAT1 and MDAT2, and that we want one sprite to move every 5 TV frames, and the second to move once in every 3 frames. We set the respective "cycle periods" to values of 5 and 3, and as usual initalize the "cycle counts" to 1, so that both sprites will move on the first call of SPRMV. All that we then have to do is:

```
LD IY, MDAT1
CALL SPRMV
LD IY, MDAT2
CALL SPRMV
```

after each interrupt.

If we have two sprints moving at the same frequency, and we wish to keep them "out of phase", perhaps because there isn't enough

room in the interrupt buffer to animate them both in the same TV frame, then we use different initializations of the cycle count. If we give both these sprites a cycle period of 2, then we can make them move on alternate TV frames by setting the first count to 1, and the 2nd count to 2. Whatever happens, the cycle period and cycle count must always be non-zero. If zero we get a movement once in every 256 calls of SPRMV.

Using the concepts of cycle count and cycle period, we can animate all the sprites involved in a program in one block. If we place all their motion data consecutively in memory, then a suitable fragment after each interrupt might be as follows:

```
          LD IY, MDAT
          LD B, (NUMBER OF SPRITES)
NXTSPRT   PUSH BC
          CALL SPRMV
          POP BC
          LD DE, 17
          ADD IY, DE
          DJNZ NXSPRT
```

Also note that the adress of the first byte of image 0 is stored at (IY+7). This value is also returned in HL after the call to PADOUT to set up the sprite data. We can use this entry in the motion data as a means of switching or cycling through different sets of images for any one sprite. For example, you may wish to make your character "walk" rather than glide, or perhaps make your spacecraft gradually disintegrate in flight after being hit by a particular nasty plasma gun shot. To realize this function, set up as many different sets of sprite data as you need, storing the values returned by PADOUT in your own look-up table. IY should be kept pointing at one set of motion data, which will obviously then be set up with the last set of sprite data generated. Then when you are running your program, use an "animation count" and "animation period" analogous to the "cycle count" and "cycle period" to step through the different sets of images, retrieving the appropriate address from your look-up table and inserting it at (IY+7) every time you want to switch data.

There are various other manipulations of the motion data that are possible. You could make the sprite move in a preprogrammed pattern by running through a table of values for VX and VY, or you could make the sprite do a "chameleon" act by changing the attribute byte at (IY+15) but remember to preserve bit 7, the OR print flag. The one thing that can't be done is have a sprite in 2 or more colors.

A Sprite Demo--TEST

To demonstrate the sprite movement programs we will temporarily

abandon our horizon and gunsight program and instead we will move a red telephone (PAPER = 8) and a cyan card in opposite directions, reversing the directions several times and having them move over each other. This is done without the horizon generator as we have to expand the interrupt buffer out to 40 characters to accomodate the 4 x 3 phone and the 3 x 4 card. This actually takes 5 x 4 and 4 x 5 print positions which adds up to the full 40 positions of the expanded interrupt buffer. For the smoothest animation, we will move them one pixel at a time each and every TV frame.

Using the formula: 8(a(m+1)+1)(n+1) for calculating the space necessary for the images, for these 2 sprites we will need:

```
TELSPC = 8(8(4+1)+1)(3+1) = 1312 bytes
CARSPC = 8(8(3+1)+1)(4+1) = 1320 bytes
DEFW 0,0,0,0
```

Remember that we need an extra 6 bytes at the end of the last table.

The horizontal speed will be VX = 1, the maximum value of XP = 7, and the distance between two successive images will be D = 1 pixel. Hence the width of the safety margin to the right of our shapes will be:

```
Ph = D(VX +XP) - 8 = 1(1 + 7) - 8 = 0
```

so we can use the whole space of 4 x 3 and 3 x 4 fully. Since the sprites will not be moved vertically, no upper safety margin is necessary. Our actual pixel data for the phone and the card are held in TELDAT and CARDAT respectively.

We start with the phone and card motion tables:

```
61740 000100FC TELMTN    DEFB 0,1,0,252
61744 00000900           DEFB 0,0,9,0
61748 00010100           DEFB 0,1,1,0
61752 0000008238         DEFB 0,0,0,130,56
61757 00FF0020 CARMTN    DEFB 0,255,0,32
61761 0000090000         DEFB 0,0,9,0,0
61766 01010000           DEFB 1,1,0,0
61770 00008538           DEFB 0,0,133,56
```

To operate the OR printing function, we must make sure that the ORMAP is cleared with CLOR (which we already have) before each complete set of sprite movements is started. If we fail to do this, the new image of a sprite would print on top of its old image, causing an undesirable trail across the screen.

```
61774 D9       TEST      EXX
61775 E5                 PUSH HL
61776 D9                 EXX
61777 AF                 XOR A :SET ZERO HORIZON, BK SKY & SEA
61778 3205E9             LD (ROWS+1), A
```

```
61781 327BE8              LD (TOPBRD+1), A
61784 3231E9              LD (BOTBRD+1), A
61787 21A0F2              LD HL, TELSPC :GENERATE PHONE SPRITE
61790 11B8F1              LD DE, TELDAT
61793 010304              LD BC, #0403 :4 X 3 SIZE
61796 FD212CF1            LD IY, TELMTN
61800 CD96EE              CALL PADOUT
61803 1601                LD D, 1:SPEED
61805 CD08EF              CALL SPREX
61808 21C0F7              LD HL, CARSPC :SAME FOR CARD
61811 1118F2              LD DE, CARDAT
61814 010403              LD BC, #0304 : 3 X 4 SIZE
61817 FD213DF1            LD IY, CARMTN
61821 CD96EE              CALL PADOUT
61824 1601                LD D, 1: SPEED
61826 CD08EF              CALL SPREX
61829 CDDEE9              CALL INT1 :INITIALIZE INTERRUPT HANDLER
61832 76                  HALT
:MOVE SPRITES BACK AND FORTH 1 PIXEL AT A TIME PER TV FRAME
61833 0E02                LD C, 2: ONCE BACK AND FORTH ACROSS
61834 0600      NXAM2     LD B, 0
61836 C5        NXAM      PUSH BC
61838 CD11ED              CALL CLOR
61841 FD212CF1            LD IY, TELMTN :MOVE & PRINT PHONE
61845 CDAEF0              CALL SPRMV
61848 FD213DF1            LD IY, CARMTN :MAVE & PRINT CARD
61852 CDAEF0              CALL SPRMV
61855 C1                  POP BC
61856 76                  HALT
61857 10EA                DJNZ NXAM :NEXT FRAME
61859 FD7E01              LD A, (IY+1) :REVERSE  DIRECTION  ALONG
:X
61862 FD77F0              LD (IY-16), A
61865 ED44                NEG
61867 FD7701              LD (IY+1),A
61870 0D                  DEC C :NEXT PASS
61871 20DA                JR NZ, NXAM2
61873 CDFEE9                 CALL  DISINT  :RESET  TO  IM  1  AND
:RETRIEVE HL
61876 D9                  EXX
61877 E1                  POP HL
61878 D9                  EXX
61879 C9                  RET

SPRITE PIXEL DATA (2 lines per character space)
61880 0F1F3F7F TELDAT     DEFB 15,31,63,127
61884 FFFEFEFC            DEFB 255,254,254,253
61888 78300103            DEFB 120,48,1,3
61892 0307070F            DEFB 3,7,7,15
61896 1F3F7F7F            DEFB 31,63,127,127
61900 7F3F3F0F            DEFB 127,63,63,15
61904 FFFFFFFF            DEFB 255,255,255,255
61908 F0606060            DEFB 240,96,96,96
61912 FFFFF8F3            DEFB 255,255,248,243
```

```
61916 E7CFDE9C              DEFB 231,207,222,156
61920 9CDECFE7              DEFB 156,222,207,231
61924 F3F8FFFF              DEFB 243,248,255,255
61928 FFFFFFFF              DEFB 255,255,255,255
61932 0F060606              DEFB 15,6,6,6
61936 FFFF1FCF              DEFB 255,255,31,207
61940 E7F37B39              DEFB 231,243,123,57
61944 397BF3E1              DEFB 57,123,243,231
61948 CE1FFFFF              DEFB 206,31,255,255
61952 F0F8FCFE              DEFB 240,248,253,254
61956 FF7F7F3F              DEFB 255,127,127,63
61960 1E0C80C0              DEFB 30,12,128,192
61964 C0E0E0F0              DEFB 192,224,224,240
61968 F8FCFEFE              DEFB 248,253,254,254
61972 FEFCFCF0              DEFB 254,253,253,240
61976 3F60D8A0 CARDAT       DEFB 63,96,216,160
61980 B9ABB985              DEFB 185,171,185,133
61984 8F858180              DEFB 143,133,129,128
61988 81838185              DEFB 129,131,129,133
61992 8F858180              DEFB 143,133,129,128
61996 81858F85              DEFB 129,133,143,133
62000 81838180              DEFB 129,131,129,128
62004 80C0603F              DEFB 128,192,96,63
62008 FF000000              DEFB 255,0,0,0
62012 00810042              DEFB 0,129,0,66
62016 E7420000              DEFB 231,66,0,0
62020 00810042              DEFB 0,129,0,66
62024 E7420000              DEFB 231,66,0,0
62028 0042E742              DEFB 0,66,231,66
62032 00810000              DEFB 0,128,0,0
62036 000000FF              DEFB 0,0,0,255
62040 FC060301              DEFB 253,6,3,1
62044 81C181A1              DEFB 129,193,129,161
62048 F1A18101              DEFB 241,161,129,1
62052 81C181A1              DEFB 129,193,129,161
62056 F1A18101              DEFB 241,161,129,1
62060 81A1F1A1              DEFB 129,161,241,161
62064 81DD951D              DEFB 129,221,149,29
62068 051B06FC              DEFB 5,27,6,253
:IMAGE AREA FOR EXPANDED SPRITES
62072-62111                 DEFS 40
62112-63423    TELSPC       DEFS 1312
63424-64743    CARSPC       DEFS 1320
64744-64751                 DEFW 0,0,0,0
```

As you can see most of the space really is taken up by the expanded image tables for the two sprites--2640 bytes. You chew up quite a bit of memory expanding a few sprites fully and may run out of memory if you want them to move one image at a time. Remember that if you move 2 pixels at a time, your memory requirement gets cut almost in half.

These programs effectively fill all the space between 59500 and top of memory although we still have about 200 bytes left. I

originally organized this code to start at address 60000 but had to move it down to 59500. I decided that a phone and a card weren't really a good demonstration of sprites so wrote another little program that "puts it all together" which just fits in the remaining space with the aid of a shoehorn--I had to leave out a few not really necessary segments to do it.

However, to get back to the SPRITE demo at hand. Assemble the present code and make another save of the source and object code.

We also have to add and change some lines in the MASTDEMO program as well. Change line 80 to read:

```
 80 PAPER 7: GOSUB 200: POKE 59500,16: POKE 59520,16: POKE 59806,16: POKE 59890,40: POKE 60280,40: REM SET CHAR TO 40
```

All these POKES are to reset the length of the INTERRUPT BUFFER to accept 40 characters. The paper change is necessary to make sure we don't have red or cyan paper from other portions of the program.

Now add the following lines:

```
82 CLS : RANDOMIZE USR 59400

84 POKE 59500,112: POKE 59520,112: POKE 59806,112: POKE 59890, 24: POKE 60280,24 :REM RESET CHAR PRINT TO 24

85 POKE 59515,5 : POKE 59697, 1

86 RETURN
```

Resave MASTDEMO and run the program, this time selecting Menu Item # 4. Was it worth the effort?

Actually the horizon generator was running through all this but was set to make no changes in the program having a black border in both sea and sky. As I said, not a good demo of everything you can do. So now let's...

PUT IT ALL TOGETHER--SHIPS

That's right, a moving horizon, a set gunsight and a couple of ships going by on the horizon all at the same time. And all in just an additional 269 bytes.

In case you were wondering, you design the ships and other sprites in basic using as many UDG characters as you need to get them just right. Even if you have big sprites, 21 characters should be sufficient to do at least one at a time.

I decided that since the ships are on the horizon they could be quite small so a destroyer took 3x1 and an aircraft carrier took 7x1 spaces. Using the formulas given earlier and deciding that they would move 1 pixel space at a time, the amount of space needed for these 2 sprites would fit into the expanded sprite space already used by the TELSPC and the CARSPC. Since the SPRITE program regenerates these each and every time, we can use this same space for our aircraft carrier and destroyer respectively. Granted, that these sprites don't use all the space but so what. We at least don't have to assign new space. We do have another problem. Ships have a bow and a stern and usually are not seen moving backwards in the open sea. We just can't reverse directions as we did for the phone and card and make them run backwards. One would have to design a second sprite to reverse images of this nature going in the opposite direction. Keep this in mind when you design your game. AT 3 and 7 characters, our new sprite data will take 80 bytes of our precious space.

We will also need new motion tables of 17 bytes each for the 2 ships for another 34 bytes. We will want to start the Aircraft carrier completely off the left side. To do this we designate its XC value at 248 (minus 7). We will only want it to move at a slow rate of 1 pixel every 3 frames, so we designate its Cycle period as 3. And it will be done in white ink with OR printing so its ATTR BYTE is 128+7 = 135 with an attribute mask of 56. The destroyer will start off screen on the right so it has a XC = 32. We want it to move one pixel per frame so use a cycle period of 1 with the same attribute and mask as the aircraft carrier. Both will be on line 9 which is just above the center of the center of the gunsight.

Now all we have to do is combine the code of TARGET with that of TEST. Here is where the shoe-horning came in. The combination was a bit too much for the remaining space. So I had to ask, what can I leave out? I decided that a value of 1 for horizon up and down motion would be all that was necessary so omitted that. I still needed more space so decided to omit the saving of HL' which is only 6 bytes but it was just enough so I put it into the entry code where I had more space. You can bend machine code rules but you can't break them.

Now, we need 13 bytes of interrupt buffer for the gunsight, (3 + 1)*(1 + 1) = 8 for the destroyer, and (7 + 1)*(1 + 1) = 16 for the aircraft carrier, or a grand total of 37 bytes. This is more than the 24 that American style TV's will allow so we have to compromise the horizon and not allow it to go all the way to the top of the screen. This would be no problem for our sub unless we were simulating a "dive" or a "surface" while looking through the periscope. We will show you how to do this from Basic as there also was not enough room to do this in code as well.

