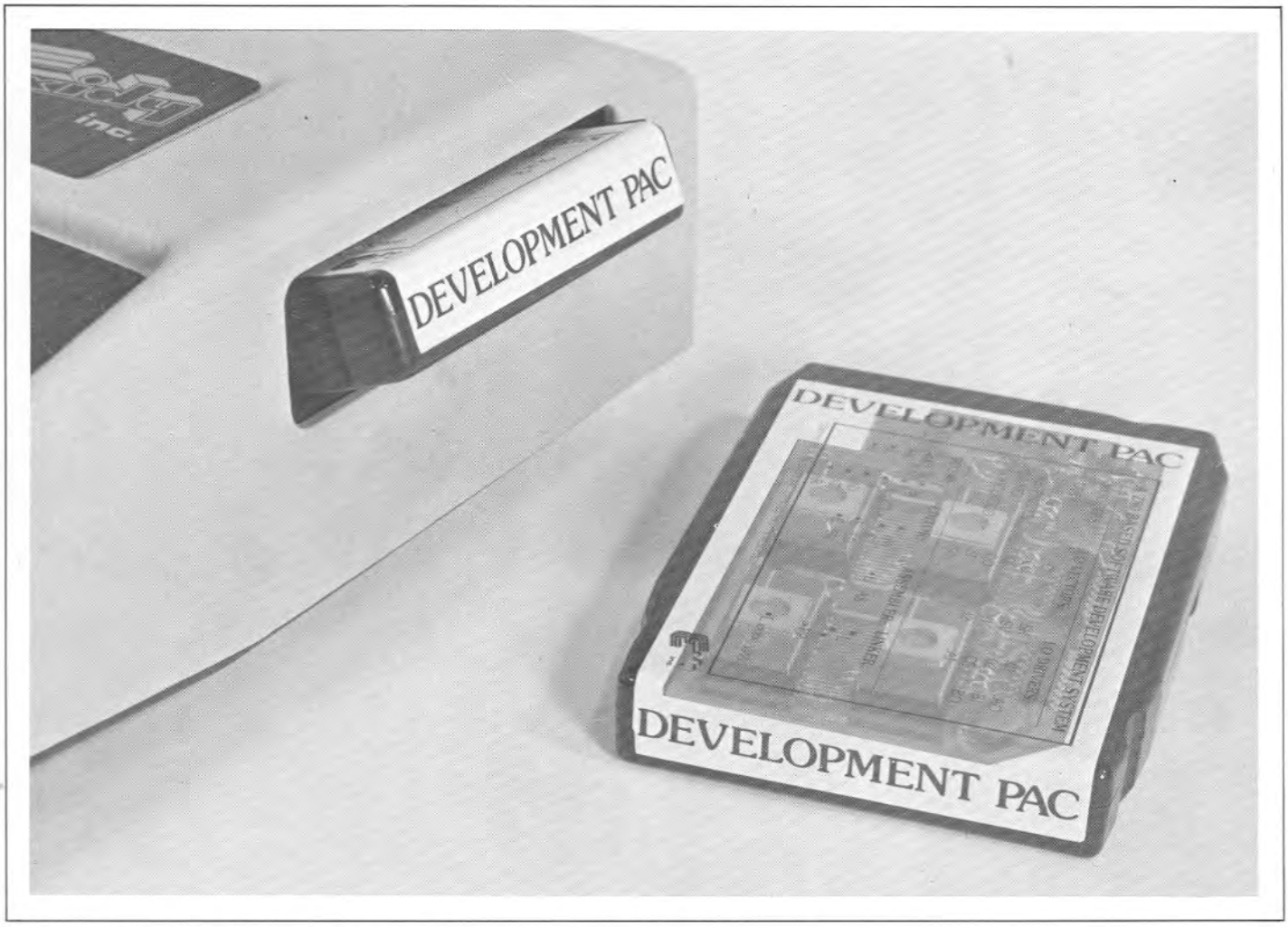
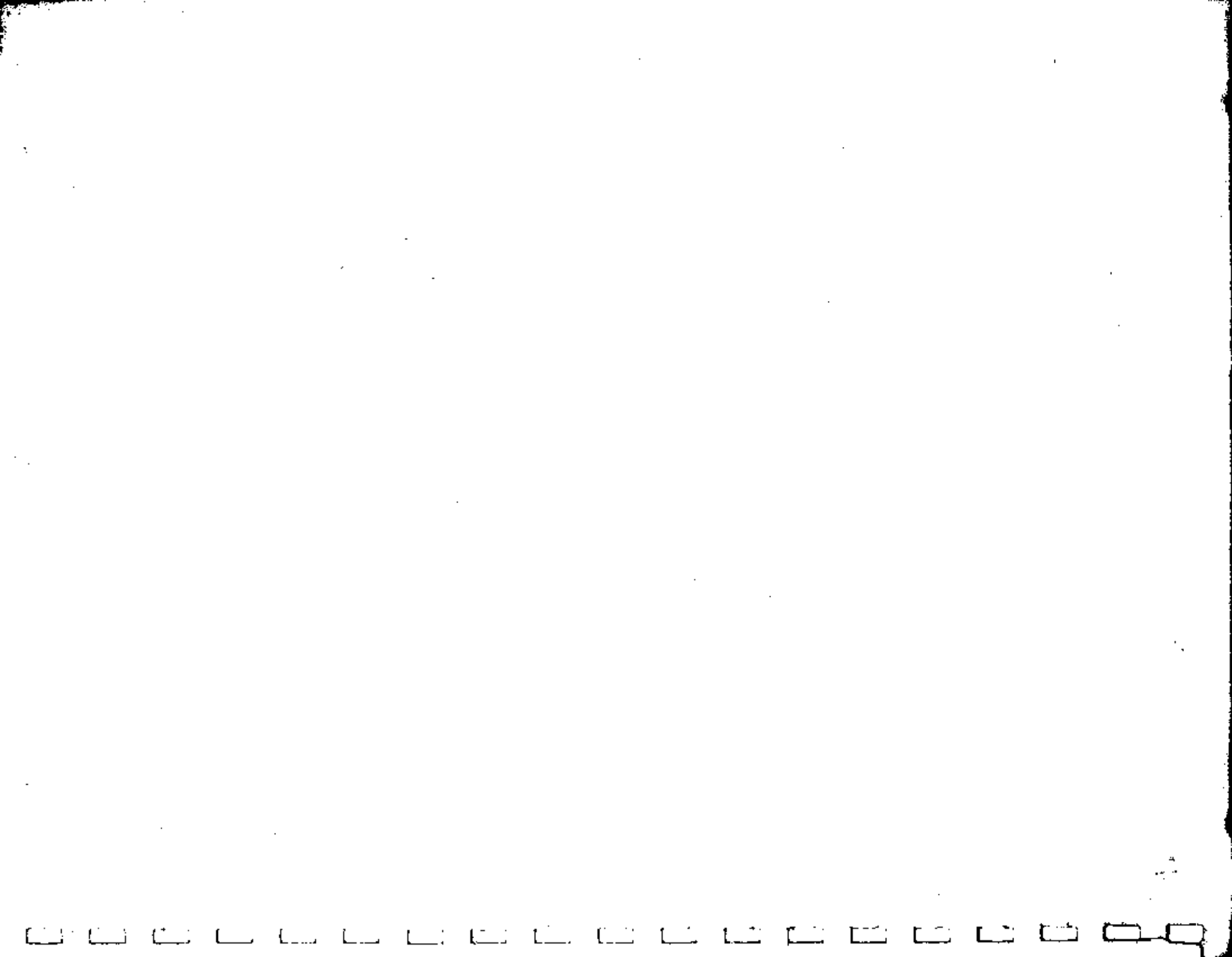
USER'S MANUAL

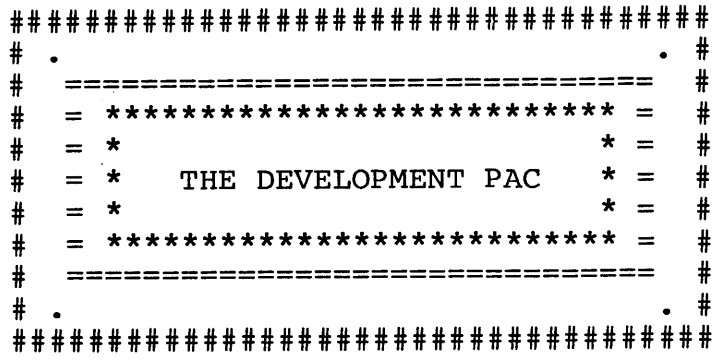


Z80 Based Software Development System for the Sorcerer Computer

Exidy inc.



exidy inc. Data Products Division



A Z80 BASED SOFTWARE DEVELOPMENT SYSTEM

FROM
EXIDY INC.
DATA PRODUCTS DIVISION

969 West Maude Avenue • Sunnyvale, California 94086 • Telex 348-329 • (408) 736-2110

THE DEVELOPMENT PAC

TABLE OF CONTENTS

CHAPTER		PAGE
I.	INTRODUCTION	1
	THE DEVELOPMENT PAC	2
	THE MODULES	4
	DDT80	4
	EDITOR	4
	ASSEMBLER	4
	LOADER	5
	I/O DRIVERS	5
II.	DDT80	7
	THE "M" COMMAND	8
	SECONDARY COMMANDS	8
	THE "R" COMMAND	11
	THE "E" COMMAND	13
	THE "H" COMMAND	15
	THE "L" COMMAND	16
III.	SORCERER I/O	17
	WRITING I/O DRIVERS	18
	THE DRIVERS	21
	THE RAM BUFFERS	24
IV.	EDITOR	27
	TEXT BUFFER AND LINE POINTER	28
	LINE EDITING	30
	EDITOR I/O	31
	CALLING THE EDITOR	33
	EDITOR COMMANDS	34
	B COMMAND	34
	<CR> COMMAND	34
	I COMMAND	35
	D COMMAND	35
	T COMMAND	36
	R COMMAND	36
	W COMMAND	36
	E COMMAND	37
	AN EDITING EXAMPLE	38

TABLE OF CONTENTS

CHAPTER		PAGE
V.	ASSEMBLER	41
	BASIC DEFINITIONS	42
	ASSEMBLY LANGUAGE SYNTAX	45
	LABELS	45
	OPCODES	46
	PSEUDO-OPS	46
	OPERANDS	48
	COMMENTS	51
	ABSOLUTE AND RELOCATABLE MODULES	52
	GLOBAL SYMBOLS	53
	ASSEMBLER I/O AND OPERATION	55
	ASSEMBLER ERRORS	57
VI.	LOADER	58
	LINKING AND LOADING BASICS	59
	LOADER SYMBOL TABLE	60
	LOADER I/O	61
	CALLING THE LOADER	62
	LOADER ERRORS	63
VII.	APPENDICES	64
	APPENDIX A - OBJECT OUTPUT DEFINITION	64
	APPENDIX B - REPARTITIONING RAM	67
	APPENDIX C - MNEMONICS RECOGNIZED BY DDT80	71
	APPENDIX D - ASSEMBLER ERRORS	72
	APPENDIX E - LOADER ERRORS	73
	APPENDIX F - COMMANDS (SUMMARIZED)	74
	APPENDIX G - ACKNOWLEDGEMENTS	75
VIII.	INDEX	76

INTRODUCTION

The Sorcerer that you already own, or have just bought, is a very sophisticated piece of equipment. You've probably already heard the familiar "It can do anything you program it to do!" Well, as you may have found out, "programming" is a non-trivial task! The Standard BASIC cartridge that is available for your Sorcerer was most likely your first introduction to the world of programming. You probably came up against the limitations of BASIC the very first week you started using it!

Well, now you have an alternative! The DEVELOPMENT PAC. The Development Pac will help you write Z80 machine language programs with a minimum of effort and a maximum of versatility. Contained in the cartridge is everything you need (software-wise) to efficiently create, assemble, debug and run Z80 machine language programs.

It is recommended that the user study the Z80 instruction set and CPU architecture before using the Development Pac. This manual is not intended to be a complete tutorial on machine language programming, but rather a reference manual to explain the use and structure of the DEVELOPMENT PAC.

We welcome you to the world of machine language programming and hope that this Package will help you use your Sorcerer computer and all of the power that it's capable of!

Exidy
Data Products Division

THE DEVELOPMENT PAC

The Development Pac consists of five main modules:

```
========================= MODULES: ========================
     DDT80      - Designer's Debugging Tool
     EDITOR     - A line oriented source editor
     ASSEMBLER - A relocatable, linking Z80 assembler
     LOADER     - A relocating, linking loader
     SORCERERIO- The Sorcerer I/O routines
===========================================================
```

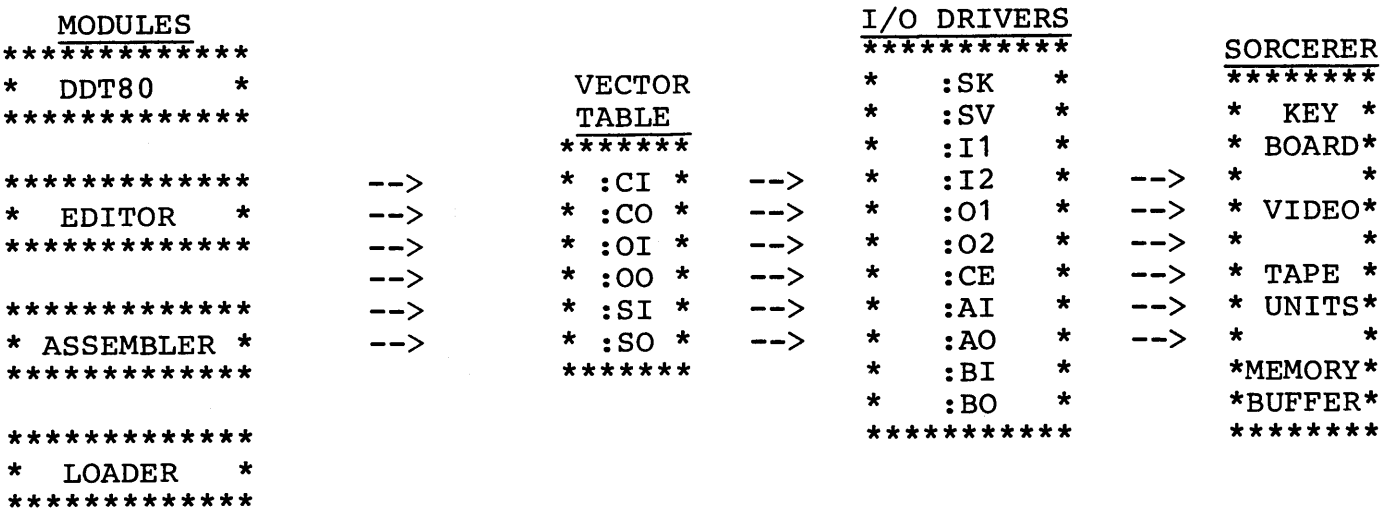
Each module will be explained in detail in their appropriate chapters in the manual. Each module "talks" to the Sorcerer through a "Vector" table. The Vector table receives all data to be sent to the Sorcerer and all data coming from the Sorcerer to the Development Pac. The Vector table is formatted as follows:

```
======================== I/O VECTORS: =====================
                :CI - Console input channel
                :CO - Console output channel
                :OI - Object input channel
                :OO - Object output channel
                :SI - Source input channel
                :SO - Source output channel
===========================================================
```

