6800/6801/6303/6811 DEBUGGER

USER'S MANUAL

5th Printing

NOTICE

This manual describes IDB Version 1.2. Software Dynamics has carefully checked the information given in this manual, and it is believed to be entirely reliable. However, no responsibility is assumed for inaccuracies. Software Dynamics reserves the right to change the specifications without notice.

IDB (INTERPRETIVE DEBUGGER) USER'S MANUAL

TABLE OF CONTENTS

IDB (INTERPRETIVE DEBUGGER) USER'S MANUAL

INTRODUCTION

IDB is a small memory, stand-alone debugger for 6800, 6801, 6303 or 6811 microprocessor systems. It is ideal for debugging assembly language programs. IDB acts as a very sophisticated replacement to the "MIKBUG" ROM available with most 6800 development systems (MIKBUG is a registered trademark of Motorola Inc., and any reference herein is to their registered trademark).

IDB allows the programmer to load and dump programs in MIKBUG format; to display large blocks of memory; to examine memory locations in several display formats; to modify memory locations; to single-step programs; to set breakpoints and execute a program; and to search or fill memory for/with a particular value.

This manual describes IDB Version 1.2.

OPERATION

IDB is usually burned into a set of ROMs and resides permanently in the development system. However, it can also be kept on external media and loaded when a debug session is about to begin; this scheme is not as safe (since the debugger might get clobbered).

IDB comes configured to communicate to the user through a serial port, usually an ACIA or 6551. This port is called the "console" device. Normally, the port is connected to a teletype or CRT device. The console device can be changed by modifying a jump table.

The programmer interacts with IDB via commands given at the keyboard. IDB gives no prompt; if no display action is occurring, IDB is in command input mode. The programmer enters commands, and if the command is valid, IDB executes the command and then returns to command input mode. IDB checks the input character by character. If an entry is syntactically or semantically incorrect, it is diagnosed immediately by a print-out of "??" followed by a carriage-return and IDB remains in the command input mode. If there is a command error, the opened location is closed.

In the examples included in this manual, underlined characters are keyed in by the operator. Comments to the right do not appear as output of the debugger; all other printed data is typical debugger output. Many of the examples use previous examples to set up a known situation.

All IDB commands and hexadecimal numbers can be entered in either upper or lower case; in this manual only uppercase commands are shown, and a small letter immediately to the left of a command represents a numeric value entered by the operator immediately before the command keystroke.

COMMAND FORMAT

All commmands to IDB fit one of the following forms:

C	(No Parameter)
nC	(Single Parameter)
n;C	(Single Parameter)
n,mC	(Double Parameter)
;C	(No Parameter)

where n is a value (hex number up to six digits depending upon the command) or simple hex arithmetic expression and m is a hex number. <CR> is a carriage-return, <LF> is a line-feed and C is a command character (letter, punctuation mark, <CR>, or <LF>). ";" is a semicolon and "," is a comma.

VALUES ENTERED INTO THE DEBUGGER

IDB accepts several formats for numbers:

Hex numbers, a string of hex letters or digits:
    0A BC9 22 BD3FA9

Single characters representing special values:

- . (Period), meaning the address of the last opened memory location, whether it is open now or not. This is referred to as the open location marker.
- * (asterisk), meaning the value that would be displayed as the P register contents on a register dump (location of next instruction to execute).
- # (pound sign), meaning the number of instructions single-stepped since last ";#" command.

'c (single quote, followed by any character), meaning "the ASCII value of the character c". 'A is equivalent to typing in 41 (hex); likewise, 'b = hex 62.

SIGNIFICANCE

Numbers entered into IDB have significance (size in bytes) based on the number of digits keyed in. This significance is used by commands which store into memory or do hex arithmetic.

1 or 2 digits gives 1 byte significance
3 or 4 digits gives 2 byte significance
5 or 6 digits gives 3 byte significance
Special values (., *, #) have 2 bytes of significance
'c has 1 byte of significance

IDB COMMANDS

IDB commands fall into the following categories:

- Set The Display Mode
- Examine and Modify Memory
- Hex Arithmetic
- Set Register
- Zero (Fill) and Search With Mask
- Compute a Relative Displacement
- Dump Memory
- Switch the Dump/Search Output Channel
- Load Memory
- Punch Memory
- Breakpoints
- Execute and Single-Step

SETTING THE DISPLAY MODE

The display mode commands affect the way the register display and memory examine commands display the currently open location.

COMMAND	OPERATION
;A	Set Display Mode to ASCII
;H	Set Display Mode to Single Byte Hex
;X	Set Display Mode to Double Byte Hex
;O	Set Display Mode to Instruction

The ;A display mode allows values to be displayed as ASCII characters. If a character is non-printable (hex 0-1F, 7F-9F, FF), then the byte is displayed in ;H mode.

The ;H display mode allows values to be displayed as single-byte hex quantities. This display mode is default upon IDB startup.

The ;X display mode allows values to be displayed as double-byte hex quantities.

The ;O display mode allows values to be displayed as instructions. If an illegal instruction begins in the location being examined, then a "?" followed by a single-byte value is displayed. Otherwise, the instruction display format depends on whether the symbolic disassembly option has been enabled.

