SOFTWARE INTERNALS MANUAL FOR THE SORCERER

by

Vic Tolomei

Contents

1. Introduction to Machine Language
2. Sorcerer Devices and Ports
3. Sorcerer Monitor
4. Sorcerer Cassette Interface
5. Sorcerer BASIC Internals
6. Sorcerer Video and Graphics
7. Sorcerer Keyboard

SOFTWARE INTERNALS MANUAL FOR THE SORCERER

by
Vic Tolomei

Contents

1. Introduction to Machine Language ...... 1-1 to 1-6
2. Sorcerer Devices and Ports .... 2-1 to 2-5
3. Sorcerer Monitor .... 3-1 to 3-13
4. Sorcerer Cassette Interface .... 4-1 to 4-4
5. Sorcerer BASIC Internals .... 5-1 to 5-13
6. Sorcerer Video and Graphics .... 6-1 to 6-17
7. Sorcerer Keyboard .... 7-1 to 7-3
INDEX .... i-1 to i-5
NOTES .... n-1 to n-4

Published by Quality Software
©1979 by Victor Tolomei. All rights reserved.
No part of this publication may be
reproduced without prior written consent.

PREFACE

This document is designed to aid the Sorcerer programmer in easily utilizing the myriad of wonderful facilities of the machine. There are many Monitor subroutines, uses of cassette tapes, BASIC programming techniques, and uses of the Input/Output ports which require a detailed explanation to be used to the fullest extent.

To obtain all the benefits from this manual, please read the two books that come with the Sorcerer: "A Guided Tour of Personal Computing" and "A Short Tour of Basic". This internals manual is a supplement to these.

The manual is divided into seven chapters. Each is intended to be an independent "mini-manual" describing fully the topic under discussion.

The names SORCERER and ROM PAC have been trademarked by Exidy, Inc. The name Z80 is a registered trademark of Zilog, Inc.

CHAPTER ONE

Introduction to Machine Language

HEX, BINARY, AND DECIMAL

Before one can understand how the Sorcerer really works, some familiarity with machine language is necessary. First of all, let's discuss the concept of "hex". "Hex" is short for hexademical. This is a number system based on 16, not 10 as we are used to (decimal). In decimal, we have 10 possible digits, 0, 1, 2, . . . and 9. In hex, we have 16. Of course the first 10 are 0 through 9 as with decimal. But there are 6 more, A, B, C, D, E, and F. "A" means 10, "B" means 11, "C" 12, "D" 13, "E" 14, and "F" 15. So a number like 1CB3 makes sense in hex. In decimal numbers each digit represents a "power" of 10, namely "ones", "tens", "hundreds", and "thousands". For example, the decimal number 1895 means 1 thousands plus 8 hundreds plus 9 tens plus 5 ones, or

$$
\begin{aligned}
1895 &= 1\times1000 + 8\times100 + 9\times10 + 5 \\
&= 1000 + 800 + 90 + 5
\end{aligned}
$$

In hex however, each digit (0 through F) represents a power of 16, "ones", "sixteens", "two hundred fifty sixes", and "four thousand ninety sixes". For example, the **hex** number 1895 can be written as in the example above

$$
\begin{aligned}
1895 &= 1\times4096 + 8\times256 + 9\times16 + 5 \\
&= 4096 + 2048 + 144 + 5 \\
&= 6293 \quad \text{(decimal)}
\end{aligned}
$$

Another hex number 3CF1 can be seen as

$$
\begin{aligned}
3CF1 &= 3\times4096 + \mathbf{12}\times256 + \mathbf{15}\times16 + 1 \\
&= 12288 + 3072 + 240 + 1 \\
&= 15601 \quad \text{(decimal)}
\end{aligned}
$$

The reason why understanding the hex number system is so important is that the hexadecimal number system is used in describing system software for the majority of computers today—big, mini, and micro. This includes the Z80 MPU, which is the basis of the Exidy Sorcerer. Its machine language instructions are in hex and characters are all coded in hex.

If you understand hex, then "binary" (the number system based on **2**) should present no problems. There are only 2 digits possible to make any binary number, 0 and 1. These **bi**nary digi**ts** are called "bits". A bit can be 0 or 1. Each of these digits represents a power of 2 (1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, and 32768). So a number in binary like 0011110011110001 is

```
0011110011110001 = 0×32768  + 0×16384 + 1×8192 + 1×4096 +
                   1×2048   + 1×1024  + 0×512  + 0×256  +
                   1×128    + 1×64    + 1×32   + 1×16   +
                   0×8      + 0×4     + 0×2    + 1
                 = 8192     + 4096    + 2048   + 1024   +
                   128      + 64      + 32     + 16     + 1
                 = 15601    (decimal)
```

But that means, according to the previous example, that since 15601 decimal is also 3CF1 hex, then

0011110011110001 (binary) = 3CF1 (hex).

This is no mere coincidence. Let's see why. If we look at a "4-bit binary number" (ie, a number in binary made up of only 4 digits of 0's and 1's), then the smallest it could be is 0000 (0 decimal), and the largest it could be is 1111 (15 decimal or F hex). Thus every digit in hex, 0-F, can be expressed exactly as a 4-bit binary number:

Binary	Decimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6

0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

In other words, a hex digit is really just another way of writing 4 bits, or, every 4 bits of a binary number can be grouped as 1 hex digit. Let's see how that works with the numbers we just did. 0011110011110001 can be broken into groups of 4 bits (right to left) as follows:

0011 1100 1111 0001

If each 4-bits group is viewed individually, they calculate to

0011= 3 decimal (3 hex)
1100=12 decimal (C hex)
1111=15 decimal (F hex)
0001= 1 decimal (1 hex)

So it can be written

0011	1100	1111	0001	binary
3	C	F	1	hex

So hex and binary are actually the same thing, with different groupings. Another example, to write 0F8D hex in binary

0	F	8	D	hex
0000	1111	1000	1101	binary

which, when pieced back together, becomes

0000111110001101 = 0F8D

BITS, BYTES, ADDRESSES, AND "K"

Enough about decimal, hex, and binary. We now know how numbers are written on the Z80. Let us take a look at how memory is organized.

The smallest unit of information that can be placed in the memory of just about any computer made, including the Z80, is a bit, the same bit we saw earlier. This only holds a 0 or a 1 however, and is too small for normal numerical use. So a larger unit was created, called a "byte". A byte is just 8 bits or 2 hex digits grouped together.

So a byte can contain a number from 00000000 binary (00 hex, 0 decimal) to 11111111 binary (FF hex, 255 decimal). Each unique byte in the Sorcerer's memory space is assigned a 4-hex digit (2-byte) number called an "address". This address identifies the particular byte and its contents. Addresses start at 0000 hex and end at FFFF hex (65535). Thus the Sorcerer (Z80) can have up to 65536 bytes of memory. Another way programmers like to put this is to use the term "K". A "K" is just another way of saying the number 1024 decimal. So 65536 boils down to 64K (64 × 1024 = 65536).

RAM VERSUS ROM

Since we are on the subject of memory, there are two types. In one type the contents can never be changed. Information can only be "read" from it. This is called **R**ead **O**nly **M**emory or ROM (computerists love abbreviations or acronyms). ROM is usually used to contain programs or data which is to be present in the same state all the time. For example, the Sorcerer Monitor program is in ROM (starting at memory byte address E000 hex) and Sorcerer BASIC is in ROM (the ROM-PAC starting at address C000 hex). ROM can have its contents "burned in" permanently at the factory, or can be burned in once by the programmer (called PROM or Programmable ROM), or can be erased by strong ultraviolet light and burned in over and over again (called EPROM or Eraseable PROM).

However, for programmers to write and run programs, we need memory in which we can change or modify the contents. This is called **R**andom **A**ccess **M**emory or RAM. When the size of a Sorcerer's memory is given (e.g., 8K, 16K, 32K), this number applies only to RAM, or user-modifiable memory. All Sorcerers have the same ROM area potential. So a 16K Sorcerer has 16 × 1024 or 16384 bytes of RAM.

STATIC VERSUS DYNAMIC

The above two terms are usually only applied to RAM. Static RAM has the ability to hold its contents indefinitely as long as electrical power is applied. Dynamic RAM on the other hand quickly (in milliseconds usually) loses or leaks its contents, and the data must be re-written or refreshed to the RAM often enough to keep the data from disappearing altogether. Typically static RAM requires more power, is more expensive, but is faster. The Sorcerer and many other Z80 based systems uses dynamic RAM because of power and cost considerations, and also because the Z80 MPU is well-suited to interface to dynamic RAM (e.g., it can be made to do the RAM refreshing).

Z80 ARCHITECTURE

The Z80 microprocessor is an 8-bit based machine. In other words, its data flow and arithmetic is usually on a 1-byte basis. It can address up to 64K bytes of memory. On the Sorcerer, a maximum of 32K bytes of this can be placed onboard (in the keyboard unit), while another 16K can be located as ROM for the Monitor and various ROM cartridges.

In addition to having 64K of possible memory, the Z80 has 22 registers. These are special high speed memories which reside on the MPU chip, and are used for arithmetic and program logic functions. These are all 1 byte in size unless otherwise noted:

A — the accumulator. This is the central register.
F — the flags register. Each bit represents a CPU status. E.g., the "Z" bit is set on if the result of the execution of an arithmetic or logical instruction is zero.
B — general use register.
C — general use register.
D — general use register.
E — general use register.
H — general use register.
L — general use register.
SP — 2-byte register containing the current stack address.
PC — 2-byte program counter containing the address of the next instruction to be executed.

IX — 2-byte index register. Usually will contain an address to be used with a constant offset or displacement.
IY — 2-byte index register with the same type of use as IX.
I — register used to allow processing of external interrupts to the Z80 from the S100 bus.
R — refresh register which can be used to provide dynamic RAM refreshing operations.

Registers A, F, B, C, D, E, H, and L have an alternate register called A', F', B', C', D', E', H', and L'. Only one set can be used at a time, while the other set allows space to save important program information. The EXX and EX Z80 instructions are used to flip back and forth between them. Also some registers can be connected together to create 2-byte, 16-bit register pairs. These are AF, BC, DE, and HL.

For more detailed information on the Z80 MPU the reader is referred to the Zilog publication "Z80 CPU, Z80A CPU Technical Manual" product number 03-0029-01.

CHAPTER TWO

Sorcerer Devices and Ports

I/O DEVICES AND PORTS

The Sorcerer has the following I/O devices or ports. Listed also is the Monitor command(s) to activate each:

a.	the keyboard	SET I=K
b.	the video screen	SET O=V
c.	cassette tape #1	
d.	cassette tape #2	
e.	serial RS-232 interface	SET I=S, SET O=S
f.	parallel interface	SET I=P, SET O=P
g.	Centronics printer interface	SET O=L

Note that these are onboard ports. This list does not include any devices added to the Sorcerer via the S-100 expansion facility.

The keyboard is implemented as part of the Z80 I/O port number FE (254), input bits 0-4, output bits 0-3. The video screen needs no port but uses the 1920-byte RAM area at address F080 as a 64 by 30 screen. There is a port FE bit (input 5) indirectly related to video processing which signals when vertical retrace is in progress on the TV screen. The two cassette interfaces are part of the serial interface and provide an audio translation of the digital data suitable for recording on tape quite reliably.

The Sorcerer's serial and parallel ports are discussed in detail in this chapter. For further discussion of the video screen, see Chapter 6. For a further discussion of the keyboard, see Chapter 7.

SORCERER SERIAL PORT

The serial port allows data transfer to occur between the Sorcerer and external devices (such as printers, modems, cassette tape, and the like). Data travels one bit at a time in a predefined conventional sequence called "asynchronous transmission protocol".