So without further ado:

```
64752 000100F8 AIRMTN    DEFB 0,1,0,248
```

```
64756 00000900            DEFB 0,0,9,0
64760 00010200            DEFB 0,1,2,0
64764 0000008738          DEFB 0,0,0,135,56

64769 00FF0020 DESMTN     DEFB 0,255,0,32
64773 00000900            DEFB 0,0,9,0
64777 00010100            DEFB 0,1,1,0
64781 0000008738          DEFB 0,0,0,135,56

64786 CDDEE9   SHIPS      CALL INT1: SET UP INTERRUPT BUFFER
64789 F3                  DI
64790 0E58                LD C, 88: SET HORIZON LEVEL
64792 CD24EA              CALL HRZST2
64795 0E0D                LD A, 13 : SET UP GUNSIGHT SPACE
64797 CD6DEB              CALL ALTRBF
64800 EB                  EX DE, HL
64801 212FEC              LD HL, TRGPOS
64804 0149EC              LD BC, TRGDAT
64807 E30D                LD A, 13
64809 CDA6EB              CALL SRVR2
64812 21807F              LD HL, #7F80: ATTR AND MASK
64815 2292ED              LD (ATT), HL
64818 21A0F2              LD HL, TELSPC: EXPAND CARRIER SPRITE
64821 11C5FD              LD DE, AIRDAT
64824 010107              LD BC, #0701: LENGTH AND WIDTH
64827 FD21F0FC            LD IY, AIRMTN
64831 CD96EE              CALL PADOUT
64834 1601                LD D,1
64836 CD08EF              CALL SPREX
64839 21C0F7              LD HL, CARSPC: NOW THE DESTROYER
64842 11ADFD              LD DE, DESDAT
64845 010103              LD BC, #0301: LENGTH AND WIDTH
64848 FD2101FD            LD IY, DESMTN
64852 CD96EE              CALL PADOUT
64855 1601                LD D, 1
64857 CD08EE              CALL SPREX
64860 FB                  EI
64861 01C003              LD BC, 960: TIME TO MOVE CARRIER ACROSS
:SCREEN = 3*256+3*8*(7+1)
64864 C5       NXAM4      PUSH BC
64865 CD8CEB              CALL SRVR1: RESET GUNSIGHT
64868 76                  HALT
64869 CD11ED              CALL CLOR
64872 FD21F0FC            LD IY, AIRMTN: CHECK CARRIER MOVE
64876 CDAEF0              CALL SPRMV
64879 FD2101FD            LD IY, DESMTN: NOW DESTROYER
64883 CDAEF0              CALL SPRMV
64886 3EFE                LD A, 254: CHECK HORIZ MOVE
64888 DBFE                IN A, (254)
64890 2F                  CPL
64891 E61F                AND 31
64893 2802                JR Z, ND3
64895 CBD1                SET 2, C
64897 3E7F     ND3        LD A,127
```

```
64899 DBFE                IN A, (254)
64901 2F                  CPL
64902 E61F                AND 31
64904 2802                JR Z, NU3
64906 CBD9                SET 3, C
64908 79          NU3     LD A, C
64909 3269EA              LD (CNTRL), A
:CHECK FOR HORIZON SPEED WOULD BE INSERTED HERE IF ROOM
64912 CDDBEA              CALL HRNNMK
64915 3E7F                LD A, 127: CHECK FOR BREAK
64917 DBFE                IN A, (254)
64919 1F                  RRA
64920 C1                  POP BC: IN CASE OF BREAK POP BC FIRST
64921 3005                JR NC, ENDS
64923 0B                  DEC BC: ADJUST COUNTER
64924 78                  LD A, B
64925 B1                  OR C
64926 20C0                JR NZ, NXAM4
64928 CDFEE9      ENDS    CALL DISINT
64931 3A6AEB              LD A,(ROLNTH)
64934 ED44                NEG
64936 4F                  LD C, A
64937 CD6DEB              CALL ALTRBF
64940 C9                  RET

64941 01010303 DESDAT     DEFB 1,1,3,3
64945 3D0FFF7F            DEFB 61,15,255,127
64949 0002029F            DEFB 0,2,2,159
64953 FF56FFFF            DEFB 255,86,255,255
64957 00000000            DEFB 0,0,0,0
64961 DEF8FFFE            DEFB 222,248,255,254

64965 00000000 AIRDAT     DEFB 0,0,0,0
64969 FF071F0F            DEFB 255,7,31,15
64973 00000000            DEFB 0,0,0,0
64977 FF99FFFF            DEFB 255,153,255,255
64981 00000000            DEFB 0,0,0,0
64985 FF99FFFF            DEFB 255,153,255,255
64989 023F1E1E            DEFB 2,63,30,30
64993 FF99FFFF            DEFB 255,153,255,255
64997 00000001            DEFB 0,0,0,1
65001 FF99FFFF            DEFB 255,153,255,255
65005 0000E080            DEFB 0,0,224,128
65009 FF99FFFF            DEFB 255,153,255,255
65013 00000000            DEFB 0,0,0,0
65017 FF90FDF8            DEFB 255,144,252,248

LAST BYTE USED 65020
```

It just fits! Assemble and save both source and object code for the last time. By this time the object code should have reached 5521 bytes.

We now have some things to add to the basic program to make this

program work. While we are at it we will also correct some other things missing in the earlier demos of the horizon as well.

Add the following lines:

```
 12 PRINT AT 13,4; "5. SHIP DEMO"; AT 15,4; "6. BORDER QUILT
DEMO"; AT 17,4; "7. GIANT M DEMO"; AT 19,4; "8. COUNTDOWN DEMO"
```

The first part of the line only is needed but you might as well add the rest while entering the line as they are the 3 demo's for the next section.

We also have to modify line 13 and 20 to read:

```
 13 PRINT '' TAB 4; "9. END"
 20 IF INKEY$ < "1" OR INKEY$ > "9" THEN GO TO 15
```

We need the same POKEs that we used to expand the interrupt buffer to 40 bytes that we used in line 80 at line 90. We also need the POKEs to get it back to 24 bytes so line 94 becomes identical to line 84 and I hope you know how to do that the fast way without having to type in the whole line in each case.

We now have to do some modifying of the buffer wait periods as we are printing too long a time and our horizon will be too low. These are the values at 64655 and 59678. Go back to the original discussion of this code to do it for your TV or monitor as it may be different than mine. For mine the following values worked:

```
 92 POKE 59655,28: POKE 59678,14: POKE 60008,1: PAPER 5: CLS:
RANDOMIZE USR 59428: REM Adjust timing

 94 POKE 64655,2: POKE 59678, 9: GOSUB 400: RETURN
```

Line 94 resets values back to what they were before we started.

There are a few things in those lines I didn't explain. The POKE at 60008 is necessary as we didn't adjust the SPEED of the Horizon move. It doesn't work so well with a zero. This makes the code we omitted work.

PAPER 5 is merely to get the whole screen to CYAN from whatever color the last demo was.

Because we jumped out of the code when we were done, the values of AIRMTN and DESMTN are whatever they ended up as. The SUBROUTINE at 400 merely resets them to their original values. This was not necessary in the phone-card demo because that program just conveniently left them where they originated when it finished. This also is the easy way to play around with different speeds and placement of the sprites. All you have to do is adjust the right data byte of line 405 and do a GOSUB 400 before calling the menu with a GOTO 10 and rerunning the SHIP

DEMO.

```
400 RESTORE 400: For X = 64752 TO 64785: READ y: POKE x,y: NEXT
x: RETURN
405 DATA 0,1,0,248,0,0,9,0,0,1,3,0,0,0,0,135,56,0,255,0,32,0,0,
9,0,0,1,1,0,0,0,0,135,56
```

We now have to write the entry code to the demo to be put at 59428. Remember that this code must save both the HL and HL' registers. But we had another problem with the horizon demo when we ran it. The problem was that the whole screen was CYAN until we ran the border up and down to do it. So how about a little routine to change the attributes below our starting point of the horizon with blue paper and white ink. This whole section of code would read: (everything in decimal this time as it is going to be added to data line 205:

```
59428 229           PUSH HL
59429 217           EXX
59430 229           PUSH HL: SAVE HL'
59431 217           EXX: BACK TO STANDARD REGISTERS
59432 33,96,89      LD HL, 22880
59435 17,97,89      LD DE, 22881
59438 1,159,1       LD BC, 415
59441 62,15         LD A,15 :BLUE PAPER, WHITE INK
59443 119           LD (HL), A
59444 237,176       LDIR
59446 205,18,253    CALL 64786: SHIPS
59449 217           EXX
59450 225           POP HL
59451 217           EXX
59452 255           POP HL
59453 201           RET
```

All we have to do is the change the end of the loop in line 202 to 59453 and add the numbers above to the end of data line 205.

We should also change line 120 to read RETURN and add 130 STOP.

The first time through the program just sit and watch and don't change the horizon. The second time use the bottom row left for "down" and the bottom row right for "up" to verify that the horizon works as the program runs. The problem is that you run the horizon down below the bottom of the ships or up way too high. One could rewrite the routine so that the code itself moves the horizon up one pixel line and then back down again to simulate waves rather than trying to do it manually. We leave that up to the student as it has to be located somewhere else. Remember that moving the horizon every frame is too fast for the eye to register. You need something slower.

There you have it, horizon, ships and gunsight. Now all that is left is to do the torpedoes. This ends our discussion of horizons and sprites. I hope that you found it worth your while.

Before going on with the next section which will also be included in MASTDEMO, we will finish up our paperwork properly. You might find this table useful in trouble-shooting your code. It's quite easy to check as I have done it backwards a line at a time as it appears in Zeus.

TABLE OF HORIZON-SPRITE ROUTINES

ENDS	FDA0	NU3	FD8C	ND3	FD81
NXAM4	FD60	DESDAT	FDAD	AIRDAT	FDC5
SHIPS	FD12	NXAM	F18D	NXAM2	F18B
CARDAT	F218	CARSPC	F7C0	TELDAT	F1B8
TELSPC	F2A0	TEST	F14E	CARMTN	F13D
TELMTN	F12C	YDN	F113	NNEG2	F106
NXUB	F0E7	NXD	F0ED	NNEG1	F0D3
SPRMV	F0AE	NXT82	F0A6	IN17	F091
NPR2	F07C	OUT82	F0A1	NXTY2	F04F
LFTLM2	F04A	HOPCL2	F0A0	RGTLM2	F046
NXTX2	F042	NXT81	F032	NINC2	F01E
IN16	F015	NOTOR3	EFD4	NPR1	EFFA
OUT81	F02D	NXTY1	EFC0	LFTLM	EFBB
HOPCL1	F02C	RGTLM1	EFB7	NXTX1	EFB0
NXTRT	EFA8	NROT1	EFAB	SPRTNO	F03A
ATTLOC	F119	NXA	EF6B	POS0	EF6E
SPRINT	EF60	NXBYT2	EF4A	NXBYT	EF39
NXSHF	EF33	NXPOS	EF2E	NXBYT3	EF26
NXROW9	EF1E	SUBDIV	EF0C	SPREX	EF08
WKSPC	EEF4	MUL1	EEDC	NXSCOL	EEB9
CL	EEEE	PADOUT	EE96	NXTDR2	EE70
NXTDR1	EE5D	CLS	EE83	MOIRE	EE3B
STEP	EE29	VERHOR	EE24	DIAG	EE1C
NXTSTP	EE15	BBC	EE0D	Y2Y1	EDFA
Y2X1	EDF1	DRAW	EDE1	VHSTP	EDDF
DIASTP	EDDD	INV1	EDDB	OVER1	EDD5
PLOOP	EDC7	PLOT	ED95	PFLAG	ED94
MASK	ED93	GO	ED69	PWAIT	ED60
PLACE	ED5D	HIPRINT	ED50	CHARS	ED4E
NOTOR	ED47	NXTROT	ED38	ATCC	ED4C
ORCHK	ED1E	CLOR	ED11	ORMAP	ECB1
NU2	EC09	ND2	EBFE	TSLP	EBEF
TRGDAT	EC49	TRGPOS	EC2F	TARGET	EBCE

NXCHR9	EBAA	SRVR2	EBA6	NXSRV1	EB93
ATT	ED91	SRVR1	EB8C	BLNK	EB84
HOPFL	EB88	ALTRBF	EB6D	ROBFPT	EB6B
ROLNTH	EB6A	RRZCOL	EB3A	NU1	EB1D
ND1	EB12	DMLP	EB04	DEMO	EAFC
HRZNMK	EADB	HCOL6	EAC7	NROWZ2	EAC3
UPINIT	EAC9	HCOL5	EAA0	SELF42	EA9E
ROWZ1	EA9C	NROWZ1	EA94	UP2	EAA5
RZSMV1	EA6A	CNTRL	EA69	HRZSPD	EA68
NOWRK	EA5D	HRZST2	EA24	HRZST1	EA0B
HRZN4	EA09	HRZN2	EA07	HRZN1	EA05
DISINT	E9FE	TBLP	E9EC	INT1	E9DE
SELF11	E9C4	HCOL4	E9B7	SELF4	E9B0
NOPLG	E9BB	NXTFL	E9A4	INIT2	E999
END	E9A8	INIT3	E991	HCOL3	E941
HCOL2	E935	HCOL1	E933	BOTBRD	E930
HRZN3	E923	SELF5	E91F	LN	E914
SCAN1	E912	GO4IT	E91D	WT1LN	E9C2
NOWAIT	E9CC	Y	E902	ZZ	E8EB
ROWS	E904	POR	E895	NTOR	E8C3
FAKE	E8F2	NXTCH	E882	TOPBRD	E87A
INTERP	E870	BORD	E86F	CHSTRE	E86E
BUFFPT	E86C	BUFFER	FF70		

Writing your own routines: Wanting to use the base programs in your own routines is quite simple for those of you who have entered all the code in Zeus and saved the source code. You merely have to remove all the demo programs, reassemble without them and then write your own routines to run what you want. The actual programs that do nothing but demonstrate the routines and their locations and lengths are:

DEMO	60156-60217
TARGET	60366-60488
MOIRE	60987-61058
TELMTN-CARMTN	61740-61773
TEST	61774-64751
SHIPS	64752-65020

If you don't need to plot lines the following can also go:

PLOT	60820-60892
DIASTP	60893-60986

This could be a whopping 3538 to 3632 bytes which would leave a kernel of only 1889 bytes. However, all is not lost for those of

you with disk who didn't go through all the hard work (?) of assembling and debugging codes. The two big routines are on the end and can easily be overwritten between addresses 61774 and 65020 for a recovery of 3247 bytes. The entry routines at 59400 also can go.

All these routines are together as one code save. You may want to add what follows or decide to omit it altogether in your programs. They do demonstrate other things that can be done.

WRITING TO THE BORDER

Well we can't quite do that! The only thing we can do is make the border various colored attributes by changing the border attribute while it is being written. However, we can produce full screen images that extend into the border area as long as we design in large sized chunks.

If you thought time was at a premium doing the horizon, imagine how much more of a premium it must be if we want to change colors in the top and bottom borders as they are being written to the screen. Going all out, we won't be able to switch border colors fast enough to make each character width a different color. The best we can do is to manage a switch every 4 characters wide. That will allow us to get 9 different colors into a border line. The fastest instruction to write to PORT 254 (which is what we have to do to switch border colors) is the OUTI instruction.

We will be using a simpler Interrupt Initialization routine as well. For this we use a set of codes we will assemble to address 58000. We will not use all that space.

We will also use a simpler Interrupt Initalization routine as well:

```
                ORG       58000
                DISP      55536 : WILL ASSEMBLE AT 48000
EXIT: BC = 0, DE = #FF01, HL = FF01
58000 3EFE      HIRON     LD A,254
58002 ED47                LD I, A
58004 010001              LD BC, #100
58007 67                  LD H, A
58008 69                  LD L, C
58009 57                  LD D, A
58010 58                  LD, E, B
58011 36FD                LD (HL), 253
58013 EDB0                LDIR
58015 ED5E                IM 2
58017 C9                  RET
```

The OUTI instruction works like this. The HL pair holds the

address of the data byte, the C register the low byte of the port address and the B address the high byte. In every execution, the B register is decremented, the port address is formed, the data byte at HL is sent out to the port, and HL is incremented. If B reaches zero the zero flag is set, if not, it is reset. It takes 16 T-states to run one cycle.