Each module will communicate through one or more of the Vector table entry points which the user can assign to any of the physical Input/Output drivers:

```
======================== I/O DRIVERS: =====================
      :SK - Sorcerer Keyboard
      :SV - Sorcerer Video Screen
      :I1 - Input from Sorcerer Cassette Unit # 1
      :I2 - Input from Sorcerer Cassette Unit # 2
      :O1 - Output to Sorcerer Cassette Unit # 1
      :O2 - Output to Sorcerer Cassette Unit # 2
      :CE - Output to Sorcerer Centronics Driver
      :AI - Input to the "A" Memory Buffer (Object)
      :AO - Output from the "A" Memory Buffer (Object)
      :BI - Dummy Input into "B" Memory Buffer
      :BO - Output from the "B" Memory Buffer (Source)
===========================================================
```

To make the connection more clear:



Each module in the Development Pac "talks" through one or more of the vector table entry points (Exactly which vector entry points should be used is discussed in the section for each module). By using DDT80, the user can assign any one of the physical I/O drivers to any one of the vector table exit points. In this way, the user can make any one of the modules communicate with any part of the Sorcerer. For example, the Editor messages can appear on the Sorcerer video, or they can be sent to either one of the cassette units, or they can be printed out on the Centronics printer. This method of assigning I/O devices allows the user a tremendous amount of flexibility. New devices can be added very easily or the existing devices can be configured in almost any combination.

THE MODULES

DDT80

DDT80 (Designer's Dubugging Tool) is the monitor of the Development Pac. It is the module that first receives control when power is applied to the Sorcerer.

DDT80 can display and/or modify any RAM location in the Sorcerer, display and/or modify any of the Z80 program registers, and execute a user program, with breakpoints, or any module in the Development Pac.

Every module in the Development Pac, except for the special case of the Loader, returns control to DDT80 upon completion of its appropriate task. DDT80 is then used to set up the I/O vector table for use by the next module.

Editor

The Text Editor aids the user in the creation of source text for use by the Assembler. With the Editor, a user can create, load, store, change, append to, delete, and print an entire source module in the Sorcerer RAM area or, spooling the text a part at a time, from the Sorcerer's cassette units.

The Editor also allows for reentry to preserve the text buffer from the previous editing session. This allows for a very efficient and quick RAM based mode of operation.

Assembler

The Assembler is a two pass, relocating, linking Z80 assembler. By creating a program as a series of modules instead of one single program, changes can be made to the appropriate module and a quick reassembly can be performed on only the altered source module. The linker can then load all of the original modules along with the new, altered module to create a complete object program. This method of program development speeds up the debugging time needed to complete a program tremendously by allowing editing and assembling of only a small part of the final program. This method also allows the creation of "universal" object modules that can be kept on cassette tape and refered to in later programs. Then, at linkage time, the appropriate module will be flagged as "undefined" so the user will then know which object module cassette to load in next. After a

period of time using this method, a huge library of object tapes will be compiled for use by the program developer!

The Assembler also allows for absolute assembly for those programs that are to be loaded and run in a particular area of memory or for those that do not need much debugging. This gives the user the best of both worlds in developing machine language programs.

Loader

The Relocating Linking Loader has the capability of loading, from cassette tape or RAM buffers, object modules created by the Assembler. A single module or a series of modules can be loaded at one time. If a series of modules is to be loaded, the linker will resolve all "GLOBAL" references among the modules. With this capability, the user can create "universal" program modules that can be "called" by other modules and then linked together as each module is loaded!

The Linking Loader can also load absolute Intel hexadecimal formatted object modules like those produced from other 8080 or Z80 absolute assemblers or the Development Pac's Assembler in the "ABS" mode. The Linking Loader also has the capability of loading both absolute and relocatable modules in the same loading session!

I/O Drivers

Each module in the Development Pac needs to have some form of communication with the Sorcerer. Sometimes this communication is with the video screen or the keyboard, and other times it's with the cassette units or some other peripheral device. The Development Pac will never know exactly which part of the Sorcerer it's talking to. Instead, each module "talks" to a "vector" point. As each module needs information from the user or wants to output information to the user, it passes control to a vector point. The location of these vector points never changes; however, by using DDT80 commands you can "tell" the vector point to go to another location. At that location will be a driver for the individual device the Development Pac wishes to communicate with. This is why the module SORCERERIO is included in the Development Pac. This module contains drivers for devices inside or connected to the Sorcerer. Using DDT80, you can assign any of the vector points to any of the device drivers in the SORCERERIO module.

Next, each module will be discussed in greater detail along with its vector table assignments and commands. As you study each module, play with it on your Sorcerer! See what it can and can't do. To get the best and quickest feeling for the DEVELOPMENT PAC you should .....

***** EXPERIMENT ! *****

DDT80

DDT80, the Designer's Debugging Tool, is the monitor and program debugger for the Development Pac. Using DDT80 you can display and/or modify any RAM memory location in the Sorcerer, display and/or modify any of the Z80 internal registers (including the stack pointer and program counter!), execute any user program, with breakpoints, or any of the other modules in the Development Pac.

When your Sorcerer is first turned on with the DEVELOPMENT PAC inserted in the side slot (DO NOT remove or insert the cartridge with the power on!), DDT80 will be given control. It will first print a sign-on message and then display the monitor prompt "." (a period).

```
EXIDY Z80 DEVELOPMENT PAC
    COPYRIGHT (C) 1979

.
```

At this point the user can enter any one of the DDT80 commands. Each command is a one letter command. All the user need do is type the letter. (No carriage return is needed at this point.) DDT80 will accept the command, print the command on the screen and print a space. The user can then enter the parameters for the command. After entering the parameters, a carriage return will execute the command.

DDT80 has five basic commands:

========== DDT80 COMMANDS: ==========

M - Memory display/modify command
R - Register display/modify command
E - Execute program command
H - Hexadecimal arithemetic command
L - Loader transfer command

==========

THE "M" COMMAND.

The "M" command is the memory display/modify command. Using this command, you can display, in hexadecimal format, the address and data of any location in the Sorcerer. You can then, optionally, alter that data to any value you chose. Note: the alter function will only work on RAM locations. Atempts to alter ROM locations will have no effect.

The format for the "M" command is:

```
============================================================
      .M Parameter #1,(Parameter #2) <Carriage Return>
============================================================
```

Parameter #1 must be specified and is interpreted to be in hexidecmal form or, if preceded by a colon (:), it is interpreted to be a mnemonic label. Parameter #2 is optional and is used to specify the ending value for a range of values.

This command has three forms. The first form is specified by using a single parameter in hexidecimal form. For example:

```
.M 1234  <Carriage Return>
```

will display the address and data for the location 1234 hex in the following format:

```
1234 5A _
```

The cursor will appear on the same line and is awaiting a secondary command. The secondary commands are:

```
============== "M" COMMAND SECONDARY COMMANDS ==============
                    . - End command and return to DDT80
   <Carriage Return> - Display next location
                    ^ - Display previous location
         <Hex value> - Alter location to <Hex value>
============================================================
```

The "." will end the "M" command mode and not alter the contents of that location. The DDT80 prompt "." will appear on the next line and await any DDT80 main command. If the "." is preceded by a hex value, that location will not be altered.

The <Carriage Return> will not alter the contents of that location (Unless preceded by a hex value). It will

cause the "M" command to display the contents of the next location after the present location. If the <Carriage Return> is preceded by a hex value, the contents of that location will be altered to the hex value and then the contents of the next location will be displayed. For example:

```
.M 1234 <Carriage Return>
1234 5A <Carriage Return>
1235 6C <Carriage Return>
1236 77 6A  <Carriage Return>
1237 .
._
```

The "^" will not alter the contents of the present location (Unless preceded by a hex value). It will cause the "M" command to display the contents of the previous location. If the "^" is preceded by a hex value, that location is altered to the hex value and then that same location is displayed again with the new value. For example:

```
.M 1234 <Carriage Return>
1234 5A ^
1233 FC ^
1232 61 F3^
1232 F3 ^
1231 66 .
._
```

The second form of the "M" command is specified by entering parameter #1 as a hex value and parameter #2 as a hex value. For example:

```
.M 1230,1257
```

will display, in hexadecimal format, the beginning address of each 16 byte block followed by the hex value of the 16 bytes. For example:

```
1230 11 3A 33 55 F6 73 41 43 65 FF FF FF FF DF FF FF
1240 FF 5F 43 21 00 00 00 FF AF 63 00 00 00 FF FF FF
1250 BA ED CB F0 65 55 63 7F
```

This form of the command is used to examine an area of memory and verify its contents. It has no "alter" provisions and is only intended to be a display command.

The third form of the "M" command is specified by sustituting a mnemonic for any of the hex values in the forms above. A mnemonic is created by typing a colon (:) followed by a two letter code. All of the two letter codes which the Development Pac recognizes are listed in appendix C. Each of the mnemonics is associated with a specific

address. The mnemonic is used to simplify the task of remembering often used memory locations. For example, to change the Source Output channel to the Centronics printer driver, you would use the first form of the "M" command with the appropriate mnemonics:

```
.M :SO <Carriage Return>

:SO  :SV  _
```

The "M" command was told to display the memory location corresponding to the mnemonic ":SO". It came back and told us that the Sorcerer Video driver, :SV, is currently assigned to the Source Output channel. To alter this to the Centronics printer driver the user would use the "^" form of the M command preceded by a mnemonic:

```
:SO  :SV  :CE^
:SO  :CE  .
.-
```

The "^" secondary command was used so that the user could verify that the alteration had been properly performed. Had the user wanted to alter more than one location, the <Carriage Return> secondary command would have been used to access the next location.

It should be remembered that the purpose of the mnemonics is to give an easily remembered label to an often used address. When accessing through memory using the "M" command, all addresses that correspond to a mnemonic label will be printed out in mnemonic form. All mnemonics consist of two letters. Any one letter mnemonics listed in appendix C should be typed in with spaces to complete the two letters.

THE "R" COMMAND

The "R" command is the register display/modify command. DDT80 maintains an area of memory to hold all of the Z80 internal registers that are used by a user's program. As the user executes and breakpoints through his program, DDT80 will save and restore the user's registers as his program is stopped and started. Before the user reenters or begins his program, he can alter any of the registers to correct a program fault or test a program condition. With the "R" command the user even has control and display capability of the stack pointer and program counter as well as all of the standard and alternate registers.

The form of the "R" command is

```
============================================================
                    .R (1) <Carriage Return>
============================================================
```

If the "R" is entered alone on the command line, then all of the registers will be displayed on the next line. No heading that identifies the registers will be printed. This is the short form of the register printout. To have register headings printed along with the register values the "1" should be entered on the command line. For example:

```
.R <Carriage Return>
A000 0181 0104 CFB3 C09A FFEE EDF6 9C3E C3DC FE9B D6ECF1BE FFB4
.
```

```
.R 1 <Carriage Return>
 PC   AF   IIF  BC   DE   HL  A'F' B'C' D'E' H'L'  IX  IY   SP
A000 0181 0104 CFB3 C09A FFEE EDF6 9C3E C3DC FE9B D6ECF1BE FFB4
```

The headings across the top of the register print out represent the register names as follows:

```
PC   = Program Counter
AF   = Accumulator / Flag register
IIF  = Interrupt status / Interrupt mode flag
BC   = B register / C register
DE   - D register / E register
HL   = H register / L register
A'F'= A' register / Flag' register
B'C'= B' register / C' register
D'E'= D' register / E' register
H'L'= H' register / L' register
```

IX = IX index register
IY = IY index register
SP = Stack Pointer

The Flag register is formatted as follows:

```
                        7     BIT     0
  FLAG REGISTER = S Z X H X P N C
```

WHERE:

S = Sign flag
Z = Zero flag
X = Indeterminate flag
H = Half carry (for BCD operations)
P = Parity or overflow flag
N = BCD add/subtract flag
C = Carry flag

The "I" and "IF" registers represent the interrupt status of the Z80 processor when the user program was stopped by the breakpoint. The "IF" register represents the interrupt flip-flop maintained by the Z80. If IF=0, then interrupts were disabled when DDT80 received control. If IF=4, then interrupts were enabled. Upon reset or power-on IF is set to 0. The "I" register represents the interrupt mode that the Z80 was in when the breakpoint was encountered. See the Z80 programming manual for the interrupt mode definitions.

The "R" command is a display function only. The "M" command should be used to alter any of the registers. Appendix C lists the mnemonics corresponding to the locations within the Development Pac containing the register values. By using the display/modify function of the "M" command the user may alter any of the registers before returning control to his progam. This feature of register display/alteration is a very powerful debugging tool when used properly.

THE "E" COMMAND

The "E" command is the program execute and breakpoint set command of DDT80. It is the only command that can transfer control to a user's program. Before executing the user's program, all of the user's registers are restored to the value when DDT80 recieved control or the modified value stored in the register storage area. Using the "E" command the user has the capability of setting a breakpoint or "stop" point in his program. This feature adds still another powerful debugging tool to the Development Pac.

The format for the "E" command is:

```
=============================================================
      .E Parameter #1 (,Parameter #2) <Carriage Return>
=============================================================
```

Parameter #1 must be specified and is interpreted to be in hexadecimal format. Parameter #2 is optional and is also expected to be a hex number. Parameter #1 is the execution address. Parameter #2 is the breakpoint address. Only one breakpoint may be set at a time with the "E" command. To execute a user's program, only parameter #1 need be specified. To insert a breakpoint in a user program, both parameters must be entered. For example:

```
.E 1234,12FF <Carriage Return>
```

will execute the user's program starting at address "1234 hex" with a breakpoint inserted at address "12FF hex". The user's program should have some sort of executable code at the execution address. When the user's program executes the instruction beginning at the breakpoint address, DDT80 will then receive control, save all of the user's registers and then go into the command mode of DDT80.