SYMBOLIC DISASSEMBLY: Instructions are displayed in a format consistent with the SD assembler. Addresses, immediate values and offsets are shown as hexadecimal constants of the appropriate significance.

NO SYMBOLIC DISASSEMBLY: If a single-byte instruction begins in the location being examined, then a single hex byte is displayed. If a double-byte instruction begins in the location being examined, then two hex bytes are displayed. If a triple-byte instruction begins in the location being examined, then three hex bytes are displayed.

Display modes are not affected by single-stepping or user program execution. They may only be changed by explicitly typing in a new display mode command. If a display mode command is entered while a location is still open, the value in that location will automatically be displayed in the new mode.

EXAMINE AND MODIFY COMMANDS

The examine and modify commands are used to display and/or change memory locations and registers.

COMMAND	OPERATION
l/	Open Location l and Display in Current Mode
(LF)	Display Next
n(LF)	Deposit and Display Next
^	Display Previous
n^	Deposit and Display Previous
(CR)	Close This Location
n(CR)	Deposit and Close Location
l:	Open Location l
"text"	Deposit ASCII Text String
?	Display Registers, Current Instruction, and Last Opened Location

The l/ command is used to open location l and display its contents in the current mode. "Opening a location" means to make it available for examination and/or modification.

The (LF) (line-feed) command is used to advance the open location marker and display the contents of the new location in the current mode. If the current mode is ;H or ;A, the open location marker is bumped by one, and the next byte is displayed. If the current mode is ;X, the open location marker is bumped by two, and the next two bytes are displayed. If the current mode is ;O, then the open location marker is bumped by the length of the instruction (1 if the instruction is illegal) and the next instruction is displayed. (LF) is only valid when a location is open.

The n(LF) command is used to deposit from one to three bytes. The open location marker is bumped by the significance of n, regardless of display mode, and the contents of the new location are displayed in the current mode. n(LF) is only valid when a location is open.

The ^ (up arrow) command is used to decrement the open location marker by one and display the contents of the new location in the current mode. ^ is only valid when a location is open.

The n^ command is used to deposit from one to three bytes. The open location marker is decremented by one, regardless of display mode, and the contents of the new location are displayed in the current mode. n^ is only valid when a location is open.

The <CR> (carriage-return) command is used to close the currently open location. The open location marker is not advanced. <CR> is a no-op when a location is not open.

The n<CR> command is used to deposit from one to three bytes into the open location. The open location marker is not advanced and the location is closed. n<CR> is only valid when a location is open.

The l: command is used to open location l. No display occurs.

The "text" command is used to enter ASCII text strings into memory. The opening " character signifies the start of this data entry mode, but does not actually deposit any data. The ASCII code for each character (keystroke) following the leading " is deposited into memory, and the open location is advanced by 1. Data entry is terminated by the second ", which does not cause any further data to be deposited. IDB then automatically displays the contents of the new value of the open location, as though <LF> had been typed.

The ? Command is used to display the registers, the next instruction (in ;O mode), and the last opened location in the current display mode. This display is referred to as a register dump elsewhere in this manual. In a register dump, the contents of the registers follow the letter naming that register; the next instruction follows the */ (* means "value of PC"), and the contents of the last open location are shown as nnnn/dddd..... Not shown in this manual, but displayed on the 6811 version, is a place for the Y register in each register dump.

Examples:

```
100: 45<CR>                 OPEN LOCATION 100 AND DEPOSIT 45.  THE
                            LOCATION IS CLOSED.
./ 45 ;A/ E <LF>            EXAMINE LOCATION 100; CHANGE TO ;A MODE,
                            SEE VALUE IN ASCII; EXAMINE NEXT.
0101/ F ;H/ 46 <CR>         CHANGE TO ;H MODE, SEE VALUE IN HEX;
                            CLOSE THE LOCATION.
./ 46 <LF>                  OPEN THE LAST LOCATION; EXAMINE NEXT.
0102/ BD ;O/ BD7E00 BD7E05<LF>
                            CHANGE TO ;O MODE, SEE VALUE AS INSTRUCTION;
                            CHANGE VALUE AND EXAMINE NEXT.
0105/ 39 01^                DEPOSIT AND EXAMINE PREVIOUS
0104/ ?05 ^                 STILL IN ;O MODE, 05 IS ILLEGAL OP CODE;
                            EXAMINE PREVIOUS
0103/ 7E0501 ^              CONTENTS OF 103 LOOKS LIKE "JMP" INSTRUCTION;
                            EXAMINE PREVIOUS.
0102/ BD7E05 ;X/ BD7E <CR>
                            CHANGE TO ;X MODE; CLOSE THIS LOCATION.

200/ 0072 "ABCD<CR>
<LF>DEF"                    DEPOSIT TEXT DATA
209/ 992A 200/ 4142 ;A/ A <LF>
201/ B <LF>
202/ C <LF>
203/ D <LF>
204/ 0D <LF>
205/ 0A <LF>
206/ D <LF>
207/ E <LF>
208/ F <LF>
209/ 99 <CR>
?                           SHOW REGISTERS
P=3005 A=01 B=FE C=C0 X=3031 S=4073 */ 7E3068 0105/ 01
;O                          SWITCH TO OPCODE DISPLAY
*/ 7E3608 39<CR>            FIX INSTRUCTION AT P COUNTER
?
P=3005 A=01 B=FE C=C0 X=3031 S=4073 */ 39     3005/ 39
```

HEX ARITHMETIC

Hex arithmetic is used to evaluate expressions.

