```
* * * * * * * * * * * * * * * * * *
*                                 *
*                                 *
*          EXTENDED   DDT         *
*                                 *
*                                 *
*        E   X   D   D   T        *
*                                 *
*                                 *
*    A Program Debugging Tool     *
*                                 *
*          by  Jim Dunion         *
*                                 *
*   (c) 1985  ANTIC PUBLISHING    *
*        THE ATARI RESOURCE       *
*                                 *
*                                 *
* * * * * * * * * * * * * * * * * *
```

EXDDT USERS GUIDE

EXTENDED DDT

by Jim Dunion

Contents:

INTRODUCTION:

THE EXTENDED VERSION OF DUNION'S DEBUGGING TOOL

The Extended version of Dunion's Debugging Tool (EXDDT) is designed to assist Atari 400, 800, XL, and XE programmers who are working at the assembly language level. This program is an outgrowth of the DDT program once marketed by APEX. As such, the power of DDT is intact, but with numerous additions and enhancements that have been added over the last year. The most significant of these are the mini-assembler and the built in system symbol table with symbolic reference.

EXDDT will work with any of the Atari 6502 based computers equipped with a disk drive and enough memory to support DOS. This includes the new 130 XE model.

EXDDT will operate invisibly after the system is turned on and EXDDT is loaded. File EXDDT.OBJ should be loaded. It will automically initialize itself and return to the loading program. To actually enter EXDDT, press the CNTL, SHIFT and the ESC keys at the same time. This is known as a "flash" entry to EXDDT.

You may use EXDDT in any operating mode that first allows for a disk access before running. The executable code file, EXDDT.OBJ can be copied to other diskettes and renamed. Additional code segments can be written and EXDDT.OBJ can be appended to the end of the object code file. For instance, EXDDT.OBJ can be renamed AUTORUN.SYS to load automatically when the system is rebooted. You can even relocate where the EXDDT code should reside and execute. I've spent a lot of time getting this program to where it is today. I feel sure that EXDDT can help make programming and debugging on the Atari computers easier and even fun. Enjoy.

LOADING EXDDT

EXDDT is loaded from DOS or other operating systems by loading the program EXDDT.OBJ. For example, if you have Atari DOS, use the "L" command and give the name EXDDT.OBJ. After loading and setting up EXDDT, control returns to the DOS calling program. At this point EXDDT is active, and can be called by pressing CTRL, SHIFT and ESC all at once. I suggest you load EXDDT and play with it as you read this documentation.

There is a test program called TEST.MAC (the assembled version is called TEST.OBJ) provided on this diskette to give you an example of how to transfer a symbol table to EXDDT, as well as how to call EXDDT from your own code. There is also an example of how to relocate the EXDDT display screen from a calling program. It will run automatically when loaded. It doesn't do much except change some color registers, put a few 'I's on the screen and then turn on player-missles.

Finally, there is a Basic program called RELOCATE. To run this program from Basic, type RUN "D:RELOCATE". This program will relocate the EXDDT.OBJ code to load and execute at a different location in memory. It creates an output object code file (which is relocated) called EXDDT.REL.

Section 1:

EXDDT DESIGN PHILOSOPHY

The heart of EXDDT is its ability to show what is happening inside the computer on a special display screen. This is coupled with the ability to easily change and monitor the internal state of the machine so that you can get a much clearer picture of exactly what's going on inside the system at any instant.

The rest of this document is an EXDDT USER's guide. Each of the features in EXDDT is described in a separate section. There is also a technical appendix section with more information on how some of EXDDT's features are implemented.

Section 2:

THE EXDDT SCREEN DISPLAY

The EXDDT Screen Display shows a user the internal state of the machine. The display screen is divided into several display areas which show different aspects of what is going on inside the computer.

The display areas are called :

- REGISTER DISPLAY	- Shows the current contents of 6502 registers
- DISPLAY WINDOW	- A window into memory
- STACK DISPLAY	- Shows the top 15 items on the system stack
- MINI SYMBOL TABLE	- Shows names and values of current symbols
- BREAKPOINT TABLE	- Shows the settings of breakpoint registers
- COMMAND WINDOW	- The keyboard command entry window

The following sections detail each of these display areas.

2-1: REGISTER DISPLAY

The lowest part of the display screen is used for displaying the current contents of the 6502 processor registers. Whenever EXDDT is entered, the contents of the processor registers are copied into register shadows which are then displayed. These shadows are used to restore the 6502 registers before control is released back to the program being tested.