Well, if it takes 224 T-states to write a TV line, then we should be able to get 224/16 = 14 into a line. Not quite. About 64 T-states out of the 224 states is spent in flyback, reducing the actual time of writing closer to 10. Our interrupt handler needs two variables:

PICDAT contains the address of the data to write.
BORSTR the string of data being written which is organized as: 1st BYTE = # of lines.
Next 10 BYTES: the colors desired across the border.

```
58018 0000      PICDAT    DEFW 0
58020-58130 00  BORSTR    DEFS 111

58131 C5        BORPIC    PUSH BC :SAVE ALL REGISTERS
58132 D5                  PUSH DE
58133 E5                  PUSH HL
58134 F5                  PUSH AF
58135 08                  EX AF, AF'
58136 F5                  PUSH AF
:THE NEXT 3 BYTES ARE A WAIT LOOP WHICH YOU MAY HAVE TO ADJUST
:YOU MAY ADD AS MANY BYTES HERE AS YOU NEED TO ADJUST TIMING.
58137 29                  ADD HL,HL
58138 1800                JR FLYBAK
58140 3E04      FLYBAK    LD A, 4
:WAIT FOR (FLYBAK +1) TV ROWS WHILE BEAM REACHES TOP OF SCREEN
:ALSO MAY NEED ADJUSTMENT
58142 060F      SCANM     LD B,15
58144 10FE      LN4       DJNZ LN4
58146 00                  NOP
58147 A7                  AND A
58148 C8                  RET Z
58149 3D                  DEC A
58159 C21EE3              JP NZ, SCANM
58153 C0                  RET NZ :5 T-STATE TIMING TRIMMER
58154 2AA2E2              LD HL,(PICDAT)
58157 0EFE                LD C, 254 :SET PORT #
58159 7E                  LD A,(HL): LOAD # OF LINES
58160 23                  INC HL
58161 08        NXTLN2    EX AF, AF': STORE IN A'
58162 7E                  LD A,(HL): ROWS THIS LINE
58163 23                  INC HL
58164 54                  LD D,H: STORE START OF THIS ROW IN DE
58165 5D                  LD E,L
58166 EDA3      NXTRW     OUTI :SEND DATA
58168 EDA3                OUTI
58170 EDA3                OUTI
```

```
58172 EDA3              OUTI
58174 EDA3              OUTI
58176 EDA3              OUTI
58178 EDA3              OUTI
58180 EDA3              OUTI
58182 EDA3              OUTI
58184 EDA3              OUTI
58186 3D                DEC A :GENERATE NEXT ROW OF DISPLAY
58187 CA593E            JP Z, NXTLN
58190 62                LD H, D
58191 6B                LD L, E
58192 0600              LD B, 0 :BUT  FIRST  WAIT  30  T-STATES
:FOR FLYBACK TO FINISH
58194 0600              LD B,0
58196 1800              JR 0
58198 00                NOP
58199 18DD              JR NXTRW
58201 08       NXTLN    EX AF, AF': NEXT LINE OF BORDER
58202 3D                DEC A
58203 E6FF              AND 255 : 7 T-STATE EQUALIER
58205 C231E3            JP NZ, NXTLN2
58208 F1                POP AF :RETRIEVE ALL REGISTERS AND RET
58209 08                EX AF, AF'
58210 F1                POP AF
58211 E1                POP HL
58212 D1                POP DE
58213 C1                POP BC
58214 FB                EI
58215 ED4D              RETI
```

BYTES 65021-3 WILL HOLD 195,19 AND 227

I found out that I couldn't bring the start of the routine down to the top of the tube to make a picture no matter what value I poked into FLYBAK+1. Maybe you can do better with your set.

Actually we can get the first border change to the TV before it gets to the normal print area of the screen. The first change occurs 3 characters LEFT of the normal print area. That brings the 2nd change of color between what is normally character 0 and character 1 of the print line, the 3rd between characters 4 and 5, etc. The last line also gets 1 character space into the right border area. To visulize this it is best to assume that the border is "behind" the print area.

We have to have the ability to extend our pattern from the border into the print area when we want to so that we can't tell where border ends and text area starts. For this routine we will assume that the text area has been devided into 6 border lines each of which is 32 pixels deep. This is equivalent to 4 characters deep. In other words, we will make perfect squares on the screen. If you prefer to have narrower border lines or possibly border lines of variable depth, then ATTSET is easily adjusted.

We enter the routine with HL pointed at the border data of the first column of the first border line in the text area. ATTSET takes this byte, multiplies it by 8 to obtain a paper value, OR's it with the INK value of that cell and then places the lot in the first byte of the attribute file. The next border value is taken and used for the next four text columns of line 0, and this is repeated for the next six border columns. The value of border column 8 is used for the final 3 text columns, and this line of border data is then reprocessed 3 times for the remaining text lines of this border line. The whole thing repeats for each of the other 5 sets of "border lines".

```
USE: TO SET THE PAPER ATTRIBUTES WHEN GIVEN BORDER DATA
ENTRY: HL = ADDR OF 1ST BYTE OF BORDER DATA IN TEXT AREA AS
          PRODUCED BY "EXPAND"
EXIT: BC = 0, HL = #5B00

58217 110058   ATTSET    LD DE,#5800 :POINT HL AT START OF ATTR
58220 EB                 EX DE,HL
58221 0606               LD B, 6 : # OF BORDER LINES
58223 0E04     NXTLN3    LD C, 4 : 4 CHARACTERS WIDE
58225 D5       NXT14     PUSH DE
58226 C5                 PUSH BC
58227 0E38               LD C, 56 : MASK FOR PAPER
58229 1A                 LD A, (DE): GET BORDER BYTE
58230 07                 RLCA :X8
58231 07                 RLCA
58232 07                 RLCA
58233 AE                 XOR (HL) :USE INK OF CELL WITH PAPER
:AS NEW ATTR
58234 A1                 AND C
58235 AE                 XOR (HL)
58236 77                 LD (HL), A : WRITE IT
:REPEAT FOR NEXT 7 BORDER COLUMNS EACH OF WHICH IS 4 WIDE
58237 13                 INC DE
58238 2C                 INC L
58239 0607               LD B, 7 :COUNTER
58241 1A       NXT12     LD A, (DE)
58242 07                 RLCA
58243 07                 RLCA
58244 07                 RLCA
58245 AE                 XOR (HL)
58246 A1                 AND C
58247 AE                 XOR (HL)
58248 77                 LD (HL),A
58249 2C                 INC L
58250 AE                 XOR (HL)
58251 A1                 AND C
58252 AE                 XOR (HL)
58253 77                 LD (HL), A
58254 2C                 INC L
58255 AE                 XOR (HL)
58256 A1                 AND C
```

```
58257 AE                XOR (HL)
58258 77                LD (HL), A
58259 2C                INC L
58260 AE                XOR (HL)
58261 A1                AND C
58262 AE                XOR (HL)
58263 77                LD (HL), A
58264 2C                INC L
58265 13                INC DE
58266 10E5              DJNZ NXT12
58268 1A                LD A, (DE) :NOW 3 RIGHT MOST ATTR COL
58269 07                RLCA
58270 07                RLCA
58271 07                RLCA
58272 AE                XOR (HL)
58273 A1                AND C
58274 AE                XOR (HL)
58275 77                LD (HL), A
58276 2C                INC L
58277 AE                XOR (HL)
58278 A1                AND C
58279 AE                XOR (HL)
58280 77                LD (HL), A
58281 2C                INC L
58282 AE                XOR (HL)
58283 A1                AND C
58284 AE                XOR (HL)
58285 77                LD (HL), A
58286 23                INC HL
58287 C1                POP BC :HL NOW AT NEXT LINE OF ATTRS
58288 0D                DEC C :REPEATFOR NEXT3 ATTR LINES
58289 2803              JR Z, OUT1
58291 D1                POP DE
58292 18BB              JR NXT14
58294 F1        OUT1    POP AF :DISCARD LAST STACK ENTRY
58295 13                INC DE :POINTER TO NEXT LINE OF  BORDER
:DATA
58296 13                INC DE
58297 13                INC DE
58298 10B3              DJNZ NXTLN3 :REPEAT FOR NEXT  5  BORDER
:LINES
58300 C9                RET
```

As a demonstration for these first 3 routines, we shall produce a multi-colored "patch quilt" pattern of eight border lines by eight border columns. Each patch of the quilt will be 4 x 4 characters in size. The first and the last lines of these will be in the top border and the bottom border. At least part of these patterns will be visible on the screen although the top pattern may not all show on your TV or monitor depending upon the amount of room between your screen mask (the outer plastic frame around the tube) and the normal print area used by the screen. Each row of the quilt will thus be 32 scan lines high.

To start generating the pattern 32 scan lines above the print area I found that my TV needed to wait only 4 scan lines. Actually, I couldn't see the top of the pattern and made this adjustment by knowing where the bottom line of the 32 scan lines should be and adjusted accordingly. Your TV set or monitor may give you more room. (FLYBAK + 1) will thus become 4.

The border pattern will be built up in BORSTR. Since we have one byte for the number of lines and 8 rows and 11 bytes for each row, we will need 89 of the 111 bytes of BORSTR. Since we have the initial number of lines byte and a byte for the number of scan lines per section followed by the 10 colors in the line, the start of the attributes for line 2 is at (BORSPC +1 +11 +1) or (BORSTR +13). We can thus set the PAPER attribute with:

```
LD HL, BORSTR +13
CALL ATTSET
```

USE: DEMONSTRATION OF BORPIC AND ATTSET

```
58301 3E04      BPDEMO    LD A,4 :MAY HAVE TO BE RESET FOR TV
:BEING USED
58303 321DE3              LD (FLYBAK+1),A
58306 21A4E2              LD HL, BORSTR :BUILT BORDER DATA INTO
:BORSTR
58309 22A2E2              LD (PICDAT), HL
58312 AF                  XOR A :START WITH BLACK BORDER
58313 3608                LD (HL), 8 : #OF BORDER LINES
58315 23                  INC HL
:LOOP TO GENERATE DATA FOR EACH BORDER LINE
58316 3620      NXBLIN    LD (HL),32 :32 SCAN ROWS THIS SECTION
58318 23                  INC HL
58319 3600                LD (HL), 0 :1ST BORDER COLUMN BLACK
58321 23                  INC HL
58322 0608                LD B, 8: RUN THROUGH 8 COLORS FOR THE
:MIDDLE 8 COLUMNS
58324 77        NXBCLM    LD (HL), A
58325 C603                ADD A, 3 :CALC NEW COLOR
58327 E607                AND 7
58329 23                  INC HL
58330 10F8                DJNZ NXBCLM
58332 70                  LD (HL), B :BLACK FOR LAST COLUMN
58333 23                  INC HL
58334 C603                ADD A, 3 :CHANGE STARTING COLOR FOR
:NEXT LINE
58336 E607                AND 7
58338 20E8                JR NZ, NXBLIN :WILL ONLY BE ZERO WHEN
:DONE SO CAN USE A AS COUNTER
58340 21B1E2              LD HL, BORSTR + 13: SET PAPER ATTR TO
:MATCH BORDER DATA
58343 CD69E3              CALL ATTSET
58346 CD90E2              CALL HIRON :RUN THE PROGRAM
58349 76        TSLP9     HALT
58350 10FD                DJNZ TSLP9 : B = 0 FROM ATTSET SO WILL
```

```
:RUN FOR 256 SCREEN REFRESHES OR 4.27 SECONDS
58352 ED56                IM 1 :RESELECT IM 1 FOR BASIC
58354 3E3F                LD A, 63
58356 ED47                LD I, A
58357 C9                  RET
```

It is time for the first assembly and saves of the new source code and new object code. I used the names of MARS.BIN for the source code and MAROBJ.BIN for the object code. However, in the process of doing this 3 different times by the time I was done I always saved the object code under different names and ended up with the final version as MAROBJ2.BIN which is what it is called in the MASTDEMO basic program. Change the names in line 1 of MASTDEMO if you use different values.

Load MASTDEMO and make the following changes in it:

Add to line 1 :CAT "marobj2.bin",58000,

Change line 100 to:

```
100 GOSUB 300: RANDOMIZE USR 59466: RETURN
```

We have to add the routine at 300 which is merely the entry to the routines. It actually is added on to the end of the first set of entries we used earlier but it does POKE in a new set of values for the Interrupt Generator.

```
300 POKE 65021,195: POKE 65022,19: POKE 65023,227
305 RESTORE 300: FOX X = 59454 TO 59471: READ Y:  POKE  X,Y:NEXT
X: RETURN
310 DATA 229,205,249,228,225,201,229,205,62,228,225,201,229,
205,189,227,225,201
```

We gave you all the entry codes for all 3 demos in this section at one time. Hope you don't mind.

After checking your new lines for correctness, save a new copy of MASTDEMO and then run the program using option 7. As was pointed out in the preceeding text, you may have to make a few adjustents for your particular screen. IF you decide to add more bytes at 58137-40 to adjust timing, remember to also adjust the entry values in line 310 for the changed entry points.

You should have a nice pattern on the screen. Of course if you want the pattern to last until a break is received you can do so at address 58350 by adding the routine we used earlier and just having the end jump back if no break is received. You should be enough of an expert to do that by now.

Of course this demo was quite specific. We should write a routine to generate data for any number of scan lines and any number of columns that we want. This is the object of EXPAND. We will use what is called "compact border data".

The routine will allow us to specify any number of border lines, each of any depth (up to 256 in each case), and use two colors for each border line, which will then be defined by the LEFTMOST 10 BITS of two "compact data bytes". Note the use of bits and bytes. Each of the bits corresponds to one border column on a border line. Using a system analogous to the 2068 Basic use of INK and PAPER, let the two colors available on each border line be BINK (border ink) and BAPER (border paper), denoting use of BINK color by a "1" in the bit and the use of BAPER color by a "0".

We only need to specify new values of BINK/BAPER whenever we want to change colors as we work down the screen. We need some way of telling EXPAND to start using new colors. The easiest way is to use the spare 6 rightmost bits of the data for a line. We set these to 3F(hex) for a change of colors, and then immediately follow that byte with two bytes the first of which is the new BINK color and the 2nd of which is the new BAPER color.

We will also use those 6 last bits to indicate the "end of data" as well merely by setting them to 3EH. These useages will be commented on when we write another demo to use them. But first the EXPAND routine:

```
USE: EXPAND BORDER DATA FOR BORPIC USING COMPACT DATA SETS
ENTRY: HL = START OF COMACT DATA
EXIT: HL = ADDR OF 1ST BORDER VALUE OF 1ST  BORDER  BORDER  LINE
       IN TEXT AREA
      A = 0, B = BAPER COLOR, C = BINK COLOR
      DE = NEXT BYTE AFTER COMPACT DATA AREA

58359 11A4E2   EXPAND     LD DE,BORSTR :BUILD UP BORSTR TABLE
58362 EDA0                LDI :TRANSFER # OF LINES
58364 AF                  XOR A :A WILL COUNT # OF TV SCAN  LINES
:ALLOTTED TO DATA
58365 08                  EX AF,AF'
58366 4E       NEWCOL     LD C, (HL) :C = BINK COLOR
58367 23                  INC HL
58368 46                  LD B, (HL) :B = BAPER COLOR
58369 23                  INC HL
:IF WE ARE AT TEXT AREA THEN STORE ADDR OF EXPANDED DATA
58370 08       NXTWD      EX AF, AF'
58371 FE20     TPMRGN     CP 32: MAY BE ALTERED TO  CHANGE  DEPTH
:OF TOP MARGIN (SHOULD BE 4)
58373 C209E4              JP NZ, NYET
58376 D5                  PUSH DE
58377 86       NYET       ADD A,  (HL)  :UPDATE  USED  SCAN  LINE
```

```
:COUNT
58378 08                EX AF, AF' :RESAVE COUNT
58379 EDA0              LDI :XFER DEPTH OF THIS SECTION
58381 03                INC BC
58382 7E                LD A, (HL) :TAKE 1ST COMPACT DATA BYTE
58383 EB                EX DE, HL
58384 D5                PUSH DE
58385 1E08              LD E, 8 :FOR EACH BIT
58387 17        ABC     RLA : ROTATE OUT LEFT INTO CARRY
:PLACE A BAPER BYTE IN BORDER DATA IF BIT IS SET
58388 70                LD (HL),B
58389 D219E4            JP NC, PAPER
58392 71                LD (HL), C :ELSE INSERT BINK BYTE
58393 23        PAPER   INC HL :NEXT BYTE
58394 1D                DEC E :TEST COUNTER
58395 C213E4            JP NZ, ABC
58398 D1                POP DE :PROCESS 2ND BYTE OF COMPACT
:DATA
58399 13                INC DE
58400 1A                LD A, (DE)
58401 17                RLA :CHOSE BAPER OR BINK FOR IST 2
:BITS ONLY
58402 70                LD (HL), B
58403 D227E4            JP NC, PAPER2
58406 71                LD (HL), C
58407 17        PAPER2  RLA
58408 23                INC HL
58409 70                LD (HL), B
58410 D22EE4            JP NC, PAPER3
58413 71                LD (HL), C
58414 23        PAPER3  INC HL
58415 1A                LD A, (DE) :TEST FOR COLOR CHANGE
58416 13                INC DE
58417 EB                EX DE, HL
58418 F6C0              OR 192 :3FH = NEW COLORS REQUIRED
58420 3C                INC A
58421 28C7              JR Z, NEWCOL
58423 3C                INC A :3EH = END OF DATA
58424 C202E4            JP NZ, NXTWD
58427 E1                POP HL :RETRIEVE ADDR FOR ATTSET
58428 23                INC HL
58429 C9                RET
```

Notice line 58371 TPMRGN CP 32

It should be the number of scan lines above the print area. In short, it stands for TOP MARGIN. EXPAND uses this value to find the correct address in the border data for use as an entry value to ATTSET, where such a use is applicable. It then stores this value away and returns it in HL ready for immediate use, if so desired, with ATTSET.