COMMAND	OPERATION
-n	Find Negative of n
n-m	Find Difference
n+m	Find Sum
n=	Print Value

The -n command is used to take the two's complement of a one or two byte value.

The n-m command yields the two's complement difference.

The n+m command yields the two's complement sum.

The n= command is used to print out the current value using the appropriate significance.

Note that all arithmetic (negate, add, and subtract) only operates on one or two byte operands, and if a three byte operand is given, the leftmost byte is ignored and the significance becomes two rather than three. Also, significance is maintained in all arithmetic operations. For instance, adding one byte to one byte yields an answer of one byte whether or not a carry-out occurred. Adding two bytes to one byte will give two bytes of significance. The significance of the result will always by one or two bytes. When in doubt as to the significance of a result, use the print value operator (=). A result of an arithmetic operation is treated as though the programmer had typed in that value itself, and may be followed by commands requiring values.

Examples:

```
-1=FF                       NEGATE 1 BYTE VALUE
-001=FFFF                   NEGATE 2 BYTE VALUE
5-6=FF                      1 BYTE DIFFERENCE
0005-6=FFFF                 2 BYTE DIFFERENCE
6-005=0001
110005-6=FFFF               3 BYTES BECAME 2
5F+6A=C9                    1 BYTE SUM
FF+1=00
00FF+1=0100
5=05                        PRINT 5
FF00=FF00                   PRINT FF00
FF0000=FF0000               PRINT FF0000
FF0000+1=0001               ANSWER IS 2 BYTES ONLY
1-2+3-4+5=03
94+C/ FE <CR>               LOOK AT LOCATION A0
.=00A0 <CR>                 PRINT ADDRESS OF LAST OPENED LOCATION
                            IGNORE VALUE (SPACE) AND GO TO NEW LINE (<CR>)
5+.=00A5/ 37 <CR>           EXAMINE LOCATION .+5
'A-1=40+'Z=9A<RUBOUT>?? USE RUBOUT TO GET RID OF VALUE
100/ 2205 ;H/ 22 'A-'0-A=07<CR>
?
P=3005 A=01 B=FE C=C0 X=3031 S=4073 */ 39     0100/ 07
*-2=3003/ FF ;O/ FF0039
```

SETTING REGISTERS

The following commands are used to change the contents of a specific register by name.

COMMAND	OPERATION
n;A	Set A Register to n
n;B	Set B Register to n
n;C	Set C Register to n
n;D	Set D Register to n
n;X	Set X Register to n
n;Y	Set Y Register to n
n;S	Set S Register to n
n;P	Set P Register to n

The n;A n;B n;C commands set registers A B C respectively to the rightmost byte of n.

The n;X n;S n;P ;D ;Y commands set registers X S P D Y respectively to the rightmost two bytes of n (the D register consists of A and B treated as a 16 bit value; the Y register is present only on the 6811). If a one byte value is given, a leading zero byte is assumed.

When the stack pointer is set, IDB assumes that the value given, minus 6 (minus 8 for the 6811), points to a (interrupt) context block (i.e., n-7+1 (n-9+1 for 6811) points to a condition code byte). The contents of this context block are used as the values of the registers.

When IDB starts up, it invents a seven (nine for 6811) byte stack for the user's context block using a value specified by the INITZ routine. If this value is not appropriate, it is a good idea to assign (via n;S) a convenient stack before doing any debugging.

Examples:

```
1;A                SET THE A REGISTER TO 01
FE;B               SET THE B REGISTER TO FE
C0;C               SET THE C REGISTER TO C0
?                  SHOW REGISTERS
P=3005 A=01 B=FE C=C0 X=3031 S=4073 */ 7E3068 0105/ 01
1234;D             SET THE A REGISTER TO 34
123456;B           SET THE B REGISTER TO 56
?                  SHOW REGISTERS
P=3005 A=34 B=56 C=C0 X=3031 S=4073 */ 7E3068 0105/ 01
1;X                SET X TO 0001
?                  SHOW REGISTERS
P=3005 A=34 B=56 C=C0 X=0001 S=4073 */ 7E3068 0105/ 01
1234;X             SET X TO 1234
?                  SHOW REGISTERS
P=3005 A=34 B=56 C=C0 X=1234 S=4073 */ 7E3068 0105/ 01
FE;P               SET P REGISTER TO 00FE
FE/ 00             LOOK AT LOCATION FE
?                  SHOW REGISTERS
P=00FE A=34 B=56 C=C0 X=1234 S=4073 */ ?00    00FE/ 00
FE/ 00 2245<CR>                 MAKE IT AN INSTRUCTION
?                  SHOW REGISTERS
P=00FE A=34 B=56 C=C0 X=1234 S=4073 */ 2245   00FE/ 22
;X                 CHANGE DISPLAY MODE
?
P=00FE A=34 B=56 C=C0 X=1234 S=4073 */ 2245   00FE/ 2245
;A                 CHANGE DISPLAY MODE
?
P=00FE A=34 B=56 C=C0 X=1234 S=4073 */ 2245   00FE/ "
```