The copywright notice will be replaced by the register names when the first command is issued. If this is your first time just type "E <RETURN>".

These registers have their contents shown in hexadecimal notation :

PC	= Program counter, a 2 byte value
A	= Accumulator
X	= X index register
Y	= Y index register
S	= Stack pointer

The Processor status register (NV BDIZC) is shown in binary form, where

N	= Negative flag
V	= Overflow flag
B	= BRK instruction flag
D	= Decimal mode flag
I	= Interrupt disable flag
Z	= Zero flag
C	= Carry bit

2-2: DISPLAY WINDOW

The display window forms a window into the system memory address space. This window is located in the upper left hand portion of the display screen, and occupies more than a quarter of the full screen. The window is set upon entry to EXDDT, and may be moved by several commands.

The window may be thought of as having one of two possible filters in front of it. The first filter, which is set upon entry to EXDDT is a disassembly filter. A GREATER THAN sign (>) points to what is called the current position. When EXDDT is entered, this will correspond to the value in the PC. The current position is modified by several commands.

The SECOND filter is a hexadecimal filter. The window shows the hexadecimal value and ATASCII representation of up to 48 memory locations. Again, the > sign indicates the current position.

The command "W" toggles between these two Window filters.

There are always three bytes shown above the current position. These are shown in hexadecimal form.

In the disassembly display, each line from the current position down is shown in a similar format. First the hexadecimal address of a location, its contents and then a disassembly readout. Standard 6502 mnemonics are used, with conventional address mode indications.

Several features have been added to aid debugging. If a mnemonic is shown in inverse video, it indicates that a breakpoint has been set at that location. In fact, if you look at the actual contents of that location, it will be a 0. If the mnemonic in inverse video is a BRK instruction, this means that particular BRK instruction was not placed there by EXDDT. This would occur for instance in looking at memory that is all zeros.

Secondly, if the instruction is one of the branch instructions, an additional portion is added to the disassembly display. An up or down arrow is used to indicate the direction of the conditional branch. The computed address of the conditional branch location is also shown.

Finally, if the address portion of an instruction contains an address that is defined as a symbol, the symbol name will be shown rather than the hexadecimal value. Such symbols can either be defined by users in the mini-symbol table, or may be standard system symbols. EXDDT has around 150 standard internal symbols that are defined in the operating system. These symbols make understanding the

disassembly display much easier. This is particularly the case if care is taken to program the system in accordance with original design considerations. Care has been taken to ensure that the standard system symbols defined are ones that are valid for all models of the Atari. There's no real rhyme or reason as to which system variables are defined, these are merely the ones I felt to be most important. The mini-symbol feature may be used to monitor a variable, locate references to a symbol in the code, or simply as labels to make the disassembly listing more readable. One more little feature is added as a convenience. If you define a symbol as a two-byte symbol, then any reference to the variable's high byte will show up in the disassembly as SYMBOL+1.

If the hexadecimal filter is in place, each line after the current position line will start on an even 4 byte boundary. This means the current position line can have 1-4 values on it. The current position line values will always be left justified.

2-3: STACK DISPLAY

The middle portion of the upper display screen is used for showing the top locations in the system stack. If the stack pointer is set at $E0 or higher (i.e. there are less than 15 entries in the stack), then only those values that are currently in the stack will be shown. The display is a top down representaion. If more than 15 entries are in the stack, then only the top 15 are shown.

2-4: MINI SYMBOL TABLE

The upper right hand portion of the screen is dedicated to a mini symbol table. There is room for 15 variables in this table. This feature is designed to let a user monitor the contents of selected variables without having to pay undue attention to where they physically reside. The display layout shows the variable name, a V column and a value column. The V column is used to mark variables (see the V function description). The value column will show either the current value of a variable, or its location in memory. Internally, each symbol in the table is in the following form:

NAME	LOCATION	BYTES to SHOW
symbol name	symbol address	0, 1 or 2
6 bytes	2 bytes	1 byte

Using the Atari Macro Assembler (AMAC), an example of setting up a mini symbol table entry would be :

```
DB  'VAR1  '  ; Exactly 6
              ; characters please!

DW  VAR1      ; let the assembler
              ; figure out what
              ; value to put here

DB  1         ; either a 0,1 or 2
              ; to indicate that
              ; the variable should
              ; be shown as no
              ; bytes, a single
              ; byte or a double
              ; byte value.
```

(Users of other assemblers may need to substitute .BYTE for DB and .WORD for DW)

The mini symbol table can be used to keep an eye on standard system variables.

```
DB  'COLPF2'
DW  710
DB  1
```

A small area of memory can be monitored by setting up several dummy variables each pointing to one or two successive bytes of memory.