To show you how this routine works, how about a giant M put on a background of four different papers, yellow, red, magenta and blue. The M will stand on the bottom of the screen. This will also show you how to set up the compact data area as well.

Our "M" will be designed in 4 x 4 character chunks. The diagram below illustrates:

```
    BIT     7   6   5   4   3   2   1   0   7   6
           ---------------------------------------
LINE 0                                                YELLOW
           ---------------------------------------    BAPER
     1                                                RED
           ---------------------------------------    BAPER
     2         XXX                       XXX
           ---------------------------------------
     3         XXX XXX                   XXX XXX      MAGENTA
           ---------------------------------------    BAPER
     4         XXX XXX XXX       XXX XXX XXX
           ---------------------------------------
     5         XXX       XXX XXX XXX       XXX      BLUE
           ---------------------------------------    BAPER
     6         XXX           XXX           XXX
           ---------------------------------------
     7        XXX XXX XXX                 XXX XXX XXX
           ---------------------------------------
```

We will use white BINK for the M itself and the squares marked with XXX will be white. Notice that we have labeled the bits on the top of the diagram. Each line will therefore take 2 bytes although the 2nd byte will only use the two high bits (7 and 6). These bytes would be the following pairs: (in hex)

```
00 00   00 00   41 00   63 00   77 00   5D 00   49 00   E3 80
  XX              XX              XX
```

We now have to intersperse the other necessary data in this sequence. We must precede this list with 4 bytes of data containing:

1. # of border lines, 2. BINK, 3. BAPER 4. # SCAN LINES FIRST SECTION.

Thus our data starts as:

```
08 -- 8 border lines
07 -- BINK
06 -- BAPER
32 -- SCAN LINES
```

At the bytes marked XX in the above sequence we must add 3F (all

1's in lower 6 bits) to the byte to indicate that we want to change colors and then insert new BINK and BAPER bytes and the number of scan lines for the next section before going on. We also have to insert a scan line number byte between each section as well. Finally, the last byte of the sequence (80H) must have 3EH added to it to indicate the "end of data".

Thus the whole sequence is:

```
08 07 06 32 00 3F 07 02 32 00 00 32 41 3F 07 03 32 63 00 32 77
3F 32 5D 00 32 49 00 32 E3 BE
```

The simple sequence of envents is:

```
LD HL, COMPRESSED DATA
CALL EXPAND
CALL ATTSET
CALL HIRON
```

USE: DEMO FOR EXPAND, ATTSET AND BORPIC TO PRINT A LARGE "M"

```
58430 3E20      MDEMO     LD A, 32 :32 SCAN LINES ABOVE PRINT
58432 3204E4              LD (TPMRGN+1),A
58435 3E04                LD A, 4
58437 321DE3              LD (FLYBAK+1), A
58440 21A4E2              LD HL, BORSTR
58443 22A2E2              LD (PICDAT), HL
58446 2164E4              LD HL, MDATA
58449 CDF7E3              CALL EXPAND
58452 CD69E3              CALL ATTSET
58455 CD90E2              CALL HIRON
58458 76        XLP       HALT
58459 10FD                DJNZ XLP
58461 ED56                IM 1 :BACK TO BASIC
58463 3E3F                LD A, 63
58465 ED47                LD I, A
58467 C9                  RET

58468 08070620 MDATA      DEFB 8,7,6,32, :COMPRESSED DATA TABLE
58472 003F0702            DEFB 0,63,7,2
58476 20000020            DEFB 32,0,0,32
58480 413F0703            DEFB 65,63,7,3
58484 20630020            DEFB 32,99,0,32
58488 773F0701            DEFB 119,63,7,1
58492 205D0020            DEFB 32,93,0,32
58496 490020E3BE          DEFB 79,0,32,227,190
```

Now that we have demo 2 of the margins in place we can again assemble and save both the source and object codes. Again reloading MASTDEMO and by changing line 120 to:

```
110 GOSUB 300: RANDOMIZE USR 59460: RETURN
```

We can then save MASTDEMO and select option 7. Again, you may have to adjust wait loops to get it to be perfect.

This is really the end of the margin routines. The routines can be used for other things as well, singly or in combinations. BORPIC on its own can supply some fancy graphics in the top and/or bottom margin if you remember that every scan line could be changed if you had that in mind rather then every 32 lines as we used.

On the other hand, remembering that the routines don't affect the display file or INK attributes, you could use BORPIC, ATTSET and EXPAND to provide a spectacular background when, say an arcade game has been frozen or you have just been exterminated or zapped.

COUNTDOWN

One last demo just to show you a sequence of pictures that spill over to the border more in line with the ships demo presented earlier. Here we go: 3, 2, 1, 0 ...

```
58501 08050220 DAT3      DEFB 8,5,2,32 :"3" COMPRESSED DATA
58505 FF802001           DEFB 255,128,32,1
58509 80200180           DEFB 128,32,1,128
58513 20FF8020           DEFB 32,255,128,32
58517 FF802001           DEFB 255,128,32,1
58521 80200180           DEFB 128,32,1,128
58525 20FFBE             DEFB 32,255,190

58528 08020420 DAT2      DEFB 8,2,4,32 :"2" COMPRESSED DATA
58532 FF802001           DEFB 255,128,32,1
58536 80200180           DEFB 128,32,1,128
58540 20FF8020           DEFB 32,255,128,32
58544 FF8020C0           DEFB 255,128,32,192
58548 0020C000           DEFB 0,32,192,0
58552 20FFBE             DEFB 32,255,190

58555 08070220 DAT1      DEFB 8,7,2,32 :"1" COMPRESSED DATA
58559 1C00207C           DEFB 28,0,32,124
58563 00200C00           DEFB 0,32,12,0
58567 200C0020           DEFB 32,12,0,32
58571 0C00200C           DEFB 12,32,0,12
58575 00207F80           DEFB 0,32,127,128
58579 207FBE             DEFB 32,127,190

58582 08010532 DAT0      DEFB 8,1,5,32 :"0" COMPRESSED DATA
58586 7F0020FF           DEFB 127,0,32,255
58590 8020C780           DEFB 128,32,199,128
58594 20CD8020           DEFB 32,205,128,32
58598 D98020F1           DEFB 217,128,32,241
58602 8020FF80           DEFB 128,32,255,128
```

```
58606 207F3E            DEFB 32,127,62

58609 85E4      BORTAB  DEFW DAT3
58611 A0E4              DEFW DAT2
58613 BBE4              DEFW DAT1
58615 D6E4              DEFW DAT0

58617 3E20      CNTDWN  LD A, 32  :USES  32  SCAN  LINES  ABOVE
:PRINT
58619 3204E4            LD (TPMRGN+1), A
58622 3E04              LD A, 4
58624 321DE3            LD (FLYBAK+1), A
58627 21A4E2            LD HL, BORSTR
58630 22A2E2            LD (PICDAT), HL
58633 F3                DI
58634 CD90E2            CALL HIRON
58637 21F1E4            LD HL, BORTAB
58640 0604              LD B, 4 :4 IMAGES TO DO
58642 5E        NXTNUM   LD  E,  (HL)  :TAKE  ADDR  OF  COMPACT
:BORDER DATA
58643 23                INC HL
58644 56                LD D, (HL)
58645 23                INC HL
58646 C5                PUSH BC
58647 E5                PUSH HL
58648 EB                EX DE, HL
58649 CDF7E3            CALL EXPAND
58652 CD69E3            CALL ATTSET
58655 FB                EI
58656 0632              LD B, 50 : APPROXIMATELY 1 SECOND
58658 76        PSE     HALT
58659 10FD              DJNZ PSE
58661 F3                DI
58662 E1                POP HL
58663 C1                POP BC
58664 10E8              DJNZ NSTNUM :TO NEXT IMAGE
58666 3E3F              LD A, 63 :BACK TO BASIC
58668 ED47              LD I, A
58870 ED56              IM 1
58672 FB                EI
58673 C9                RET
```

This is the last byte used. Therefore the space between 58674 and 59399, a total of 725 bytes, is not used and these routines could be moved up that much if you like. If you decide to do so remember to change the CALLs in the entry routine. We now have one last assembly to go along with one last save of source and object codes and of course one last change to the MASTDEMO program, namely:

```
120 GOSUB 300: RANDOMIZE USR 59454: RETURN
```

If you have been able to follow this all along you now have the

complete program with enough basic to switch to any one of eight demo programs provided in this chapter. The text that goes with these programs and these demonstrations should be enough for even the most novice of machine code writers to adopt them to their uses.

Again we will finish up our paperwork properly:

PSE	E522	NXTNUM	E512	CNTDWN	E4F8
BORTAB	E4F1	DAT0	E4D6	DAT1	E4BB
DAT2	E4A0	DAT3	E485	XLP	E45A
MDATA	E464	MDEMO	E43E	PAPER3	E42E
PAPER2	E427	PAPER	E419	ABC	E413
NYET	E409	TPMRGN	E403	NXTWD	E402
NEWCOL	E3FE	EXPAND	E4F7	TSLP9	E3ED
NXBCLM	E3D4	NXBLIN	E3CC	BPDEMO	E3BD
OUT1	E3B6	NXT12	E381	NST14	E371
NXTLN3	E36F	ATTSET	E369	NXTLN	E359
NXTRW	E336	NXTLN2	E331	LN4	E320
SCANM	E31E	FLYBAK	E31C	BORPIC	E313
BORSTR	E2A4	PICDAT	E2A2	HIRON	E290

The student will again realize that most of the code ends up being demo programs to run the basic routines and thus is not necessary to run his/her programs. Eliminating these programs is quite easy with Zeus if you assembled your own code. They are:

BPDEMO	58301 to 58357
MDEMO	58430 to 58500
CNTDWN	58501 to 58673

Removing these reduces the actual routines you need to support your program down to a mere 371 bytes. Recalling all the demo programs of part one of the code, one could easily assemble them to take the place of one of those demos and thus consolidate things down to a mere 4003 bytes and get everything above the 61000 mark. Your code could then be loaded below that and take as much space as you want.

Again all is not wasted to the user of the already assembled disks or tapes as you could overwrite the demos with sections of your code. There should be almost enough room for everything except maybe areas to run a whole horde of sprites. If you remember most of the sprite area is consumed with the sprite expanded data. You could conveniently expand them into lower addresses, and as was demonstrated, use the areas over when done with a particluar sprite and needing another.

We started this chapter quite a few pages ago. Although I have

not demonstrated an actual arcade game in this section, I hope I have given the reader enough food for thought when designing his/her next game.

Tips to Game Designers

I just want to write a few lines to help you avoid some of the mistates I have seen other people make in designing games in the hope that my readers can avoid making the same mistakes in writing their own games. There is a lot more to designing modern computer games then just scrolling "space invaders" across the screen and erasing them when zapped. Detection schemes of hits are sometimes based on the color of the attribute the bullet enters.

The selling point of most games is a combination of the gaming idea, the progressing difficulty of the game, and the graphics. Early games didn't spend too much time on graphics but the newer 2nd edition of revised and revamped games spend much more effort on the graphics. Games that simulate three dimensional representations of walls and spaces take quite a bit of predesigned memory which is just printed to the screen in sections depending upon the desired screen. Because of the repetativeness of "standardized designs" they still leave something to be desired. Sometimes graphics are the real selling point of why people like to play a game.

Unless there is some reason for doing it, try to design some randomness into your sequence of events so that the same screens don't always follow one another in unvarying monotonous sequence each time the game is played. Granted that some games of skill rely on this but if your game isn't one of them change things so that the results aren't always predictable.

Spend some time testing everything section by settion. I have seen programmers misuse the ON ERR command to cover programming errors. The player always knows something went wrong when all of a sudden the program has skipped something. Programs have to be tested extensively. You don't necessarily have to test them as friends are always too willing to run your games for you and tell you when things go wrong especially when you instruct them as to what sort of things to look for.

Something else that really isn't an error but only a slipup in programming is the sometimes lack of sufficent testing for movement of the joysticks. Make sure that you read the joystick often enough to prevent the designing of an impossible situation. I have witnessed more "sermonizing" on the sluggishness of joysticks on a particular game then I would care to mention. In reality, its not the joystick at all but too few readings of it for the program to respond properly. For you purist who do not know the meaning of the word "sermonizing" I

just want to remind you that the favorite word in a sermon is the name of God.

If your game is multi-person, have at least one version of the game where the computer takes on the roll of an opponent so that only one person can play the game. The real challenge of game writing comes in making a stupid computer play a game. (If you don't believe me, design the game of tic-tak-toe in Basic so that the computer plays an opponent.) This gets into some serious logic statements at times. Make sure your logic is absolutely right. This is even more true when adapting a game everyone already knows for the computer. The best way to attack the problem is to do it in Basic first. If it is fast enough, don't bother converting it to machine code. If it is sluggish, it is demandatory to speed up things by converting slow portions to code. It is unforgivable to write a program where the human has to wait for minutes for the computer to make up its mind as to what to do for its turn. If you can't do it in 30 seconds or less plan on some sort of graphic or sound routine to entertain the human or to let him know what is going on. This is especially true for the thinking type games as there is usually enough graphic action in the shoot em up type to keep things entertaining.

Best of luck.

CHAPTER 8. FACTUALS AND ADVENTURE PROGRAM GRAPHICS

I call these two programs "Diamonds in the rough". They could be real gems someday but in the form presented here need more work and polishing and, since they are in Basic, conversion to machine code. However, dispite this, they are far enough along to demonstrate what they are supposed to be doing. It also leaves something for the interested student to do.

Terrain--A Factual Program

If you have never heard about factual geometry, its about time. It is a technique used by many profesional graphic design programs. The technique was originally invented by Benoit Mandelbrot and is now used by computers to creat all the quirks and irrgularities of a mountain or other landscape by a random method. No two runs of this program ever creat quite the same landscape.

What this program does is takes a simple triangle, finds the midpoint of each side and adds a random positive or negative displacement to its position. Then it draws new lines connecting these points with each other as well as to the old points . It does this 5 times and is quite interesting to watch it happen as more and more trianges fill the space. We thus go from 1 to 4 trianges in the first pass, from 4 to 16 in the second, from 16 to 64 in the third, from 64 to 256 in the 4th, and from 256 to 1024 trianges in the final form. I think that's chopping up a triangle finely enough.