Caution: setting the stack pointer (S register) causes the remaining registers to take on arbitrary new values according to their positions in the context block pointed to by the new value of the S register!!

```
FE;S               SET THE STACK POINTER TO 00FE
?                  SHOW REGISTERS
P=0022 A=F4 B=45 C=C0 X=789F S=00FE */07      00FE/ "
```

ZERO (FILL) AND SEARCH WITH MASK

The fill commmand is used to fill memory with a one, two or three byte value from a mask. This is effectively a zero command when the mask is zero. The search command is used to search memory for a one, two or three byte value using the mask.

COMMAND	OPERATION
nM	Set Mask
M	Show Mask
nS	Set Search Target
S	Show Search Target
n,mS	Search Using Mask Between n and m
n,mZ	Zero (Copy Mask to Memory) Between n and m
n,m?	Checksum memory

The nM command is used to define a mask for the search and zero (fill) commands. The mask may be one, two or three bytes long with one bits specifying the bit positions to ignore (mask out) while searching. The mask is defaulted to a single-byte zero upon IDB startup.

The M command is used to show the last value defined as the mask as a one, two or three byte value.

The nS command is used to define a search target to be used with the search command. The search target may be one, two or three bytes long specifying the exact sequence of bits to search for. Selected bit positions of the search target may be overriden by one bits in the mask.

The S command is used to show the last value defined as the search target as a one, two or three byte value.

The n,mS command is used to search memory between n and m inclusive for the occurance of the search target. The mask is used while searching to specify bits in the search target and the memory to ignore. The mask must be the same length as the search target. The search command will print out the address and contents of that address for each match found. Note that m-n+1 search attempts are made regardless of search target length.

If the search target and the mask are three bytes long, then three bytes are printed out for each match. IDB will compare against loc n, n+1, n+2 for a match; then n+1, n+2, n+3, through loc m, m+1 and m+2.

If the search target and the mask are two bytes long, then two bytes are printed out for each match. IDB will compare against loc n, n+1, for a match; then n+1, n+2, through loc m, m+1.

If the search target and the mask are one byte long, then one byte is printed out for each match. IDB will compare against loc n for a match; then n+1 through loc m.

Note that a match may occur if a search target begins within the limits (inclusive), even though the remaining bytes may cross the limit. The output of the search command is normally directed to the console device. The output can be switched to the dump channel by using the T command. The search command may be interrupted at any time by typing an escape character on the console device.

The n,mZ command (fill) is used to copy the mask to memory between n and m inclusive. If the mask is one byte long then the mask is copied m-n+1 times into locations m, m+1, m+2... n-1, n. If the mask is two or three bytes then the mask is copied INT((m-n+1)/2) or INT((m-n+1)/3) times respectively into memory with any leftover bytes being filled with leading mask bytes. For example, if the mask is three bytes and "100,107Z" is entered on the console device, then the mask is copied to locations 100 through 102 and 103 through 105, and locations 106 and 107 get the left-most two bytes of the mask. The zero (fill) command never modifies a location past the address given as the second parameter.

The n,m? command is used to compute a simple checksum over the address range n thru m, inclusive. The value of the checksum is printed. This is used mostly for fast determination of whether a portion of RAM has changed or not.

Examples:

```
0M                SET MASK TO ZERO (ONE BYTE)
M 00              SHOW MASK
100,1FFZ          FILL 100 THRU 1FF WITH ZEROES
FFM               SET MASK TO FF (1 BYTE)
100,1FFZ          FILL 100 THRU 1FF WITH FF
BD3F92S           SET SEARCH TARGET TO BD3F92 (3 BYTES)
000000M           SET MASK TO 3 BYTES (IGNORE NO BITS)
2000,3000S        FIND SEARCH TARGET BETWEEN 2000 & 3000 INCLUSIVE
20FE/ BD3F92      FOUND IT HERE
219A/ BD3F92      FOUND IT HERE
3000/ BD3F92      FOUND IT HERE
S BD3F92          SHOW SEARCH TARGET
0000FFM           SET MASK TO IGNORE LAST BYTE OF SEARCH
2000,20FFS        FIND ALL JSR'S TO 3FXX
2010/ BD3F5A      FOUND IT HERE
20FE/ BD3F92      FOUND IT HERE
7E0100M
5001: FFFF<CR>
4000,5000Z        INSTALL "JMP $100" INSTRUCTIONS BETWEEN 4000 & 5000
;O
4000/ 7E0100 <LF>        LOOK AT WHAT WE DID
4003/ 7E0100 <CR>
4FFF/ 7E01FF      NOTE THAT LOC 5001 WAS UNTOUCHED
1000,2000? 57     CHECKSUM LOCATIONS 1000 THRU 2000
```

COMPUTE RELATIVE DISPLACEMENT COMMAND

This command is used to compute the relative displacement byte of relative branch type instructions.