The protocol defines how the data is to look, and the speeds at which it is to travel. For example, each 8-bit byte of data is actually sent as a 10- or 11-bit stream, sometimes even longer. The 8-bits must be preceeded by a bit called a "start bit", and must be followed by 1 or usually 2 or more "stop bits". These bits also must be send and received at a particular speed, predetermined by the sender and receiver. The speed is given in bits per second, or commonly called "baud" (derived from Baudot, the name of one of the forerunners of terminal communications). Thus 300 baud means 300 bits per second. Since it takes about 10-11 bits to transmit a byte or character, this means about 30 characters per second (cps). The Sorcerer serial interface "speaks" this common language, and operates at one of two speeds, either 1200 baud (120 cps) or 300 baud (30 cps).

The serial port is actually two devices, an RS-232C interface and the dual cassette interface. RS-232C is the name given to a widely accepted standard of signal voltage and logic levels and the pinouts of the 25-pin plug or connector used for cabling between the sender and receiver. The asynchronous protocol signals are usually sent via this RS-232C standard. Another part of the Z80 port FE, output bit 7, determines whether the serial port is RS-232C (bit on) or dual cassette (bit off). Dual cassette is the default. Output bit 6 controls the baud rate (1 for 1200 baud, 0 for 300 baud, default is 1). Port status is placed on port FD while data transfer occurs on FC. To connect a 300 or 1200 baud RS-232C serial printer to the Sorcerer, follow instructions given with the printer and from Exidy. However, the following guidelines may be used:

1. Connect pin 7 of the serial DB25 connector to printer ground pin 7.
2. Connect pin 3 to printer pin 2.
3. Connect pin 2 to printer 3.

Reset the Sorcerer, enter the Monitor (BYE in BASIC), enter the command SET O=S, and all output which would have gone to the screen will go to the printer until RESET or SET O=x is entered (x is usually V to return to video).

There is also software available from Exidy providing a serial device driver, and the ability to use the serial interface to turn the Sorcerer into a terminal connected to another computer. Typically a modem and possibly an acoustic coupler may be required here. For this use connect pin 2 to pin 2 and pin 3 to pin 3.

The cassette interfaces may also be used with motor control. Pins 12 and 24 can be used to turn cassette number 1 off and on for the commands SAVE, LOAD, FILE, and BATCH. Pins 13 and 25 work similarly for cassette number 2. Pins 15, 5 and 20 are the mike input, auxiliary input, and earphone output connections for cassette number 1. Pins 16, 18 and 21 perform the same services for cassette number 2. Note that cassette number 1 has these mike and ear connections duplicated as RCA plugs on the back of the Sorcerer.

SORCERER PARALLEL PORT

The parallel port differs from the serial port mainly in that data is transferred an entire byte at a time. This is ideal for fast printers and sometimes even some floppy disk units. The Sorcerer also provides an interface to the popular Centronics printer. The same parallel port is used, but unique software "handshaking" is done by the monitor I/O driver. An example of the handshaking which occurs between the Sorcerer and printer might be the following "electronic conversation" over port FE, the parallel interface status port:

Printer:	"Wait, I'm still busy, send no data." "OK, now you can send."
Exidy:	"Here it is, let me know when I can send more."

The 8-bit data (and at times status) rides on port FF.

To successfully hook up a Centronics or Centronics-like printer to the parallel port, again follow the printer's and Exidy's instructions. Here are some additional guidelines:

1. Connect parallel pins (DB25 connectors again) 5-7 and 16-19 (data bits 0-6) to the printer's data lines 0-6 (see printer's pinouts).
2. Connect pin 4 (data output bit 7) to the printer's input strobe line, a negative (true is low, false is high) pulse indicating data is ready to be transmitted.
3. Connect pin 1 to the printer ground.
4. Connect pin 25 (input data bit 7) to the printer busy line, indicating the printer is not ready to accept any data (probably still printing previous data or is out of paper).
5. Pins 2 and 3 (output accepted and available) and others may also be required depending on the printer model.

Once this is done, RESET the Sorcerer, enter the Monitor, type in the command SET O=L, and from that point on all output will be routed to the screen and the printer, until RESET occurs or until another SET O=x command is entered.

CHAPTER THREE

Sorcerer Monitor

SORCERER MEMORY MAP

To get an overall picture of how the Sorcerer utilizes the 64K of (possible) memory, a "memory map" is given in Figure 3-1. The term "HIMEM" in Figure 3-1 refers to the highest address in RAM and is described in detail later in this chapter. Table III-1 provides a more thorough description of the memory map of Figure 3-1.

Memory is cut up into pieces and each piece is used for a different purpose. In Table III-1, the address of the first byte of each piece is listed along with the use of that area. The address is given in both hex and a form of decimal that is usable directly in BASIC with the PEEK and POKE commands. Note that some of these decimal numbers are negative. If the address exceeds 7FFF (32767), then BASIC requires that the two's complement form of the number be used, i.e. the negative form. For decimal numbers greater than 32767, 65536 is subtracted from the number before use in PEEK and POKE instructions.

Be aware also that this is an **overall** wide angle view of memory. Detailed maps of the Monitor Workarea and the BASIC Control Area will be found in Tables III-2 and V-1 respectively.

MONITOR WORKAREA

This section is a detailed description of the Monitor Workarea, which is the area of memory beginning at location 1F91, 3F91, or 7F91, depending on the amount of RAM in the Sorcerer.

Figure 3-1. SORCERER MEMORY MAP

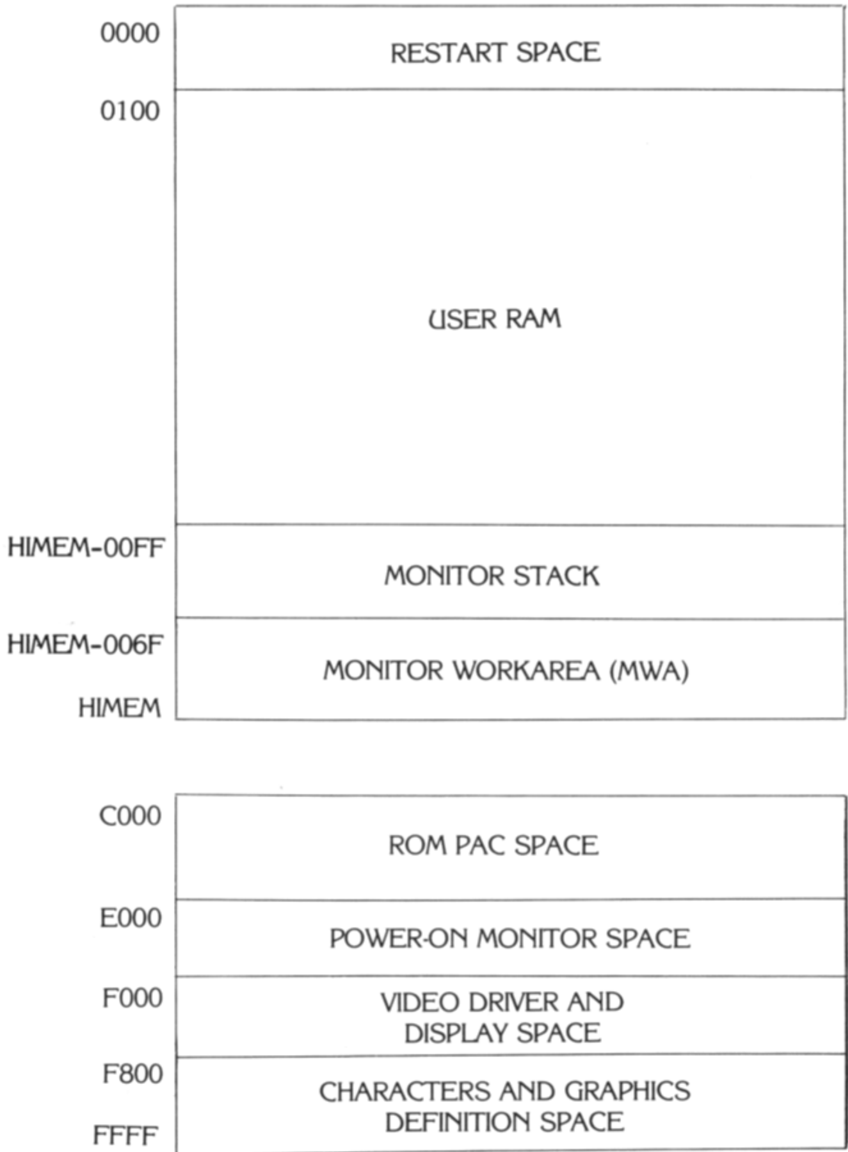


Table III-1. SORCERER MEMORY MAP DESCRIPTION

Hex	Decimal	
0000	0	256-byte Z80 Restart space (RAM)
0100	256	User RAM start, begin BASIC Control Area (RAM)
1F00	7936	Monitor Stack end (8K machines) (RAM)
3F00	16128	(16K machines)
7F00	32512	(32K machines)
1F90	8080	Monitor Stack start (8K machines) (RAM)
3F90	16272	(16K machines)
7F90	32656	(32K machines)
1F91	8081	Monitor Workarea start (8K machines) (RAM)
3F91	16273	(16K machines)
7F91	32657	(32K machines)
1FFF	8191	End User RAM (8K machines) (RAM)
3FFF	16383	(16K machines)
7FFF	32767	(32K machines)
C000	-16384	Begin 8K ROM PAC (e.g., begin BASIC) (ROM)
E000	-8192	Begin 4K Monitor Program (ROM)
F000	-4096	128-byte video driver space (RAM)
F080	-3968	1920-byte video screen (64x30) (RAM)
F800	-2048	1K standard Sorcerer ASCII alphanumerics (00-7F) (PROM)
FC00	-1024	512-byte Sorcerer keyboard standard graphics character set, accessed by depressing GRAPHICS key, character codes hex 80-BF (128-191) (RAM)
FE00	-512	512-byte User Programmable graphics character set, accessed by depressing SHIFT and GRAPHICS keys, codes hex C0-FF (192-255) (RAM)
FFFF	-1	End Sorcerer address space (64K)

The Monitor Workarea, hereafter called MWA, is used by the Sorcerer Monitor program to save important information needed for its successful operation. This area is always located adjacent to the Monitor Stack, and is always placed at the very top of available RAM space. For an 8K machine, the top of RAM is at 1FFF (8191), for 16K 3FFF (16383), and for 32K 7FFF (32767). This address is called HIMEM and its value is placed by the Monitor in the two bytes at address F000-F001 (-4096 to -4095) in the video driver RAM space. Remember that, as with most micros, the two bytes of data are reversed in storage. For example, for a 16K Sorcerer, F000-F001 contains FF3F, not 3FFF. The address of the MWA is directly related to HIMEM, so MWA values can be found by the same method regardless of what size machine is being used. To calculate the address of the MWA, get the HIMEM value at F000-F001 and subtract 6E (110) or add FF92 (-110). For example, in Z80 Assembly Language:

```
LD       HL, (0F000H)    ;GET HIMEM
LD       BC,0FF92H       ;PUT -110 IN BC
ADD      HL,BC           ;HL POINTS TO THE MWA
```

Or in BASIC,

```
100  AD = 256*PEEK(-4095)+PEEK(-4096)
110  IF AD>32767 THEN AD = AD-65536
120  AD = AD-110
```

There is also a Monitor subroutine designed to do this calculation for you. It is at address E1A2(-7774). When CALLed, it puts the MWA address in Z80 register IY.

```
CALL 0E1A2H ;IY POINTS TO THE MWA
```

Table III-2 is a detailed map of the contents of the MWA. It is presented in the same format as the overall memory map of Table III-1, except that the addresses will be shown in a different form. First the offset in hex from the beginning of the MWA will be given. This can be used in Z80 Assembly Language as a displacement away from the starting address of the MWA, where the starting address has been placed in an index register such as IY. For example, if the

displacement is listed as +41 to a particular field, then that field can be addressed in Z80 by (IY+41) or by 41(IY).