The first line of the program is the starting data. The first data bit actually being a standard deviation. It can be varied from 0, which means don't deviate at all to 1 which is too much for our uses. Values of from .35 to .65 give the best results. The other 6 data bits are the x and y coordinates of the starting triangle. Taking the values given in the data line gives a nice pattern although you are allowed to alter them for your own purposes. The program only works on triangles. It is left to the student to:

1. Modify it for other shapes.
2. Keep one already finished area while doing another.
3. Add appropite coloring.
4. Convert it to machine code as it already takes about 2 minutes to run.

The program prints the parameter (a) in the lower corner of the screen to inform you of progress. At the end, the value of S used is printed.

```
 10 DATA .4,0,100,100,145,255,100
 20 GOTO 100
 30 PLOT x(i),y(i): DRAW x(j)-x(i),y(j)-y(i): RETURN
 40 LET u = (x(k) + x(j))/2: LET v = (y(k) + y(j))/2  :REM  find
center point on a line
 50 LET r = s * (ABS ((x(j) - u)+ ABS (y(j) - v)) :REM calculate
 deviations
 60 LET x = INT ABS ((RND - RND)*r + u): LET y = INT  ABS  ((RND
- RND)*r + v) :REM Notice use of RND - RND!
 70 IF x > 255 THEN LET x = 255 : REM Keep on screen
 80 IF y > 175 THEN LET y = 175
 90 LET x(i) = x : LET y(i) = y : RETURN : REM Store values
100 LET d = 561: DIM x(d) : DIM y(d) : REM Initialize program
110 READ s: FOR J = 1 TO 3: READ x(j): READ y(j): NEXT j
120 BORDER 6
130 LET c = 2: FOR t = 1 TO 6
140 CLS
150 INK 4: REM Green
160 LET a = 0: FOR m = 1 TO c-1
170 FOR n = 1 TO m
180 LET j = a+n: LET i = j+m: GOSUM 30: REM Plot triangles
190 LET j = i+1: LET i = a+n: GOSUN 30
200 NEXT n: LET a = a+m: NEXT m
210 INK 0 :REM Black
220 LET d = a: FOR m = c TO 2 STEP -1
230 FOR n = 1 TO m-1:LET i = a+n: LET j = i+1: GOSUB 30
240 NEXT n: LET a = a-m+1: NEXT m
250 IF t =6 THEN GOTO 9000
260 FOR m = c TO 2 STEP -1
270 LET a = d: LET d = d-m+1: LET b = a+a-m
280 FOR j = a+1 TO a+m: LET i = j+j+b
290 LET x(i) = x(j): LET y(i) = y(j)
300 NEXT j: NEXT m
310 LET c = c+c-1: LET a = 1
320 FOR m = 2 TO c STEP 2
330 PRINT AT 21,29; a
340 FOR i = a+1 TO a+m STEP 2: LET j  =  i-m+1:  LET  k  =  i+m:
     GOSUB 40: NEXT i
350 FOR i = a+2 TO a+m STEP 2: LET j  =  i-m:  LET  k  =  i+m+1:
     GOSUB 40: NEXT i
360 FOR i = a+m+2 TO a+m+m STEP 2: LET j = i-1:  LET  k  =  i+1:
     GOSUB 40: NEXT i
370 LET a = k: NEXT m
380 NEXT t
9000 PRINT AT 20,24;"s=";s: COPY
9900 STOP
9910 CLEAR : MOVE "Terrain.bas",10
```

As you can see, the program is a lot of juggling of variables and could chew up a lot of memory. One saving grace is that each number that has to be saved is less than 256 so could fit into a single byte rather than a slugged variable.

Running the program and then trying to add colors other then

green and black presents a problem. You cannot use Bright green ink and regular green paper in the same character space as you would like.

As I said, the program is presented here to give those interested in this technique some food for thought.

ADVENTURE GRAPHICS

Adventure type games such as Dungeons and Dragons (copyrighted by TRS Inc) when converted to the computer have to be able to generate literally hundreds to thousands of different scenes on the screen. The memory of the 2068 would not be able to handle all these scenes as screens unless one resorted to what I call generic picture elements.

Imagine a small MAP of only 16 by 23 spaces. That is a total of 368 spaces. Now each space has 4 scenes to it as one looks North, East, South and West from it. As one looks north, one should see the local scene in the forground and the scene one step north in the background. Similarily, as one gazes East, the local scene should be in the forground and the scene East of the local scene in the background. Similar conditions apply to the South and West scenes. Drawing 4 scenes for 368 spaces is 1472 scenes. We would never finish if each was different. Also, since we see adjacent scenes in the background of the local scene, we must make sure that when we move to an adjacent scene that we have in the foreground what we saw in the backgound of the scene moved from. This is the function of the Map grid. Each local scene number is held in it. A companion background scene is written in the background by looking up the number of the appropiate adjacent scene from the MAP.

As Columbus said, "The world is round and and if I go far enough West I should eventually arrive home from the East." The MAP of an adventure world has the "Dark Ages" type borders--you can see off in the distance but you never go there. To simulate this type of impentratable boundary, adventure designers resort to:

1. Impassible mountains
2. Unventured seas
3. Vast stretches of tundra and glaciers
4. Unending desserts of sand dunes

This is done by surrounding our Map with a row of pixture elements that one can see but never get to. Our sample pattern thus grows to a grid of 18 by 25 needing even more scenes.

We thus have to resort to what is called GENERIC pictures. Generic pictures can be drawn of a Mountain, a forest, hill country, open plains etc. in two sizes, FOREGROUND and BACK GROUND. If we resort to overprinting of scene pictures with special event picutres we can cut down on the number of scenes

we need.

Special event scenes are things like a cottage, a town, a city, a castle, a ruin, a keep etc. which will be placed randomly at selected sites. Thus by printing a background of a hill or a mountain or a plain and then overprinting the special event scene we can get a, for example, cottage in the mountains, cottage in the forest, cottage on the plains etc.

We thus want generic scenes and special event scenes. These are easiest to do as line drawings using UDG graphics. Unfortunately, we only have 21 available at one time. These along with the 16 standard graphics and a few inverse and regular character graphics such as the colon, the underline and the period afford some additions. The slash and the backslash aren't too useful as they have a blank pixel line at the top and the bottom and so don't attach to other lines from adjacent characters.

How about designing some GENERIC UDG's that will be useful in numerous scenes. If instead of doing them in straight lines we do them in rough graphics they can still be used where we want straight line graphics. Some extremely useful generic UDG's in addition to a full slash and backslash are the half space slash and backslash going up a full character in 1/2 a character space. We need four of these for a right and left half of a space and the two slants. Similarly, a line that needs two characters to rise or fall a character space are useful as well, requiring another 4 for upper and lower halfs. We could also use a center of character line. Vertical lines at the start and middle of a character space would be nice as well. Half circles in a space have a use other than tree tops as well. Are you beginning to get the idea? It really is a case of saving as much space as we can and still do a lot of specialized scenes. We need a lot of space to run the rest of the program what with the monsters and the characters and the things that are found around the scene.

Lets start by defining 2 sets of UDG characters. We will print our generic picture by first printing the scene with one set and then switching UDG sets and adding the necessary elements from the 2nd set. We usually start with the set using the most UDG elements. To switch between sets we will use the lines:

```
98 POKE 23675,88: POKE 23676,255: RETURN  for set 1
99 POKE 23675,159: POKE 23676,254: RETURN for set 2
```

The UDG code entry program goes like this:

```
1 CLEAR 65183 :FOR X = 65184 TO 65535: READ Y: POKE X,Y: NEXT Y
100 DATA 113,134,1,6,25,6,8,49,142,97,128,96,152,96,16,140:  REM
      PINE TREES
110 DATA 14,16,100,25,98,136,112,129,112,8,38,152,70,17,14,129
120 DATA 98,56,66,132,123,1,1,1,98,28,66,33,222,128,128,128
```

```
130 DATA 112,128,0,0,0,0,0,0,14,1,0,0,0,0,0,0
140 DATA 1,2,2,2,2,66,162,162,128,64,64,64,64,64,64,64: REM
    CACTI
150 DATA 162,158,128,124,2,2,2,2,64,64,64,66,69,69,69,57
160 DATA 2,2,2,2,2,2,2,15,2,60,64,64,64,64,64,64,240
170 DATA 24,152,153,249,25,31,24,24
180 DATA 0,72,206,40,17,146,124,18: REM THORN/BRUSH
190 DATA 0,0,0,0,48,76,130,126: REM SCRUB BUSH
200 DATA 249,0,0,0,47,0,0,126,0,0,63,0,240,0,2,64: REM WATER
210 DATA 0,0,144,0,0,0,127,0
220 DATA 0,0,0,0,4,36,165,165: REM GRASS
230 DATA 33,0,88,1,192,2,22,7,114,35,11,120,177,32,249,201,62:
    REM SHORT PRINT ROUTINE
240 DATA 3,28,32,64,64,64,128,128,128,64,64,36,26,2,2,4: REM
    TREES
250 DATA 192,56,4,2,2,2,1,1,1,2,2,36,68,64,64,32,
260 DATA 56,68,68,130,130,108,16,16,131,68,68,40,40,198,1,0
270 DATA 128,64,64,32,32,192,0,0,3,4,4,8,8,6,1,1
280 DATA 128,96,28,2,1,0,0,0,0,0,0,0,192,56,4,3: REM DIAGONALS
290 DATA 3,4,56,192,0,0,0,0,0,0,0,0,7,24,96,128
300 DATA 128,96,16,8,8,6,1,1,0,1,2,2,12,16,32,192
310 DATA 128,64,64,64,32,6,16,16,8,8,8,4,4,4,2,2,2,1
320 DATA 1,1,1,2,4,4,8,8,16,16,32,32,32,32,64,128
330 DATA 16,16,16,16,16,16,16,16,128,128,128,128,128,128,128,
    128: REM VERT LINES
340 DATA 0,0,0,0,0,255,0,0,0
```

You can see what these graphis look like by entering lines 98 and 99 as given above along with the following line. My designation for user graphics is to enclose such letters in brackets. Thus, PRINT "(A)" means print a UDG "A".

```
  5 GOSUB 98
 10 PRINT "SET 1"
 15 PRINT "A B C D E F G H I J K L M N O P"'"(A B C D E F G  H
I J K L M N O P)"'"Q R S T U"'"(Q R S T U)"
 20 GOSUB 99
 25 PRINT "SET 2"
 30 PRINT "A B C D E F G H I J K L M N O P"'"(A B C D E F G  H
I J K L M N O P)"'"Q R S T U"'"(Q R S T U)"
 35 STOP
```

Either delete line 1 or use a GO TO 5 statement to run the program. You may want to run this program and copy down the look of the symbols if you ever decide to add additional graphics to these or refine some of the graphic scenes already built into the up coming program.

Use MOVE "ADVENTURE.BIN",65183,353 to save the code portion of the program. Don't bother with the basic loader program.

On to the program itself. Let's start at the end of the program and use a few space saving techniques.

```
9000 LET a = PI/PI: LET b = a+a: LET c = b+a: LET d = c+a: LET
e = d+a: LET f = e+a: LET g = f+a: LET z = g+a: LET i = z+a: DIM
a$(d,a): LET a$(a)="N": LET a$(b)="E": LET a$(c)="S": LET
a$(d)="W": LET q$="": LET c$=CHR$ 22+ CHR$ g + CHR$ 0+ " (32
underlines) ": LET S1=98: LET s2=S1+a: INK g: PAPER a-a: CLS:
BORDER f: GO TO 900

9990 POKE 23730,158: POKE 23731,254: CAT "ADVENTURE.BIN",65183,
353: GOTO 9000
9995 MOVE "ADVENTURE.BAS",9990: MOVE "ADVENTURE.BIN,65183,252
     or if you have only tape
9995 SAVE "ADVENTURE" LINE 9990: SAVE "ADVENTURE" CODE 65183,
383
```

The save lines are quite standard and one should have no problem understanding them. However, try line 9000. What are we doing and why? Simply, a = 1, b = 2, c = 3, etc. but z is out of place as h is going to be Horizontal. V will be vertical. Our basic program is going to be so full of small numbers and if you remember anything about numbers each one has an invisible to the screen slug of 6 bytes following it. Letter variables don't, so every time we use an "a" instead of 1 we save 6 bytes, even zero as a-a saves 4 bytes. S1 of course stands for SET 1 and SET 2 for Set 2 which will call the rountines at lines 98 and 99 respectively when changing UDG sets. The a$ array should be quite straight forward as to what we ere going to use that for. But what is this c$ for? Well, try:

```
CLS: PRINT c$
```

You get a line acrossed the screen at line 7. The unprintable characters 22, 7 and 0 at the start of the string tell the Basic interperter to do an AT 7, 0.

Like all good Basic programs we will cram our most important SUBS into low line numbers. Lines 1 to 8 handle the print a scene routines with overprints as encoded per line 9000. Lines 9 to 99 finish the actual routines and the switch graphic sets subs. I will give you that much of the program first and then talk about putting it all together. You won't understand the rational of lines 1 to 8 until after the MAP description.

Most everything inside quotes is either UDG graphics or standard graphics or inverse symbols. Therefore, we are going to omit the brackets and use them only to indicate an inverse character, e.g., (:) means inverse colon, (5) means inverse regular graphic 5. Add the REMs for now to remind you what things stand for. You can always delete them later.

```
 1 FOR w=a-a TO z: IF v<(g*c) THEN GO TO c :REM print routine
 2 LET v=v-g*c:NEXT w
 3 GO SUB (v+i+(z*e AND u)+(a and (v=17 OR v=19) amd (s=b OR
s=d))): IF w THEN GO SUB (w+g*d+a+(z*e and u): REM over print
 4 RETURN
```

```
 5 CLS: INK a-a: PRINT c$: LET  u=a-a:  LET  h=a-a:  LET  v=CODE
m$(y,x): GO SUB a: LET u=a: LET v=CODE m$(y+a,x): GO SUB a:  LET
u=a-a: LET h=z*b: LET v= CODE m$(y,x+a): GO SUB a: LET u=a:  LET
v= CODE m$(y+a,x+a): GO SUB  a:  RANDOMIZE  USR  65351:  INK  g:
RETURN: REM North

 6 CLS: INK a-a: PRINT c$: LET h=a-a: LET  u=a-a:  LET  v=  CODE
m$(y,x+b): GO SUB a: LET u=a: LET v= CODE m$(y,x+a): GO  SUB  a:
LET u=a-a: LET h=b*z: LET v= CODE m$(y+a,x+b):  GO  SUB  a:  LET
u=a: LET v= CODE m$(y+a,x+a): GO SUB a:  RAANDOMIZE  USR  65351:
INK g: RETURN: REM East

 7 CLS: INK a-a: PRINT c$: LET h=a-a: LET  u=a-a:  LET  v=  CODE
m$(y+b,x+b): GO SUB a: LET u=a: LET v= CODE m$(y+a,x+b): GO  SUB
a: LET h=z*b: LET u=a-a: LET v= CODE m$(y+b,x+a): GO SUB a:  LET
u=a: LET v= CODE m$(y+a,x+a): GO SUB a: RANDOMIZE USR 65351: INK
g: RETURN: REM South

 8 CLS: INK a-a: PRINT c$: LET h=a-a: LET  u=a-a:  LET  v=  CODE
m$(y+b,x): GO SUB a: LET u=a: LET v= CODE m$(y+b,x+a): GO SUB a:
LET h=z*b: LET u=a-a: LET v= CODE m$(y+a,x): GO SUB a: LET  u=a:
LET v= CODE m$(y+a,x+a): GO SUB a: RANDOMIZE USR 65351:  INK  g:
RETURN: REM West

 9 RETURN
```