The mini symbol table has other uses also, e.g. you can define a program label as a symbol. By selecting a display size of 0, no value will be displayed, but the disassembly listing in the display window will be more readable.

```
DB  'LOOP1 '
DW  LOOP1
DB  0
```

Indeed you can even define a symbol as "------" or some such to separate different usage areas of the symbol table. Finally, the mini symbol table can be used to help locate a portion of your code after a reassembly. See the 'M' function and symbolic references for more information on this usage.

You need not define any more symbols than you want to use. If you set up less than 15 symbols in a symbol table, a value of $FF should terminate the table.

EXDDT comes with a built in mini-symbol table that contains some of the variables the operating system uses in controlling the system. You may make individual changes to the standard table with the M command, or you may load in your own table. A symbol table is loaded in a straightforward manner. Create a symbol table in your object code as described earlier. Near the beginning of your code, use a series of instructions like :

```
LDA # high SYMTAB
PHA
LDA # low  SYMTAB
PHA
JSR $6000    ; address of the
             ; symbol table
             ; transfer routine
             ; for this assembled
             ; version. This is
             ; always the first
             ; byte of EXDDT, even
             ; if it has been
             ; relocated.
```

2-5: BREAKPOINT TABLE

The Breakpoint table is located just above the register display. There are six user definable breakpoints and two trap breakpoints, each of which will be shown with its current setting. If a register is not set the value shown will be 0000. If a breakpoint register is set, the value in that register will be the location of where in memory a BRK instruction has been placed. However, in the case of the TRAP breakpoints, no BRK instruction is used. These values are used in interpretive mode to create the equivalent of a break instruction.

2-6: COMMAND WINDOW

The extreme right hand part of the bottom of the screen is devoted to the command window. This is the area that shows the command that a user is typing in.

2-7: TRAP

The Trap breakpoints are reserved for interpretive mode. In this mode, breakpoints in memory are ignored, since EXDDT already has control of the system. Instead EXDDT checks the values in the TRAP registers. If either equals the address of the next instruction to be executed, EXDDT wil halt the interpretive mode. This allows pseudo breakpoints to be placed in ROM locations. Thus it becomes much easier to reach a certain spot in the ROM code by setting a trap, and running in interpretive mode than by single stepping up to the desired location.

Section 3:

BREAKPOINTS

One of the most common debugging techniques is to make use of a breakpoint.

The breakpoint mechanism is the most common way for you to transfer control to EXDDT. When a program is running, executing a BRK instruction will call EXDDT. This will cause the EXDDT screen display to activate, and also will turn on the keyboard and the push button command interpreter. The breakpoint will remain set even after it has been encountered in code execution.

After a breakpoint has been encountered, and control has been transferred to EXDDT, there are several ways to leave EXDDT. The 'N' command will set a breakpoint at the next location and then continue code execution. START simply continues code execution. 'G' can be used to transfer control to another location.

Up to six breakpoints can be in place at any one time. The location of the breakpoints is shown in the breakpoint register display. If a breakpoint is clear (i.e. not set), it will show up as 0000. Setting a breakpoint register to a new location will automatically restore an existing breakpoint, if one is already set for that register. Note also that there is an internal system breakpoint 0 which is used by the 'N' command. If any breakpoint is encountered and control is transferred to EXDDT, then the internal 'N' breakpoint is cleared.

Section 4:

PUSH BUTTONS CONTROLS

The three ATARI console push buttons are used by EXDDT for special effects.

START - is used to continue code execution at the location indicated by the PC register. All 6502 registers are updated with the current displayed contents before control is transferred.

SELECT - is used to toggle back and forth between the EXDDT screen and whatever screen dynamics were active before EXDDT was called. An attempt has been made to allow most alternative features such as mixed Display lists, VBLANK routines, alternative character sets, display list interrupts, playfield size changes, and player-missiles.