The second part of the address given in Table III-2 is an absolute address of the field in RAM. Since the MWA moves depending on the size of the machine, the first two hex digits of these addresses can change. However, the last two digits are always the same, and so only the last two digits are listed. The first two will either be 1F (8K), 3F (16K), or 7F (32K). Note: if the user resets the top of RAM, such as with a USER reentry to the Monitor, with a size that does not end in FF, then the above addressing scheme is not applicable and only the displacement method may be used.

Table III-2. SORCERER MONITOR WORKAREA (MWA) MAP (1 of 4)

ADDRESS		DESCRIPTION
Relative to Start of MWA	Last Two Hex Digits of Absolute Address	
+00	91	60-byte Monitor command input buffer. Any command entered from the current RECEIVE device (SET I=x) such as the keyboard, serial or parallel ports is placed in this area. It is left-justified, and terminated by an ASCII carriage return character (hex code 0D), 13 decimal, hereafter called a CR). The Monitor subroutine at E13A (-7878) builds this buffer from the input.
+3C	CD	Port FE interface status
+3D	CE	Serial interface and dual cassette interface baud rate save area. 1200 baud is indicated by hex 40, 300 baud by the value 00. Serial port or cassette baud rates are set to the default of 1200 baud (hex 40) by the Monitor COLD Reset routine at E000 (-8192) and by the Monitor USER Reset entry point at E006 (-8186). Such a coldstart is done, for example, when the RESET keys are depressed. This byte is also set by the SET T=0 and SET T=1 commands (at Monitor routines at E5A2, -6750).

Table III-2. SORCERER MONITOR WORKAREA (MWA) MAP (2 of 4)

ADDRESS		DESCRIPTION
Relative to Start of MWA	Last Two Hex Digits of Absolute Address	
+3E	CF	SEND delay time. This value is used to delay before a SEND (to video, serial, or parallel) is done. The actual delay is about 1500 times this value in machine cycles. This delay can therefore range from 0 to approximately 400000 cycles. The value is set by the SET S=n command.
+3F	D0	Current SEND routine address. The default address set by COLD starts is the video routine at E9F0 (-5648). It can be changed by the SET O=x command
+41	D2	Current RECEIVE routine address. The default is set by COLD starts to be the keyboard routine at EB1C (-5348). It can be changed by the SET I=x command.
+43	D4	Batch mode status. 00=normal input, nonzero=batch mode. This byte is used by the Monitor command input routine at E13A (-7878) to determine whether commands are to be fetched from the RECEIVE device or from the batch tape serial port. The OVER command turns this off and the BATCH command turns this on.
+44	D5	Monitor output prompt character. The default is the character " > " or ASCII code 3E (62) set by COLD starts. It can be changed by the PROMPT x command. It is output to the SEND device every time a Monitor input command is being requested at E0ED (-7955).
+45	D6	Tape status, baud rate, motor control save area. This is zeroed when the tape(s) is turned off, and otherwise remembers the status of the tape baud rates (00=300, 40=1200) and motor controls (10=motor #1 on, 20=motor #2 on).

Table III-2. SORCERER MONITOR WORKAREA (MWA) MAP (3 of 4)

ADDRESS		DESCRIPTION
Relative to Start of MWA	Last Two Hex Digits of Absolute Address	
+46	D7	Tape input and output CRC (**C**yclic **R**edundancy **C**heck). The CRC is used to check whether the data has been transmitted successfully to/from the tape. This technique is is described in detail in Chapter 4.
+47	D8	Beginning of the 16-byte tape output file header area. The first 5 bytes here contain the 5-character ASCII file name as entered on the SAVE or CSAVE command. It is left justified and padded to the right with ASCII blanks (code 20 hex, 32 decimal).
+4C	DD	File header identification byte, usually hex 55.
+4D	DE	File type. Usually C2 (194) for a BASIC save file. If the high order bit (80, 128 decimal) is on, the file cannot be automatically executed with the LOADG command. This is set by the SET F=xx command.
+4E	DF	2-byte length of the file in bytes.
+50	E1	2-byte program loading address. For BASIC files, this is always 01D5 (469) because BASIC programs always start at that address. See the BASIC Control Area description in Chapter 5. For other programs such as those in machine language, this address is the "ssss" of the command "SAVE name ssss eeee".
+52	E3	2-byte program "go-address" for auto execution files. The Monitor will automatically begin execution of the program at this address with the LOADG command. This address is set by the SET X=nnnn command.
+54	E5	3 bytes of reserved space, ending the output tape header.

Table III-2. SORCERER MONITOR WORKAREA (MWA) MAP (4 of 4)

ADDRESS: Relative to Start of MWA	ADDRESS: Last Two Hex Digits of Absolute Address	DESCRIPTION
+57	E8	16-byte tape input header area. The format is identical to that of the area at +47. This area is filled in from reading the tape for commands such as CLOAD, LOAD, FILES, and so on.
+67	F8	Character under the cursor. Since the cursor is an underscore character (ASCII code 5F, 95 decimal), it actually replaces the character at the cursor location. This hidden character is saved to be put back when the cursor is moved. The save is done by E9CC (-5684), and it is replaced by E9E8 (-5656).
+68	F9	2-byte line number where the cursor is times 64. This ranges from 0x64 (0) to 29x64 (1856), and is the offset from the beginning of the screen to the cursor line start.
+6A	FB	2-byte cursor column number (0-63). When added to +68 the actual cursor offset into the screen is found.
+6C	FD	Last character entered from the keyboard. This is used for the processing of the REPT (repeat) key logic. This character is entered to the keyboard input routine about every 30000 machine cycles as long as the REPT key is depressed. It is always the last key entered, and is saved and used by the keyboard processing routine at EB1C (-5348).
+6D	FE	Two bytes of reserved space. This brings us to the end of the MWA, and in fact the end of the user RAM.

MONITOR SUBROUTINES

The Sorcerer ROM Monitor is packed with well-written and useful subroutines which can be called from BASIC and assembly language. All are resident in the 4K ROM between locations E000 and EFFF. Table III-3 is a brief description of all the useful routines, and how to interface to them.

The subroutine addresses will be given in hex, of course, but will also be given as a pair of decimal integers in the order necessary to POKE into the BASIC USR jump vector at locations 260-261.

An example of how this can be done in BASIC is as follows:

```
100 POKE 260, 177 : REM HEX B1
200 POKE 261,233 : REM HEX E9
300 X=USR(X) : REM CALL HEX E9B1
```

This routine clears the screen and returns the cursor to home (the upper left hand corner).

TABLE III-3. SORCERER ROM MONITOR SUBROUTINES

ADDRESS Hex	Decimal Poke Values	DESCRIPTION
E000	0,224	Monitor Cold Start (on RESET)
E003	3,224	Monitor Warm Start (on BYE command)
E006	6,224	Monitor User Cold Start - similar to E000 except HL is is input containing what the user wants to use as HIMEM
E009	9,224	RECEIVE: returns NZ and a character from the current RECEIVE device in the accumulator (A), or Z if no character yet
E00C	12,224	SEND: sends character in A to the current SEND device
E00F	15,224	SERIAL IN: reads a character into A from the serial input device or from cassette tape
E012	18,224	SERIAL OUT: writes character from A to serial/tape
E015	21,224	QCKCHK: returns NZ if Control-C or ESC (RUN/STOP) is depressed, otherwise it returns Z
E018	24,224	KEYBOARD: the RECEIVE routine if SET I=K (default). See E009.
E01B	27,224	VIDEO: the SEND routine if SET O=V (default). See E00C.
E01E	30,224	PARALLEL IN: the RECEIVE routine if SET I=P.
E021	33,224	PARALLEL OUT: the SEND routine if SET O=P.
E024	36,224	CASSETTE MOTOR CONTROL ON: will turn motor on and set the baud rate of the requested cassette. MWA + 3D must contain the baud rate (00=300, 40=1200) and reg B must contain the cassette number (1 or 2).
E027	39,224	CASSETTE OFF: turns off both tapes

E02A *→ E65A*	42,224	TAPE SAVE: Save memory onto tape. MWA+50, MWA+51 must contain the memory address where SAVEing is to start. It must also be pushed on the stack. DE must contain the ending address. HL must point to a byte containing a CR (hex 0D). MWA +47 through MWA +4B must contain the ASCII file name; MWA +4D must contain the file type; MWA +52, MWA +53 the GO address if any.
E02D *→ E799*	45,224	TAPE LOAD: load a file into memory from tape. MWA+47 through MWA+4B must contain the file name to load. If a LOADG is to be done, a Z flag must be on the stack, otherwise an NZ flag. Then if the program name is specified, put NZ in the flags, otherwise Z (i.e., load the next file on the tape).
E13A *note: will not continue with CR entered at keyboard.*	58,225	MONITOR INPUT: will put the command in the command input buffer at MWA+0. IY must point to the MWA. MWA+43 must contain 0 (not Batch).
E1A2	162,225	Will find MWA and put the address in IY without causing screen flicker (only does so during vertical retrace on the TV to avoid Direct Memory Access conflicts — see Chapter 6).
E1BA	186,225	SENDLINE: sends an entire line to the SEND device. HL points to the line, which must end in a 00. LF's are always sent when CR's are found.
E1C9	201,225	ERROR: sends "ERROR" followed by the diagnostic message (which is pointed to by HL).
E1D4	212,225	OVER Command Processor (CP). Handles all work necessary for the OVER command.
E1E8	232,225	Sends 4-byte ASCII equivalent of the 2-byte integer in DE. If DE=3F29, then "3F29" is sent.
E1ED	237,225	Send 2-byte ASCII of byte in A.
E205	5,226	Send a CR followed by a LF, CRLF
E23D	61,226	Convert a 1-4 byte ASCII hex number (pointed to by HL) into DE. If HL points to A93 followed by a "Monitor Delimiter" (e.g., blank, CR, etc.), then DE will contain 0A93. This is the reverse process of the routine at E1E8.

Table III-3. SORCERER ROM MONITOR SUBROUTINES, Continued

ADDRESS		DESCRIPTION
Hex	Decimal Poke Values	
E2D2	210,226	Send as many blanks as the number in B.
E4D3	211,228	DUMP CP.
E538	56,229	ENTER CP
E562	98,229	MOVE CP
E597	151,229	GO CP
E5A2	162,229	SET CP
E638	56,230	SAVE CP
E6B9	185,230	FILES CP
E78A	138,231	LOAD CP
E845	69,232	PROMPT CP
E858	88,232	BATCH CP
E85C	92,232	CREATE CP
E884	132,232	LIST CP
E8A1	161,232	TEST CP
E98A	138,233	PP CP
E993	147,233	CENTRONICS OUT: the SEND routine for SET O=L.
E9B1	177,233	Clear the video screen and refresh/rewrite the graphics character set at FC00.
E9CC	204,233	Move the cursor to line/column specified in the MWA. See cursor positioning discussion in Chapter 6.
E9D6	214,233	Find the cursor. HL is set to the screen address (which starts at F080) and DE is set to the column number.
EB10	16,235	Refresh character set at FC00
EC1E	30,236	Keyboard input tables (to EDFD). See Chapter 7 on the keyboard.
EDFE	254,237	Character set for the 64 standard graphics 80-BF to be copied to FC00.