At this point, the way in which breakpoints are implemented should be discussed. When the "E" command sets a breakpoint, it saves the byte of data at the breakpoint address. This is for later restoral after the breakpoint has been encountered. The "E" command inserts a RESTART 7 instruction at the breakpoint address. The RESTART 7 instruction will, when encountered to be executed, transfer control to location "0038 hex". At that location the "E" command will insert a jump back into DDT80. In this way DDT80 can receive control from a user's program and still remember where in the user's program it left off.

Knowing this about breakpoints, there are a few rules to follow when using breakpoints. Since the RESTART 7

instruction transfers control to 0038 hex, no user's code can be at 0038, 0039, or 003A hex. These three bytes are reserved for the jump instruction back into DDT80. It should be remembered that this restriction only exists if the user is using breakpoints to debug his program. Another restriction to follow is that DDT80 has a RAM storage area beginning at 0100 hex and ending at 0138 hex. This area should not be used by a user's program that is receiving control from DDT80. The Sorcerer Power-On Monitor should be used to transfer control to programs executing in this area of memory. An easy method of program development would consist of assembling and linking a program at an address above the DDT80 RAM area while in the debugging process and then, after the program is completly debugged, relink it at the actual execution address.

The "E" command is also used to transfer control to each of the other modules in the Development Pac. The Linking Loader is a special case, however and is not given control by the "E" command. To execute another module, such as the Editor, the user should first use the "M" command to set up the appropriate I/O vector channels for the appropriate module and then use the "E" command, substituting the appropriate mnemonic for parameter #1. For example:

.E :ED <Carriage Return>

will execute the EDITOR by looking up the Editor's address from the mnemonic table.

THE "H" COMMAND

DDT80 has an arithmetic capability that allows hexidecimal addition and subtraction. This arithmetic capability can be used with any of the above commands in place of a parameter in the command line. DDT80 will process the series of additions and subtractions to form a single 16 bit value to use in place of the parameter. The "H" command allows for using this arithmetic capability without affecting any of the other commands.

The form of the "H" command is:

```
============================================================
  .H +operand+operand-.....+operand=zzzz <Carriage Return>
============================================================
```

The user would enter the "H" and then the arithmetic expression. Only the "+" and "-" are legal operations. If the sign of the first operand is omitted, it is assumed to be +. The "=" (equal) causes the 4 digit (least significant 4 digits) result to be displayed. When the <Carriage Return> is entered, DDT80 returns to accept another command.

For example:

```
.H 1234+32-AABF+DDD3=457A <Carriage Return>

._
```

It should be noted that the user does not have arithmetic capability of only the "H" command in DDT80. Any of the DDT80 commands will perform hexadecimal arithmetic to form its parameters.

THE "L" COMMAND

The "L" command is the loader transfer command of DDT80. It operates similarly to the "E" command in that it is used to transfer control to another program. However, this command is a special case for transfer operations.

When the Linking Loader is called from DDT80, it is used to load object modules that were created by the Assembler into the Sorcerer's RAM area. As the Loader does this, it maintains a "GLOBAL" symbol table to resolve all references among the modules that it is loading. Thus, the loader uses two areas of RAM. One for the actual program object code, and the other for the module "GLOBAL" symbol table. The program object code "grows" up in memory. That is, if it starts at location 0000 hex, the next byte loaded will be at 0001 hex. The "GLOBAL" symbol table "grows" down in memory. That is, if it starts at 0100 hex, the next byte loaded into the table will be at 00FF hex.

The format for the "L" command is:

```
============================================================
      .L Parameter #1, Parameter # 2 <Carriage Return>
============================================================
```

Parameter # 1 specifies the beginning of the program object code. It supplies a "base" address that the Loader will add to each relocatable module that it loads in.

Parameter # 2 specifies the beginning of the "GLOBAL" symbol table. It should be remembered that this value should be a "high" value that the table will "grow" down from.

Altogether, DDT80 provides a simple, efficient and powerful means of debugging and executing user programs in the Development Pac.

SORCERER I/O

Each module in the Development Pac communicates with the Sorcerer through the Vector channels. Each one of the six Vector channels can be assigned to any address in the Sorcerer by use of the "M" command in DDT80. (See the "M" command in DDT80 for details.) For the convenience of the user, some of the most commonly used I/O drivers that communicate with the Sorcerer have been included in the module SORCERERIO. All of the included Sorcerer I/O drivers are:

```
=================== SORCERER I/O DRIVERS ===================
      :SK - Sorcerer Keyboard
      :SV - Sorcerer Video Screen
      :I1 - Input from Sorcerer Cassette Unit # 1
      :I2 - Input from Sorcerer Cassette Unit # 2
      :O1 - Output to Sorcerer Cassette Unit # 1
      :O2 - Output to Sorcerer Cassette Unit # 2
      :CE - Output to Sorcerer Centronics Driver
      :AI - Input to the "A" Memory Buffer (Object)
      :AO - Output from the "A" Memory Buffer (Object)
      :BI - Dummy Input into "B" Memory Buffer
      :BO - Output from the "B" Memory Buffer (Source)
============================================================
```

Each module in the Development Pac communicates to the user through the vector channels. As the module needs to output information to the user or get information from the user, it transfers control to the appropriate vector point. This vector point will then transfer control to the physical device driver that is assigned to it. That driver may be any of the device drivers listed above or one that the user has written and located somewhere in memory. The user can even use the Development Pac to write and assemble the device driver!

Writing I/O Drivers

An I/O driver in the Development Pac is used to pass a character to or receive a character from some sort of device. For the sake of this example we will use the Sorcerer video screen as an output device and write an I/O driver for it. Note: there is already an I/O driver for the Sorcerer screen (:SV) but, it is recreated here for demonstration.

After the vector channel has passed control to the device driver, three registers contain information for the device driver to use. These registers are:

I/O DRIVER REGISTER USAGE:

REGISTER	ON INPUT	ON OUTPUT
A	DATA	NOT USED
D	DATA	DATA
E	CONTROL	CONTROL

The "A","H" and "L" registers are free to be used or destroyed by the I/O driver. All other registers should be intact when the I/O driver is exited. Upon entry to the driver a valid stack will be established. This may be used for pushing registers or calling subroutines. The stack must be maintained and returned to the proper level before executing the "RET" instruction to exit the I/O driver.

The data for the I/O device should be passed or received from the "A" or "D" register. For output, the data will be in the "D" register only. For input, the data should be returned in both the "A" and "D" registers.

The "E" register is used to pass control parameters to the I/O device driver. Each bit in the "E" register is used to "flag" some sort of control operation. For example:

CONTROL REGISTER

BIT	USAGE
0	NOT USED
1	NOT USED
2	NOT USED
3	OPEN DEVICE FLAG
4	CLOSE DEVICE FLAG
5	DUAL DEVICE SELECT FLAG
6	NOT USED
7	IMMEDIATE RETURN FLAG

Bit # 3, Open Device Flag, is set to a one when the Development Pac is using the device for the first time or wants an iteration or a "rewind" operation performed. When the driver sees this bit set, it should perform any operation necessary to "rewind" the device- Such as closing and opening a disk file or issuing a message to the user to tell him to rewind the cassette units. If the device has no provisions for "opening", then this bit may be ignored.

Bit # 4, Close Device Flag, is set to a one when the Development Pac is finished with the device for that module. This flag should be used for such things as writing out unfilled buffers to the device or closing a disk file that was previously opened. Again, if the device has no provisions for "closing", then this bit may be ignored.

Bit # 5, Dual Device Select Flag, is used internally by the I/O driver to select one of two devices that can use the same I/O driver. For instance, the two Sorcerer cassette units or a dual drive disk system. This flag is only an internal flag to the device driver and is not set by the Development Pac. It is used in the Sorcerer cassette unit drivers, but may be used by any driver the user wishes to write.

Bit # 7, Immediate Return Flag, is set to a one when the Development Pac wants to "bypass" the I/O driver. This flag is not used in the present version of the Development Pac, but is mentioned here and may be included in future releases.

Now, let's return to writing the I/O driver for the Sorcerer video screen. The easiest way to send a character to the Sorcerer screen is to call the Sorcerer's Power-On Monitor Video Driver routine. There is only one problem with using this method. The Power-On Monitor expects the character to be in the "A" register and the Development Pac Vector channel sends the character in the "D" register. So, before the Power-On Monitor is called, the "D" register will have to be loaded into the "A" register. Knowing this, the Sorcerer Video I/O driver would be written:

```
7A          LD      A,D
CD 1B E0    CALL    VIDEO
C9          RET
```

That's all there is to it!

Hopefully, this section contains enough information for the user to write an I/O driver for any device he may interface to his Sorcerer. After the driver is written, it should be loaded at some memory location not used by the Develoment Pac, such as 0000 to 00FF hex, and then enabled by assigning it to any of the Vector channels by use of the "M" command in DDT80. For example, the user could interface

an RS232 printer, assign it to the Source Output channel (:SO), and have all of the assembly listings printed on it.

For the convience of not having to reload them every time the Sorcerer is turned on, most of the common I/O drivers for the Sorcerer have been included in the Development Pac. Only two of them are assigned by DDT80 when it is first turned on. These are :SK and :SV for the Vector channels :CI and :CO respectfully. This provides a means of immediate communication to the Development Pac. It should be remembered that these devices do not necessarily have to be left connected to these Vector channels, but may be reassigned to some other Vector channel or have some other device assigned to the Console Vector channels. Also, any device may be connected to more than one Vector channel at one time.

Next, each of the included device drivers will be discussed in more detail.

THE DRIVERS

:SK - THE SORCERER KEYBOARD DRIVER

This driver will scan the Sorcerer keyboard until any key is pressed. It will then return with the ASCII code for that key in the appropriate registers. This driver may be assigned to any input Vector channel. It should be noted that this driver may be assigned to the Source Input channel (:SI) during an assembly to create a "dynamic" assembler. This provides a great Z80 opcode learning tool for the beginner!

:SV - THE SORCERER VIDEO DRIVER

This driver was demonstrated above. It will pass the character in the data register to the Sorcerer screen, performing any necessary scrolling or erasing operation. This driver may be assigned to any output Vector channel.

:I1 - CASSETTE UNIT # 1 INPUT DRIVER

This driver will perform all of the buffering necessary to use the cassette units. Each time the driver is called with the "open" flag set, it will print the following message on the Console Output channel (:CO):

REWIND CASSETTE. HIT ANY KEY WHEN READY._

The user should then stop the cassette recorder, rewind the tape to the beginning of the file, put the recorder back on "play" or "record" and then hit any key on the device connected to the Console Input channel (:CI). The same message will be printed for both input and output operations. So, the user should keep track of which units are performing which operation.

When the cassette driver is called with the "close" flag set, it will output the remaining contents of the buffer to the cassette unit and then return.

It is recommended that the user use the Serial Data cable with the cassette routines provided for the Development Pac; however, it is not necessary, provided that long source files are not read into the Assembler. This is because the Development Pac provides for full motor control of both cassette units. It can then "buffer" the output and input in 256 byte blocks.

The cassette files maintained by the Development Pac

are not compatable with the cassette files maintained by the Sorcerer's Power-On Monitor. The Development Pac uses only ASCII formatted files for source and hex files. There are no provisions for saving machine code files in the Development Pac. Instead, the Linking Loader returns control to the Power-On Monitor after loading a program. It can then be saved using the tape I/O routines in the Power-On Monitor.

All of the cassette files used and created by the Development Pac are unnamed. That is to say that they have no real beginning point; however, they should always be read in from their physical starting point on the tape to avoid reading from the middle of a block on the tape. All of the files have an "end of file" marker on them that is used by the Assembler, Linker and Editor to detect the end of a file.

Extreme care should be exercised when using the cassette interface with the Development Pac. The user should make certain that all volume and tone settings on the recorder are correct and always be aware of which cassette unit is being used for play-back and which unit is being used for recording. Rewinding the wrong unit at the wrong time could result in a loss of information!.

The user should also be careful to have the tape head positioned off leader in the tape during a "record" operation. Always make sure after rewinding a cassette unit, to move the tape forward and off the leader before putting the recorder on "record".

:I2 - CASSETTE UNIT # 2 INPUT DRIVER

This driver performs the exact same operations as the driver described above, except this driver will input characters from cassette unit #2. It should be noted that the user should never use both cassette input drivers at the same time. Doing so would result in a buffer conflict and a probable loss of information.

:O1 - CASSETTE UNIT # 1 OUTPUT DRIVER

This driver performs the exact same operation as the driver described above, except this driver will output characters to cassette unit # 1. It should be noted that the user should never use both cassette output drivers at the same time.

:O2 - CASSETTE UNIT # 2 OUTPUT DRIVER

This driver is the same as the one above, except this driver will output characters to cassette unit # 2.

:CE - CENTRONICS PRINTER DRIVER

This driver is used to print on a Centronics type printer that has been connected to the Sorcerer's parallel port according to the specifications described in the Sorcerer technical manual. This driver will output to only the printer and not the Sorcerer screen.

THE RAM BUFFERS

To speed the development process of writing programs, RAM buffer drivers have been included with the Development Pac. These RAM buffer drivers are extremely fast and provide a very quick and efficient means of data storage. However, the user must use care when using the buffers to insure that they are not inadvertantly destroyed.

When DDT80 first receives control it partitions the available memory in the Sorcerer into four parts. These four parts are formed as follows:

Partition # 1 = Top of RAM - 768 bytes TO Top of RAM

This partition is for I/O storage and buffering and stack space. From the top of RAM down 256 bytes is the Sorcerer Power-On Monitor data area and stack. This stack is also used by the Development Pac. From that point down for the next 512 bytes are two 256 byte cassette unit buffers. One for input and one for output. The ending address of this partition will normally be the top of RAM for the Sorcerer being used located in locations F000 and F001 hex. The beginning address for this partition is stored by DDT80 in locations 0136 and 0137 hex. This partition should always be left intact and never loaded into by any of the modules in the Development Pac. There are no buffer drivers for this area of memory.

Partition # 2 =(Partition #1 to 0000 hex)/2 TO Partition #1

This partition is called the "B" memory buffer. It is normally used in the Development Pac for source code storage. The Editor uses this partition for its buffer and leaves with the buffer intact for later reentry and modification. As long as this partition is not overwritten or loaded into, the user may reenter the editor and correct or modify any source code line very quickly without the need to reload the entire text. There are two drivers for this partition available to the user, :BI and :BO. :BI is not really a driver, but a "dummy" driver. This is because the user never really needs to "input" into the "B" buffer. The Editor "I" command or "R" command should always be used to "input" into this buffer. The :BI driver will simply return ETX characters (03 hex) to signify that there is no input. The :BO will output each character in the "B" buffer one at a time. When the "open" flag is set the buffer pointer will be reset to the beginning of the buffer.