COMMAND	OPERATION
nR	Compute Relative Displacement

This command is used to find the difference in addresses as a one byte value between .+1 and n. If the address given (n) is outside the range of a relative branch-type machine instruction, an error will occur. The way this command is used is to open a location where a relative displacement byte is to be deposited, and specify the target address (n) followed by "R".

Examples:

```
100/ 2021 ;H/ 20 <LF>  WE HAVE A "BRA $123"
101/ 21 105R=03<CR>    TELL IDB TO MAKE A "BRA $105", DISPLAY THE
                       DISPLACEMENT, THEN DEPOSIT IT
.-1/ 20 ;O/ 2003       NOW GO CHECK ENTIRE INSTRUCTION

.+1/ ?03 0R??          TELL IDB TO MAKE A "BRA $0"; HE SAID
                       THAT'S TOO FAR!
./ ?03 90R<CR>         TELL IDB TO MAKE A "BRA $90"
.-1/ 208E              NOW CHECK ENTIRE INSTRUCTION
```

DUMP MEMORY COMMAND

This command is used to display large areas of memory in hex and ASCII on the dump device.

COMMAND	OPERATION
1,n/	Dump Memory to Dump Device

The area dumped is specified by 1 and n. 1 is used as an address; n may be a byte count (significance of one) or an address (significance of two). If a byte count is used as the second parameter, dumping begins at 1 and continues for n bytes. If an address is used as the second parameter, dumping begins at 1 and continues until address n is reached (inclusive). Beware of specifying a second parameter address that is smaller than the first parameter address; an awful lot of memory will be dumped!! The output of the dump is normally directed to the console device. The output can be switched to the dump channel by using the T command. The dump device may be a console, printer, or a disk file, depending upon the configuration. See the section on I/O entry points to find out how to change the dump device. The dump may be interrupted at any time by typing an escape character on the console device. This causes the dump to stop and IDB to return to the command input mode.

After dump is complete, location 1 is opened for changes or re-display in a different display mode.

Examples:

```
50,10/          DUMP 16 BYTES
0050/ 00 7D CD 9D 80 9F 84 00 00 00 00 20 39 30 31 31  .}M........ 901

55,061/                    DUMP FROM ADDRESS 0055 TO 0061
0055/ 9F 84 00 00 00 00 20 39 30 31 31                 ...... 9011
0060/ 5A 9A                                            Z.

100,1FF/        DUMP FROM 100 TO 1FF INCLUSIVE
0100/ 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  ................
0110/ 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F  ................
0120/ 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-.
0130/ 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>
0140/ 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMN
0150/ 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^
0160/ 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F  `abcdefghijklmn
0170/ 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~
0180/ 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F  ................
0190/ 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F  ................
01A0/ A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF   !"#$%&'()*+,-.
01B0/ B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF  0123456789:;<=>
01C0/ C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF  @ABCDEFGHIJKLMN
01D0/ D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF  PQRSTUVWXYZ[\]^
01E0/ E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF  `abcdefghijklmn
01F0/ F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF  pqrstuvwxyz{|}~
```

The address on the left side of the page is the address of the first byte printed. Addresses increase by one for each byte displayed from left to right, so that the address of the $5F byte is $15F. The characters to the right are in one-to-one correspondence from left to right with the displayed hex bytes, and are the ASCII equivalents of the bytes dumped. Control characters and $7F, $FF are printed as a period. The parity bit is ignored.

SWITCHING THE DUMP/SEARCH CHANNEL

COMMAND	OPERATION
nT	Switch Dump/Search Output Channel

The nT command is used to switch the dump/search output between the console and the dump channel. If n is zero, then the console is selected for dump/search output. If n is non-zero, then the dump channel is selected for dump/search output. Console output is defaulted upon IDB startup.

Examples:

<u>ØT</u>	SET DUMP/SEARCH OUTPUT TO CONSOLE
<u>1T</u>	SET DUMP/SEARCH OUTPUT TO DUMP

LOAD COMMAND

This command is used to load programs in MIKBUG format from the load device. Certain implmentations of IDB can load SDOS load records instead of MIKBUG, and other implmentations may not have a load command at all.

COMMAND	OPERATION
;L	Load object records from Load Device

A successful load will print the address of the last byte loaded. This can be helpful in situations where you don't know how big a program is, or if you do, you have an extra verification that all is ok. If a checksum error occurs while loading, or an illegal character is encountered in a load record, the first address of the block being loaded is printed out followed by "??" indicating the error. The load may be interrupted by typing an escape character on the console device at any time. The address of the block being loaded will be printed out, showing how far the load had progressed before being interrupted, and then IDB will return to the command input mode. See the section on I/O entry points to find out how to change the load device.

Examples:

```
;L 17E3                     SUCCESSFUL LOAD, LAST BYTE WENT INTO 17E3
;L 01B0??                   CHECKSUM ERROR ON BLOCK 1B0
;L 1200                     ESCAPE WAS TYPED AFTER THE LOAD BEGAN.
                            IDB SAYS THAT HE WAS ON BLOCK 1200
                            WHEN INTERRUPTED
```

PUNCH COMMAND

This command is used to dump ("punch", a term inherited from paper tape days) memory out in to the punch device. The standard object file format produced is MIKBUG, but certain implementations of IDB may produce SDOS object records, or the punch command may not be present at all.