CHAPTER FOUR

Sorcerer Cassette Interface

CASSETTE TAPE FILE FORMAT

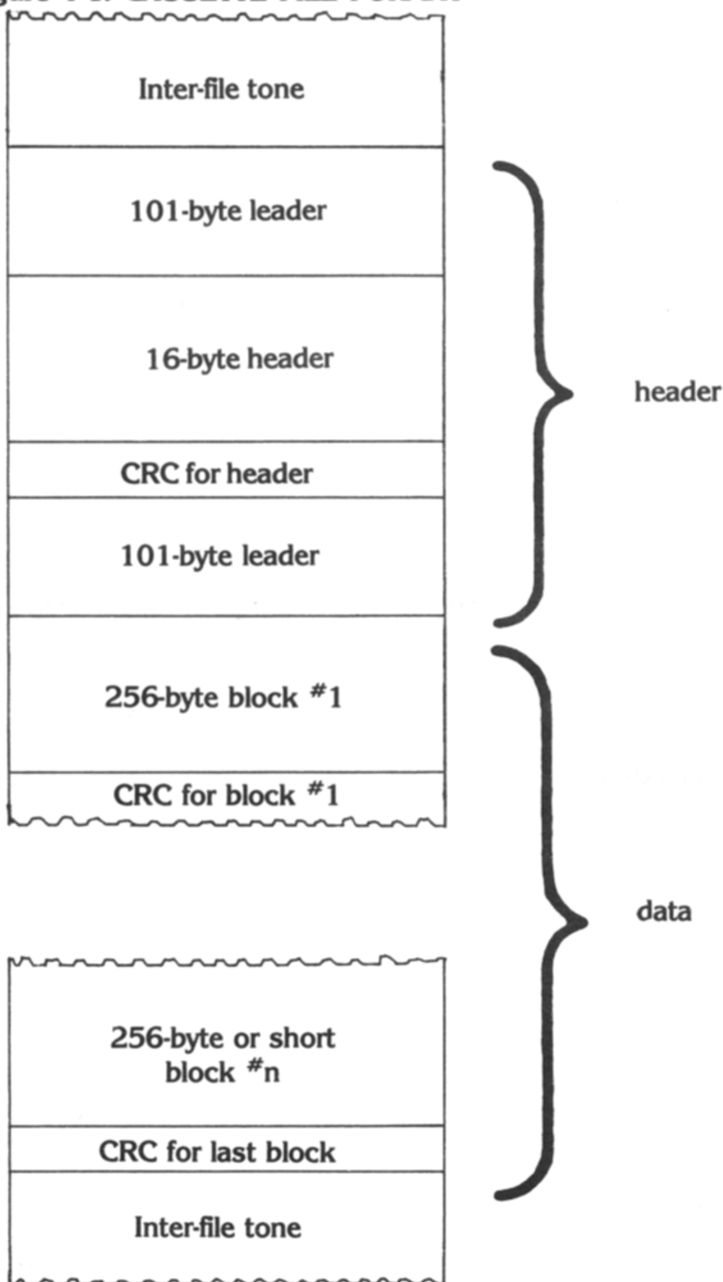
When a SAVE, LOAD, or FILES command is done from the Monitor, or when a CSAVE or CLOAD is done from BASIC, files are processed from the cassette tape device on the serial interface. This applies to both cassette #1 and #2. The cassette tape motor-on routine can be found at E024 (-8156), motor-off at E027 (-8153), cassette save at E02A (-8151), and cassette load at E02D (-8148).

Cassette files on the Sorcerer can be written at either 300 or 1200 baud. The cassette file format is the same for both BASIC and machine language tapes at either baud rate and is depicted in Figure 4-1. A full description of this format follows:

1. Inter-file tone
   a. a high frequency tone always output by the cassette interface when data is not present.

2. 101-byte leader
   a. 100 bytes of 00 (nulls)
   b. 1 byte of 01 (control-A or SOH, Start-Of-Header)
3. 16-byte file header (described in Table III-2)
4. CRC for header
   a. 1-byte Cyclic Redundancy Check (CRC) for error checking. The CRC byte is described in a later section of this chapter.

Figure 4-1. CASSETTE FILE FORMAT



5. Up to 256 bytes of data
6. CRC for above data block (1 byte again)
7. Repeat 5 and 6 until data exhausted. The last data block may be short (less than 256 bytes). CRC still follows.
8. Inter-file tone (same as before the file).

To LOAD or CLOAD a file, or to perform a FILES command, the Monitor scans the tape (whichever is on) for the leader. Then the header is read into the MWA and the "FOUND . . ." message is sent to the current SEND device. The data portion is then either skipped (wrong file, or FILES command) or loaded. All CRC's are always validity checked for any of these commands. Thus to check all the bits on an entire tape for errors, it is sufficient to perform a FILES command.

Note that the default tape transfer rate is 1200 baud. A much more reliable method of saving data is to use 300 baud. However it will take 4 times longer to SAVE and LOAD, and use a lot more tape. This is accomplished with the SET T=1 command.

TIPS ON LOADING AND SAVING FILES ON TAPE

The following hints can be used to minimize problems with cassette recording of files:

To Load:

1. Use a relatively inexpensive cassette recorder ($30-$60) with ALC (Automatic Level Control). This means you have no control over the volume or tone of the recordings. All are made exactly the same way. Strangely enough, experience shows that expensive recorders work worse.
2. Connect the MIC wire to the microphone input. Do **not** use the auxiliary input on most recorders. The signal will be too weak.
3. Connect the EAR wire to the earphone or monitor jack.

To Play:

1. You must find the correct volume and tone for your recorder. As a first guess, set volume and tone to 7-8 out of 10, or 3/4 high.

2. Listen to the tape play through the speaker. The interfile tone should be louder than normal listening volume, maybe even as loud as possible without distortion and noise. The data should sound high-pitched and clear, like static.
3. Try loading a file. Tinker with volume and tone until at least a file header is read without a CRC error ("FOUND . . ." message appears). Now you are close enough to the correct settings.
4. Once found, the correct settings should be able to be used for all tapes recorded on that recorder.

CASSETTE TAPE ERROR CHECKING

The CRC (Cyclic Redundancy Check) method is used to detect bit transmission errors in cassette data recording and reading. The CRC is stored in the Monitor Workarea (see Chapter 3) at MWA+46. CRC checking is done as follows:

1. Before the file is first written to tape (i.e. when the 101-byte leader is written), the CRC byte is set to zero.

2. For every data byte, in program or header, the current CRC byte is subtracted from the data byte. The result is complemented (all 1's changed to 0's and all 0's changed to 1's) and stored as the new CRC byte.

3. After the data in the file or block is completely written, the current CRC is then written. Note: this procedure causes BASIC programs to grow by one byte each time they are saved on tape.

4. When the file is loaded, the CRC is again calculated per steps 1 and 2 and then compared to the CRC byte associated with the block being read in (the previously written CRC byte). A match indicates that there are no errors, while a mismatch means an error.

5. This procedure is the same for BASIC files as it is for machine language files, because the same Monitor routines are used to write/read tapes.

CHAPTER FIVE

Sorcerer BASIC Internals

BASIC FLOATING POINT FORMAT

Numbers stored in Sorcerer BASIC are not integers but are hexadecimal numbers in which the decimal point can move or "float". For example, the decimal point floats when 13.25 is divided by 10, resulting in 1.325. It is from this idea that the term "floating point" was derived.

Sorcerer BASIC stores floating point numbers in four bytes of memory. Each number has 3 parts:

1. the sign (+ or -)
2. the "mantissa" (the actual number, but with the point shifted to the left of the leftmost 1 bit of the number). So the number 127 decimal (7F, 01111111) is a mantissa if it is thought of as .1111111
3. the "exponent", which is how much the point had to be shifted in the number to produce the mantissa with the point at the left

This may sound very complex, but it actually is not. Let's take an example, say 13.5 decimal. In hex this would be equal to D.8 (13+8* 1/16). Remembering that hex is just groups of four bits, the binary equivalent of 13.5 would be 1101.1000. To create a mantissa from this, we must shift the point (in this case, the "binary point", not the decimal point) to the left four places, producing .11011000. The exponent can now be calculated. It is always a **positive** if the mantissa shift was to the **left, negative** if to the **right**, and **zero** if **no shift**

was necessary. Thus the exponent in this example would be +4 (4 to the left). However, we are not quite done. Rather than worrying about how to express a negative number exponent, 80 (128 decimal) is always added to the exponent to produce the final result. Thus the final exponent is 84 (132). Now we come to the sign. Since the digit to the far left in the mantissa is always a 1 (because we shifted until that was the case), then the sign can be stored in this bit without losing any information. If the number is positive or zero, then the sign bit will be 0. If negative, then the sign bit will be a 1. So the mantissa for 13.8 which is .11011000 in floating point binary changes to .01011000. To assemble this number, first we put the exponent 84 then the mantissa filled out to the right to fill out the 4 bytes:

10000100 .01011000 00000000 00000000

Now if we ignore the point, since it is always in the same place, and convert to hex, we have:

If the original number were - 13.5 instead, then nothing would change except the sign. That is, the mantissa would change from .01011000 to .11011000, so the new number would be

84D80000

In the reverse direction, to convert floating point back to decimal, let's use 88FF4000 as an example:

1. Examine the exponent (88) and subtract hex 80 (128). In this example 88-80=08. But this may produce a negative number.

2. Examine the mantissa with the implied point (.FF4000).

3. If the left bit (high order, the one next to the point) is on (it is), then the number is negative, otherwise it is positive.

4. In either case, turn that bit on.

5. Shift the point according to the exponent from step 1 (08 here). If plus, shift right, if minus, left, if zero, no shift. Since we have +8, shift the point right 8 **bits.**

.11111111,0100000000000000

6. The number is now FF.4000, and with the sign, -FF.4000, or -255.25 decimal.

The only special case is the number 0. Here the exponent is 00. Some examples are:

1815	=	hex 717	=	8B62E000	(floating point
1		1	=	81000000	hexadecimal)
-1		-1	=	81800000	
-.5		-.8	=	80800000	
0		0	=	00610000	(the mantissa is ignored)