In lines 9 to 97 all the things inside quotes are UDG or other graphics including underlines. Spacing with blanks is also critical. None of these lines are right justified in printing this book so you can count empty spaces from the line above. We also put a blank line between lines to make it easier to read.

```
10 GO SUB s1: LET v=c: PRINT AT v,h+d;"NO"; AT v+a,h+b;"LKMP";
AT v+b,h+a;"N  NOMNM"; AT v+c,h;"N  N P ONMNMNM"; AT v+d,h;
" LKM NMP  IJIJIJ"; AT v+e,h;"N LJQ  OIJ KMKIJI"; AT v+f,h;
"LK  ILKIJLKIJ"; AT v+g,h+c;"LK     IJ  IJ": RETURN

11 GO SUB s1: LET v=z; PRINT AT v,h;"EEEEEEEEEEEEEEEE"; AT v+a,
h;"HFFFFFFFFFFFFFFG": RETURN

12 GO SUB s1: LET v= g: PRINT AT v,h;" LUJLJLJLUJLJLJ"; AT v+a,
h;"KLJK  I I KLKLJI"; AT v+b,h;"K  IK IJLULKKLJI"; AT v+c,h;
" LKIJLJKLKILKLUJ"; AT v+d,h;"LUJLKKLJI LUJK LJ": RETURN

13 GO SUB s1: LET v=g: PRINT AT v,h;"_LJ________LUJ_"; AT v+a,
h;"_TTT T T T TTTTT "; AT v+b,h;" IUUJT TLUUJTTTT": GO SUB 99:
PRINT AT v+b,h; OVER a;"Q   Q___   Q____" AT v+c,h;
"Q    Q       Q": RETURN

14 GO SUB s2: LET v=c: PRINT AT v,h+b;"Q UQ RST QU  Q"; AT v+a,
h;"S   STR Q  Q  TSR"; AT v+b,h;"QRS Q  STR  Q  Q": RETURN

15 GO SUB s2: LET v=g: PRINT AT v,h+b;"_QQ";: GO SUB s1: PRINT
```

```
AT v+a,h;"LK  IUJ";: GO SUB s2: PRINT "Q": GO SUB s1: PRINT AT
v+b,h;"IUJ";: GO SUB s2: PRINT "QQ";: GOSUB s1: PRINT "UK":
RETURN

16 GO SUB s2: LET v=f: PRINT AT v,h+b;"_"; TAB h+f+g;"_"; AT
v+c,h+b;"O"; AT v+a,h;"Q Q Q      Q  JQ"; AT v+b,h+z+f;"Q": GO
SUB s1: PRINT AT v+a,h+a;" OVER a;"N M QTLUJ_ N M"; AT v+b,h+e;
"ST   QT"; AT v+b,h+e;"_"; AT v+c,h+d;"K I   ST"; AT v+c,h+z+b;
"_"; AT v+b,h+i;"K I": RETURN

17 GO SUB s2: LET v=z: PRINT AT v,h;" __  _ _Q __  _"; AT v+a,h;
"__ Q__  ___Q _" AT v+b,h;" Q___  __ Q__ _": RETURN

18 GO SUB s1: LET v=z: PRINT AT v,h+i+d;"LUK: AT v+a,h+i+b;"LK";
AT v+b,h+i+g;"_LUK": GO SUB s2: PRINT AT v,h+i+c; OVER a;"Q Q";
AT v+a,h+i;" Q Q"; AT v+b,h+f;"Q  Q": RETURN

19 GO SUB s2: LET v=g: PRINT AT v,h+a; OVER a;"O O     O   O";
AT v+a,h; OVER 0;"Q  RS      Q O"; AT v+a,h;"O  QQ O   O RST";
AT v+c,h+b;"OQ    OR   QQ": RETURN

20 GO SUB s2: LET v=z: PRINT AT v,h;"RSTTT TTTST  TRS"; AT v+a,h
0;"R  E TT TT   RST"; AT v+b,h;"ST  RST   RST": GO SUB s1: PRINT
AT v,h+d; OVER a;"UKIU  UKIU"; AT v+a,h;"LJ_UKIUKIU_LJ_"; AT v+b
,h+a;"_LJ_ _LJ_   UKI": RETURN

21 GO SUB s1: LET v=z: PRINT AT v,h;"IJ_        ____LK"; AT v+a,h
;"   IUUUUUK": GO SUB s2: PRINT AT v,h+d; OVER a;"ST RS  STT":
RETURN

22 GO SUB s2: LET v=z: PRINT AT v,h;"ABABABABABABABAB"; AT v,h;
OVER a;"GHGHGHGHGHGHGHGH"; AT v+a,h;"HABABABABABABABG"; AT v+b,h
;"G"; AT v+b,h+i+f;"H"; AT v+c,h; OVER 0;"HGHGHGHGHGHGHGHG":
RETURN

23 GO SUB s1: LET v=z: LET h=h-(a AND h): PRINT AT v,h+e;"IJJ";
AT v,h+f; OVER a;"U"; AT v+a,h; OVER 0;"MIJ"; AT v+b,h+z;"IJIJ";
LET h=h+(a AND h): RETURN

24 GO SUB s1: LET v=g: PRINT AT v,h+g; OVER a;"T"; AT v+a,h+d;
"LJQO LJ"; AT v+a,h+c;"_________"; AT v+a,h+c;"T   7    T"; AT
v+b,h+c;"_________": RETURN

25 GO SUB s1: LET v=g: PRINT AT v,h+i;"LUJ"; AT v+a,h;
"__LUEUUUJ_E_LUUJ"; AT v+b,h;"LUJ": RETURN

26 GO SUB s1: LET v=z: LET h=h+b: PRINT AT v,h;"LUJ    _LK"; AT
v+a,h+c;"_LUK    LUJ"; AT v+b,h+a;"LK    LUJ": LET h=h-b:
RETURN

27 GO SUB s1: LET v=g: PRINT AT v,h+i;"LUJ"; AT v+a,h;
"_ LEU UU E_LU J"; AT v+b,h;"LUJ"; AT v+b,h+g;"LUJ": RETURN

28 GO SUB s1: LET v=z: LET h=h+b: PRINT AT v,h;"LUJ    L"; AT
```

```
v+a,h+d;"L K    LUJ"; AT v+b,h+a;"L     LUJ": LET h=h-b: RETURN

29 GO SUB s1: LET v=z: PRINT AT v,h+f;"3 3E";AT v+a,h+f;"3 3 1";
AT v+b,h+e;"E(4) (4)E(4)E": RETURN

30 GO SUB s1: LET v=i: LET h=h+i; PRINT AT v,h;"AC"; AT v,h;
OVER a;"__"; AT v,h;".":LET h=h-i: RETURN

31 GO SUB s1: LET v=g: LET h=h+i; PRINT AT v,h;"QT"; AT v+a,h;
"ST"; AT v+b,h;"RT"; AT v+b,h; OVER a;"_": LET h=h-i: RETURN

32 LET v=f: LET h=h+i: FOR l=a TO b: PRINT AT v+l,h;"(5)": NEXT
l: PRINT AT v+l,h;"27": LET h=h-i: RETURN

33 LET v=z: LET h=h+i: PRINT AT v,h;"(4)7"; AT v+a,h;"21":
RETURN

34 LET v=f: LET h=h+i: PRINT AT v,h+a;"T"; AT v+a,h;"21"; AT v+b
,h;"88"; AT v+c,h;"5(5)"; AT v+c,h; OVER a;"__": LET h=h-i:
RETURN

35 GO SUB s1: LET v=i: PRINT AT v,h+g;"(2)3": RETURN

36 GO SUB s1: LET v=z: PRINT AT v,h+f;"NM"; AT v+a,h+f; OVER a;
"__E": RETURN

Lines 37 to 48 can be used for more scenes.

49 RETURN

50 GO SUB s1: LET v=g: PRINT AT v,h+g;"NO"; AT v+a,h+e;"LKMP";
AT v+b,h+d;"N  NOMNM"; AT v+c,h+c;"N  N P ONO"; AT v+d,h+b;
"Q LK  NMP P"; AT v+e,h+b;"R    Q   IJM"; AT v+f,h+a;
"NIJ   R    NOIJ"; AT v+g,h;"N  LJLK  LJK P KM"; AT v+z,h;
" LK  IJLK  IJ": RETURN

51 GO SUB s1: IF h THEN LET h=h-a: LET v=i+a: PRINT AT v,h;
"ACACACACACACACAC"; AT v+a,h;"TACACACACACACAC T"; AT v+b,h;
"ACACACACACACACAC"; AT v+c,h;"TACACACACACACACAC"; AT v+d,h;
"BT T T T T T T T"; AT v+e,h;"BDBDBDBDBDBDBDBD"; AT v+b,h;
OVER a;"I"; AT v+b,h+i+e;" K": IF h THEN LET h=h+a: RETURN

52 GO SUB s1: LET v=i+a: PRINT AT v,h+b;"_          __"; AT v+a,h;
"LK IJLKIJLK  IJ"; AT v+b,h;"_  LKLUJ IJ    IJ"; AT v+c,h;
" IJKLK   IJ_IUU"; AT v+d,h;"JKL     LK IJ"; AT v+e,h+b;
"LKIJLK     IJ_": RETURN

53 GO SUB s1: LET v=i: PRINT AT v,h+d;"N"; AT v+a,h;
"__NN      M_____"; AT v+a,h+b; OVER a; "__"; AT v+b,h; OVER 0;
" TSSIUUJLUKT T"; AT v+c,h;" TSP  T TT T T"; AT v+d,h;
"  R T T TT T T T"; AT v+e,h;"  T T TN TQ TT T"; AT v+f,h;"__K":
GO SUB s2: PRINT AT v+f,h+b; OVER a;"Q"; AT v+f,h+i; "Q  QQ":
RETURN
```

```
54 GO SUB s2: LET v=i: FOR l=0 TO d STEP d: PRINT AT v+l,h+b;
"Q UQ RST QU  Q"; AT v+l+a,h;"S  STR Q  Q  TSR"; AT v+l+b,h;
"QRS Q  STR  Q  Q"; AT v+l+c,h;"UQ R TRS  U RSTU": NEXT l:
RETURN

55  GO SUB s1: LET v=z+b: PRINT AT v,h+d;"EE EEE EE"; AT v+a,h+
c;"E_LUUUJ__EE"; AT v+b,h;"___N        M__E"; AT v+c,h;
"E             E"; AT v+d,h+a;"E_         _EE"; AT v+e,h+c;
"IJ____LK": GO SUB s2: PRINT AT v+a,h+d;"Q"; AT v+a,h+i+a; "Q";
AT v+b,h+a; OVER a;"QQ  RS       Q"; AT v+c,h+b;"RSTR  ST"; AT
v+d,h+d;"TS   TS"; AT v+e,h+a;"Q"; AT v+e,h+e; OVER 0;"Q":
RETURN

56 GO SUB s1: LET v=i: PRINT AT v,h+i;"_LJ_"; AT v+a,h+z;
"Q SS O"; AT v+b,h+d;"_   R RP P"; AT v+c,h+c;"Q T N N  M M"; AT
v+d,h+c ;"S_TN N    M M"; AT v+e,h+c;"S T"; AT v+f,h+c; "R_T";
AT v+g,h+c;"T_T"; AT v+z,h+a;"LK  IJ": GO SUB s2: PRINT AT v+a,
h+c;"RST"; AT v+b,h+e;"ST"; AT v+d,h+g; OVER a;"Q  QQQ  Q Q"; AT
v+e,h;"ST     S RST  R"; AT v+f,h;"RT     QQ     Q"; AT v+g,
h+i+a;"RST Q"; AT v+g,h+g+b;"Q RST R": RETURN

57 LET v=i+b: FOR l=0 TO c STEP c: GO SUB s1: PRINT AT v+l,h+c;
"_LJ_    LUUJ"; AT v+l+a,h;" LUUJ   _LJ_"; AT v+l+b,h;
"LJ   LUUJ   _LJ_": GO SUB s2: PRINT AT v+l,h+b;"Q"; AT v+l,h+i+
a;"Q"; AT v+l+a,h;"U"; AT v+l+a,h+g;"U"; AT v+l+a,h+i+c;"Q"; AT
v+l+b,h+d;"Q"; AT v+l+b,h+i;"U Q": NEXT l: RETURN

58 GO SUB s1: LET v=i+a: PRINT AT v,h+e;"ELEE"; AT v+a,h+d;"LKEE
"; AT v+b,h+b;"LKEEE"; AT v+c,h+d;"EEE"; AT v+d,h+b;"E  EE"; AT
v+e,h+c;"M  IJE"; AT v+f,h+d;"E   IJE"; AT v+g,h+e;"M MJ IJE":
GO SUB s2: PRINT AT v+b,h+a;"Q"; AT v+c,h;"QQRS"; AT v+d,h+c;
"ST"; AT v+e,h+d;"ST"; AT v+f,h+e;"TST"; AT v+g,h+f; OVER a;
 "T  S": RETURN

59 GO SUB s2: LET v=z+b: PRINT AT v,h;"IJ  Q      O"; AT v+a,h;
"KL     Q  RST"; AT v+b,h;"MNQ RS  IJ TSIJ"; AT v+c,h+e;
"RT KL   KL"; AT v+d,h;"RT IJ  PMNQ QMN"; AT v+e,h+c;
"KL   RST   PP"; AT v+f,h+b;"QMN Q   TRS": RETURN

60 LET v=i+b: FOR l=0 TO d STEP c: GO SUB s2: PRINT AT v+l,h;
"RSTTT TTTST  TRT"; AT v+l+a,h;"R  R TT TT   RST"; AT v+l+b,h;
"ST  RST   RST": GO SUB s1: PRINT AT v+l,h+d; OVER a;"UKIU  UKIU
"; AT v+l+a,h;"LJ_UKIUKIU_LJ_"; AT v+l+b,h;"_LJ_ _LJ_   UKI":
NEXT l: RETURN

61 GO SUB s2: LET v=i: FOR l=a TO f STEP c: PRINT AT v+l,h;
" ST   RST    RS"; AT v+l+a,h+c;"RS   RST    R"; AT v+l+b,h+c;
"TR   TRS": NEXT l: GO SUB s1: PRINT AT v+l,h;"IJ_       ___LK";
AT v+l+a,h+c;"IUUUUUK": GO SUB s2: PRINT AT v+l,h+d; OVER a;
"ST RS  STT": RETURN

62 GO SUB s2: LET v=z+c: PRINT AT v,h;" ABABABABABABAB"; AT v,h;
OVER a;" GHGHGHGHGHGHGH"; AT v+a,h; OVER 0;"ABABABABABABABAB";
AT v+a,h; OVER a;"G"; AT v+a,h+i+f;"H"; AT v+b,h; OVER 0;
```

```
"CABABABABABABABd"; AT v+c,h;"ABABABABABABABAB"; AT v+d,h;
"CDCDCDCDCDCDCDCD"; AT v+e,h;"EFEFEFEFEFEFEFEF": RETURN

63 GO SUB s2: LET v=i: PRINT AT v,h+e;"TS"; AT v+c,h;
"TS         RS"; AT v+d,h+a;"MTIJ____"; AT v+b,h+b;"MI TLK"; AT
v+c,h+c;"M TIJ"; AT v+d,h+d;"MTTJIJ"; AT v+e,h;
" ____M J  IJ_____"; AT v+f,h+d;"IJTT S  TLK"; AT v+g,h+f;
"MT SU TIJ"; AT v+z,h+g;"M SJ TTJIJ"; AT v+c,h+f; OVER a;"T"; AT
v+e,h+e;"_"; AT v+g,h+i+d;"T"; AT v+e,h+i;"S": RETURN

64 GO SUB s1: LET v=i+b: PRINT AT v,h;"R LJ R  RLJ S R"; AT v,h;
OVER a;"P    P  P     P"; AT v,h;"_________________";AT v+a,h;
"TT 7 TT TT 7  TT"; AT v+b,h;"TT   T88TT    TT"; AT v+c,h;
"TT   T88TT    TT"; AT v+c,h;"_________________": RETURN

65 GO SUB s2: LET v=i+a: PRINT AT v,h+c;"U"; AT v.h+i+c;"Q"; AT
v+a,h;"Q      U   U"; AT v+b,h;"__Q_____  Q ____"; AT v+c,h;
" __ __   ___  _"; AT v+e,h;____U__Q____UQ_": GO SUB s1: PRINT
AT v+b,h+e; OVER a;"E"; AT v+b,h+i+f;"E"; AT v+d,h;
"U UU  UUUU   UU"; AT v+f,h+c;"E"; AT v+f,h+i+a;"E": RETURN

66 GO SUB s1: LET v=i+a: PRINT AT v,h+i+f;"LK"; AT v+a,h+i+d;
"LKNN"; AT v+b,h+i+b;"LKL N"; AT v+c,h+i;"LK K N"; AT v+d,h+g;
"LKL  NN"; AT v+e,h+e;"LK K"; AT v+f,h+c;"LK LK N N"; AT v+g,h+a
;"LK L   N N": GO SUB s2: PRINT AT v+a,h+i+d;"Q"; AT v+b,h+i+a;
"U"; AT v+c,h+i+g;"U"; AT v+d,h+g;"U"; AT v+e,h+i+c;"Q"; AT v+f,
h+c;"U": RETURN

67 GO SUB s2: LET v=i+a: PRINT AT v,h+c;"U"; AT v,h+i+c;"Q"; AT
v+a,h;"Q      U   U"; AT v+b,h+c;"Q       Q"; AT v+c,h;
" __ __   ___  _; AT v+e,h+c;"U  Q    UQ": GO SUB s1: PRINT AT
v+b,h+e; OVER a;"E"; AT v+b,h+i+f;"E"; AT v+d,h;
"U UU  UUUU   UU"; AT v+f,h+c;"E"; AT v+f,h+i+a;"E": RETURN

68 GO SUB s1: LET v=i+a: PRINT AT v,h+i+f;"L"; AT v+a,h+i+d;"LK"
; AT v+b,h+g;"L LK"; AT v+c,h+i;"LKLK"; AT v+d,h+g;"L LK"; AT
v+e,h+f;"KL"; AT v+f,h+c;"LK K": GO SUB s2: PRINT AT v+a,h+i+c;
"U"; AT v+c,h+z; OVER a;"U   U"; AT v+e,h+d;"U   U": RETURN

69 GO SUB s1: LET v=i+a: PRINT AT v,h;"____      ____"; AT v+a,
h;"T   TCACACT   T"; AT v+a,h; OVER a;"____A     ____"; AT v+b,
h; "_"; AT v+b,h+i+b;"_"; AT v+c,h;"TNM_ACNM ACNM_T"; AT v+c,h;
"_"; AT v+d,h;"N  MTN  MTN  M"; AT v+e,h;"T8  TT 8 TT8  T"; AT
v+f,h;"T8  TT 8 TT8  T"; AT v+f,h;"_ E_E__ _E_  E_": RETURN

70 GO SUB s1: LET v=i+d: LET h=h+g: PRINT AT v,h+d;"_"; AT v+a,
h+c;"A C"; AT v+b,h+b;"N   M"; AT v+c,h+a;"A  K  O"; AT v+d,h;
"N     MP"; AT v+d,h; OVER a;"__AC____": LET h=h-g: RETURN

71 GO SUB s1: LET v=z: LET h=h+f: PRINT AT v,h+b;"_"; AT v+a,h+a
;"NTO"; AT v+b,h+a;"TTP"; AT v+c,h+a;"TT T"; AT v+d,h;"QQ  T";
AT v+e,h;"RSK T"; AT v+f,h;"TS  O"; AT v+g,h;"TSO S"; AT v+z,h;
"TSP S"; AT v+i,h; OVER a;"MSLK": LET h=h-f: RETURN
```

```
72 LET v=g: LET h=h+g: PRINT AT v,h+a;"3"; AT v+a,h;"(5)85": FOR
l=b TO z: PRINT AT v+l,h+a;"(.)": NEXT l: PRINT AT v+l,h;"2(I)1"
: LET h=h-g: RETURN : REM the (I) is inverse I

73 GO SUB s1: LET v=i+b: LET h=h+g: PRINT AT v,h+c;"S"; AT v+a,h
+a;"28881"; AT v+b,h+a;"(.+++.)"; AT v+c,h+a;"88888"; AT v+d,h;
"(5)88(I)885": LET h=h-g: RETURN

74 GO SUB s1: LET v=f: PRINT AT v,h+z;"S"; AT v+a,h+z;"S"; AT
v+b,h+f;"11822"; AT v+c,h+c;"111(.....)222"; AT v+d,h+c;
"8(+)8(+)8(+)8(+)8(+)8"; AT v+e,h+c;"(S)88(T)8888(T)8(S)"; AT
v+f,h+c;"888(T)8888(T)88"; AT v+g,h+c;"8(+111222+)8"; AT v+z,h+
c;"88 (5)3335 88"; AT v+i,h+c;"88 (5):S:5 88"; AT v+VAL "10",h+
c;"88 (5):SZ5 88"; v+VAL "10"; OVER a;"_______": RETURN: REM (S)
& (T) are inverse UDG's

75 GO SUB s1: LET v=i+a: PRINT AT v,h+c;"NMACAC"; AT v+a,h+b;
"N M T T___"; AT v+b,h+b;"T88 TTNM   M"; AT v+c,h;
"NTT88 DN  M___M"; AT v+d,h;"T      T8  TTT T"; AT v+e,h;
"      ET8_ET___T"; AT v+f,h;"       NN       "; AT v+g,h;
"TTTTTTTTTTTTTTTT"; AT v+g,h; OVER a;"UUUUUUUKUUUUUUUUU"; AT v+a
,h+i;"_"; AT v+c,h+b;"_  _TB      _"; AT v+d,h+i+c;"_"; AT v+e,h
+g;"_": RETURN

76 GO SUB s1: LET v=g: PRINT AT v,h+g;"LJ"; AT v+a,h+e;
"LK(4)7IJ"; AT v+b,h+c;"LK 8  8 IJ"; AT v+c,h+a;
"ACT         TC"; AT v+c,h+i+d; OVER a;"A"; AT v+d,h+a; OVER 0;
"T T8 8  8 8 T T"; AT v+e ,h+a;"T T         T T"; AT v+f,h+a;
"T T88 88 88 T T"; AT v+g,h+a ;"BDT___88____TD"; AT v+g,h+c;
OVER a;"_          B"; AT v+z,h+b; OVER 0;"($&)  N Q"; AT v+i,h;
"__T_TN  R______"; AT v+i+a,h;"TTTTT   TTTTTTTT"; AT v+i,h+b;
OVER a;"_      _"; AT v+i+a,h;"____N   _______": RETURN
```