COMMAND	OPERATION
1,nP	Punch object records to Punch Device

The area punched is specified by the addresses l and n. Punching begins at l and continues until address n is reached (inclusive). Beware of specifying a second parameter address that is smaller than the first parameter address; an awful lot of memory will be punched!! See the section on I/O entry points to find out how to change the punch device. Punching may be interrupted at any time by typing an escape character on the console device. This causes the punching to stop and IDB returns to the command input mode. Note that an end record ("S9" in MIKBUG format) is not punched. This allows the punching to the same file of different and not necessarily contiguous areas of memory. When all punching is complete, the end record can be punched by entering the command "Ø,ØP".

Examples:

```
100,200P        PUNCH OUT 100 THRU 200 ...
3F00,3FFFP      FOLLOWED BY 3F00 THRU 3FFF
0,0P            END FILE ON PUNCH
```

BREAKPOINT COMMANDS

"Breakpoints" are used to stop a program at a certain place so that the state of the machine can be examined. The programmer places breakpoints in his program where he would like to be able to interrogate the machine registers and whatever else may be interesting; then he tells IDB to run his program (see G commands). When the program hits a breakpoint, control is passed to IDB, which does a register dump. The programmer can then examine or change memory, place new breakpoints, start his program again or continue execution from where it left off. The breakpoint commands are used to set up to four realtime conditional or unconditional breakpoints, showing breakpoints, and deleting breakpoints.

COMMAND	OPERATION
l!	Set Unconditional Breakpoint on Address l
l,c!	Set Conditional Breakpoint
!	Show Breakpoints
l\	Remove Breakpoint from Address l
K	Kill All Breakpoints

An IDB breakpoint instruction (BKPT) may be either a SWI instruction or a three byte extended JSR instruction, depending on configuration. Associated with each BKPT is an iteration counter and a conditional subroutine. The BKPT instruction is "planted" at the breakpoint location during realtime execution to regain control when encountered. The conditional subroutine is used to return "true" or "false" depending upon some arbitrary user-specified conditions. The iteration count is used to count down the "true" responses from a conditional subroutine until the counter becomes zero, at which time the breakpoint is considered to be "hit".

There are two types of breakpoints: conditional and unconditional. Conditional breakpoints are associated with a user-defined conditional subroutine. Unconditional breakpoints are really conditional breakpoints that are associated with an IDB-defined conditional subroutine that always returns "true".

IDB uses the BKPT instruction at each breakpoint to regain control after encountering a breakpoint during realtime execution. [Since JSR BKPT takes three bytes, no JSR-styple breakpoint may be set within two bytes of another breakpoint (IDB won't allow it)]. Note that setting breakpoints in ROM doesn't work, as they cannot be stored at execution time. This may not be obvious since the breakpoints are invisible to the user (they can't be seen in the user's code) while IDB is in the command input mode. The BKPT instructions are not "planted" in the user code until realtime execution is requested (see G commands), so that if a breakpoint were set at location 100 (by entering "100!"), examination of location 100 will still show the original user code rather than IDB'S BKPT instruction.

When the user's program is executing, and it encounters a BKPT, the conditional subroutine is invoked and the iteration count for that breakpoint is decremented by one if the subroutine returns "true". If the counter goes to zero, then the breakpoint is "hit"; all BKPTs are removed, the original user code is restored and a register dump is displayed on the console device. IDB then enters command input mode. Otherwise (the breakpoint was not hit), the single-stepper will simulate instructions until the P register is outside the region that the BKPT instruction occupies (in case it is a JSR), then realtime execution continues without any notification to the user that a BKPT was encountered (and not "hit").

A conditional breakpoint "hit" happens when the conditional routine for that breakpoint signals condition true for n times, where n is the iteration count for that breakpoint (initially set to one at breakpoint setting). Note that the iteration count is not decremented if the conditional routine returns "false" condition.

An unconditional breakpoint "hit" happens when the breakpoint is encountered n times, where n is the iteration count for that breakpoint (initially set to one at breakpoint setting). Note that the iteration count is always decremented because the conditional subroutine used by IDB always returns "true" condition.

When a breakpoint hits, the next instruction to execute is the one at the breakpoint address (the instruction at the breakpoint has not yet been executed). Entering the G command on the console after hitting an unconditional breakpoint will result in an immediate breakpoint "hit" without having executed any instructions because the P register still points to the breakpoint location and breakpoints are re-installed when realtime execution is requested and, exhausted breakpoints have their iteration counts reset to one. The only way to continue from a breakpoint is to use the single-step (N, X, nX or nU) or the proceed (P or nP) commands. Since the proceed commands and unexhausted breakpoints (iteration count non-zero) single-step until the P register is outside the region of the BKPT, it is safe to breakpoint on the beginning of any legal instruction (the single-stepper refuses to execute an illegal instruction) provided that some other instruction does not branch into the region occupied by the BKPT instruction.