This partition will occupy approximately 1/2 of the available memory in the Sorcerer. The beginning address for this partition is stored by DDT80 in locations 0134 and 0135 hex. The ending address for this partition is the beginning of partition # 1.

Partition # 3 = (Partition #2 to 0000 hex)/2 TO Partition # 2

This partition is called the "A" memory buffer. It is used in the Development Pac mainly as the object buffer. It is normally assigned to the Assembler Object Output Vector to accept the hex file and then to the Loader Object Input Vector to load the hex file. This provides for very quick assembly and loading. There are two drivers provided for this buffer, :AI and :AO. :AI is used to input characters into the buffer and :AO is used to output characters from the buffer.

This partition occupies approximately 1/4 of the available memory in the Sorcerer. The beginning address of this partition is stored by DDT80 at locations 0132 and 0133 hex. The ending address of this partition is the beginning of partition # 2.

Partition # 4 = 0100 hex to Partition # 3

This partition is used for storage by DDT80, the Editor and the Assembler. The Assembler symbol table is also formed in this partition. The Loader does not have any storage in this partition and can therefore load user programs in this area of memory. A normal mode of operation with the Development Pac is to write the source code for a program in the "B" buffer (partition # 2), assemble it placing the hex file in the "A" buffer (partition # 3), and then load it into partition # 4.

DDT80 uses locations 0100 hex to 0138 hex. The Assembler and Editor use locations 0139 hex on up. The user may assemble and edit his program to run at any address from 0139 hex to the end of partition # 4. In this way the user has access to the powerful breakpoint and register commands in DDT80 to debug his program. If the user then finds a "bug" in his program, he can easily reenter the Editor, modify his program, reassemble it and then reload it for debugging again! All in the span of a few seconds!

It should be remembered that these partitions are set up to allow the user the flexibility of a very fast and effecient development process. However, the user is not restricted to use this RAM based mode of developing a program. If the user's program is too large to fit in the size Sorcerer he has or if the user wants to run his program in one of the other partitions, he may use the cassette drivers to "spool" his files and therefore not use the partitions at all! This makes development more time

consuming, but allows for more flexibility in program size and running parameters.

:AI - "A" BUFFER INPUT DRIVER (OBJECT)

This driver will input characters into the "A" memory buffer described above. When this driver is called with the "open" flag set in the control register, the memory pointer is reset to the beginning of the buffer and the character in the data register is loaded into the buffer.

:AO - "A" BUFFER OUTPUT DRIVER (OBJECT)

This driver will output characters from the "A" memory buffer described above. When this driver is called with the "open" flag set in the control register, the memory pointer is reset to the beginning of the buffer and the first character in the buffer is returned.

:BI - DUMMY DRIVER (RETURNS ETX (03 hex) CHARACTERS)

This driver will always return with an ETX character (03 hex) in the data register. The user must assign the :BI driver to the :SI channel when creating a new file in the Editor. It is used with the Source Input channel to tell the Editor that there is no input file when creating a new file. The Editor's "E" command will read all input from the :SI channel into the "B" buffer before returning to DDT80.

:BO - "B" MEMORY BUFFER OUTPUT DRIVER (SOURCE)

This driver outputs characters from the "B" memory buffer described above. When this driver is called with the "open" flag set in the control register, the memory pointer will be positioned to the beginning of the "B" memory buffer and the first character in the buffer will be returned in the data registers.

The user should study the drivers listed above and experiment with each one to learn its full potential. By using the drivers effeciently, the user will have a very powerful machine language development tool!

EDITOR

The Development Pac has a line-oriented Text Editor for creating and altering ASCII source files. A source file is made by calling the Editor and entering the source lines from the console. It can later be corrected by reading the source in and adding or deleting code.

The Text Editor commands are summarized below:

========== EDITOR COMMANDS: ==========

```
    B - point to the beginning of text
n<CR> - move line pointer n lines and display
    I - insert source lines
   nD - delete n source lines
   nT - type n source lines on console display
   nR - read n source lines into the buffer
   nW - write out n source lines from the buffer
    E - exit from editor and close file
```

==========

TEXT BUFFER and LINE POINTER

This is all implemented through the use of a text buffer. The size restraints of this buffer are determined by the available RAM in the Sorcerer that the Pac is being used with. As a rule of thumb, the text buffer can take up just under half of the available RAM area. The actual size of the buffer at any given time depends on the amount of source code it contains; the buffer expands as it is being filled with source lines until the maximum byte count allowed is reached.

So that the lines of text within the buffer can be manipulated, a conceptual (or imaginary) line pointer is maintained. The pointer is always positioned at the beginning or end of the buffer or at the first character of a source line; a source line, hereon referred to as "line"- is defined as a string of ASCII characters ending with <CR><LF>. Generally, the pointer is positioned at the point in the buffer where the next action of the editor will be initiated. This pointer can be moved by the user.

The text buffer operation is relatively simple. Upon entry to the Text Editor (not reentry, which is explained later), the buffer is empty. At this time, the pointer points to both the beginning and the end of the buffer. If the user is creating a new source file, lines will be inserted into the text buffer through the console input channel (:CI) before the line pointer. Therefore, the pointer will point to the end of the buffer when the user stops inserting. If the user wishes to then edit these lines, the pointer can be moved and other operations may be done.

If the user is changing or updating an already existing source file, lines can be read from an input device, such as a cassette player, into the text buffer via the source input channel (:SI). These lines will be put at the end of the buffer; so if it is empty, they will be the sole contents of the buffer. The pointer position will remain unchanged.

If the buffer expands to its maximum size and the user wishes to insert more lines into the text or read more lines in from the source input channel, some of the lines in the buffer will have to be written out. This is done via the source output channel (:SO) to a device such as a second cassette unit. Lines are written from the beginning of the buffer and the line pointer is not affected. The user can then delete these lines from the buffer and continue line insertion or read in more lines at the end of the buffer.

When the user wishes to leave the Text Editor and

renter DDT80, all the lines in the text buffer are written out through the source output channel. If there are any more lines that haven't been read in via the source input channel yet (unprocessed source lines), they are read and output to the source output channel.

The user can renter the Text Editor at a later time if so desired. The contents of the text buffer will usually be unchanged (exceptions will be explained later); whatever was in it upon leaving the editor previously will still be there. The pointer will be positioned at the beginning of the buffer.

LINE EDITING

As the user is entering commands to the editor or source lines to be placed in the buffer from the console input channel, line editing is available. Before the <CR> is typed in by the user, the line is maintained in a line buffer; when the <CR> is entered, this buffer is effectively emptied until a new line is started. Within this buffer, characters can be deleted or tabulations can occur.

Typing SHIFT/RUB will delete the last character in the buffer entered by the user and will move the curser back one position on the console output channel (:CO), usually a video display. Tab stops are set at character positions 9, 17, 25, 33, 41, 49, and 57 of the line. Hitting the TAB-SKIP key will cause spaces to be inserted in the line buffer from the curser's current position up to but not including the next tab stop position. On the console out channel, the curser will jump to the next tab stop. Tabs mistakenly entered can be deleted, but they must be done so space by space.

The line buffer is 64 characters long but the 64th is never used by the user. The editor will always insert a <LF> after a user's <CR> and a byte must be reserved. When at position 63, the user can only enter a SHIFT/RUB or <CR>. Therefore, excluding the <CR><LF>, the user can enter lines including up to 62 characters. It should be noted, however, that all 62 characters are not necessarily displayed via the console output channel. Depending on what device is assigned, some characters, particularly control characters, may not be displayed; these characters are entered into the buffer though and are counted as characters in the line even if the user cannot see them.

EDITOR I/O

To call the Text Editor, the user must be in DDT80. Before calling the Editor though, the I/O channels should be set up. The method of assigning devices to the I/O channels is explained in the section on DDT80.

When a source file is first created, all source lines ae entered through the :CI channel. So are all Editor commands. The Editor's prompt, any messages it has for the user, any lines the Editor is commanded to display, and echoes of every character entered by the user via the :CI channel are output through the :CO channel. The :OI and :OO channels are not used by the Editor. The :SI channel is used to input, to the Editor, the user's source file. When the user is creating a new source file and there is no source to be input, a dummy input driver, :BI, should be assigned to :SI. This will keep the Editor from looking for the end of a source file, an <ETX> character (which in this case would not exist since no source exists), when the user exits from the Editor. The :SO channel is the channel through which the user's edited file is output upon leaving the Editor. Also, anytime the user writes lines from the text buffer, they go out through the :SO channel.

As an example, if the user was working in the cassette-based mode of operation, the channels could be assigned as follows:

```
:CI=:SK
:CO=:SV
:SI=:I1 (or :I2 for unit # 2)
:SO=:O2 (or :O1 for unit # 1)
```

The user here communicates with the Editor, and vice versa, through the Sorcerer keyboard and video display. The source input comes from cassette unit 1 and the source output is sent to cassette unit 2.

If the user was working in the RAM-based mode of operation, the channels would normally be assigned as shown:

```
:CI=:SK
:CO=:SV
:SI=:BI
:SO=:SV
```

As in the above example, the user and the Editor would communicate through the Sorcerer. :BI would be the dummy driver for source input unless the user had a file to read into the buffer; then the driver for the correct device

would be assigned. :SV is used as the source output channel so when the user leaves the Editor, the source is just displayed on the screen and still remains in the buffer RAM. It should be noted here that no matter what device is assigned to the :SO channel, after leaving the Editor, the text buffer remains unchanged. The only ways to alter its contents besides entering the Editor again and updating the buffer are to use the Relocating Linking Loader to load into that area, to run a user program that writes into that area in RAM, or to use the DDT80 M" command to modify that area of RAM.

The following shows what devices can be assigned to the channels the Editor uses:

:CI	:CO	:SI	:SO
:I1	:O1	:I1	:O1
:I2	:O2	:I2	:O2
:SK	:SV	:SK	:SV
:AO	:AI	:AO	:AI
:BO	:CE	:BO	:CE
		:BI	

Appendix C specifies the driver symbol for each device.

CALLING THE EDITOR

After the I/O channels have been set up in DDT80, the user can then call the Text Editor. To enter at the normal entry point, the user types

.E :ED<CR>

and the Editor prompt, an asterisk, will appear when the Editor is ready to receive commands. To enter the Editor at the reentry point, the user should type

.E :ER<CR>

and the Editor will respond with its prompt. The line pointer is positioned at the beginning of the buffer when either entry point is used.

EDITOR COMMANDS

The editor commands, which were summarized previously, give the user many capabilities in the manipulation of source files. Following is a description of each command including command format, operation, possible restrictions, and editor errors, if any.

B command

The format for this command is

B

The command may be on the same command input line with other commands, and there are no positional restrictions. The command will also work if a decimal number is on the line before the command letter, but the number has no effect. The result of this command is that the line pointer is moved to the beginning of the buffer. If it is already there, nothing happens. This command is important because the operation of the other commands only moves the pointer forward, if at all. This is the only way to move the pointer backward in the buffer.

<CR> command

The format for this command is

n<CR>

where n is a decimal integer in the range 0 <= n <= 65535. If n=0 or isn't included before the command, n=1 is assumed. This is the same for all commands that use the value n. (NOTE: If either the single quote character or the @ character is entered on a command input line where a number could be entered, they will be interpreted as decimal digits with the values 247 and 9 respectively.) The <CR> command must be the only command in a command input line. Otherwise, it is assumed to be just a delimeter for the line and n is ignored.

This command moves the buffer pointer forward n lines in the buffer with no effect upon the text. The line that the pointer is left pointing at is displayed via the :CO channel. If the value of n is too large relative to the number of lines in the buffer, the pointer points to the end of the text buffer and nothing is displayed.

I command

The format for this command is

I<CR>
lines of source delineated by <CR>'s
<ESC>

The command may be on the same command input line with other commands, but it should be the last command on the line. Anything entered in the command input string after the command letter and before the <CR> is ignored. The <ESC> character is entered by the user to terminate the command and it must be the only character entered on its line. This prohibits insertion of anything but entire lines delineated by <CR>'s.

The lines of source entered on lines after the command letter's line and before the line containing <ESC> are inserted into the text buffer before the current line pointer position. This position doesn't change. As each line is entered through the :CI channel, the editor makes sure there is room in the buffer for it. If a line is entered to be inserted that would cause the buffer to overflow, the editor ends the insert command, does not insert ANY of that line, and sends a message via the :CO channel letting the user know the text buffer is full. If the user wishes to continue insertion, lines will first have to be removed from the buffer.

D command

The format for this command is

nD

where n is as defined above in the explanation of the <CR> command. This command may appear anywhere in a command input line. This command deletes n lines from the text buffer starting with the current line; the current line is defined as the line that the line pointer is presently pointing to. The pointer is left pointing to the first undeleted line after what was the current line before the command was executed. If the value of n is larger than the number of lines available to delete, the lines that can be deleted are deleted and the pointer is left positioned at the end of the buffer.

T command

This command's format is

nT

where n is as previously defined. This command displays, via the :CO channel, the next n lines in the buffer starting with the current line. The line pointer position and the buffer state remain unchanged. If n is too large, only the lines available to display are shown.

R command

This command's format is

nR

This command causes the next n unprocessed source lines to be read into the buffer. These lines are taken from the :SI channel and are placed at the end of whatever text is already residing in the buffer RAM. The line pointer position remains unchanged.

If the <ETX> character (03H) defining the end of the source input file is found, the read command is terminated. If the <ETX> character has already been found during a previous read command, the command has no effect.

If the value of n is too large relative to possible buffer size, lines are read in until there are only about three lines worth of space left available in the buffer. After the last full line has been read in, the Editor outputs through the :CO channel a message letting the user know the buffer is full. The extra bytes not filled are available to the user if line insertion is necessary, although not too many lines can be inserted before the buffer becomes completely full.

W command

The format for this command is

nW

This command causes the first n lines in the text buffer to be written out via the :SO channel. The line pointer position is not changed and neither is the state of the buffer. If n is larger than the number of lines in the buffer, the entire buffer is written out. The user has the option of deleting the lines after they are written out by using the D command.

E command

The format for this command is

E<CR>

This command must be the only command on the line. It will work if there is an integer on the line directly before it although the number has no effect. If there are other commands on the line, the E command is ignored as are any other commands on the line that were entered after the E command. This is so that the user doesn't inadvertantly exit from the editor.