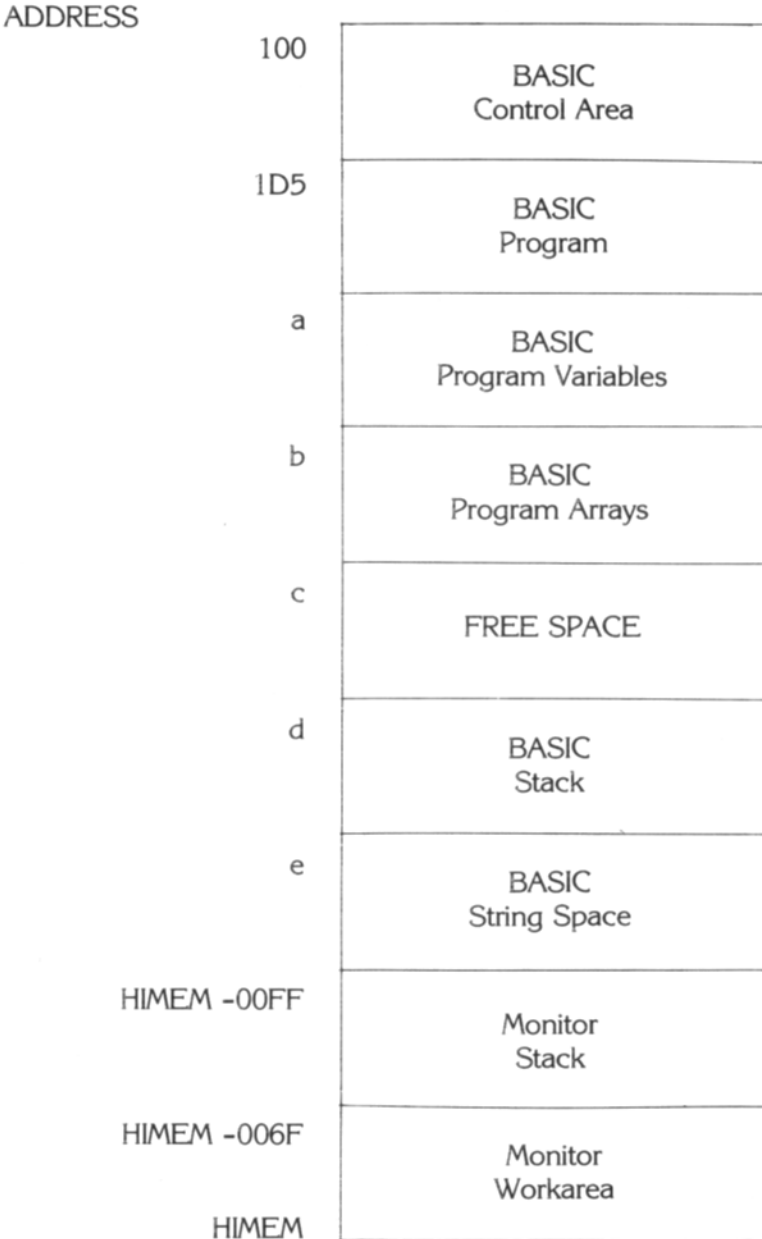
The last idea that must be mentioned is that the bytes representing the number are actually stored in memory in **reverse** order, so that the number eemmnnpp is stored ppnnmmee. For example, decimal 1815 in the above example is stored as:

00 E0 62 8B

BASIC CONTROL AREA

This is a discussion of the area in RAM called the BASIC Control Area, or BCA. This area is used by BASIC to store information necessary to run or save a BASIC program. The BCA begins at address 100 (256), as shown in Figure 5-1.

Figure 5-1. FREE RAM AS USED BY BASIC



In detail RAM locations 100-14E (256-334) are copied from the BASIC ROM (address C258) when a BASIC Cold Start occurs (i.e., after RESET or a PP X command is entered). The BCA is described in Table V-1. This description includes only those areas which are of direct use to the programmer. It is intentionally sketchy, especially due to the great number of fields.

FORMAT OF BASIC PROGRAM STATEMENTS

The first line of every BASIC program begins at location 1D5. All BASIC lines have the following variable length format:

OFFSET	DESCRIPTION
+0	2-byte link pointer address of the next sequential full line in the program. This is independent of multiple statements on one line separated by colons). The last line of the program points to to location 0000 to indicate the end.
+2	2-byte BASIC line number of the line in integer binary (a number between 0000 and FFF9, 0-65529).
+4	The BASIC statement(s), variable in length. Let us say they are "n" bytes long. Each BASIC "reserved word" such as GOTO, IF, END, DIM, PRINT, etc is encoded here to a 1-byte character not belonging to the ASCII character set (ie, hex codes greater than 7F). This speeds up processing and saves program memory space. When the program is LISTed, these special bytes are decoded back into their corresponding reserved words.
+4+n	Byte of 00 indicating the end of this line and beginning of the next.

FORMAT OF BASIC FLOATING POINT VARIABLES AND ARRAYS

A BASIC floating point variable resides in the BASIC Program Variable Area (see Figure 5-1). Each one takes a constant six bytes:

OFFSET	DESCRIPTION
+0	2-byte variable name (two characters). The high order bit is always 0. The letters are also reversed as usual.
+2	4-byte floating point value currently held by this variable. See the format description at the beginning of this chapter.

Table V-1. BASIC CONTROL AREA MEMORY MAP

ADDRESS hex/dec	DESCRIPTION
100/256	3-byte JUMP instruction to C06B (Warm Start). Done when PP command is entered without operands.
103/259	3-byte JUMP to C7E5 default (displays "FC ERROR" message). This is the USR function hook. See BASIC Assembly interface section for details.
145/325	2-byte address of top of string space (letter "e" in Figure 5-1) or the beginning of the BASIC stack. This is set by the BASIC CLEAR n command.
147/327	BASIC line input buffer and Direct Mode execution line.
18E/398	Column number that the cursor is currently in.
1B1/433	2-byte address of instruction in the BASIC program about to be executed when Control-C break is entered. This could be in the middle of a line of multiple statements separated by colons.
1B3/435	2-byte BASIC line number of current line
1B5/437	2-byte address of the next **full** line to execute from the link pointer of the current line (see below).
1B7/439	2-byte address of the end of the program and the beginning of the BASIC Program Variable Area (letter "a" in Figure 5-1).
1B9/441	2-byte address of the end of the Variable Area and the start of the BASIC Program Array Area (letter "b" in Figure 5-1). Whenever changes are made to the BASIC program (adding, deleting, updating lines) the above two addresses are used to define a new Variable and Array area below the new BASIC program. Thus a program cannot be continued with old variable/array values once a change has been made.
1BB/443	2-byte address of the end of the Array Area and the pointer to the top of free space (room for expansion — letter "c" in Figure 5-1).
1BD/445	2-byte address of the last used data operand of a DATA statement so that the next READ will find the appropriate item. This is reset by a RESTORE command.

ADDRESS hex/dec	DESCRIPTION
1BF/447	4-byte input parameter (usually floating point format) to the USR function, and output parameter from the USR function. If USR(3.5) is called, 3.5 is passed to the subroutine in floating point. See a later section for BASIC/Assembly interfacing details.
1D5/469	Beginning of all BASIC programs

BASIC arrays all reside together in the BASIC Program Array Area (see Figure 5-1). A floating point array is variable in length. An array in BASIC can have up to 255 dimensions; call that number "n". Each can have any number of elements. Each array takes a minimum of seven bytes and looks like this:

OFFSET	DESCRIPTION
+0	2-byte array name. The high order bit is always 0. The letters are reversed.
+2	2-byte total array length minus 4 (ie, the length of the array starting after these 2 bytes). This is used to find the next array in the area quickly.
+4	1-byte number of dimensions (we called it n).
+5	2-byte size (number of elements) in the 1st dimension.
+7	2-byte size of the 2nd dimension (if any).
.	.
.	.
.	.
+5+2(n-1)	2-byte size of the nth dimension
+5+2n	Beginning of a list of contiguous 4-byte floating point array elements. These are in row order.

FORMAT OF BASIC STRING VARIABLES AND ARRAYS

A BASIC string variable is similar to a floating point variable. It is also 6 bytes long. It looks like:

OFFSET	DESCRIPTION
+0	2-byte variable name. The high order bit is always 1.
+2	1-byte current length of the variable length string value.
+3	00
+4	2-byte address of the string itself. It resides either in the string space or in the program statement itself (eg, 1005 A$="HI").

A string array is identical to a numeric array except for two very important features:

1. **the high order bit of the array name is always 1**

2. **the 4-byte value is not floating point format but the length/00/stringaddress fields described above. All dimensioning remains the same.**

BASIC TO Z80 ASSEMBLY LANGUAGE INTERFACE

To call Z80 Assembly Language subroutines from Sorcerer BASIC, certain general conventions and procedures must be followed:

1. The machine language program must reside either in the first 256 bytes of memory (00-FF, 0-255 — usually a bad idea) or in the BASIC free space area as shown in Figure 5-1. Either BASIC control, program, variables, arrays or strings, or Monitor/video control resides in the rest of memory. This is the only way a BASIC and machine language hybrid can coexist without complicated machinations such as putting the machine language routine right after the BASIC program and fooling BASIC into thinking that it is part of the program. The BASIC free space is the best and easiest choice. However there are some potential problems:

    a. Free space is dynamic. As the program changes, as variables/arrays are added or change size, the start of the free space moves. A machine language program placed too close to the end of the Program Array Area can get walked on when the program is enlarged. The end of the free space changes too, since the BASIC stack (and/or string space) will grow and shrink, especially with the CLEAR command. Since this change is usually not as radical as that of the start of the free space, I recommend putting the program close to the **end** of the free space. But there are now other considerations.

    b. The free space ends near HIMEM of the machine (where the BASIC stack is). This changes with each different Sorcerer size. So a generalized machine language subroutine designed to run on any size Sorcerer (probably with several BASIC

programs) would either have to be relocatable (able to be moved without affecting anything), or there will have to be different versions of the program to run on different size machines. Either of these approaches allows the BASIC program to use the maximum amount of free space. A subroutine designed for a **particular** BASIC program could be placed at the top of the free space as long as the BASIC program does not grow too much.

c. If the program is placed at the end of the free space an excessive CLEAR n BASIC statement could kill it.

d. Thus no matter where the machine language program is placed, certain precautions are necessary in order to coexist with BASIC.

2. Assume a good location is found, and the Z80 program is written and placed at that RAM location. Assume the start address to be 312A hex (12586). To call this subroutine from BASIC, it must already be in memory, and the USR function must be used. When BASIC executes it, it converts the argument in the USR function to floating point and places this number in the 4-byte USR parameter area at 1BF-1C2 (447-450). It then calls the subroutine at location 103 (259). For example, when the statement

```
2030 X=USR(25.7)
```

is executed, 25.7 is placed at 1BF and a CALL is made to 103.

3. Now, by default 103 contains the following Z80 instruction

```
JP      0C7E5H
```

or in machine language — hex C3E5C7. This is an unconditional JUMP to the instruction at address C7E5 in BASIC ROM. This default subroutine prints the error message "FC ERROR" (function call invalid) an stops the program. To call **your** subroutine, you must change the JUMP instruction address to the address of the beginning of your program. Again the instruction after a BASIC Cold Start looks like

ADDRESS	CONTENTS	DESCRIPTION
103/259	C3	JUMP Z80 operation code
104/260	E5	Low part of address
105/261	C7	High part of address

Leave the C3 JUMP, but change the address. If your program was at 312A as in our example, you must make the jump to 312A, or

```
JP          312AH
```

or in machine language — hex C32A31. It is a good idea to change the two address bytes every time the subroutine is to be called. Use the BASIC POKE statement for this (which requires **decimal** operands). Put 2A (42) at location 104 (260), and put 31 (49) at location 105 (261):

```
10000   POKE   260,42
10010   POKE   261,49
10020   XX  =  USR(Y)
```

When the USR function is executed in line 10020, your routine at 312A will be called. It could use the value in variable Y placed at 1BF as input. It could also put another value back as output. This value will be returned to the BASIC statement as the "result" of the USR function. In the above example, the value returned will be placed in variable XX. Note that the short BASIC routine shown above can easily be made into a GOSUB subroutine by adding the statement

```
10030 RETURN
```

Thus, to call your routine you need only say

```
GOSUB 10000
```

4. To terminate your subroutine, one of four things can be done:

   a. Return directly to the Monitor and exit BASIC altogether, e.g. for catastrophic errors. For Monitor Warm Start jump to address E003. For Cold Start use E000. The user will be shown the Monitor prompt (" > ").

   b. For lesser errors detected give an FC ERROR MESSAGE, stop the program, and return to BASIC READY level. This is simply done by jumping to C7E5.

   c. If errors are detected and your routines have displayed the error message(s), you can stop the program and exit directly to BASIC READY level. For a BASIC Warm Start jump to DFFA, for a Cold Start DFFD.

   d. Of course you can return normally to BASIC so it will continue the program where it left off after the USR statement. This is simply done by the RET instruction. Put a floating point number at 1BF first if necessary.