OPTION - is used to single step the processor. This causes the disassembly filter to be turned on, but will not automatically toggle the display screen. Continuting to press the Option button will continue to single step the processor. See Single Step section for more information.

Section 5:

THE COMMAND INTERPRETER

The command interpreter is a code module that allows a user to issue keyboard commands to EXDDT. The command window is shown in the lower right hand portion of the display screen. The left-hand part of this display is used for showing the register state of the machine.

Each command is a single keystroke command. However, depending upon the command, additional arguments might be required. If the first key typed is not a valid EXDDT command, it will be ignored.

The EXDDT Keyboard Commands are :

B <1-6>,<addr>..	- Breakpoint 1-6 at location addr
D <hstring,@>..	- Deposit hex string or enter mini-assem
E <addr>.....	- Examine address addr
G <addr>......	- Go to address addr
H <hnum,dnum>..	- Hex/Decimal con-verter
I	- Interpretive mode
M <0-2>,<addr>..	- Monitor 0, 1 or 2 bytes at addr
N	- Next instruction
R <A,X,Y,S,P>,<byte>..	- Register selected is loaded with byte
S <hstring>.	- Search for hex string
T <1-2>,<addr>....	- Trap at address addr
V	- Variable trap function
W	- Window filter toggle
down arrow .	- Pull display window down
up arrow ..	- Push display window up
* ***<addr>*** .	- Set the Program counter

These commands are described in the following pages.

5-1: ENTERING A VALUE

Several of the keyboard commands require that you enter one or two values. All entries are terminated by pressing the RETURN key. When two values are needed, as with the Breakpoint command, a COMMA should be used to separate them. Pressing RETURN without having entered a value will result in the entire command being ignored.

(Exceptions - see the Breakpoint, the Trap and the Monitor Commands)

In the explanations that follow, these abbreviations are used :

<addr>	= an address value, up to 4 hexadecimal digits
<1-6>	= either a 1,2,3,4,5 or 6
<A,X,Y,S,P>	= either A,X,Y,S or P
<byte>	= a single byte value, up to 2 hexadecimal digits
<hstring>	= a hex string up to 12 characters (i.e. 6 hex digits)
<@>	= special character used to enter mini-assembler
<>>	= special character used to indicate the current position
<*>	= special character used to indicate the contents of the PC
<'>	= special character used to signify a symbolic reference
<hnum>	= a hex number, with an 'H' followed by up to 4 hex digits
<dnum>	= a decimal number, 0<= dnum <= 65535

The command interpreter (CI) ignores certain characters, such as spaces, '$'s and even ')'s. When values such as address values are expected, typing characters that are not hex characters will result in an input error. The CI signifies an input error by flashing the screen. To erase a character, use the DELETE key.

For input convenience, there are several special characters, "*",">", and "'". These are used as shorthand ways of entering commands. * means the current value of the PC and > means the current position. The ' is used to indicate a symbolic reference. For instance 'BRKKEY means the address of the symbol BRKKEY. Any symbol in the mini symbol table or any of the predefined system symbols can be symbolically referenced. This is a very powerful and useful feature, and is equivalent to having a built in partial memory map. Anytime an address is expected in a command (<addr>), these special characters can be typed. The * by itself is used to change the contents of the PC.

5-2: BREAKPOINT   B <1-6>,<addr>

The Breakpoint command is used to set one of the six breakpoint registers to a location. If a value other than a 1-6 is entered for the breakpoint register, the command will be terminated. Note that two values (the breakpoint register number, and the breakpoint location) are required for this command. Thus a comma must be typed after the breakpoint number and before the address for the breakpoint.

When a breakpoint is set, that location should show up in the breakpoint register display. Physically, a '0' for the BRK instruction is stored in memory at the requested location. If an Examine command is issued to look at that part of memory, a '0' will be seen, even though the proper mnemonic is shown in the disassembly. If a breakpoint is set at an examined location, the mnemonic will be shown in inverse video. If a breakpoint register is already in use when a new breakpoint is requested, the old breakpoint is first restored.

To clear a breakpoint register and restore the source code, type RETURN after selecting the desired breakpoint register (e.g. typing 'B' then '1' then RETURN will clear breakpoint 1 and restore the source code). Trying to clear a breakpoint that is not set will not harm anything. EXDDT checks to see if a desired breakpoint is actually set, and if not flashes the screen and clears the breakpoint register. This could happen, for instance, if you try to set a breakpoint in ROM.
You can use the special shorthand characters in this command. For instance, to set a breakpoint at the current location being examined in the display window, you could type : B 1,> (and press RETURN).