Lines 77 to 97 open for further graphics

```
98 POKE 23675,88: POKE 23676,255: RETURN
99 POKE 23675,159: POKE 23676,254: RETURN
```

Lines 1-8 are the print routine which is entered at lines 5 to 8. These lines are the routines to print the N, E, S and W scenes respectively. These lines in turn call the subroutine at line 1 which sorts out the scene to be printed depending upon the vaule of u which is used as a "print near scene flag" when on and prints a far scene when off. Line 1 actually decides if the scene is an overprint scene or not. To save typing I didn't put in REM statements for the actual scenes. Lines 9 to 48 are for the far scenes while lines 49 to 89 are for the same scene printed near. Lines 90 to 97 can be other scenes.

Each picture to the screen is a composite of 2 far scenes and two near scenes where the actual point on the map is the lower right corner scene. If you would visualize the following grid:

0,0	0,1	1,2
1,0	1,1	1,2
2,0	2,1	2,2

Being at point 1,1, and looking north, we print scene 0,0 in the upper left background, scene 1,0 in the lower left foreground, scene 0,1 in the upper right background and scene 1,1 in the lower right foreground. This is done by line 5 which will print all north scenes.

Looking East from 1,1 we would use line 6 to print 2,0 in the left background, 1,2 in the right background, 0,1 in the left foreground and 1,1 in the right foreground.

For South we merely select line 7 to print 2,2 in the left background, 2,1 in the right background, 1,2 in the left foreground and 1,1 in the right foreground.

For West we repeat the same idea with line 8 to print 2,0 in the left background, 1,0 in the right background, 2,1 in the left foreground and 1,1 in the right foreground.

Notice that 1,1 is the lowest we can get. This leaves us always with background scenes. But we have another problem. If we are holding our map in an array called m$(y,x), the lowest value of y is 1, not zero. Similarly for x. Therefore, we have to do what in mathematics is called a transposition of axes. Position 1,1 is actually held in m$(2,2). Or to put it simply, we have to add 1 to both x and y when we read values from m$. Got it? That is why the 0,0 position actually becomes 1,1 when both y and x are 1 as given in the first part of line 5.

A look at lines 5 to 8 will show you that we use v to hold the peeked value of the Map array held in M$. The horizontal print position is set to zero for left printed scenes and to 16 for right printed scenes. U is set to zero for background and to 1 for forground.

After getting the scene number, the direction lines call the subroutine at line 1. If the number held in v is less than 20 it knows that the scene is a normal scene and goes to line 3 and adds 9 to the number and prints that scene if u is off. It thus would jump to lines 9 to 29. If u is on, it adds another 40 and prints a Near scene from lines 49 to 69. Notice that there are 2 offsets: the first of 9 and to get the right line number of far scenes after the first 8 lines of the program, the second of 40 for the far scene. Notice that as soon as a jump is made to the scene line, v is reset to the vertical line to start printing at and as such does double duty as a variable.

Scenes in line numbers 30 to 36 and 70 to 76 are what I call special event scenes and thus are scenes to be overprinted over a previous scene. The lines 1-2 For loop determines which two scenes these are from the value of v. How can this be done?

Think about it. We have 21 scenes to be printed with each over scene. If the scene at line 30, which is scene # 21 (subtracting off the 9 offset), we need another 20 numbers beyond this to designate the background. In other words numbers 21 to 42 are used to indicate scene 21 + the scene # of the background. Similarly, numbers 42 to 62 are for scene 22, 63 to 83 for scene 23, 84 to 104 for scene 24, 105 to 125 for scene 25, 126 to 146 for scene 26 and 147 to 167 for scene 27. I emphasize this in case you plan to add additional scenes on your own.

With this in mind, let's start doing an edit routine to adjust our scenes and enter a playing map. We will start with the Menu of the various scenes. The numbers listed in line 190 are with only the overprint offset.

```
190 CLS: PRINT TAB i;"MAP SCENES"'" 0 Open plain","16 Road
(horiz)"," 1 Mountain","17 Road (vert)"," 2 Forest","18 Path
(horiz)"," 3 Hills/Dunes","19 Path (vert)"," 4 Cliff","20 Small
town"," 5 Swamp","21 Cave"," 6 Lake","42 Monolith"," 7 Tablelan
d","63 Tower"," 8 Bog/Moor","84 Keep"," 9 River","105 Castle",
"10 Desert","126 Farmhouse","11 Rough sea","147 Inn","12 Shore
& sea"'"13 Pine Forest"'"14 Crevass"'"15 Walled Town": IF
Q$<>"h" THEN RETURN
```

The last statement in the line is not part of the menu but is used by the rest of the edit program which is to follow. You now could print a few test scene without the aid of lines 1-8 with a direct entry line that would go something like this: (Since v is not initialized in line 9000, do a LET v=a if you are doing this for the first time.)

CLS: Print c$: Let h=0: GO SUB (9 to 29): GOSUB (49 to 69): Let h=16: GO SUB (9 to 29): GO SUB (49 to 69)

The numbers in () give you a choice of those lines. For over printing a scene add: GO SUB 30 to 36 for a far scene or 70 to 76 for a near scene at the right spot--either left or right. Don't use the numbers given on the scene menu as they are encoded with the offset for the assembler.

Why don't we finish the Editor routine and then explain it by entering an actual Map after we have given the routine.

```
100 PRINT " ASSEMBLE"'"Hit ENTER to abort"
102 INPUT "(A)ssemble or (E)dit?";r$: GO TO 100 +(d AND r$="a"
OR r$="A") +(5920 and r$="e" OR r$="E")+(800 and r$="")
104 INPUT "Width of landscape grid?"; LINE r$: GO TO 106 +(794
AND r$="")
105 LET k=VAL r$
106 INPUT "height ": LINE r$: GO TO 109 +(791 AND r$="")
107 LET r=VAL r$
108 DIM M$(r,k)
110 LET xmin=a: LET ymin=a: LET xmax=r-a: LET ymax=k-a: IF q$
```

```
"" THEN GO SUB VAL q$
112 GO SUB 6020 -(20 AND r$="e" OR r$="E"): GO TO 900
  (Line 190 will be here)
200 PRINT " SAVE DATA ": PRINT #0;"Landscape complete? (y/n)":
PAUSE 0: IF PEEK 23556=13 THEN GO TO 900
201 INPUT "(T)ape or (D)isk? ";r$: IF r$="d" OR r$="D" THEN GO
TO 210
202 IF PEEK 23556<>89 THEN LET m$(r,k-b)=STR$ y: LET m$(r,k-a)=
STR$ x: LET m$(r,k)=STR$ s
205 INPUT "Title?"' LINE r$: SAVE r$ DATA m$(): GO TO 900
215 INPUT "Title?"; LINE r$: FOR t=LEN r$ TO 10: LET r$=r$+" ":
NEXT t: LET r$=r$+".var": MOVE "r$",: GO TO 900: REM pad out
name to 10 spaces and then save all variables--you can't just
save m$ array
300 PRINT " LOAD DATA": LET q$="301": GO TO 104: REM m$ must be
dimensioned if it isn't
301 INPUT "From (T)ape or (D)isk? ";q$: IF q$="d" OR q$="D" THEN
GO TO 310
305 INPUT "Title? ";r$: LOAD r$ DATA m$()
308 LET y=VAL m$(r,k-b): LET x=VAL m$(r,k-a): LET s=VAL m$(r,k):
GO TO 900
310 INPUT "Title? ";r$: FOR t= LEN r$ TO 10: LET r$=r$+" ": NEXT
t: LET r$=r$+".var": CAT "r$",: GO TO 900
400 GO TO 5000
500 PRINT ''"Use GO TO 900 to restart": STOP
900 CLS: PRINT '"OPTIONS"''"1 Assemble/Edit"''"2 Save Data"''
"3 Load"''"4 View"''"5 Stop"''"Press option number"
901 PAUSE 0: LET r$=INKEY$: IF r$>="1" AND r$<="5" THEN GO TO
VAL r$*100
4999 STOP
5000 LET s=a: LET x=a: LET y=a: GO SUB 5200
5001 PAUSE 0: LET r$=INKEY$: IF PEEK 23556=13 THEN GO TO 900
5005 IF r$="c" THEN GO TO 5300
5010 IF r$="m" THEN GO TO 5100
5020 LET s=(1 AND r$="7")+(2 AND r$="8")+(3 AND r$="6")+(4 AND
r$="5")
5030 GO SUB 5200: GO TO 5001
5100 PRINT #0;"AT ";(y);",";(x);"Move which way? Use crsors ":
PAUSE 0: LET r$=CHR$ PEEK 23556
5105 IF (r$<CHR$ 53 and r$>CHR$ 56) THE GO TO 5100
5110 LET y=y+(a AND r$="6" AND y<ymax)-(a AND r$="7" AND y>ymin)
5120 LET x=x+(a AND r$="8" AND x<xmax)-(a AND r$="5" AND x>xmin)
5130 LET r$="": GO SUB 5200: GO TO 5001
5200 IF m$(y,x)=" " THEN CLS: PRINT "loc ";y;",";x;": view ";a$
(s);" Empty": RETURN
5210 GO SUB d+s: PRINT #0;AT 0,0;"loc ";y;",";x;": view ";a$(s)
: RETURN
5300 LET py=a: LET px=a: GO SUB 190: PRINT AT 19,0;"Coordinates
: ";py;",";px;'"Use cursors to change start"'"then press SPACE"
5301 IN INKEY$="" THEN GO TO 5301
5302 LET px=px+(INKEY$="8" AND px<xmax)-(INKEY$="5" AND px>xmin)
5303 IF INKEY$=" " THEN GO TO 5315
5304 LET py=py+(INKEY$="7" AND py>pmin)-(INKEY$="6" AND py<pmax)
5305 PRINT AT 19,13;"      "; AT 19,13;py;",";px
```

```
5306 FOR 1=a TO i*b: NEXT 1: GO TO 5001
5315 ET y=py: LET x=px: GO SUB 5210: GO TO 5001
6000 FOR y=a TO r: PRINT "line ";y;": ";: FOR x=a TO  LEN(m$(y):
PRINT CODE m$(y,x);",";: NEXT x: PRINT ':NEXT y
6001 INPUT "Do you  wnat to change any picture element? (y/n) ";
q$
6002 IF q$="y" OR q$="Y" THEN GO SUB 6050
6010 RETURN
6020 CLS: PRINT "Enter H for help screen"; FOR y=a TO  r:  PRINT
"line ";y;" ";: FOR x=a TO LEN m$(y): INPUT "coord:"(y);",";(x);
" ";q$: IF q$="h" OR q$="H" THEN GO SUB 190: INPUT "coord:";(y);
",";(x);" ";q$: CLS: PRINT "line ";y;",";x;" ";
6025 LET t=VAL q$: LET m$(y,x)=CHR$ t: NEXT x: NEXT y
6030 RETURN
6050 CLS: LET q$="": GO SUB 190
6055 INPUT "What y? ";y: INPUT "What x?;x:  INPUT  "Coord:";(y);
",";(x);" Now:"(CODE m$(y,x));" What new value? ";q$
6056 LET t=VAL q$: LET m$(y,x)=CHR$ t
6060 INPUT "Another? (y/n) ";r$: IF r$="y" OR r$="Y" THEN GO  TO
6055
6070 RETURN
8999 STOP
```