Example:

```
        BRA     L1
        .
        .
        .
L0      BEQ     L3
L1      LDAA    #5
```

Breaking on L0 is hazardous during realtime execution if the "BRA L1" is executed, and IDB is using JSR for BKPT instructions. The reason for this is that the breakpoint JSR is planted at L0 and it will take up the first byte of L1, so that during realtime execution, L1 does not contain a "LDAA #5" instruction!! This will not be a problem during single-stepping because the BKPT instructions are not "planted".

Example:

```
        BSR     XYZ
```

Breakpointing the BSR is fatal when the RTS in subroutine XYZ is executed because the third byte of a breakpoint JSR covers the first byte of the instruction following the BSR. When the called subroutine returns, the instruction will most likely be invalid, and at the very least will cause unpredictable results. For this reason, IDB will not let you set a breakpoint on a BSR or a JSR indexed. If you wish a breakpoint there anyway, change the opcode to a NOP, set the breakpoint, and change the opcode back. This will not be a problem during single-stepping because the BKPT instructions are not "planted".

The set breakpoint command (l!) is used to set an unconditional breakpoint on a particular location with an iteration count of one. No more than four breakpoints (conditional or unconditional) may be set at a time.

The set conditional breakpoint command (l,c!) is used to set a conditional breakpoint on a particular location with an iteration count of one. l specifies the break address and c specifies the address of the conditional breakpoint subroutine. The conditional subroutine must be coded using 6800 machine instructions. When a conditional break is encountered during realtime execution, IDB will JSR to the user-defined conditional subroutine. At this point a context block exists on the user's stack representing the state of the user's registers at the time the break location was encountered. IDB will pass to the subroutine a pointer to the context block in the X register exactly as the S register would point if seven bytes were pushed on the user's stack.

Example:

If X points to n, then the registers are found in the following locations when the conditional breakpoint subroutine is entered:

```
         ----------
    n    ! TRASH  !   <------- X
         ----------
    n+1  ! C      !
         ----------
    n+2  ! B      !
         ----------
    n+3  ! A      !
         ----------
    n+4  ! X HIGH !
         ----------
    n+5  ! X LOW  !
         ----------
    n+6  ! P HIGH !
         ----------
    n+7  ! P LOW  !
         ----------
```

The user's S register at the time of the break is equal to X+7. (The above diagram is different in the obvious way for a 6811).

The conditional subroutine may test for any condition or combination of conditions (including keeping its own iteration count) and signal to IDB the truth of the condition by returning the Z bit on in the condition code byte if condition is true and Z bit off for false. The conditional subroutine returns to IDB by executing a "RTS" instruction. If the user wishes to set a conditional breakpoint at location 100 to break when register A is equal to the contents of location 5, he might decide to install the conditional subroutine at location 5000, so he enters "100,5000!" on the console device. The conditional subroutine code could look like the following:

```
5000   9605    LDAA    5       GET LOCATION 5
5002   A103    CMPA    3,X     COMPARE TO REG A IN CONTEXT BLOCK
5004   39      RTS             Z BIT SET ON IF EQUAL, OFF IF
                               NOT EQUAL
```

The user then installs this code at location 5000 before running his program.

Let's say the user wishes to build a conditional subroutine to return "condition true" if the S register (stack pointer) was not equal to $3280 (hex constant). Since the S register at the time of the breakpoint is equal to X+7 while inside the conditional subroutine, installing the following subroutine would do the trick:

```
5000   8C3279  CPX     #$3280-7        COMPARING X TO n-7 IS EQUIVALEN
5003   07      TPA             I WANT Z ON IF NOT EQUAL
5004   8804    EORA    #4      SO I MUST INVERT THE Z BIT
5006   06      TAP             BEFORE I RETURN TO IDB
5007   39      RTS             Z IS SET ON IF NOT =, OFF IF =
```

Note that the conditional subroutine is using IDB'S stack which is not infinitely deep, so don't push too far. Also, IDB is running with interrupts disabled, so please don't turn them on.

Examples:

```
100!                      SET BREAK AT LOCATION 100
102!??                    CAN'T BREAK HERE, TOO CLOSE TO 100
!                         SHOW BREAKPOINTS
0100
4852!                     SET BREAK AT 4852
!                         SHOW BREAKPOINTS
4852 0100
5000/ 7E ;O/ 7E1276 8C3279<LF> INSTALL CONDITIONAL SUBROUTINE TO
                               TEST FOR S <> $3280
5003/ ?00 07<LF>
5004/ 9601 8804<LF>
5006/ DE03 06<LF>
5007/ 0B 39<CR>
100,5000!??               BREAKPOINT ALREADY HERE
100\                      DELETE BREAKPOINT 100
!                         SHOW BREAKPOINTS
4852
100,5000!                 SET CONDITIONAL BREAKPOINT AT 100
!
0100 4852
K                         KILL ALL BREAKPOINTS
!                         SHOW BREAKPOINTS
                          NONE LEFT
```

EXECUTION COMMANDS

The execution commands are used for single-stepping instructons, realtime execution, proceeding from breakpoints and setting the iteration counter for breakpoints.