Note that all the Monitor subroutines are available to the Z80 subroutine, including turning the tape on, reading a file, and turning it off; or getting input from the keyboard. See Chapter 3 for a description of available Monitor subroutines.

Debugging of the Z80 routine is a little more difficult than debugging BASIC programs. BASIC loses control of the situation and of what you are doing while your routine is running, and can't "keep an eye out" for potential errors as it can within a BASIC program. Great care, desk checking, and modular programming are necessary.

An assembly language routine can also use as input and output actual BASIC variables and arrays. Using the pointers in the BCA described earlier, the program can find the variable/array lists and scan for the one(s) with the correct name(s). Then using the floating point or string formats, the values can be examined or changed.

CHAPTER SIX

Sorcerer Video and Graphics

CHARACTER CODES

To understand the Sorcerer's video interface, the programmer must be familiar with what a character code is. This code indicates to the video driver which character to display on the screen. Conveniently, character codes consist of exactly one byte, or eight bits. Thus there are 2^8 or 256 possible character codes, numbered 0 to 255 (00-FF in hex). Table VI-1 on the following pages lists for each character code the key press related to the code and the character that the code represents.

Explanation of Table VI-1

Column 1 lists the decimal code for each character, which is used in the BASIC function CHR$ and when POKEing characters.

Column 2 lists the hexadecimal code for each character, for use by machine language programmers.

Column 3 indicates the keypress that corresponds to the respective character code. There are three keys on the Sorcerer keyboard that can be depressed simultaneously with the other keys to modify the code transmitted. They are SHIFT (**S**), CTRL (**C**), and GRAPHICS (**G**). The SHIFT LOCK key also affects the keyboard selection and will be discussed in Chapter 7. The letters **S**, **C**, and **G** in column 3 of Table VI-1 indicate that the respective key or keys must be held down to transmit the desired code. The letters NP in this column stand for Numeric key Pad.

Column 4 depicts the character displayed on the video screen when the relevant code is sent to the video driver. In the case of the user-programmable characters (codes 128 to 255, part 4 of Table VI-1), this column gives the memory address at which the character definition is stored.

Column 5 gives the BASIC reserved word, if any, that each code represents. This is the shorthand notation used to condense BASIC program statements (see Chapter 5).

Table VI-1. CHARACTER CODES AND KEYPRESSES

Part 1: CONTROL AND NUMERIC CHARACTERS

CODE (DECIMAL)	CODE (HEX)	KEY PRESS	CHARACTER	BASIC RSVD WORD
0	00	**C**-@	□	
1	01	**C**-A	┌	
2	02	**C**-B	⊥	
3	03	**C**-C	┘	
4	04	**C**-D	ϟ	
5	05	**C**-E	⌧	
6	06	**C**-F	√	
7	07	**C**-G	○	
8	08	**C**-H	↖	
9	09	**C**-I	⊳	
10	0A	**C**-J	≡	
11	0B	**C**-K	◆	
12	0C	**C**-L	◆	
13	**0D**	**C**-M	⊲	
14	0E	**C**-N	⊗	
15	0F	**C**-O	⊙	
16	10	**C**-P	⊟	
17	11	**C**-Q	◷	
18	12	**C**-R	◶	
19	13	**C**-S	◵	
20	14	**C**-T	◴	
21	15	**C**-U	√	
22	16	**C**-V	⎍	
23	17	**C**-W	┤	
24	18	**C**-X	⧗	
25	19	**C**-Y	+	
26	1A	**C**-Z	ʕ	
27	1B	**C**-[	θ	
28	1C	**C**-\	◰	
29	1D	**C**-]	◱	
30	1E	**C**-^	◲	
31	1F	**C**-underscore	◳	

CODE (DECIMAL)	CODE (HEX)	KEY PRESS	CHARACTER	BASIC RSVD WORD
32	20	space bar	blank	
33	21	**S**-1	!	
34	22	**S**-2	"	"
35	23	**S**-3	#	
36	24	**S**-4	$	$
37	25	**S**-5	%	
38	26	**S**-6	&	
39	27	**S**-7	'	
40	28	**S**-8	(	(
41	29	**S**-9	)	)
42	2A	**S**-:	*	
43	2B	**S**-;	+	
44	2C	comma	,	,
45	2D	hyphen	-	
46	2E	period	.	.
47	2F	/	/	
48	30	0	0	
49	31	1	1	
50	32	2	2	
51	33	3	3	
52	34	4	4	
53	35	5	5	
54	36	6	6	
55	37	7	7	
56	38	8	8	
57	39	9	9	
58	3A	:	:	:
59	3B	;	;	;
60	3C	**S**-comma	<	
61	3D	**S**-hyphen	=	
62	3E	**S**-period	>	
63	3F	**S**-/	?	

Table VI-1. CHARACTER CODES AND KEYPRESSES

Part 2: ALPHABETIC CHARACTERS

CODE (DECIMAL)	CODE (HEX)	KEY PRESS	CHARACTER	BASIC RSVD WORD
64	40	@	@	
65	41	**S**-A	A	
66	42	**S**-B	B	
67	43	**S**-C	C	
68	44	**S**-D	D	
69	45	**S**-E	E	
70	46	**S**-F	F	
71	47	**S**-G	G	
72	48	**S**-H	H	
73	49	**S**-I	I	
74	4A	**S**-J	J	
75	4B	**S**-K	K	
76	4C	**S**-L	L	
77	4D	**S**-M	M	
78	4E	**S**-N	N	
79	4F	**S**-O	O	
80	50	**S**-P	P	
81	51	**S**-Q	Q	
82	52	**S**-R	R	
83	53	**S**-S	S	
84	54	**S**-T	T	
85	55	**S**-U	U	
86	56	**S**-V	V	
87	57	**S**-W	W	
88	58	**S**-X	X	
89	59	**S**-Y	Y	
90	5A	**S**-Z	Z	
91	5B	[	[	
92	5C	\	\	
93	5D	]	]	
94	5E	^	^	
95	5F	underscore	_	

CODE (DECIMAL)	CODE (HEX)	KEY PRESS	CHARACTER	BASIC RSVD WORD
96	60	**S**-@	`	
97	61	A	a	
98	62	B	b	
99	63	C	c	
100	64	D	d	
101	65	E	e	
102	66	F	f	
103	67	G	g	
104	68	H	h	
105	69	I	i	
106	6A	J	j	
107	6B	K	k	
108	6C	L	l	
109	6D	M	m	
110	6E	N	n	
111	6F	O	o	
112	70	P	p	
113	71	Q	q	
114	72	R	r	
115	73	S	s	
116	74	T	t	
117	75	U	u	
118	76	V	v	
119	77	W	w	
120	78	X	x	
121	79	Y	y	
122	7A	Z	z	
123	7B	**S**-[	{	
124	7C	**S**-\	¦	
125	7D	**S**-]	}	
126	7E	**S**-V	~	
127	7F	**S**-underscore	▒	

Table VI-1. CHARACTER CODES AND KEYPRESSES

Part 3: SORCERER-DEFINED GRAPHIC CHARACTERS

CODE (DECIMAL)	CODE (HEX)	KEY PRESS	CHAR.	CHAR. ADDRESS	BASIC RSVD WORD
128	80	**G**-1	|	FC00	END
129	81	**G**-2	|	FC08	FOR
130	82	**G**-3	|	FC10	NEXT
131	83	**G**-4	|	FC18	DATA
132	84	**G**-5	●	FC20	BYE
133	85	**G**-6	|	FC28	INPUT
134	86	**G**-7	|	FC30	DIM
135	87	**G**-8	|	FC38	READ
136	88	**G**-9	∘	FC40	LET
137	89	**G**-0	-	FC48	GOTO
138	8A	**G**-:	-	FC50	RUN
139	8B	**G**-hyphen	-	FC58	IF
140	8C	**G**-∧	—	FC60	RESTORE
141	8D	**G**-tab	π	FC68	GOSUB
142	8E	**G**-Q	X	FC70	RETURN
143	8F	**G**-W	╭	FC78	REM
144	90	**G**-E	╮	FC80	STOP
145	91	**G**-R	┌	FC88	OUT
146	92	**G**-T	┐	FC90	ON
147	93	**G**-Y	◤	FC98	NULL
148	94	**G**-U	◥	FCA0	WAIT
149	95	**G**-I	■	FCA8	DEF
150	96	**G**-O	■	FCB0	POKE
151	97	**G**-P	‾	FCB8	PRINT
152	98	**G**-[	♠	FCC0	CONT
153	99	**G**-]	♥	FCC8	LIST
154	9A	**G**-A	♦	FCD0	CLEAR
155	9B	**G**-S	♣	FCD8	CLOAD
156	9C	**G**-D	└	FCE0	CSAVE
157	9D	**G**-F	┘	FCE8	NEW
158	9D	**G**-G	◣	FCF0	TAB(
159	9F	**G**-H	◢	FCF8	TO

CODE (DECIMAL)	CODE (HEX)	KEY PRESS	CHAR.	CHAR. ADDRESS	BASIC RSVD WORD
160	A0	**G**-J	▪	FD00	FN
161	A1	**G**-K	■	FD08	SPC(
162	A2	**G**-L	│	FD10	THEN
163	A3	**G**-;	◆	FD18	NOT
164	A4	**G**-@	♣	FD20	STEP
165	A5	**G**-¦	▚	FD28	+
166	A6	**G**-underscore	▞	FD30	—
167	A7	**G**-Z	▌	FD38	*
168	A8	**G**-X	▐	FD40	/
169	A9	**G**-C	▀	FD48	^
170	AA	**G**-V	▄	FD50	AND
171	AB	**G**-B	/	FD58	OR
172	AC	**G**-N	\	FD60	>
173	AD	**G**-M	+	FD68	=
174	AE	**G**-comma	-	FD70	<
175	AF	**G**-period	-	FD78	SGN
176	B0	**G**-/	-	FD80	INT
177	B1	**G**-minus, NP	░	FD88	ABS
178	B2	**G**-7, NP	⊥	FD90	USR
179	B3	**G**-8, NP	=	FD98	FRE
180	B4	**G**-9, NP	├	FDA0	INP
181	B5	**G**- ÷, NP	░	FDA8	POS
182	B6	**G**-4, NP	│	FDB0	SQR
183	B7	**G**-6, NP	│	FDB8	RND
184	B8	**G**-X, NP	░	FDC0	LOG
185	B9	**G**-1, NP	┤	FDC8	EXP
186	BA	**G**-2, NP	═	FDD0	COS
187	BB	**G**-3, NP	┬	FDD8	SIN
188	BC	**G**- +, NP	┌	FDE0	TAN
189	BD	**G**-0, NP	┐	FDE8	ATN
190	BE	**G**-period, NP	└	FDF0	PEEK
191	BF	**G**- =, NP	┘	FDF8	LEN