Let's start at the menu at line 900. It gives the option of:

1. Assemble/Edit the scene map
2. Save the map to Tape or Disk
3. Load a map from Tape or Disk
4. View various map scenes
5. Stop

Notice the satatement of never using run to restart or we lose everything we have already done including our number variables.

The Assembly routine should only be used to initialize a new map or to start over a map. Edit is used to change the picture in a certain position on the map. It is suggested you draw your map on a piece of grid paper and encode it. Finally use the editor to enter a scene. If you make any errors entering the numbers make a note of the locations and enter Edit to change them.

Let's do a typical map. I picked a 25 wide by 18 deep map. This includes the spaces around the edge as discussed earlier. On my original design I chose the following single letters to represent the various scenes. Some are quite obvious:

M Mountain	F Forest	H Hills/Dunes	E Cliff
S Swamp	L Lake	A Tableland	B Bog
R River	D Desert	Y Sea w waves	Z Sea & shore
P Pine forest	V Crevass	W Walled town	1 Road horiz
2 Road Vert	3 Path horiz	4 Path Vert	Q Small town
C Cave	N Monolith	T Tower	K Keep

X Castle G Farm House I Inn

The A, E, Y, Z, 1, 2, 3, 4, Q and G are not obvious as I ran out of letters. I have also separated the overprint scenes from the regular ones to remind me I have to add a background to them as well.

Now for the Map with spaces for easier reading:(Blanks are open plains)

```
     1           6           11          16          21
 1   Y Y M M M   M M M M M   M M M M M   M M M M M   M M M M M
 2   Y Y H H H   H H H V H     H M H C   H P N H H   H H C M M
 3   Y H   P P   P P P   V     R M P P   P P P R G   3 3 H M M
 4   Y H   P P   P P         T R 3 3 3   K 3 R 3 3   P H E E M
 5   Y H     P   G P     F   R R R R R   R R R 4 P   M H E E M

 6   Y H               I R   F F F F       G   R H   X M H E M
 7   Y H H W 1   1 1 1 R F   F F F F G     Q   F R   F F F H M
 8   Y Y R R R   R R R T F   F F F F F           G F   R F F N M
 9   Y S S S S   S   R F     F F F F             F F   F F H H M
10   Y S S B S   S B   R       F F F       W     F V   F H M M M

11   Y S B B S   S     S L   T   I       L       Q F   V M C M M
12   Y S B B B         B S   R R R R R   S B     D D   N V D D M
13   Y E E           H H H   R K     B   R B     D D   D A A D M
14   Y E E E E         H H   R 4         B R R   Q A   A D A D M
15   Y Y Y Y     E E     H     R 4 D       B H   A W   D D D H H

16   Y Y H H H   E E   H H   H I R 4 D   D X D A A   A D C H H
17   Y M M M M   M E E Y Y   Y H E D D   H H H H H   H H H H H
```

I have to explain about the horizontal and vertical road and path. These are encoded as per a NORTH map. The print routine will reverse them for an East or West scene. Neat? You will also note the lack of the use of the sea with shore scene. It just didn't look good and is one of the scenes that needs work. While we are on scenes needing work, there also are combina- tions which have a problem between the left and right halves which leaves something to be desired and needs more work. As I said, a diamond in the rough--not quite done.

Notice that the map contains sea on the west and mountains on the North and partly down the East side. The SE corner contains a desert with sand dunes blocking the way. The South allows for some further expansion as a bay of the sea comes into the SW corner so that I could expand the Map at a later time--like say in another load of another map. When designing maps make sure that you don't change scenes too rapidly. Forests have to be big and so do plains and farmland. Pine Forests are found in the North and other forests further south with jungles in the tropics. Deserts have tablelands and sand dunes associated with them and all moutains have foothills before you get to them. Cliffs and hills are associated with the shores of oceans and seas. Swamps have bogs associated with them.

We go on to the encoding of the map. We have to remember the adding of the background values to the special event squares. Since the numbers will range from 0 to 167 I have trouble printing it in 64 columns for this book. We will split it at the 15 mark and do it in two sections.

```
Line/Column      5                        10                  15
 1   11 11 1  1  1  1   1  1  1   1  1  1  1  1  1
 2   11 11 3  3  3  3   3  3  14  3  0  3  1  2  22
 3   11 3  0  13 13 13  13 13 0   14 0  9  1  13 13
 4   11 3  0  13 13 13  13 0  0   63 9  18 18 18 18
 5   11 3  0  0  13 139 13 0  0   2  9  9  9  9  9

 6   11 3  0  0  0  0   0  0  147 9  2  2  2  2  0
 7   11 3  3  15 18 18  18 18 9   2  2  2  2  2  129
 8   11 11 9  9  9  9   9  9  67  2  2  2  2  2  2
 9   11 5  5  5  5  5   0  9  2   0  0  2  2  2  0
10   11 5  5  8  5  5   8  0  9   0  0  2  2  2  0

11   11 5  8  8  5  5   0  0  5   9  65 0  147 0  0
12   11 5  8  8  8  0   0  0  8   5  9  9  9  9  9
13   11 4  4  0  0  0   0  3  3   3  9  84 0  0  8
14   11 4  4  4  4  0   0  0  3   3  9  18 0  0  0
15   11 11 11 11 0  4   4  0  0   3  0  9  18 10 0

16   11 11 11 11 3  0   0  126 0   3  3  147 9  18 10
17   11 11 3  3  3  4   4  0  3   3  3  3  10 10 18
18   11 1  1  1  1  1   4  4  11  11 11 34 4  10 10

Line 16              20                  25
 1   1  1   1  1     1  1  1  1  1  1
 2   3  13  55 3     3  3  3  45 1  1
 3   13 13  13 9     145 19 19 3  1  1
 4   97 18  9  18    18 13 3  4  4  1
 5   9  9   9  18    13 1  3  4  4  1

 6   0  126 0  9     18 13 1  3  4  1
 7   0  20  0  2     9  2  2  2  3  1
 8   0  0   0  128   2  2  2  3  45 1
 9   0  0   0  2     2  2  2  3  3  1
10   0  15  0  2     14 2  3  1  1  1

11   6  0   0  20    2  14 1  106 1  1
12   5  8   0  10    10 10 52 14  10 10
13   9  8   0  10    10 10 7  7   10 1
14   8  9   9  20    7  7  10 7   10 1
15   0  8   3  7     15 10 10 10  3  3

16   10 115 10 7     7  7  10 24  3  3
17   52 10  49 7     7  3  3  3   3  3
18   3  3   3  3     3  3  3  3   3  3
```

We can now use the assebly portion of our editor program to enter the map. We can edit anything we want to change with the edit section as long as we know the coordinates. Please note that the coordinates start with 1,1 not 0,0. The program is user friendly in that it gives you the present value in the square so that if you happen to pick the wrong coordinates all you have to do is enter the same unchanged value and try one of the coordinates one removed from it.

The view section of the editor program (lines 5000-5400) allows you to view any scene. The cursor keys will allow you to view North with up, South with down, East with right and West with left. To move to new coordinates one uses an "m" and then the cursor keys. Once moved, the scene of the new coordinates is displayed. Thus once your final map is set you can delete everything in the editor except the 5000 lines. This secton of program can be kept in your own game program and does everything for you. Well almost everything. Since part of an adventure is getting lost, as it is up to the player to keep track of where he is, you sould delete the coordinate printing sections, and, unless they have a lode stone, the direction printing as well.

Of course all of this is redundant for those of you with the disc or tape of the programs as all you have to do is load "Adventure". The demo map is already included and the program is pointed at the editor menu.

Things left to do for "Adventure":

As I have already mentioned, the program is not done. It is not an adventure. It is just the scene portion of an adventure and even here the graphics are not complete. For example, the graphic scenes written so far are all outside adventure scenes. Generally at the special events one enters the cave, keep, tower, castle, inn, town or whatever. These interior scenes are not included and would have to designed and entered at the available line numbers mentioned earlier. You of course would not use a horizon nor would you of necessity need a near and far version of a scene. Since the present grraphics are generic, it is possible to design these interior scenes without designing any more UDG's.

Some typical cave or dungeon scenes would be:

1. Steps up and down
2. Tunnels going ahead and to the right or left.
3. Side entrances to tunnels
4. Vast rooms with special features such as an underground river with a waterfall with stalagtite ceiling.

Typical rooms for an inn would be:

1. The room for the general public where drinks and food are served.
2. Kitchen
3. Bedroom
4. Steps up and down
5. Storage rooms.

Some of these same rooms can be used for towers, keeps and castles although you would have to add the round tower rooms and an armory. One outside scene I forgot was the ruined keep, tower, castle or whatever. Rooms of rubble would have to be added for these as well.

Large towns and cities require:

1. Small narrow streets with buildings on both sides and occasional doors to shops and/or private homes.
2. If you have shops, internal shop scene.
3. If you have a church (not necessarily Christian), you need an interior altar scene.
4. If you have a kings castle, a throne room is necessary.
5. A general square or outside market scene is a place for a lot of interesting events.

Remember in all of this that you design a general scene. You don't have to add the shapes of people to these scenes as you always have 4 lines of script below the scene to describe anything special that is going on such as discovering a chest, running across a group of people or a monster or two, a horde of treasure, etc.

To do just the graphics for this, one has to modify the routines at 5000 probably adding an enter command which would jump to a subroutine to handle a new set of maps which would describe the sequence of scenes inside whatever you are in. Structures like castles, inns, keeps and towers and particularly, dungeons have levels so that when one goes down or up a level, a new map may be necessary.

By using a monster/treasure map (probably generated randomly so its different every time you run the program) especially when inside dungeons etc., you can manage everything. Here one of the big advantages of disc drives can come in handy as you could just load a new variable table replete with new sets of maps and string arrays of text from inside the program and continue to have the program run. This is not possible from tape as someone has to start the tape recorder and possible rewind a tape now and then. One reason why we want to keep the graphic section of the program as compact as possible is that one needs enough room for everything else. Monster graphics could be designed to be smaller than scene graphics and would always be written in the extreme foreground over everything else.

We have designed the present graphics to be white on black since

they are primarily line graphics. There of course is no reason to do this and one could use various different paper colors as background if one desired. Remember, however, that paper graphics go by character space. Just doing the lines in different colors doesn't work too well. Using wide splashes of paper color acrossed certain areas could be done.

The ultimate in compactness of course is to put all the scenes in code. This time its not for speed but to save space. With the full editor in place along with the 25x18 map the present program still leaves 24k of free space for other things which shows you something about how efficient our squeezing technique has been. We leave it up to the student interested in writing adventure programs to improve this shell program revising it as he/she sees fit.

Obviously, the present graphics won't do for a space odyssey but the idea of generic type UDG's could be used to design practically any scene one wants. You of course could get rid of the tree, pine and cactus graphics and replace them with other exotic life forms, planets, suns and star groups as well as distant space ships. The line graphics would be reserved for interior rooms of space ships etc.

The whole idea of this program is to give you a general idea on how various effects can be achieved and get you to thinking. Let your imagination go a bit wild and do something creative in generalized graphic routines for a change. As you can see, you first get it right in Basic and then cast it into the unbending mode of machine code. Designing something like this right into machine code would be a bit horrendous. Remember that in code most of your program is going to be strings of print data called from an address lookup table of some nature.

BIBLIOGRAPHY and COPYRIGHTS

Corcoran, V. C., and Branigin, M. H., "Timex Sinclair 2068 Personal Color Computor Technical Manual", Time Computer Corporation, Waterbury, CT 06720 $25.00

Dreger, Dr. Lloyd H., "The Timex/Sinclair 2068 ROM Manuscript", S.M.U.G., Box 101, Butler, WI 53007 (c) 1985 $16.95 +$2.50 S&H

Dreger, Dr. Lloyd H., "Introduction to 2068 Machine Code", S.M.U.G., Box 101, Butler, WI 53007 (c) 1986 $16.95 +$2.50 S&H

Franson, David, "Extended Paint" S.M.U.G., Box 101, Butler, WI 53007 (c) 1987 $19.95 +S&H

Hayes, Camron, "Timachine", Novelsoft, 106 7th St., Toronto, Ontario, Canada M8V 3B4 (c) 1984.

Kingsley, Ray, "HOT Z-2068", SINWARE, Box 8032, Santa Fe, NM 87504 (c) 1984

"Relocatable AERCO Print Driver", AERCO, Box 18093, Austin, TX 78760 (c) 1984

Smith, Joe C. Jr., Editor, "200 Computer Programs in Basic for the T/S 1000, ZX81, T/S 2068 (c) 1985 (No address given.) "Terrain" by Smith, Blanchard D., Alexandria, Va.

"Zeus Z80 Assembler", Softsync, 14 E. 13th ST., New York, NY 10016 (c) 1983

Webb, David, "Advanced Machine Code Programing for the Spectrum", Melbourne House (c) 1984 (Out of print).

The following are copyrights of the companies listed:

T/S 2068 Timex Compter Corporation

MSCRIPT Micro Systems Inc.

AERCO Acme Electric Robot Co.

Dungeons and Dragons TSH Inc.