This command ends the editing session and exits the Editor. First, the buffer contents are written out through the :SO channel. If the <ETX> character defining the end of a source input file has not yet been found, characters are then read from the :SI channel and written to the :SO channel until an <ETX> character is found. All the unprocessed source lines are written out as is. If there is no source input file, the dummy driver, :BI, should have been assigned to the :SI channel. This driver just sends out <ETX> characters. After the output file is closed, the Editor writes out an ETX character (03 hex) to the output file and returns control to DDT80.

AN EDITING EXAMPLE

A typical editing session consists of either creating a file or updating one. Examples of both will be shown here. First, creating a file will be demonstrated. Recall that the Editor's prompt is displayed by the Editor and is not entered by the user; when it appears in the following examples, it can be assumed that the Editor output it. The same goes for DDT80's prompt, the period. When displaying what the user is entering, non-printable characters are shown in <>'s. When showing the :CO display, exactly what the user would see is shown.

For this example, the Sorcerer is to be assigned to the console channels, the dummy driver, :BI, should be assigned to the :SI channel, and cassette unit # 1 is to be assigned to the :SO channel. The user first turns on the Sorcerer with the Development Pac inserted. The following will then appear.

```
EXIDY Z80 DEVELOPMENT PAC
    COPYRIGHT (C) 1979

.
```

The I/O drivers are assigned:

```
.M :CI<CR>
:CI  :SK  <CR>
:CO  :SV  <CR>
:OI  XXXX <CR>
:OO  XXXX <CR>
:SI  XXXX :BI^
:SI  :BI  <CR>
:SO  XXXX :O1^
:SO  :O1  <CR>
FF2A XX .
.
```

The user then calls the Text Editor:

```
.E :ED<CR>
```

and the Editor signs on:

```
*
```

The user is going to create a file containing a small subroutine, so the code is inserted as follows:

```
*I<CR>
;<CR>
;THIS SUBROUTINE DETERMINES IF THE ASCII<CR>
;CHAR. IN A IS A LETTER.  ON RETURN, IF<CR>
;IT IS, A'S CONTENTS ARE UNCHANGED.  IF<CR>
;NOT, A WILL CONTAIN ZERO.<CR>
ALPHA<TAB>EQU<TAB>$<CR>
```

```
        <TAB>CP<TAB>05BH<CR>
```
On the next line, the user enters the wrong code:
```
        <TAB>JRNC
```
and corrects it with the line editing capabilities. Two SHIFT/RUB's are entered:
```
        <TAB>JR
```
and the user continues:
```
        <TAB>JR<TAB>NC,NOAL-$<CR>
        <TAB>CP<TAB>041H<CR>
        <TAB>RET<TAB>NC<CR>
        <TAB>XOR<TAB>A<CR>
        <TAB>RET<CR>
        <ESC>
```
The pointer is now positioned at the end of the buffer. The user will generally want to check and see if the code entered is correct. The pointer must then be moved to the beginning of the buffer so the buffer contents can be displayed. By using a large value for n, the user can insure that the entire buffer contents are shown. The user enters:
```
        *B200T<CR>
```
and the Editor responds with the following:
```
        ;
        ;THIS SUBROUTINE DETERMINES IF THE ASCII
        ;CHAR. IN A IS A LETTER.  ON RETURN, IF
        ;IT IS, A'S CONTENTS ARE UNCHANGED.  IF
        ;NOT, A WILL CONTAIN ZERO.
        ALPHA   EQU     $
                CP      05BH
                JR      NC,NOAL-$
                CP      041H
                RET     NC
                XOR     A
                RET
```
The user is satisfied and leaves the Editor writing the Editor buffer out to cassette unit # 1 through the :SO channel:
```
        *E<CR>
        REWIND CASSETTE. HIT ANY KEY WHEN READY.<CR>
```
After the program has been assembled, the user realizes that a label was forgotten. This example will show a file being updated. First, the :SI channel assignment is changed to cassette unit # 1 and the :SO channel assignment is changed to cassette unit # 2:
```
        .M :SI<CR>
        :SI   :BI   :I1^
        :SI   :I1   <CR>
        :SO   :O1   :O2^
        :SO   :O2   <CR>
        FF2A XX .
        .
```
Then, the user enters the Editor again:

```
.E :ED<CR>
```

The file must be brought into the buffer from cassette unit # 1. A large value is used for n to insure all source lines are read in:

```
*200R<CR>
REWIND CASSETTE. HIT ANY KEY WHEN READY.<CR>
```

The buffer pointer, unchanged by the R command, is positioned at the beginning of the buffer, but the line that needs to be corrected is the eleventh line. The pointer must be moved:

```
*10<CR>
```

and the Editor responds:

```
        XOR     A
```

Now, the line is deleted and entered again properly:

```
*DI<CR>
NOAL<TAB>XOR<TAB>A<CR>
<ESC>
```

The user checks the line by positioning the pointer before the line again and displaying it. Remember that after insertion, the pointer was positioned after the inserted line.:

```
*B<CR>
*10<CR>
NOAL    XOR     A
```

The user writes the entire buffer out to cassette unit # 2 and exits:

```
*500W<CR>
REWIND CASSETTE. HIT ANY KEY WHEN READY.<CR>
*E<CR>
```

Normally, it isn't necessary to write out the buffer before leaving the Editor, but this was done here for purposes of example.

ASSEMBLER

The Development Pac has a powerful Z80 Relocating Assembler which, in conjunction with the Text Editor and Relocating Linking Loader, provides the means for editing, assembling, and loading Z80 assembly language programs. The Z80 Relocating Assembler reads Z80 assembly language source code and outputs an assembly listing and object code. The Assembler recognizes all standard Z80 source mnemonics and supports global symbols and relocatable programs. The object code is industry standard hexadecimal format modified for relocatable, linkable assemblies. The Assembler with the Text Editor and the Relocating Linking Loader provide the user with state-of-the-art software for building, assembling, and loading Z80 programs. The Z80 Relocating Assembler can assemble any length program, limited only by the symbol table size. This is determined by the size of RAM in the Sorcerer the Pac is being used with. A little less than one-fourth of the Sorcerer's RAM is available to be used for the symbol table.

BASIC DEFINITIONS

There are a few terms used quite often when discussing the Z80 Relocating Assembler that the user must understand in order to facilitate proper usage. These will be explained here.

The term "module" is often used interchangably with the term "program," and both refer to a unit of code that is either worked with or produced by the Editor, the Loader, or the Assembler. A source module is the user's source program, lines of text created by the Text Editor and output upon exit from it. Each source module is assembled into one object module by the Assembler. The end of a source module is defined by an ETX character (03H). An object module is produced by the Assembler from a source module. Each object module contains machine code, linking information, address and relocating information, and checksum information (all in ASCII) that is used by the Relocating Linking Loader. A description of its format can be found in Appendix A. A load module is the binary code of one complete program. It is created by the Relocating Linking Loader from one or more object modules and is generally defined in RAM.

When working with groups of relocatable modules, as opposed to single absolute modules, much more significance is attached to symbols and their usage. A local symbol is a symbol in a source module that appears in the label field of a source statement. That symbol is given an offset into the module by the Assembler when the module is assembled. The symbol can be used within that module, obviously, but it can also be used by other modules.

This is brought about by global definition which occurs when a symbol appears in the operand field of the assember pseudo-op GLOBAL in a source module. Note that a symbol does not necessarily have to be a local symbol in a source module in which it is given global definition. Any symbol which is made a global symbol within a source module will appear in the corresponding object module. If a symbol is defined as a global symbol in some or all object modules in a group of modules that will be loaded together, it can be used within each corresponding source module. When all the object modules are loaded by the Relocating Linking Loader, all references to global symbols from modules in which those symbols were not local symbols will be resolved as the symbols are specifically defined. Global symbol handling is discussed further in the section on Global symbols.

Two terms are used to differentiate between global symbols that are local symbols within a source module and

global symbols which are only referenced within a source module. An internal symbol is a global symbol which is a local symbol in the source module being referred to. When the corresponding object module is loaded by the Relocating Linking Loader, the internal symbol will be defined and relocated, or given a specific value relative to the offset into the module given it by the Assembler. Internal symbols are assumed to be addresses, not constants. When discussing a different source or object module in which the symbol is a global symbol, the symbol is called an external symbol. An external symbol is not a local symbol in the source module. When the object module is loaded, along with the module to which the symbol is internal, the reference to the symbol is resolved. Note that external symbols may not appear in an expression with operators or as the operand of an EQU pseudo-op in a source line.

Just as there are different types of symbols when using a relocatable assembler, there are different types of modules or programs. A position independant program is one written in such a way that it may be placed anywhere in memory and still run properly. No relocating information is needed in the object module, although it may be there anyway. An absolute program is one in which there is no relocation information in the object module. The declaration of a program as absolute through the use of the Assembler pseudo-op PSECT means by definition that no relocation information will appear in the object module. An absolute program may or may not be postition independant; if it isn't, it can only be loaded in one place in memory in order for it to run properly. A relocatable program has extra information in the object module which allows the Relocating Linking Loader to place the program anywhere in memory. A relocatable program also may or may not be position independant, but because of the relocation information in the object module, it doesn't matter which it is. More on relocatable and absolute module handling can be found in the section on Absolute and Relocatable Modules. A linkable program is one in which the object module contains data about internal and external global symbols. The loader uses this to connect, resolve, or link external references to internal symbols in modules. A linkable program may be either absolute or relocatable and either position independant or not. Linking is discussed in more detail in the section on Global Symbols

In reference to the Z80 Relocating Assembler itself, there are some ideas which the user should be familiar with. Especially important is the concept of a two pass assembler, which this assmebler is. The term "two pass" refers to the fact that the Assembler scans each source module that it assembles twice. Each scan is known as a pass. During the first pass, space is allocated in the symbol table for all symbols used in the module. Relative offsets, in relation

to the first byte of the object module produced, are provided in the symbol table for local symbols. A linked list is created for each external global symbol in the module with the beginning of this list placed in the symbol table. During the second pass, using the data picked up during the first pass, the Assembler decodes opcodes, operands, and expressions. This is why external global symbols cannot be used in expressions; only a linked list, and no relative offset, is provided for these symbols and the relative value of the expression cannot be determined. As the assemler assembles each line of source code, it maintains a program counter that counts each byte of object code produced. It is assigned a starting value of zero and can be altered at anytime by the ORG pseudo-op, explained in the section on Pseudo-ops. The Assembler also creates the object module and produces a listing during the second pass. Recall this information when pass one or pass two of the Assembler is discussed later.

ASSEMBLY LANGUAGE SYNTAX

An assembly language program, or a source module, consists of labels, opcodes, pseudo-ops, operands, and comments in a sequence which defines the user's program. The assembly language conventions are discussed here. First, though, a quick mention should be made of delimeters. Labels, opcodes, pseudo-ops, and operands must be separated from each other by one or more ASCII commas or spaces. An easy way to insert spaces is to use the TAB/SKIP which moves the curser to a tab stop and fills in the "spaces" jumped over with ASCII spaces. The ASCII <TAB> character itself is not used in order that source modules produced by the Development Pac be more compatible with other Z80 assemblers. Comments are delineated from the rest of the elements on a line by a semicolon. The following illustrates the source code format:

```
============================================================
     (label) opcode  operand(,operand)        (;comment)
============================================================
```

A. Labels

A label is composed of one or more characters. Only the first six characters of a label with more than six characters are recognized by the Z80 Relocating Assembler. The characters used in a label cannot be any of the non-printable ASCII characters, an ASCII blank, or any of the following characters:

' ( ) * + , - < > = . / : ;

In addition, the first character of a label cannot be a decimal digit. All labels must begin in column one. No colon should be used after the label. Some examples of valid and invalid labels are shown:

```
VALID                    INVALID

LAB                 9LAB     ;STARTS WITH A DECIMAL DIGIT
L923                L)AB     ;ILLEGAL CHARACTER IN LABEL
$23                 L:ABC    ;ILLEGAL CHARACTER IN LABEL
```

A label may be used on any line in the source module. The relative value assigned to the label, assuming it is not before an EQU pseudo-op, is that of the current program counter.

B. Opcodes

There are 74 opcodes, such as "LD"; 25 operand keywords, such as "HL"; and 693 legitimate combinations of opcodes and operands in the Z80 instruction set. The full set of these opcodes is briefly documented in the Z80-CPU Technical Manual and is fully documented in the Z80-Assembly Language Programming Manual, both published by Zilog, Inc., Cupertino Ca. The Z80 Relocating Assembler allows one other opcode which isn't explicitly shown in the Zilog publications:

LABEL IN F,(C)

This instruction sets the Z80 CPU condition bits in the "F" flag register according to the contents of the port defined by the C register.

C. Pseudo-ops

The Z80 Relocating Assembler recognizes seven pseudo-os. These appear in the opcode field of a source statement. Labels for these source lines are optional for all of the pseudo-ops except one. They do not necessarily generate object code, as all opcodes do, but can cause certain values to be loaded into certain bytes or can reserve bytes. All the pseudo-ops direct the Assembler to cause some action to occur.

One of the pseudo-ops which was already mentioned is the PSECT pseudo-op which has the following format:

========== (label) PSECT opr ==========

where opr is the operand. This pseudo-op defines a program section as absolute or relocatable. The pseudo-op should appear before any source lines which can be assembled into object code and should appear only once in any source module. If not included in a source module, the module is assumed to be relocatable. For an absolute module, opr=ABS; and for a relocatable module, opr=REL.

Another pseudo-op already mentioned is the ORG pseudo-op. The format of this pseudo-op is

========== (label) ORG nn ==========

where nn is a sixteen bit value. This sets the program counter to the value nn. When used in an absolute module before any source code which can be assembled to produce object code, ORG determines the starting address for the program. In a relocatable program, ORG provides a base address that can be given an offset when it is loaded. There can be more than one ORG pseudo-op in a source module. This is useful for look-up table placement on even boundries or separating variable RAM areas from program areas. If a source module contains no ORG pseudo-ops, the program counter is set to zero at the beginning of the assembly.

Another pseudo-op which has already been discussed is the GLOBAL pseudo-op with the format

========== (label) GLOBAL symbol ==========

The GLOBAL pseudo-op defines as global the symbol which is its operand. Any symbol referenced from a source module to which it is not local must be defined as global, both in the source module where it is a local symbol and in the modules to which it is an external symbol.

A pseudo-op can be used to assign a value to a label. This is the EQU pseudo-op and has the format

========== label EQU nn ==========

where nn is a sixteen bit value. The unrelocated value of the label is nn. The label cannot be redefined by another EQU pseudo-op or by appearing in the label field of another source statement in the source module. If the label is a global symbol, this restriction also applies to occurances of the symbol in any module which will be loaded with the module where the label is EQUated. The value of that label, a global symbol, is relocated even though it appears as a constant.