5-3: DEPOSIT   D <hstring,@>

The Deposit command is used to place a string of bytes in memory. A string of hexadecimal values (up to 12 characters, 6 hex bytes) may be entered. The values entered will be placed in successive locations starting at the current position indicated in the display window, replacing whatever was there. The input string is decoded two characters per hex byte at a time. If there is an odd character left at the end, it will be interpreted as the low order nibble of a hex value. For example, entering a string of 01AAB0 will result in three bytes (01, AA, and B0) being placed in memory. However, entering 01AAB will result in 01, AA, and 0B being deposited. Note that depositing a byte or a series of bytes will not move the display window. This must be done with the examine or the push and pull window commands.

EXDDT is able to switch screens by saving 13 locations the operating system uses in managing the system graphics. Thus, before each value is deposited, it is examined to see if it should be deposited to these graphics locations. If so, the value is placed instead in an internal save table. Similarly, if one of these saved locations are defined as a mini-symbol, then the value shown will be taken from the saved location. Thus, for example, you can deposit values directly to the color shadow registers and affect the color of the user screen and not the EXDDT screen. You might note that this works only if you deposit directly to these locations using the D command. If you're single stepping through a program that deposits to these locations, then the new value will be deposited to the current locations, i.e. to the EXDDT's settings. Toggling the screen twice will restore EXDDT's normal screen.

If the character '@' is typed after a D, EXDDT enters a special mini-assembler mode, as will be indicated in the command window. In this mode, assembly language commands can be deposited directly, e.g. LDX A000. Or, STA ($12,X). Once this mode is entered one instruction at a time is deposited. To exit, merely press RETURN with no value entered. Note also that in this mode the > sign for the current position is removed. Instead a < sign is shown to indicate the current deposit position. This position moves after each assembly language instruction is deposited, even though the window itself remains fixed. If this indicator moves off the bottom of the screen, the mini-assembler deposit mode is automatically exited.

You may use the $ or not, EXDDT ignores it. All values must be in hex, and symbolic references aren't allowed within the mini-assembler. EXDDT also ignores spaces and even right parentheses. Therefore you can type in some funny looking instructions that EXDDT interprets as being hunky dory.

EXDDT checks to see if the instruction you entered is actually deposited in memory. If the result after depositing isn't equal to the desired deposit values, the screen is flashed and the mini-assembler is exited.

5-4: EXAMINE   E <addr>

The Examine command is used to set the display window to view an area of memory. The extreme left hand edge of the display window has a GREATER THAN sign (>) in the 4 th row. This points to the current position that was entered as the address in the 'E' command. Note that the 'E' command does not change the state of the display window filter, nor will it affect which instruction will next be executed by a single step command.

Remember, you can use symbolic references in this command to examine standard system symbols or mini-symbol table symbols (e.g. E 'CIOV).

5-5: GO   G <addr>

The Go command is used to begin execution at a specific location in memory. Before control is transferred to this location, all registers are updated based upon the current contents of the displayed registers. This is true for all commands involving code execution.

5-6: HEX/DECIMAL CONVERTER   H <hnum,dnum>

The H command is used for a hex to decimal / decimal to hex converter. To enter a hex number to be converted, type an 'H' followed by up to 4 hex digits and then press return. This means you will type 2 Hs, one for the command, and one to signify a hex number. A colon will be displayed, followed by the decimal value for the hex number you entered. The result will remain in the command window until you type any character.

To enter a decimal number, just type the number itself after the H command and then press return. EXDDT will display a colon, an H to signify the hex value for the decimal number you entered, and then the hex value. If you enter a number bigger than 65535, EXDDT will flash indicating an error.

5-7: INTERPRETIVE MODE   I

The Interpretive Mode command is used to place the system in an automatic single step mode. After each instruction is interpreted, the screen display is updated if the EXDDT screen is turned on. The display window is automatically placed in the disassembly mode. Pressing the BREAK key halts the interpretive mode. It is possible to run ROM programs such as portions of the O.S. interpretively, but problems with the display and timing can arise. The Trap register is used for setting up the equivalent of a breakpoint in this mode. Interpretive mode will run with either the user screen or the EXDDT screen being shown, but you pay a severe time penalty for selecting the EXDDT screen. Interpretive mode runs much faster if the user screen is selected because EXDDT does not have to update it's screen if it is not active.