Table VI-1. CHARACTER CODES AND KEYPRESSES

Part 4: USER-DEFINED GRAPHIC CHARACTERS

CODE (DECIMAL)	CODE (HEX)	KEY PRESS	CHARACTER ADDRESS	BASIC RSVD WORD
192	C0	**GS**-1	FE00	STR$
193	C1	**GS**-2	FE08	VAL
194	C2	**GS**-3	FE10	ASC
195	C3	**GS**-4	FE18	CHR$
196	C4	**GS**-5	FE20	LEFT$
197	C5	**GS**-6	FE28	RIGHT$
198	C6	**GS**-7	FE30	MID$
199	C7	**GS**-8	FE38	
200	C8	**GS**-9	FE40	
201	C9	**GS**-0	FE48	
202	CA	**GS**-:	FE50	
203	CB	**GS**-hyphen	FE58	
204	CC	**GS**-Λ	FE60	
205	CD	**GS**-tab	FE68	
206	CE	**GS**-Q	FE70	
207	CF	**GS**-W	FE78	
208	D0	**GS**-E	FE80	
209	D1	**GS**-R	FE88	
210	D2	**GS**-T	FE90	
211	D3	**GS**-Y	FE98	
212	D4	**GS**-U	FEA0	
213	D5	**GS**-I	FEA8	
214	D6	**GS**-O	FEB0	
215	D7	**GS**-P	FEB8	
216	D8	**GS**-[	FEC0	
217	D9	**GS**-]	FEC8	
218	DA	**GS**-A	FED0	
219	DB	**GS**-S	FED8	
220	DC	**GS**-D	FEE0	
221	DD	**GS**-F	FEE8	
222	DE	**GS**-G	FEF0	
223	DF	**GS**-H	FEF8	

CODE (DECIMAL)	CODE (HEX)	KEY PRESS	CHARACTER ADDRESS	BASIC RSVD WORD
224	E0	**GS**-J	FF00	
225	E1	**GS**-K	FF08	
226	E2	**GS**-L	FF10	
227	E3	**GS**-;	FF18	
228	E4	**GS**-@	FF20	
229	E5	**GS**-|	FF28	
230	E6	**GS**-underscore	FF30	
231	E7	**GS**-Z	FF38	
232	E8	**GS**-X	FF40	
233	E9	**GS**-C	FF48	
234	EA	**GS**-V	FF50	
235	EB	**GS**-B	FF58	
236	EC	**GS**-N	FF60	
237	ED	**GS**-M	FF68	
238	EE	**GS**-comma	FF70	
239	EF	**GS**-period	FF78	
240	F0	**GS**-/	FF80	
241	F1	**GS**-minus, NP	FF88	
242	F2	**GS**-7, NP	FF90	
243	F3	**GS**-8, NP	FF98	
244	F4	**GS**-9, NP	FFA0	
245	F5	**GS**- ÷, NP	FFA8	
246	F6	**GS**-4, NP	FFB0	
247	F7	**GS**-6, NP	FFB8	
248	F8	**GS**-X, NP	FFC0	
249	F9	**GS**-1, NP	FFC8	
250	FA	**GS**-2, NP	FFD0	
251	FB	**GS**-3, NP	FFD8	
252	FC	**GS**-+, NP	FFE0	
253	FD	**GS**-0, NP	FFE8	
254	FE	**GS**-period, NP	FFF0	
255	FF	**GS** - =, NP	FFF8	

BASIC RESERVED CHARACTER CODES

The input/output (I/O) routine in BASIC "intercepts" control codes (character codes less than 20 hex) and, instead of printing out a character, performs a function such as a carriage return or a cursor movement. The specific control codes are listed in Table VI-2.

Table VI-2. BASIC CONTROL CODES AND THEIR FUNCTIONS

CODE (decimal)	CODE (hex)	KEYPRESS	FUNCTION PERFORMED	EQUIVALENTS
1	01	**S**-4,NP	Cursor left	**C**-A; **C**-4,NP
3	03	**C**-C	Non-destructive carriage return and line feed, stops execution, if any	
8	08	**S**-under score	destructive back-space (rubout)	**C**-H
10	0A	LINE FEED	Line feed	**C**-J
12	0C	CLEAR	Home cursor, clear screen, set Sorcerer graphics	**C**-L
13	0D	RETURN	Carriage return, line feed, enter data	**C**-M
15	0F	**C**-O	Disables keyboard, except for RETURN, RESET, or another C-O, which enables keyboard	
17	11	**S**-5,NP	Cursor home	**C**-Q; **C**-5, NP
19	13	**S**-6,NP	Cursor right	**C**-S; **C**-6,NP
23	17	**S**-8,NP	Cursor up	**C**-W; **C**-8,NP
26	1A	**S**-2,NP	Cursor down	**C**-Z; **C**-2,NP
27	1B	ESC	Suspends execution, if any	RUN/STOP

THE VIDEO SCREEN

The Sorcerer Video uses direct memory access (DMA). The video driver repeatedly reads 1920 bytes of RAM to determine the 1920 characters to be displayed on the screen. The RAM area for video is located in memory at F080 to F7FF (-3968 to -2049). The first 64 bytes of this area define the first row of the screen, the second 64 the second row, and so on, such that the byte that contains the character code for the character in row R, column C, is located at address A, where

$$A = F080 + (40 * R) + C \quad \text{(hexadecimal)}$$

$$0 \leq R \leq 1D$$

$$0 \leq C \leq 3F$$

$$A = -3968 + 64 * R + C \quad \text{(decimal)}$$

$$0 \leq R \leq 29$$

$$0 \leq C \leq 63$$

CHARACTER DEFINITIONS

The first 128 characters, codes 00-07 (0-27), are not under user control. The definition of these characters is located in PROM at F800-FBFF (1K). The next 64 characters, codes 80-BF (128-191), can be programmed if desired, but they are already programmed to be standard keyboard graphics. The 64x8 (512) bytes that define these characters are located in RAM at FC00-FDFF. This RAM can be changed at any time by the programmer to redefine these characters. However, the Monitor refreshes this area from its ROM every time a RESET occurs, or whenever the video screen is cleared (e.g., when CLEAR is pressed or when C-L is pressed). This will clobber any user modifications.

The last 64 characters (CO-FF) are completely under programmer control. They are always displayed as nonsense until they are defined by turning on and off the bits of the eight bytes associated with the character. These bytes are in RAM from FE00 to FFFF (-512 to -1). For example, the character C0 (192) is at FE00-FE07 (-512 to -505), C1 (193) at FE08-FE0F (-504 to -497), C2 at

FE10-FE17, and so on through FF (255), which is at FFF8-FFFF (-8 to -1). The formula to calculate where the eight bytes in RAM begin for any of the 128 programmable characters (80-FF) is:

FC00 + (8 * (c - 80)) (hexadecimal)

(8 * (c - 128)) - 1024 (decimal)

where "c" is the character code of the character to be programmed and ranges from 80-FF (128-255).

The procedure for defining characters involves designing a character and then translating the design into binary code. Each character definition takes eight bytes of memory, or 64 bits. On the screen each character consists of eight lines of eight dots. Thus each of the eight bytes defining the character in memory corresponds to one of the eight lines of the character in the display, and each of the eight bits in that byte is a dot in that line. If the bit is on (1), then the dot is white. If the bit is off (0), then the dot is black. For example, a circle with a dot in the middle could be defined as a character. It would require defining each of the 64 (8x8) dots as 64 (8x8) bits in memory. So

```
. . . . . . . .     00000000 binary    00 hex     0 decimal
. . X X X . . .     00111000           38        56
. X . . . X . .     01000100           44        68
X . . . . . X .     10000010           82       130
X . . X . . X .     10010010           92       146
X . . . . . X .     10000010           82       130
. X . . . X . .     01000100           44        68
. . X X X . . .     00111000           38        56
```

The following BASIC program demonstrates how user-defined characters can be generated and displayed using BASIC. The program prints a "blot" (all dots on, a white square) on the screen followed by the above circle with the dot in the middle. The blot will be the first programmable graphic code C0 (192), and the circle/dot will be C1 (193).

```
10  FOR I=0 TO 7: REM 8 BYTES AT FE00 (-512) FOR BLOT
20  POKE -512+1,255: NEXT: REM TURN ON ALL BITS/DOTS
30  FOR I=0 TO 7: REM 8 BYTES AT FE08 (-504) FOR CHR #193
40  READ J:REM GET A BYTE VALUE FROM THE TABLE AS ABOVE
50  POKE -504+I,J: NEXT: REM TURN ON CORRECT DOTS
60  PRINT CHR$(192) ;CHR$(193): REM PRINT THE 2 NEW CHRS
70  DATA 0,56,68,130,146,30,68,56: REM DATA CHR #193
80  END
```

CURSOR POSITIONING

Cursor positioning is the process of moving the cursor (that underscore character) on the screen to locations other than where it usually is when standard BASIC or Monitor video output is done (eg., PRINT, DUMP, etc.). This is very useful especially when data is to be placed on the screen but not in a line by line fashion. For example, if a graphic diagram is displayed and certain segments are to be labeled, the cursor can be moved directly to each one and the output generated in a random fashion on the screen. Also, in some programs output statements will destructively erase what is already on the screen. For example, if something is to be printed in the middle of a line but there is information already in the beginning of that line, an output statement will erase it. Proper cursor positioning will leave the beginning of the line intact.

To perform cursor positioning from Assembly Language or BASIC is quite simple:

1. Restore character under cursor. In machine language,

   ```
   CALL 0E9E8H
   ```

   From BASIC use the USR technique as follows:

   ```
   900 POKE 260, 232 : REM HEX E8
   901 POKE 261, 233 : REM HEX E9
   902 X=USR(X)   : REM CALL E9E8
   ```

2. Decide what row the cursor is to be on. There are 30 rows numbered 0-29. Call this "R".
3. Decide what column of that row the cursor is to be in. There are 64 columns numbered 0-63. Call this "C".

4. Calculate 64 times R. This is the offset from the beginning of the screen to the first column (0) of row R. This is easy in BASIC (Q=64*R). In machine language, just shift R left six times, or, assuming R were in register E:

```
        LD      D,0     ;D=0,E=R
        LD      B,6     ;TIMES TO SHIFT
X:      SLA     E       ;SHIFT E
        RL      D       :SHIFT D
        DJNZ    X       ;6 TIMES, DE=64*R
```

An alternate calculation would be to put R in the register pair HL and execute the ADD HL,HL instruction six times in a row to double R six times, or multiply by 64.

5. Find the Monitor workarea (MWA). This is described in detail in Chapter 3. For the examples below, assume register IY points to the MWA for Assembly, or assume AD equals the MWA address for BASIC.

6. At offset 68 hex (IY+68 or AD+104) are two bytes where 64*R is to be stored:

```
        LD      (IY+68),E
        LD      (IY+69),D
```

or in BASIC, POKE the low part (low byte) of the number 64*R, (64*R) MOD 256, into AD+104, and POKE the high part (byte) of 64*R, INT (64*R/256) at AD+105. Now, (64*R)MOD 256 is just the remainder when 64*R is divided by 256, and this can be calculated as follows in BASIC:

```
905 R2 = 64*R
910 MD = R2-INT (R2/256)*256
```

To do the POKEs, assuming AD is already pointing to the MWA:

```
915 POKE AD+104,MD
920 POKE AD+105,INT(R2/256)
```

7. At offset 6A in the MWA (IY+6A, AD+106) are two bytes where "C" is to be stored. If it were in register A:

```
        LD      (IY+6A),A
        LD      (IY+6B),0
```

or in BASIC:

```
930 POKE AD+106,C
940 POKE AD+ 107,0
```

BASIC also requires you to put C at location 18E (398) in the BASIC control area (BCA):

```
950 POKE 398,C
```

8. Call the Monitor cursor move routine. This will replace the current cursor with the character which was at that spot ("underneath" it), move the cursor to the requested spot and save the character there. In machine language,

   CALL 0E9CCH

   From BASIC the USR technique must be used:

```
960 POKE 260,204: REM HEX CC
965 POKE 261,233: REM HEX E9
970 X=USR(X): REM CALL E9CC
```

9. Now a standard output statement like PRINT can be done and the output will begin at this new cursor location.

This technique enables very fast horizontal and vertical tabbing that is non-destructive. If speed and non-destructibility are not required, tabbing can be done with BASIC statements. Horizontal tabbing can be done directly with the use of the TAB(n) function. This function overwrites all letters on the row, spacing out to the nth column before writing.