Particular bytes within a program can have their contents determined by one of two pseudo-ops, as opposed to having their contents determined by the assembling of a source line and the resultant object code. The first of these is the DEFB pseudo-op which has the following format:

========== (label) DEFB n ==========

The value n is an eight bit value which becomes the contents of the byte located at the current program counter value. Note that this program counter value is the unrelocated value for the location. The other pseudo-op is

========== (label) DEFW nn ==========

where nn is a sixteen bit value. The least significant bits of the value nn are loaded into the byte at the program counter address and the most significant bits are loaded into the byte located one after the byte at the current program counter. These two bytes together comprise what is termed a "word." Recall that an offset may be later added to this program counter value when the object module is loaded. These pseudo-ops are useful for constructing tables and ASCII messages.

If the user wishes to use a certain area of RAM but does not wish to initialize it with values, the DEFS pseudo-op can be used. The format is

========== (label) DEFS nn ==========

where nn is a sixteen bit value. Using this pseudo-op reserves nn bytes of memory starting at the current (unrelocated) program counter value by adding nn to the program counter.

The following is a summary of the Z80 Relocating Assembler pseudo-ops:

PSEUDO-OP			FUNCTION
(label)	PSECT	opr	define module as ABSolute or RELocatable
(label)	ORG	nn	origin-set program counter to nn
(label)	GLOBAL	symbol	define global symbol
label	EQU	nn	equate-set value of label to nn
(label)	DEFB	n	define byte contents as n
(label)	DEFW	nn	define word contents as nn
(label)	DEFS	nn	define storage for nn bytes

D. Operands

There may be zero, one, or two operands present in a source statement depending on the opcode or pseudo-op used. An operand can take one of the following forms: a generic operand, a constant, a label, the "$," or an expression.

A generic operand is a keyword that has special meaning to the Assembler. These keywords are recognized as having only one meaning and should not be used as labels. The following is a list of these operands and their meanings:

OPERAND	MEANING
A	A register (accumulator)
B	B register
C	C register
D	D register
E	E register
F	F register (flags)
AF	AF register pair
AF'	AF' register pair
BC	BC register pair
DE	DE register pair
HL	HL register pair
SP	stack pointer register
$	program counter
I	I register (interrupt vector MS byte)
R	refresh register
IX	IX index register
IY	IY index register
NZ	not zero
Z	zero
NC	not carry
C	carry
PO	parity odd/not overflow
PE	parity even/overflow
P	sign positive
M	sign negative

A constant used as an operand must be in the range 0 through 0FFFFH or 0 through 65,535. There are five types of constants, but the default is decimal. A number can be denoted as decimal by following it with the letter "D". Hexadecimal constants must start with a digit from 0 to 9 and end with the letter "H". Octal constants must end with either of the letters "Q" or "O". A binary constant must end with a "B". ASCII constants are characters enclosed in single quotes and will be converted to their equivilent value ('A'=041H).

Labels may be used as operands with some limitations. The label must either appear elsewhere in the source module or be an external global symbol defined as such in the source module by a GLOBAL pseudo-op. Labels cannot be defined by labels which have not yet appeared in the user program. This is an inherent limitation of a two pass assembler. Labels also cannot be defined by external global symbols. If an external global symbol is used as an operand, it cannot be part of an expression using operators.

ALLOWED			NOT ALLOWED		
I	EQU	7	L	EQU	H
H	EQU	I	H	EQU	I
L	EQU	H	I	EQU	7

The symbol "$" can be used as an operand. It represents the value of the program counter at the current instruction.

The Z80 Relocating Assembler accepts a limited group of expressions as operands in a source statement. Integer two's complement arithmetic is used. All expressions are evaluated from left to right although parenthesis can be used to insure correct expression evaluation. Note that enclosing an entire expression in parenthesis denotes a memory address. The contents of the location equivalent to the expression is used as the operand. There can be any number of terms in an expression. The following are the allowed operators:

OPERATOR	MEANING
+	unary plus
-	two's complement unary minus
+	addition
-	subtraction
.	shift right eight with zero fill

Here are examples of how some expressions using these operators are evaluated:

```
-5        = 0FFFBH
+5        =  0005H
-5-(4+1)  = 0FFF6H
0AABBH    = 0AABBH
0AABBH.   =  00AAH
```

The allowed range of an expression depends upon where it is used. If this range is exceeded, an error message is generated. Usually, the limits on this range are 0 through 0FFFFH. The limits on the range of a relative jump, "JR," are -126 through +129 bytes. When using relative addressing, the current value of the program counter must be subtracted from the label if the branch is to be made to that label address. For example,

```
NAME    JR      NC,LOOP-$
```

will transfer control to the location "LOOP."

For relocatable programs, the Assembler will output relocation information in the object module for all addresses which are to be relocated by the Relocating Linking Loader. The following rules are used to determine if an expression is a relocatable address or a non-relocatable constant; C stands for constant, * represents an operator, and R stands for a relocatable value.

```
C * C=C
C * R=R
R * C=R
R * R=C
```

Shown are examples of these rules in use:

```
I       EQU     1               ;CONSTANT DEFINITION
        DEFW    I               ;NON-RELOCATABLE CONSTANT
LAB     EQU     $               ;RELOCATABLE DEFINITION
         .
         .
         .
        JP      LAB             ;RELOCATABLE OPERAND
        JR      LAB-$           ;CONSTANT OPERAND
        JR      +5+(I)          ;CONSTANT OPERAND
```

It should be mentioned once again that external global symbols connot be used in expressions. External symbols are considered to be relocatable address constants in relocatable programs.

E. Comments

A comment is defined as any characters following a semicolon in a source line and can begin in any column. Comments are used to document source code. They are ignored by the Assembler, but are output in the listing. Note that a semicolon within single quote marks is treated as an expression, not a comment delimeter.

ABSOLUTE AND RELOCATABLE MODULES

A module is defined as absolute by the pseudo-op

(label) PSECT ABS

An absolute object module will be loaded by the Relocating Linking Loader at the exact addresses at which it is assembled. This is useful for software drivers whose positions must always be known, constants, or common blocks of global symbols. A list of global constants can be defined, as shown, and the global symbols will have constant values that may be used by other modules.

```
        PSECT   ABS
        GLOBAL  VIDEO
        GLOBAL  KEYBRD
        GLOBAL  INTAPE
        GLOBAL  OUTAPE
VIDEO   EQU     0E01BH
KEYBRD  EQU     0E018H
INTAPE  EQU     0E00FH
OUTAPE  EQU     0E012H
```

Modules default to relocatable if

(label) PSECT ABS

is not used or if

(label) PSECT REL

is specified. During loading, only sixteen bit address values will be relocated. No eight bit quantities, whether derived from sixteen bit address values or not, will be relocated and neither will sixteen bit constants that aren't internal global symbols. All internal and external global symbols will be relocated. Labels equated to labels which are constants will be treated as constants while labels equated to labels which are relocatable will be relocated. The sample program may help in understanding these principles.

```
ONE     EQU     0A13H       ;ABSOLUTE VALUE
        LD      A,(ONE)     ;ONE NOT RELOCATED
TWO     EQU     $           ;RELOCATABLE VALUE
        LD      A,(TWO)     ;TWO WILL BE RELOCATED
        LD      A,TWO       ;TWO NOT RELOCATED (8 BIT VALUE)
        LD      A,(IX+TWO)  ;TWO NOT RELOCATED (8 BIT VALUE)
THREE   EQU     ONE         ;ABSOLUTE VALUE
        LD      A,(THREE)   ;THREE NOT RELOCATED
FOUR    EQU     TWO         ;RELOCATABLE VALUE
        LD      A,(FOUR)    ;FOUR WILL BE RELOCATED
```

GLOBAL SYMBOLS

When using a relocating assembler, not only is the concept of relocation important to understand, but so are the principles of linkage and how they relate to global symbols. Linkage refers to the process of resolving global references between object modules. To review, a global symbol is one that can be referenced by more than one source or object module. It is given a relative offset into the module in which it is a local symbol by the program counter during assembly and can be used by that and any other module in which the symbol is defined as global. The symbol is an internal global symbol in the module in which it is a local symbol and an external global symbol in any other modules that refer to it.

When object modules are loaded together by the Relocating Linking Loader, all global symbol references are resolved. Each location where an external global symbol is used is modified to the value of the corresponding internal symbol. This is linking. The use of global symbols and linking allow large programs to be broken up into smaller modules.

The object modules produced by the Z80 Relocating Assembler contain the information that the Loader uses. Knowledge of how the Assembler handles global symbols is therefore necessary to obtain proper relocation and linkage. During pass 1, the Assembler recognizes and defines symbols so indicated as global and builds up an external reference link list. For relocatable assemblies, all references to an external global symbol except the first reference in a module are marked relocatable. The object code produced by the Assembler during pass 2 for these references is actually a backward link list terminating with the constant 0FFFFH. All internal global symbols are always marked relocatable, except in absolute assemblies, and will be relocated even if they look like constants. The example demonstrates this point.

```
        PSECT   REL     ;RELOCATABLE MODULE
        GLOBAL  YY      ;INTERNAL SYMBOL
YY      EQU     0AF3H   ;YY ALWAYS MARKED RELOCATABLE
        LD      A,(YY)  ;YY WILL BE RELOCATED WHEN LOADED
```

If the module was absolute, YY would be an absolute value. Both internal and external global symbols in a relocatable assembly are always considered relocatable sixteen bit addresses.

Because of the way global symbols are handled by the

Assembler, there are some rules and limitations associated with their usage. The syntax rules applicable to labels are also applicable to global symbols. In a group of object modules that are to be loaded together, there can be no duplication of internal global symbol names. An internal symbol can be a local symbol in a source module only once in a set of modules that eventually will be loaded together. Because external global symbols are always considered to be sixteen bit addresses, they cannot appear in instructions requiring eight bit operands. External symbols, as mentioned before, cannot appear in an expression with operators either. An external global symbol also cannot appear in the operand field of an EQU pseudo-op. The following examples help show the restrictions on external symbols.

```
        GLOBAL  SYM         ;EXTERNAL SYMBOL
        LD      A,SYM       ;CANNOT BE USED AS 8 BIT CONSTANT
        LD      (IX+SYM),A  ;CANNOT BE USED AS A DISPLACEMENT
        LD      HL,(SYM)    ;LEGAL
        LD      HL,SYM+25   ;CANNOT BE IN AN EXPRESSION
SYM1    EQU     SYM         ;CANNOT BE OPERAND OF "EQU"
```

ASSEMBLER I/O AND OPERATION

Before the user calls the Assembler, the I/O channel assignments must be properly set up. The section on DDT80 describes how this is done.

The Z80 Relocating Assembler uses every I/O channel except the object input channel (:OI). The Assembler and the user communicate basic information via the console channels; the user calls the Assembler through the console input channel (:CI) and any messages to the user regarding I/O or abort errors are sent by the Assembler through the console output channel (:CO).

The object module produced by the Assembler for each source module is output via the object output channel (:OO). The format of this object code is described in Appendix A. This output can be loaded by an Intel hexadecimal loader for non-linkable, non-relocatable programs. Extra information is output in the object modules for relocatable and/or linkable programs for use by the Relocating Linking Loader.

The source input channel (:SI) is the channel through which the Z80 Relocating Assembler reads the user's source module. As mentioned before, the Assembler is a two pass assembler and therefore must scan the source module two times. If an external device, as opposed to a RAM buffer, is used for the :SI channel, the user will have to send the module to the Assembler twice. For example, if a cassette unit is assigned to the :SI channel, the user will have to rewind the tape after the first pass of the Assembler so it can be scanned a second time.

The assembly listing is output by the Assembler through the sourc output channel (:SO). If the user wishes to obtain columns between fields in the listing, the TAB/SKIP key should be used to insert spaces when the source module is created via the Text Editor. The relative value of each EQUated symbol in the listing is printed with a pointer, ">", next to it. Any relocatable address or operand is identified by the single quote character. The statement numbers are printed in decimal. If the user wishes to assemble a source module with no listing, any valid output device except the Centronics printer (:CE) may be assigned to the :SO channel.

Here is an example of how the I/O channels could be set up for a cassette based mode of operation:

```
:CI=:SK
:CO=:SV
:OO=:O2
:SI=:I1
:SO=:CE
```

The user and the Z80 Relocating Assembler communicate via the Sorcerer keyboard and video display. The source module is read in from cassette unit # 1 and the object module is output to cassette unit # 2. The Assembler listing is sent to the Centronics printer.

Suppose the user was working in the RAM-based mode of operation. The channels would be set up as follows:

```
:CI=:SK
:CO=:SV
:OO=:AI
:SI=:BO
:SO=:SV
```

As before, the Sorcerer is used for the console channels. The source module is read directly from the text buffer of the Text Editor, :BO. The object module is output to the object buffer of the Assembler, :AI. The listing is output to the Sorcerer Video Driver.

This table shows what devices can be assigned to each of the I/O channels the Assembler uses:

:CI	:CO	:OO	:SI	:SO
:I1	:O1	:O1	:I1	:O1
:I2	:O2	:O2	:I2	:O2
:SK	:SV	:SV	:SK	:SV
:AO	:AI	:AI	:AO	:AI
:BO	:CE	:CE	:BO	:CE

Appendix C describes what each device symbol represents.

To call the Z80 Relocating Assembler, the user enters, via the console input channel, the following:

```
.E :AS
```

When the Assembler is finished, control is returned to DDT80.

ASSEMBLER ERRORS

The Assembler detects thirteen errors. Each error has a single letter abbreviation. One of these is an abort error. This means that the Assembler operation is aborted any time this error is detected, control is returned to DDT80 immediately, and a message is sent via the console output channel:

ABORT F

For all other errors, the letter abbreviation is printed in the left margin of the assembly listing next to the statement which is in error.

```
0 '>0000            0012 LABEL   LDR     A,6     ;OPCODE ERROR
```

Following is a list of the Assembler errors:

B - In an expression, an operator exists which does not belong. This usually refers to a trailing operator.

D - In a number that is used as an operand in the source statement, there is a digit of the wrong base or a character which is not allowed.

E - In the source statement, an External global symbol is being used in an expression with operators or as the operand of an EQU psuedo-op, or as the operand where an 8 bit value is required.

F - The symbol table is full as a result of too many symbols being defined. This is an abort error.

I - There is an operand or combination of operands that is invalid for the given opcode in the source statement.

L - There is an invalid character in a label or symbol in the source line. This error can also occur for expressions when the Assembler scans for a symbol.