If you have V marked symbols in the mini-symbol table, EXDDT checks to see if their values change after an instruction. If so, I mode is halted. This feature is active only if EXDDT is displaying its screen.

5-8: MONITOR   M <0-2>,<addr>

The Monitor command is used to modify the mini symbol table. After typing M, a '0', '1' or '2' should be typed to indicate how many bytes should be displayed. The <addr> is the location you want to monitor.

After entering this command, the cursor will be positioned in the symbol table. Typing RETURN will move the cursor successively to each variable in the table. This allows you to select where in the table to put the new variable. When the cursor is positioned where you want to start the new entry, typing any character other than RETURN will begin the definition of the new label. Press RETURN after entering the label and the new variable will be entered into the symbol table.

There is one special case with the M function. Entered by itself (i.e. pressing RETURN right after typing M), it is a signal to switch the value display from variable values to variable locations or vice versa. This display will toggle

back and forth.

5-9: NEXT

The Next command is used to place an internal breakpoint after the next instruction to be executed. Control is then transferred to the code. In most situations, this will act like a single step instruction, with EXDDT being immediately recalled. However, if the instruction is a JSR, a branch instruction, or any other instruction that can alter program flow, then control might not immediately return. This allows (in most instances) a way to single step over a JSR instruction.

If you try to do an N command when examing code in ROM, the screen will be flashed and control will not be transferred to the next instruction.

5-10: REGISTER   R <A,X,Y,S,P>,<byte>

The Register command is used to modify the contents of some of the 6502's registers. Typing a character other than A,X,Y,S or P after the R will result in the command being terminated. Note that this command requires two separate values, so a comma must be typed after the register designation.

5-11: SEARCH   S <hstring>

The Search command is used to locate a specific sequence of hex characters in memory. You may enter a hex string of up to 12 characters (6 bytes). Memory will be searched from the current position indicated in the display window up through memory. If the search is successful, the display window will be repositioned. If it is unsuccessful, the command window will simply be cleared for the next command.

If no value is entered after the 'S' (i.e. just a delimiter is typed), the previous search string will be used. This will allow for easily finding multiple occurences of the search string.

If, however, a Deposit command is issued, the area where the Search string was stored is overwritten. In this case, the old search string is wiped out.

5-12: TRAP   T <1-2>,<addr>

The Trap command is used for setting one of the Trap breakpoints to a specific location. The address entered should show up in the proper Trap register. Note the trap will only work when in interpretive mode. To clear the trap, type 'T', a '1' or '2' for the trap register you want to clear and then press RETURN. A

0000 should show up in the register.

5-13: VARIABLE TRAP   V

The Variable trap function is used to activate an interpretive trap on the alteration of any byte or word variable. Typing V moves the cursor to the V column on the mini symbol table area. Typing subsequent RETURNs moves the cursor circularly thru the list. Typing a minus sign (-) causes a dash to appear on the screen at the selected variable. Typing the SPACE BAR removes a previously placed dash. Typing any other character causes a return to the command intrepter.

Any time a displayed variable changes, a vertical bar (|) is placed in the V column next to the variable entry. If the variable was previously selected with a dash, a plus sign (dash and vertical together) is displayed. If EXDDT is running in the interpretive mode (I command), modification of a dash selected variable will immediately halt interpretive execution. This can be used to trace where in a program a particular location is modified for instance.

5-14: WINDOW   W

The Window command is used to change the 'filter' over the display window. 'W' toggles between the filters. Two filters are available, a disassembly filter and a hexadecimal filter.

5-15: PULL WINDOW DOWN   <down arrow>

The Pull Window command is used to pull the display window down. Depending upon the display filter in place, this will pull the window down one byte (hex filter) or by one full instruction (disassembly filter). Note that Auto Repeat on the keyboard is active, so that continuing to press the down arrow key (i.e. the '=' key) will continue to pull the window down.

If the shift key is held down while typing the down arrow character, the screen will be pulled down a full screen each time.

5-16: PUSH WINDOW UP   <up arrow>

The Push Window command is used to push the display window up. Depending upon the display filter in place, this will push the window up one byte (hex filter) or by one full instruction (disassembly filter). Again the Auto Repeat on the keyboard is active, so that continuing to press the up arrow key (i.e. the '-' key) will continue to push the window up.