Vertical tabbing may be done with C-Z (down arrow) characters. For example, to tab to row 15, home the cursor with a C-Q (hex 11, decimal 17) and then do a C-Z 15 times (C-Z is hex 1A, decimal 26):

```
2220 PRINT CHR$(17);: REM HOME
2240 FOR I=1 TO 15
2260 PRINT CHR$(26);: REM DOWN ONE ROW
2280 NEXT
```

PRINT TAB(n) can then be used to tab horizontally on that row.

CHAPTER SEVEN

Sorcerer Keyboard

KEYBOARD ARCHITECTURE

The keyboard on the Sorcerer has a unique physical (hardware) and logical (software) architecture. Communication between the keyboard and the computer is via small parts of the input and output ports FE (254). The keyboard has a potential for 80 keys, and the software organization is similar to the physical keyboard layout: five rows of sixteen keys each. Keys within rows are indicated by output port FE, bits 0, 1, 2, and 3, and range in value from 0 to F (0 - 15). The five rows correspond to the five input FE bits 0, 1, 2, 3, and 4. For example, row 0 key 0 is ESC, row 3 key 9 is a P, and row 4 key 15 is the = key on the numeric keypad. Not all 80 possibilities are in use (about three are meaningless).

Each of the 80 possibilities can assume any one of five states:

1. SHIFT depressed — used for upper case and punctuation; no numerics or graphics; cursor arrow keys remain operative.
2. LOCK depressed — this is a CAPS LOCK so upper case latters, numerics, and punctuation are valid, but no graphics or cursor movement keys.
3. CTRL depressed — this produces control characters, some numerics, and cursor movement; no graphics.
4. GRAPHICS depressed — this is standard Sorcerer keyboard graphics (codes 80-BF). If SHIFT is also depressed simultaneously, the programmable graphics codes C0-FF are used.

5. None of the above depressed — standard lower case and numerics and punctuation are used; no graphics or cursor movement.

The Monitor ROM area EC1E-EDFD contains the tables necessary to allow the keyboard input routine to translate the signals from the keyboard into a one-byte character code, depending on which of the five states the keyboard is in. These tables are actually broken down into six tables total: the first is a what-to-do table to calculate the state, and the last five are the character codes for the five states.

PERFORMING KEYBOARD INPUT

To get keyboard input from the user from BASIC or Z80 Assembly Language without INPUT statements, a very useful Monitor subroutine can be used. In fact, this can be done such that the program sees each character as it is typed without having to wait (or ever get) a carriage return (RETURN). For example, a program can react and respond immediately to input commands as they are typed.

From BASIC, characters can be input with the following example assembly routine. Place this simple and relocatable Monitor keyboard routine driver interface at, say, location F0 (240). It can go anywhere, but F0 is a good start.

```
F0: CD15E0  SCAN:   CALL   QCKCHK     ;CONTROL-C PRESSED?
F3: C2FADF          JP     NZ,BASIC   ;YES, BACK TO BASIC(WARM)
F6: CD09E0          CALL   RECEIVE    ;NO, GET INPUT CHARACTER
F9: 28F5            JR     Z,SCAN     ;NOTHING YET, CONTINUE
FB: 32FF00          LD     (CHR),A    ;GOT IT, SAVE AT LOC FF
FE: C9              RET               ;RETURN AFTER USR CALL
FF: 00      CHR:    NOP               ;WHERE BYTE STORED
            QCKCHK EQU     0E015H
            BASIC   EQU    0DFFAH
            RECEIVE EQU    0E009H
```

The routine first checks to see if **C-C, ESC, or RUN/STOP** have been entered, meaning the user wants to quit. If so (Not Zero) back to READY level. If not, the current RECEIVE device (usually keyboard) is scanned for a character. If none (Zero), scanning continues. If found, the character is put at location FF (255). Control is then return to BASIC after the USR call. The following example BASIC program can use this routine:

```
10 PRINT "ENTER CHARACTER"
20 POKE 260,240: POKE 261,0: REM LOC 00F0 IS 240,0
30 Z=USR(Z): REM CALL SCAN
40 REM IF WE GET HERE LOC FF HAS A CHARACTER
50 A$ = CHR$(PEEK(255))
60 IF A$ = "S" THEN STOP: REM STOP IF S ENTERED
70 PRINT A$: REM ECHO THE CHARACTER
80 GOTO 20: REM LOOP TILL S ENTERED
```

These are both simple routines that can be modified to be as fancy as necessary.

From Z80 machine language there is no need to store the character in RAM. It is returned in the accumulator by the RECEIVE routine.

The above programs accept their input from the current RECEIVE device. To set this device the Monitor command SET I=x is used, where x is the desired input device (see Chapter 2).

INDEX

Acoustic coupler **2**-3
addresses **1**-3
alphabetic characters **6**-4, **6**-5
arrays, BASIC **5**-5, **5**-8
assembly language **3**-9, **5**-9, **5**-12, **6**-13
automatic level control **4**-3

BASIC
- arrays **5**-5, **5**-8, **5**-12
- calling the monitor from **3**-9
- control area **3**-3, **5**-3, **5**-6, **5**-7
- cursor positioning **6**-13
- defining characters with **6**-12
- floating point numbers **5**-1, **5**-5
- machine language interfacing **3**-9, **5**-9
- memory map **5**-4, **5**-6, **5**-7
- program statements **5**-5
- reserved character codes **6**-10
- reserved words **5**-5, **6**-1
- ROM-PAC **1**-4, **5**-5
- shorthand notation **6**-1
- stack **5**-4, **5**-6
- string space **5**-4, **5**-6
- string variables **5**-8
- tabbing **6**-15
- variables **5**-5, **5**-12

BATCH command **3**-6, **3**-12
baud rate **2**-2
BCA **3**-3, **5**-3, **5**-12
BYE **3**-10

Carriage return (CR) **3**-5, **3**-11
cassette interface **2**-1, **4**-1
cassette motor controls **3**-6, **3**-10, **4**-1
Centronics printer interface **2**-1, **2**-3
character codes **6**-1
character definitions **6**-11
clearing the screen **3**-9
CLOAD **4**-1, **4**-3
cold start **3**-5, **3**-6, **3**-10, **5**-11

control codes **6**-2, **6**-10
CRC **3**-7, **4**-1, **4**-2, **4**-3, **4**-4
CRLF **3**-11
CRTL key **6**-1, **7**-1, **7**-3
CSAVE **3**-7, **4**-1
cursor **3**-12, **5**-6, **7**-1, **7**-1, **7**-2
cursor movement **6**-10
cursor positioning **6**-13
cyclical redundancy check **3**-7, **4**-1

DB25 connector **2**-2
debugging **5**-12
decimal, relation to hex and binary **1**-1
direct memory access **3**-11, **6**-11
display screen **6**-1
DMA **6**-11
dynamic RAM **1**-5

EPROM **1**-4
ERROR message **3**-11, **5**-11
ESC key **7**-3
exponent **5**-1

FILES command **3**-12, **4**-1, **4**-3
floating point numbers **5**-1
FOUND message **4**-3, **4**-4
free space **5**-4, **5**-9

Graphics **6**-1
graphic characters **6**-6, **6**-7, **6**-8, **6**-9
graphics with BASIC **6**-12
GRAPHICS key **3**-3, **6**-1, **7**-1

Header, tape **4**-1, **4**-2
hex **1**-1
hex digits **1**-1, **1**-2
hex representation of floating point **5**-3
HIMEM **3**-1, **3**-2, **3**-4, **3**-10, **5**-4, **5**-9
horizontal tabbing **6**-15

I/0 devices 2-1
input ports 2-2
interfaces 2-1
inter-file tone 4-1, 4-2

"K" 1-3
keyboard 2-1, 7-1
keypresses 6-1, 6-2

Line feed (LF) 3-11
LOAD 3-11, 3-12, 4-1, 4-3
LOADG 3-7, 3-11

Machine language 1-1, 3-9
machine language interfacing to BASIC 5-9
mantissa 5-1
memory 1-3, 1-4
memory maps
  BASIC 5-4
  SORCERER 3-1, 3-2, 3-3
modem 2-3
Monitor
  commands 2-1
  description 3-1
  interfacing with BASIC 3-9, 5-9
  stack 3-2, 3-3
  subroutines 3-9 through 3-12, 4-3, 5-12
  workarea 3-1 through 3-8, 6-14

motor controls, tape 3-6, 3-10, 4-1
MPU 1-1
MWA 3-1 through 3-8, 3-10, 3-11, 3-12, 4-3, 6-14

Numeric characters 6-3

Output ports 2-2
OVER command 3-4, 3-6

Parallel interface 2-1, 2-3
ports 2-1
printer interface 2-1

programmable characters **6**-11
PROM **1**-4, **6**-11
PROMPT command **3**-6, **3**-12

RAM **1**-4, **2**-1
RECEIVE **3**-5, **3**-10, **7**-3
registers **1**-5
REPT key **3**-8
reserved character codes, BASIC **6**-10
reserved words, BASIC **5**-5, **6**-1
RESET **2**-3, **2**-4, **3**-5, **3**-10, **6**-11
restart space **3**-2, **3**-3
ROM **1**-4
ROM, BASIC **1**-4, **5**-5
ROM PAC **1**-4, **3**-2, **3**-3
RS-232 **2**-1
RUN/STOP key **7**-3

S-100 expansion facility **2**-1
SAVE command **3**-11
screen flicker **3**-11
SEND command **3**-6, **3**-10, **3**-11, **3**-12, **4**-3
serial interface **2**-1, **2**-2, **3**-10
SET command **2**-1, **2**-3, 3-5, **3**-6, **3**-7, **3**-10, **3**-12, **4**-3, **7**-3
SHIFT key **3**-3, **6**-1, **7**-1
SHIFT LOCK key **6**-1, **7**-1
SORCERER
- BASIC **5**-1
- cassette interface **4**-1
- keyboard **7**-1
- I/O devices **2**-1
- ports **2**-1
- Monitor **2**-1, **3**-1
- Video interface **6**-1, **6**-11

Sorcerer-defined graphic characters **6**-6, **6**-7
string arrays **5**-8
string space **5**-4, **5**-6
string variables **5**-8
static RAM **1**-5

Tabbing **6**-15
tape interface **2**-1, **4**-1
tape motor controls **3**-6, **3**-10, **4**-1

User-defined graphic characters **6**-8, **6**-9, **6**-11
USR function **3**-9, **5**-10

Variables, BASIC **5**-5, **5**-8
variables, string **5**-8
vertical tabbing **6**-15
video
  driver **3**-2, **6**-1, **6**-11
  interface **6**-1
  screen **2**-1, **6**-11
  screen flicker **3**-11

Warm start **3**-6, **5**-11

Z-80 **1**-1, **1**-5, **2**-1
Zilog **1**-6

DISCLAIMER

The information in this manual is the work of the author. Exidy Inc. is in no way responsible for the accuract of its contents. Much of the research for this manual was conducted by wading through personal disassemblies of the Monitor and BASIC trying to figure out what was going on (the author has also written a Z80 disassembler which is available from Quality Software).

Neither the author nor Quality Software makes any guarantee or warranty, expressed or implied, of the accuracy of the information contained in this manual. Many Sorcerer programmers, including the author, have found this manual to be extremely useful, and we hope the same holds true for you. Good luck in your programming!