M - A symbol appeared in the label field in more than one source statement, or was m-ltiply defined.

N - There is no label, as is required, in the label field of an EQU pseudo-op.

O - There is an invalid opcode in the source statement.

R - For the given opcode in the source statement, there is an operand whose value is out of the allowed range. This often occurs for a "JR" instruction where the operand is too large.

S - There is a syntax error in an expression in the source line. Usually this refers to unbalanced parenthesis or quotes, or extra characters in the expression.

U - A symbol used in an expression is undefined. This will occur when a symbol is defined in terms of a local symbol that hasn't appeared yet in the source module, a limitation imposed by a two pass assembler.

V - An expression caused an overflow error in the Z80 CPU when it was evaluated.

LOADER

The Development Pak has a Relocating Linking Loader which will load and link both relocatable and non-relocatable, or absolute, object modules produced by the Z80 Relocating Assembler. The Relocating Linking Loader enables separately assembled object modules to be linked together and to be relocated anyplace in the user's RAM. A large program, for example, can be created as a group of relatively short individual modules which can be separately assembled and debugged. They can then be combined into a complete program when loaded.

LINKING AND LOADING BASICS

The Relocating Linking Loader automatically links global symbols which provide communication or linkage between program modules. Recall that a global symbol is a symbol which is defined within one program module but can be referenced by other modules. As the object modules are loaded, a table containing the global symbol references and definitions is built. After each module is loaded, the Loader then resolves all references to global symbols that have been encountered.

The number of object modules which can be loaded is limited only by the amount of RAM available for the modules and the symbol table. The top 768 bytes of RAM are used by the Sorcerer Power-On Monitor and the Development Pak I/O routines and under no circumstances should any modules be loaded there. The remainder of RAM in the user's system may be used if the user is working in a cassette based or I/O based mode of operation.

If the user is working in the RAM-based mode of operation, the user has several considerations to make in deciding what areas of RAM can be loaded. The values ENDA (In 0132 & 0133), ENDB (In 0134 & 0135) and ENDC (In 0136 & 0137), which the user can determine when in DDT80; separate the user RAM into three areas. The RAM area starting at location zero and ending at ENDA (approximately one-fourth of the Sorcerer RAM) is always free for loading. The RAM between ENDA and ENDB is the object buffer (:AO) which contains the object file to be loaded when working in the RAM-based mode of operation. The user must be careful not to overwrite this area when working in this mode. The RAM between ENDB and ENDC is the Text Editor buffer, :BO. If the user wishes to save this area for later re-editing or patching, loading must be restricted to the RAM below ENDA; otherwise, this area is also free for loading.

Each object module, relocatable or non-relocatable, is loaded in via the object input channel (:OI). The PSECT pseudo-op of the Z80 Relocating Assembler defines a module as relocatable or absolute. Absolute modules are never relocated and are always loaded at their starting address as defined by the ORG pseudo-op during assembly. Relocatable modules are located at an offset address plus the module's defined starting address. This offset address is the address one higher than the last address of the previously loaded module or, in the case of the first module of a group being loaded, is the address specified via the console input channel (:CI) when calling the Relocating Linking Loader.

LOADER SYMBOL TABLE

In the linking process, the Relocating Linking Loader builds a symbol table of global references between program modules and resolves these references after each individual module is loaded. Both symbol definitions and references are placed in this table. Space for this table is allocated dynamically downward in RAM from its origin. This origin is specified by the user when calling the Loader. The length of this table is

$$(N + 1) \times 11$$

where N is the number of unique global symbols.

Each time a module is loaded, the global symbol table is output through the console output channel (:CO) displaying each symbol and its address. Unknown global symbol addresses are marked as undefined. These would occur when symbols have appeared in the operand field of a module but have not yet been defined. A global symbol becomes defined when a module containing both the symbol in the label field and a reference to it by a GLOBAL pseudo-op is loaded.

LOADER I/O

Before the Relocating Linking Loader is called from DDT80, the proper devices should be assigned to the I/O channels that are used by it. How these assignments are done is explained in the section on DDT80.

The user and the Loader communicate with each other via the console. All command inputs to the Loader are received through the :CI channel. The Loader displays error messages, the global symbol table, and the beginning and ending address of each program module through the :CO channel. The object modules are loaded into the system via the :OI channel. The Loader does not use the :OO, :SI, or :SO channels.

In the cassette based mode of operation, the user could assign the channels as shown:

```
:CI=:SK
:CO=:SV
:OI=:I1 (or :I2 for unit # 2)
```

The user and the Relocating Linking Loader communicate through the Sorcerer keyboard and display. The object module(s) would be loaded in via cassette unit # 1.

In the RAM-based mode of operation, the channels would be assigned as follows:

```
:CI=:SK
:CO=:SV
:OI=:AO
```

Again, the user and Loader would communicate through the Sorcerer. The object module(s) here would be loaded directly from the object buffer in RAM.

The following table shows what devices can be assigned to the channels the Loader uses:

:CI	:CO	:0I
:I1	:O1	:I1
:I2	:O2	:I2
:SK	:SV	:SK
:AO	:AI	:AO
:BO	:CE	:BO

Appendix C describes what each of the device symbols refers to.

CALLING THE LOADER

After the user has set up the I/O channels in DDT80 and a module is ready to be loaded in via the :OI channel, the Loader can be called. To enter the Relocating Linking Loader, the user types

.L a,b<CR>

where a and b are hexadecimal addresses.

The operand a is the offset address. If an absolute module is to be loaded, its starting address will be the address defined by the ORG pseudo-op in the Z80 Relocating Assembler and the offset address is ignored. If a relocatable module is to be loaded, its starting address will be the sum of the address defined by the ORG pseudo-op and the offset address. The operand b is the address of the origin of the global symbol table.

After the Loader is called, it will load the module, display via the :CO channel the starting and ending address for the module and the global symbol table, and return with the Loader's prompt, an asterisk. If the user has more modules to load, the command used is

*L

The offset address for each new module is one more than the ending address of the last module loaded. Absolute modules are loaded without regard to the offset address but they do produce a new offset address for the next module.

To leave the Loader, the user just enters a period. If anything is entered after the Loader prompt except a period or the L command, the Loader ignores it and returns a new prompt. Upon exiting the Relocating Linking Loader, control is given to the Sorcerer monitor and is NOT returned to DDT80. This is because DDT80 uses RAM areas that might contain user loaded code. If the user wishes to renter DDT80, use this monitor command:

*.
>PP <Carriage Return>

LOADER ERRORS

The R-locating Linking Loader detects four errors, two of which are fatal and return control to DDT80. The non-fatal errors leave the Loader symbol table intact allowing the user to continue loading.

When an error occurs, the Loader displays via the :CO channel a message:

****ERROR e

The number e indicates the type of error as shown here:

e	ERROR	CONTROL RETURNS TO:
1	checksum error	Loader
2	double definition of a global symbol	Loader
3	attempt to overwrite symbol table	DDT80
4	symbol table full	DDT80

If a checksum error occurs in a data record, as opposed to an EOF or relocating record, the address of the location where the last byte of that record was loaded is displayed directly before the error message. If there were checksum errors in more than one data record, all the addresses will be shown. The user then has the capability to correct the data in memory using DDT80 if so desired. If no addresses are displayed when a checksum error occurs, the error was detected in a non-data type record and it is recommended that the load sequence be completely redone from the beginning.

In the process of loading a module, if a global symbol definition which already existed in the symbol table from a previously loaded module is encountered as a record, the global symbol name is displayed before the error message. Global symbols may be referenced any number of times but can only be defined once within any group of modules being loaded together.

When a module is loaded and is going to be using the same bytes that the global symbol table is using, a fatal error occurs. The other fatal error happens when the end of the symbol table has reached the bottom of RAM, location 0000H. As stated before, both of these errors cause an abort of the loading process and return control to DDT80.

APPENDIX A

OBJECT OUTPUT DEFINITION

Each record of an object module begins with either a colon or a dollar sign and ends with <CR><LF>. A colon is used for data records and for the end-of-file record. A dollar sign is used for records containing relocating and linking information and for the module definition record. An Intel hexamdecimal loader will ignore all records not beginning with a colon and can therefore load non-relocatable, non-linkable programs. All of the information in these records is represented by ASCII characters.

Following is a description of the format for each type of record. A few terms used should be defined. A binary byte refers to one eight bit byte of data which will be represented in a record by two ASCII bytes. Record type refers to two ASCII bytes that represent a number indicating the kind of record as follows:

00 - data record
01 - end-of-file record
02 - internal global symbol record
03 - external global symbol record
04 - relocating information record
05 - module definition record

Checksum refers to a method for checking the accuracy of the data in a record. The checksum for each record is computed by negating the binary sum of all the bytes in the record excluding the beginning delimeter and the <CR><LF>.

I. Data Record

Byte 1	Colon
2-3	Number of binary bytes of data in this record, with a maximum of 32 binary bytes
4-5	Most significant byte of start address of data
6-7	Least significant byte of start address of data
8-9	Record type
10-	Data bytes to be converted to binary when loaded
Last two bytes	Checksum

II. End-of-file Record

Byte 1	Colon
2-3	ASCII zeroes
4-5	Most significant byte of starting execution address of program
6-7	Least significant byte of starting execution address of program
8-9	Record type
10-11	Checksum

III. Internal Global Symbol Record

Byte 1	Dollar sign
2-7	Up to 6 ASCII characters of the symbol name, left-justified, blank filled
8-9	Record type
10-13	Address of internal symbol, most significant byte first
14-15	Binary checksum; note that the ASCII letters of the symbol are converted to binary before the checksum is calculated

IV. External Global Symbol Record

Byte 1	Dollar sign
2-7	Up to 6 ASCII characters of the symbol name, left-justified, blank filled
8-9	Record type
10-13	Last address in module which references the external symbol; beginning of a backward link list in the object data records pointing to references to the symbol and terminated by 0FFFFH
14-15	Binary checksum

V. Relocating Information Record

Byte 1	Dollar sign
2-3	Number of sets of 2 ASCII characters, each 2 sets defining an address that must be relocated
4-7	ASCII zeroes
8-9	Record type
10-	Addresses which must be relocated, most significant byte first
Last two bytes	Binary checksum

VI. Module Definition Record

Byte 1	Dollar sign
2-7	Random ASCII characters and/or graphic characters
8-9	Record type
10-11	Flag byte whose least significant bit, when the byte is converted to binary, is zero for absolute assemblies and one for relocatable assemblies and is determined by the PSECT pseudo-op
12-13	Binary checksum

APPENDIX B

REPARTITIONING RAM

In the section on Sorcerer I/O each of the RAM Partitions, #1 - #4, is discussed. The beginning and ending addresses of these Partitions are variable, depending on the size of the Sorcerer being used with the Development Pac. Initially each Partition is set up to use a certain amount of RAM in the Sorcerer. For example:

PARTITION # 1 - Always uses the top 768 bytes of RAM
PARTITION # 2 - Initially set up to use approx. 1/2 of RAM
PARTITION # 3 - Initially set up to use approx. 1/4 of RAM
PARTITION # 4 - Initially set up to use approx. 1/4 of RAM

The Partitions are configured in this way to make the most efficient use of RAM in a RAM-based mode of operation. However, this may not be the most efficient configuration if the user is in an I/O-based mode of operation, wishes to increase the Assembler symbol table size, or wishes to have a larger Editor buffer to work with.

Using the "M" command in DDT80, the user can alter the boundaries and therefore the size of any one of the Partitions. As mentioned above, there are a number of reasons for a user to want this capability:

1. To increase the Assembler symbol table size.
2. To increase the Editor buffer size.
3. To increase the Object buffer size.
4. To reposition the stack area.
5. To move the buffer areas out of the way of other programs.
6. To move the buffer areas out of the way of user I/O drivers.
7. To move the buffer areas out of the way of disk driver routines.

Before it's explained exactly how to "move" the Partitions around in the Development Pac, it should be mentioned that some parts of the Partitions are not moveable! First, let's summarize exactly what each Partition consists of:

PARTITION # 1

1. Power-On Monitor RAM Data area.
2. Power-On Monitor Stack area.
3. Development Pac Stack area.
4. Cassette Input Buffer.
5. Cassette Output Buffer.

PARTITION # 2

1. Editor Buffer.
2. "B" Memory buffer.
3. Free RAM if Source File is overwritten.

PARTITION # 3

1. Object Buffer.
2. "A" Memory Buffer.
3. Free RAM if Object File is overwritten.

PARTITION # 4

1. DDT80 RAM Data area.
2. Editor RAM Data area.
3. Assembler RAM Data area.
4. Assembler Symbol Table area.
5. Free RAM if DDT80 Data area is overwritten.

The boundaries of the Partitions are determined by three values: ENDA, ENDB and ENDC. These values separate the partitions according to the following format:

1. PARTITION # 1 = TOP OF RAM TO ENDC

2. ENDC = TOP OF RAM - 768 BYTES

3. PARTITION # 2 = ENDC TO ENDB

4. ENDB = ENDC / 2

5. PARTITION # 3 = ENDB TO ENDA

6. ENDA = ENDB / 2

7. PARTITION # 4 = ENDA TO 0100 HEX

8. 0000 TO 00FF HEX IS FREE

Since partition #1 contains the Sorcerer's Power-On Monitor Data and Stack area, it should be moved by using the Sorcerer's Monitor Reentry Point, E006 hex. By entering the Power-On Monitor at this point, the HL register pair will be used as the new TOP OF RAM and the Power-On Monitor will reinitialize its data area at the now location.

A program for moving the Power-On Monitor Data and Stack area would then be:

```
21 YY XX        LD      HL,XXYYH
C3 06 E0        JP      0E006H
```

The "XXYY" represents the address to which the Data and Stack areas are to be moved. It is the "TOP" address and space will be allocated from that location downward. The Power-On Monitor will then transfer control to the Development Pac WARM start entry point. The Development Pac will clear the screen and print a "." alone on the screen. The Development Pac must be "RE-STARTED" after moving the Power-On Monitor's Data area. This is because the Development Pac uses part of the Power-On Monitor's Data area for obtaining the TOP OF RAM address and setting up the Cassette Buffers. The Development Pac is "RE-STARTED" by using the DDT80 "E" command:

```
.E C000  <Carriage Return>
```

The Development Pac will sign-on again, using the normal COLD start message, and repartition the RAM according to the new TOP-OF-RAM address from the above program. The user must then use the DDT80 "M" command to alter the ENDA, ENDB and ENDC values for the "real" Partitions. The locations to alter for the boundaries are:

```
ENDA = Top of Partition # 4 = 0132 & 0133 hex
ENDB = Top of Partition # 3 = 0134 & 0135 hex
ENDC = Top of Partition # 2 = 0136 & 0137 hex
```

These are Intel format addresses with the least significant 8 bits first and the most significant 8 bits last.