If the shift key is held down while typing the up arrow character, the screen will be pushed up a full screen. A problem occurs however when you arbitrarily examine an area of memory with the disassembly filter in. If you try to push the window up, there is not enough information to be able to tell if the preceding instruction was one, two or three bytes long. EXDDT keeps track of how many bytes the window is moved each time you pull the window down. Thus you can push the window back up if you have previously pulled it down past an instruction or group of instructions. Refer to the technical appendix for information on this feature.

5-17: RESET PROGRAM COUNTER   * <addr>

The * followed by an address is used to reset the program counter. You can use the special short hand characters in this command. So, for instance, to set the PC to where you're examining in memory, type * >. To set the PC to one of your mini-symbols, type * 'LOOP1.

Section 6:

EXDDT ENTRY POINTS

There are three ways of entering EXDDT:

FLASH ENTRY
WARM ENTRY
BREAKPOINT ENTRY

6-1: FLASH ENTRY

This entry point is provided to allow immediate entry to EXDDT regardless of other circumstances. This is a single keyboard special character, and is set up as [CTRL] [SHIFT] ESC (i.e. pressing the Control, the Shift and the Escape keys at the same time). When EXDDT is initialized, the operating system code that looks at the keyboard is modified so that it looks for the special character first before handling normal keyboard input. If this character is found, EXDDT is entered immediately through the FLASH ENTRY point.

Pressing START will return control to wherever the processor was when the EXDDT special character was typed. For more information on the Flash entry mechanism, see the Keyboard Scanner section in the Technical Details appendix.

6-2: WARM ENTRY

This entry point is the starting point for the EXDDT code. The first three bytes of EXDDT are a JMP to the set symbol table routine. The EXDDT code itself immediately follows this. If this location is called via a JSR instruction, then the START button exit will return control to the calling point. This allows EXDDT to be called at various program locations for setting up breakpoints, changing values, etc.

Example

```
      .
      .
--your code--
      .
     PHA       ;this doesn't mean
               ;anything, only an
               ;example
  JSR $6003    ;the address of the
               ;unrelocated version
               ;
--Pressing START will return here--
      .
--your code--
      .
      .
```

6-3: BREAKPOINT ENTRY

Breakpoint entries are the most common way of entering EXDDT. The breakpoints first have to be set up via a FLASH or WARM entry to EXDDT. After they are set, EXDDT will be called if those specific instructions are executed. Exits from EXDDT breakpoints return to the code sequence where the breakpoint was located. Notice that the breakpoints will remain in place unless they are explicitly cleared, or RESET is pressed. This is true even if a breakpoint has been tripped.

Recall also that if the trap register is set in interpretive mode, then attempting to execute the instruction at that address will halt the interpretive mode. Thus to move past a trap breakpoint in interpretive mode, you have to either clear the trap or single step past the instruction that was trapped and then enter interpretive mode.

APPENDIX (TECHNICAL DETAILS):

A-1: INTERACTIONS WITH DOS

EXDDT is designed to be compatible with most programs that adhere to Atari O.S. design considerations. EXDDT can either be loaded first separately or placed as a header file for the program of interest. EXDDT is set up to return to the program that loaded it after loading and initializing. Normallly, the preparation sequence would be to load whichever editor you use and then create the assembly language source file. Next, load your assembler and create an object code file. At this point you would load EXDDT.OBJ, which would load, initialize, and then return to the DOS you're using. Finally, From there you would load the actual object code itself.

A-2: KEYBOARD SCANNER

During EXDDT initialization the system keyboard vector is redirected to a preprocessor which checks for the EXDDT FLASH ENTRY special character ([CNTL][SHIFT] ESC). If this character is seen, control transfers to the FLASH ENTRY point, otherwise control passes to the normal keyboard processing routine.

Note that keyboard interrupts MUST be enabled. Pressing RESET will usually reset the system vectors, even if they have been changed.

A-3: EXDDTS USE OF SYSTEM RESOURCES

EXDDT itself occupies 8K of memory space. The preassembled version is ORGed at $6000, with 600 additional bytes at $5C00. Every attempt was made to minimize EXDDT's use of system resources. Unfortunately, it is not possible to completely avoid using resources in 3 areas: Page Zero Ram, a storage area for EXDDT variables and some screen storage, and the major portion of the screen display.

Page Zero - After much deliberation, I convinced myself that EXDDT would simply have to use a large portion of the upper half of Page Zero. One main implication of this is that if a cartridge is installed that uses the upper half of page zero, then you won't be able to run that cartridge interpretively using EXDDT. In many instances you will be able to call EXDDT from the cartridge, and then reset to get back to the cartridge.
All of the Page Zero locations that are not normally touched by BASIC or the Assembler/Editor cartridge are kept open for your usage ($B9 - $DB are unused. Thus you can write and debug programs that use these locations in Page Zero. The untouched locations include the ones Basic uses for error information and the floating point register that interfaces with USR calls. You can easily create assembly language routines that will run when called from Basic.

EXDDT Variable storage - The area from $400 - $584 is used by EXDDT for local variable storage, the symbol table and parts of the screen display. One thing this means is that you won't be able to debug programs which use the cassette buffer, or the floating point scratch area.

EXDDT Screen Display - Finally, EXDDT needs another 600 bytes for the main portion of its screen display. In the preassembled version this is set at $5C00. However, this is easily relocated, as location $FE contains a pointer to the intended screen loacation area. Each time EXDDT sets up its display screen, it looks at loacation $FE to get the location where the screen should be built. To put the display window screen in another place, for example $5800, simply create a small assembly language program like:

```
        ORG $FE
        DW  $5800
```

(Users of some assemblers may need to use *= instead of ORG).

Assemble this code fragment and copy the object code to the EXDDT.OBJ code using the append option. Thus when EXDDT is activated, it will see your pointer to the desired screen area.

You can also set this location from within your program under test. Look at the program TEST.MAC to see an example of this.

A-4: DISPLAY WINDOW MOVEMENT

EXDDT maintains a "pull stack" while the disassembly filter is in place. This means that each time you pull the display window down, EXDDT places the number of bytes that the window was pulled in a stack. Thus when you want to push the window up, EXDDT checks to see if there are any values left in the pull stack. If so, you can push the window up. If not, nothing happens. The pull stack is cleared whenever EXDDT is entered, or when an Examine command is typed. To conserve memory, 4 pull values (which will be a 1, 2 or 3) are packed into each byte in the stack. A total of 64 bytes are reserved for the stack. Thus you can pull the window down 256 times before the stack runs out. After this occurs the first values in the stack are lost and you can't back up as far. In computer terms, the stack is implemented as a circular buffer.

A-5: THINGS TO WATCH OUT FOR

There are a few areas where you have to be careful in using EXDDT. In general, these occur when you are single stepping or running interpretively. If the interpreted code messes around with the display list, or with ANTIC, or CTIA/GTIA, then you might end up with a scrambled screen. Usually this is non fatal, just distracting. To restore the normal EXDDT screen, press the BREAK key to halt the interpretive mode, then press SELECT twice.

Trying to do I/O from disk or any other real time activity in either interpretive mode or single step mode will probably not work. You should set up breakpoints so that this type of I/O is done in real time, and then call EXDDT.

A-6: RELOCATING EXDDT.OBJ

The version of EXDDT.OBJ that comes with this diskette is ORGed to load and run at $6000 hex. The display screen is set to default to $5C00. This loacation was picked to put EXDDT.OBJ up pretty high above areas where you might want to ORG your code, yet not run into the normal areas for standard O.S. display screens. If for some reason you want to relocate EXDDT, there is a Basic program provided called RELOCATE which you can run. Listen, if you're looking for elegant relocaters, don't look at RELOCATE. It is implemented in the most simple fashion I could think of. Namely RELOCATE has a list of the locations that need to be changed to have EXDDT run at some new location in memory. EXDDT must be set up so that it starts on a Page boundary, so when it asks for a relocation address, only a two character hex value need be entered. RELOCATE assumes that EXDDT.OBJ is located on in drive 1. It also assumes the new file will be called EXDDT.REL. Naurally, you can rename the new file after it has been created. RELOCATE then asks where you would like the new version of EXDDT to load and execute. Finally, RELOCATE asks for a new default location for creating the display screen. Again, this should begin on a page boundary, so only a single hex byte value should be entered. RELOCATE will work automatically from then on. Might as well go get something to drink now, 'cause it's gonna take a few minutes (about eleven) for this pore li'l Basic program to work. Adios Amigos. Now get out there and get those bugs before they get you.