The Power-On Monitor move routine, at E006 hex, may be used to simply "lower" all of the Partitions and make room in the Top of RAM for I/O drivers or disk drivers. In this case, the user would not have to alter any of the Partition boundaries since this would already have been accomplished by "RE-STARTING" the Development Pac.

It should be mentioned that the user must be careful of moving any of the Partitions over Partition # 1 described above. This Partition needs exactly 768 bytes below the TOP-OF-RAM address given it in the program described above. If any of the 768 bytes is altered, a loss of information or possibly a complete loss of user control could result!

The user should also be careful of moving Partition # 4. This is because DDT80, the Editor, and the Assembler all have their data storage in this Partition. DDT80 uses from 0100 to 0138 hex. The Editor and Assembler have overlapping data areas starting at 0139 hex and working

upward. The Editor uses from 0139 to 0183 hex and the Assembler uses from 0139 to 0255 hex. The Assembler also starts its symbol table at 0256 hex and "grows" upward. These values and data areas are NOT changable! They are ingrained in the Development Pac to start at 0100 hex. The TOP part of Partition # 4 can be moved up or down to increase or decrease the size of the Assembler's symbol table, but the BOTTOM address (0100 hex) of Partition # 4 CANNOT be altered! The user should be extremely careful not to move the TOP of Partition # 4 below 0258 hex. In fact, the user should always allow about 40 or 50 bytes above 0258 hex for at least a small symbol table.

Partition # 2 and Partition # 3 do not have the restrictions of the other Partitions. They can be moved freely by the user to almost any RAM location. The Partitions will not function properly however if moved into ROM areas or into non-existent addresses. The only other restriction in moving Partitions # 2 and # 3 is to be careful of moving them over Partition # 1 or over Partition # 4.

As a general rule, the user should be extremely careful when moving or shifting the Partitions. It is much easier to change the size of the Partitions than to move them to another location. The user should always "experiment" with a small test program after adjusting the Partitions to verify that they work in their new locations.

This feature, Repartitioning RAM, is mentioned here to inform the user of this capability. It is not necessary to move any of the Partitions when doing "normal" development with the Development Pac. However, this feature is very valuable to the advanced Development Pac user.

APPENDIX C

MNEMONICS RECOGNIZED BY DDT80

DEVELOPMENT PAC MNEMONICS

MNEMONIC	ADDRESS	DATA AT ADDRESS
:PC	0118	User's Program Counter
:A	0117	" A Register
:F	0116	" Flag Register
:I	0115	" Interrupt Register
:IF	0114	" IFF Register
:B	0113	" B Register
:C	0112	" C Register
:D	0111	" D Register
:E	0110	" E Register
:H	010F	" H Register
:L	010E	" L Register
:A'	010D	" A' Register
:F'	010C	" Flag' Register
:B'	010B	" B' Register
:C'	010A	" C' Register
:D'	0109	" D' Register
:E'	0108	" E' Register
:H'	0107	" H' Register
:L'	0106	" L' Register
:IX	0104	" IX Register
:IY	0102	" IY Register
:SP	0100	" Stack Pointer
:CI	F01E	Console Input Vector
:CO	F020	Console Output Vector
:OI	F022	Object Input Vector
:OO	F024	Object Output Vector
:SI	F026	Source Input Vector
:SO	FF28	Source Output Vector
:SK	C547	Sorcerer Keyboard Driver
:SV	C54E	Sorcerer Video Driver
:I1	C59B	Input from Tape Unit # 1
:I2	C59D	Input from Tape Unit # 2
:O1	C557	Output to Tape Unit # 1
:O2	C559	Output to Tape Unit # 2
:CE	C552	Sorcerer Centronics Driver
:AI	C5F5	"A" Buffer Input (Object)
:AO	C60A	"A" Buffer Output (Object)
:BI	C61F	Dummy Driver (ETX Chars.)
:BO	C624	"B" Buffer Output (Source)
:AS	CD9B	Assembler start address
:ED	CAA9	Editor start address
:ER	CAB0	Editor reentry address

APPENDIX D

ASSEMBLER ERRORS

Each of the Errors described below is in summarized form. The complete definition and probable causes of each error can be found in the Assembler section of the manual under "ASSEMBLER ERRORS".

ERROR	DEFINITION
B	Bad operator, Trailing operator
D	Digit of wrong base in operand, Invalid character
E	External Global symbol misused in expression
F	Full Symbol Table
I	Invalid operand or expression for opcode
L	Label error, Invalid character in label or symbol
M	Multiply defined label or symbol
N	No label appears where one is expected
O	Opcode invalid
R	Range error in expression evaluation, Out of Range
S	Syntax error, Unbalanced parenthesis or quotes
U	Undefined symbol
V	Overflow in expression evaluation

Except for the "F" error, these errors will appear as a one letter symbol in the right-most column of the listings produced by the Assembler. The "F" error generates an "ABORT F" on the console output. The Assembler will also count each error and print the total number of errors (From 0000 to 9999) for each assembly.

APPENDIX E

LOADER ERRORS

Each of the Errors described below is in summarized form. The complete definition, cause and correction for each error can be found in the Loader section of the manual under "LOADER ERRORS".

ERROR	DEFINITION	RETURN
1	Checksum Error	LOADER
2	Double Definition of Global	LOADER
3	Symbol Table being Overwritten	DDT80
4	Symbol Table Full	DDT80

The Loader will flag each error by displaying:

****ERROR X

The "X" is one of the numbers listed above. Errors 3 and 4 are "fatal" errors and will end the loading session. Errors 1 and 2 are "non-fatal" errors and can usually be recovered from. Some additional information is output for error 1. For a full description of this type of error the section "LOADER ERRORS" should be read.

APPENDIX F

SUMMARIZED COMMANDS OF THE DEVELOPMENT PAC

DDT80

========== DDT80 COMMANDS: ==========

M - Memory display/modify command
R - Register display/modify command
E - Execute program command
H - Hexadecimal arithemetic command
L - Loader transfer command

==========

EDITOR

========== EDITOR COMMANDS: ==========

B - Point to the Beginning of text
n<CR> - Move line pointer n lines and display
I - Insert source lines
nD - Delete n source lines
nT - Type n source lines on console display
nR - Read n source lines into the buffer
nW - Write out n source lines from the buffer
E - Exit from Editor and close file

==========

LOADER

========== LOADER COMMANDS: ==========

.L XXXX,YYYY	Code at XXXX - Symbol Table at YYYY
*L	Load next Object Module
*.	End Loading Session
>PP	Reenter DDT80 from Power-On Monitor

==========

APPENDIX G

ACKNOWLEDGEMENTS

A great deal of effort and cooperation went into the Development Pac. We at Exidy would like to acknowledge the efforts of the individuals responsible for creating the package and documentation.

CODING

FROM MOSTEK INC.

Dave Leitch
Dan Hammond
P. Formaniak
John Bates

FROM EXIDY INC.

John K. Borders Jr.
Janice E. Cheng
Jan A. Neff

DOCUMENTATION

FROM EXIDY INC.

John K. Borders Jr.
Jan A. Neff
Janice E. Cheng

INDEX

A

:A - Register A ..... PAGE 11,48,71
:A' - Register A' 11,48,71
Absolute Module 42,52
Acknowledgements 75
:AI - "A" Buffer Input 2,17,26
:AO - "A" Buffer Output 2,17,26
A Ram Buffer 24,26,59
:AS - Assembler mnemonic 71
Assembler 4,41
Assembler I/O 55

B

:B - B Register ..... PAGE 11,48,71
:B' - B' Register 11,48,71
B Command (Editor) 34
B Error (Assembler) 57
:BI - Dummy Driver 2,17,26
Binary Constants 49
:BO - "B" Buffer Output 2,17,26
B Ram Buffer 24,26,59
Breakpoints 13

C

:C - C Register ..... PAGE 11,48,71
:C' - C' Register 11,48,71
Calling the Assembler 56
Calling the Editor 33
Calling the Loader 16,62
Cassette Buffers 21,24
Cassette Files 21,22
Cassette Mode of Operation 31,55,61
:CE - Centronics Printer Driver 2,17,23
Checksum Error (Loader) 63
:CI - Console Input Vector Channel 2,3
Close Device Flag 18,19
:CO - Console Output Vector Channel 2,3

Coding 75
Commands - DDT80 7
- Editor 27,34
- Loader 62
Comments (Source Code Line) 51
Console Vector Channels 2,3
Constants 49
Control Register (I/O Driver) 18,19
CR Command (Editor) 34

D

:D - D Register ..... PAGE 11,48,71
:D' - D' Register 11,48,71
Data Record (Object File) 65
Data Register (I/O Driver) 18,19
D Command (Editor) 35
DDT80 - Designer's Debugging Tool 4,7
Decimal Constants 49
DEFB - Define Byte Pseudo-op 47
DEFS - Define Storage Pseudo-op 47
DEFW - Define Word Pseudo-op 47
D Error (Assembler) 57
Documentation 75
Double Definition Error (Loader) 63
Dual Device Select Flag (I/O Driver) 18,19
Dummy Driver 26,31

E

:E - E Register ..... PAGE 11,48,71
:E' - E' Register 11,48,71
E Command (DDT80) 13
E Command (Editor) 37
:ED - Editor mnemonic 71
Editing Example 38
Editor 4,27
Editor I/O 31
E Error (Assembler) 57
END A (Partition Marker) 24,25,59
END B (Partition Marker) 24,25,59
END C (Partition Marker) 24,25,59
End of File Marker 31,37,42
End of File Record 65
EQU - Equate Pseudo-op 47
:ER - Editor Reentry mneumonic 71
Errors - (Assembler) 57
Errors - (Editor) 28
Errors - (Loader) 63
Escape Character 35

ETX Character 31,37,42
Expressions (Source Code) 49
External Symbols 43,50
External Symbol Record 66

F

:F - Flag Register ..... PAGE 12,48,71
:F' - Flag' Register 12.48,71
F Error (Assembler) 57
First Form (DDT80 "M" Command) 8
Flag Register 11,71

G

GLOBAL - Global Pseudo-op ..... PAGE 42,46,47
Global Symbols 42,53
Global Symbol Table (Loader) 16,60,63

H

:H - H Register ..... PAGE 11,48,71
:H' - H' Register 11,48,71
H Command (DDT80) 15
Hex Arithmetic 15
Hex Constant (Source Code) 49

I

:I - Interrupt Register ..... PAGE 12,48,71
:I1 - Cassette Input from Unit # 1 2,17,21
:I2 - Cassette Input from Unit # 2 2,17,22
I Command (Editor) 35
I Error (Assembler) 57
:IF - Interrupt Register 12,48,71
Immediate Return Flag (I/O Driver) 18,19
Internal Symbols 43
Internal Symbol Record 65
Interrupts 12
Introduction 1
I/O Drivers 2,5,21
I/O Drivers (Writing Drivers) 18
I/O Vectors 2,3
:IX - IX Index Register 12,48,71
:IY - IY Index Register 12,48,71

K

Keywords ..... PAGE 46,48

L

:L - L Register ..... PAGE 11,48,71
:L' - L' Register 11,48,71
Labels 45
L Command (DDT80) 16
L Error (Assembler) 57
Line Buffer (Editor) 28,30
Line Pointer (Editor) 28,30
Linker 58
Link List 44
Listing (Assembler) 55
Loader 5,58
Loader I/O 61
Loader Symbol Table 60
Local Symbols 42

M

M Command (DDT80) ..... PAGE 7,8
M Error (Assembler) 57
Mnemonics 9,10,71
Module Definition Record (Object File) ... 66
Modules 2,4,42
M Command Secondary Commands (DDT80) 8
"." 8
"^" 9
"<CR>" 8
"HEX" 8,9

N

N Error (Assembler) ..... PAGE 57

O

:O1 - Cassette Output to Unit # 1 ..... PAGE 2,17,22
:O2 - Cassette Output to Unit # 2 2,17,22
Object Buffer 25,26
Object Output 64
Object Vector Channels 2,3
Octal Constants 49
O Error (Assembler) 57

Offset (Loader) 42,43,44
:OI - Object Input Vector Channel 2,3
:OO - Object Output Vector Channel 2,3
Opcodes 46
Open Device Flag (I/O Driver) 18,19
Operands 48
Operators (Assembler) 49
(DDT80) 15
ORG - Origin Pseudo-op 44,46,47,59
Overwrite Symbol Table Error (Loader) 63

P

Partition # 1 ..... PAGE 24
2 24
3 25
4 25
:PC - Program Counter 11,71
Position Independent Programs 43
Power-On Monitor 24,62,67
Program Counter (Assembler) 49
(DDT80) 11
Prompt (Editor) 33,38
(DDT80) 7
(Loader) 62
(Power-On Monitor) 62
PSECT - Program Sectioning Pseudo-op 43,46,47,52
Pseudo-ops 46

R

RAM - (Assembler) ..... PAGE 25
(DDT80) 25
(Editor) 25
(Loader) 16,25
Ram Buffers 24
Ram Mode of Operation 31,56,61
R Command (DDT80) 7,11
(Editor) 36
Register Headings 11
Relative Offset 42,43,44
Relocatable Module 42,52
Relocating Information Record (Object File) 66
R Error (Assembler) 57
RESTART 7 13
Rewinding Cassette Files 21

S

Second Form (DDT80 "M" Command) ..... PAGE 9
Serial Data Cable 21
S Error (Assembler) 57
:SI - Source Input Vector Channel 2,3
Sign-On 7,38
:SK - Sorcerer Keyboard mnemonic 2,17,21
:SO - Source Output Vector Channel 2,3
Sorcerer I/O 17
Source Buffer 24,28
Source Vector Channels 2,3
:SP - Stack Pointer 12,71
:SV - Sorcerer Video Driver mnemonic 2,17,21
Symbols 42
Symbol Table Full Error (Loader) 63
Syntax - Assembly Language 45

T

TAB-SKIP Key ..... PAGE 30,45
Tab Stops 30
T Command (Editor) 36
Text Buffer (Expansion) 24,28
Text Buffer (Size) 24,28
Third Form (DDT80 "M" Command) 9
Two Pass Assemblers 43

U

U Error (Assembler) ..... PAGE 57

V

Vector Channels ..... PAGE 2,3
V Error (Assembler) 57

W

W Command (Editor) ..... PAGE 36

Z

Zilog Publications ..... PAGE 46