COMMAND	OPERATION
G	Start Realtime Execution (GO)
nG	Set P Register and GO
P	Continue Realtime Execution From Breakpoint (Proceed)
nP	Proceed From Breakpoint and Set Iteration Counter
X	Single-Step One Instruction
nX	Single-Step Multiple Instructions / Until Address
nU	Single-Step Until Condition Occurs
N	Single-step past current instruction
#	Value representing number of instructions stepped
;#	Zeros number of single-stepped instructions

The G command is used to start realtime execution from the current context block (the context block consists of all the registers displayed by the "?" command). All of the registers are loaded up (including S register) and control is transferred to the user program. Instruction execution begins with the instruction pointed to by the P register, and execution continues in real time. If a breakpoint JSR is encountered, IDB will regain control and do one of two things:

1) If the breakpoint is conditional, then IDB calls the user-defined conditional subroutine for this breakpoint. If a "true condition" is returned, then the iteration counter for this breakpoint is decremented by one.
2) If the breakpoint is unconditional, then the iteration counter for this breakpoint is decremented by one.

Now IDB will give a register dump and enter command mode if the iteration counter for this breakpoint is zero. Otherwise, it will carefully single-step instructions until the P register is outside the area occupied by the breakpoint JSR instruction, then continue realtime execution.

If no breakpoint is hit then, well, I hope your program is debugged (see non-maskable interrupt). If the program runs away and restart of IDB is necessary, and breakpoints were already planted when the problem occurred, then the locations with breakpoints will have to be manually repaired; that is, the original user code at those locations must be restored by hand. If you don't do this and an old breakpoint is encountered that IDB doesn't remember (IDB initializes his breakpoint table upon startup), a breakpoint display will occur. One cannot proceed, go, or single-step past the forgotten breakpoint.

If the user types an escape character on the console device and IDB encounters any breakpoint, IDB will return to command input mode and give a register dump. The user may immediately continue by using P commands.

Also note that breakpoints change the characteristics of realtime execution. That is, each instruction that the single-stepper must execute as a result of encountering a breakpoint consumes about three milliseconds. For example, a breakpoint installed on the sequence:

```
       LDAA    #2       or       LDAA    #2
       CLRB                      LDAB    #3
```

may have to single-step through two instructions (about 6 milliseconds), because a breakpoint JSR covers a portion of the second instruction.

The nG command sets the P register in the context block to n, then does a G command. If the significance of n is one, a leading zero byte is assumed.

The P command is used to continue realtime execution from a breakpoint. Instructions are single-stepped until the P register is out of the range occupied by the breakpoint JSR instruction, then execution continues in realtime execution as if a G command was used. Note that the G command could not be used in place of a P command immediately after a breakpoint was hit. Entering a G command at this point would cause another immediate breakpoint.

The nP command sets the iteration counter for the last breakpoint hit, and then does a P command. Which breakpoint was hit is remembered by an IDB variable called the "breakpointer". The P commands will not proceed if the breakpointer is invalid. Here are some possible conditions that can invalidate the breakpointer:

1) Restart IDB.
2) Encounter a conditional breakpoint whose conditional subroutine returns "condition false".
3) Killing all breakpoints.
4) Deleting the breakpoint that was last hit.

A way to set the iteration counter for a breakpoint is to set the breakpoint, go to the location, giving an immediate breakpoint, then set the P register as desired, then use the nP command. Also, a conditional subroutine could have its own iteration count.

The X command is used to single-step one instruction at a time. A register dump on the same command line occurs followed by a carriage-return after single-stepping each instruction. The single-stepper refuses to step past an illegal instruction or an old and forgotten breakpoint (this is an unusual circumstance because IDB only forgets breakpoints when the user restarts him -- see the G command). If an unusual condition exists (including breakpoint hit while stepping -- see below) an extra carriage-return will be printed out before the register dump. The purpose of this is to attract the user's attention to an unusual condition by a conspicuous change in the display format.

The single-stepper steps through an instruction first and then checks to see if the next instruction has a breakpoint. If it does, and the breakpoint is conditional, the conditional subroutine is called to see if the condition is true. If the condition is true or it's unconditional, and the iteration count goes to zero, a carriage-return is printed out before the register dump to call the user's attention to the fact that a breakpoint hit. Any proceed command may be used if desired after a breakpoint is hit, even though the user was single-stepping. Note that while single-stepping the breakponts are not physically planted in the code, but they are still checked. This is nice if the program lives in ROM.

IDB remembers the last breakpoint encountered even while single-stepping. As long as the breakpointer (see P commands for explanation) remains valid, P commands are valid. Let's say that an unconditional breakpoint was installed at 100, and a conditional breakpoint was installed at 105. If we single-step through 100, the breakpointer remembers that 100 was the last breakpoint hit so that if P commands are used, they can set the iteration count for this breakpoint. Let's step once, P register shows 102; P commands would be valid at this point. Step again, P register shows 105, the conditional subroutine was already called, it returns a false condition (no hit on this one), the breakpointer is invalidated; P commands would be invalid at this point because it is unclear to IDB (and us) whether the iteration count for 100 or 105 should be set. So the moral is: P commands are not valid after stepping through conditional breakpoints that don't hit!!

The nX command is used to single-step n times if the significance of n is one (note that 0001 has a significance of two). Entering "0X" (execute zero instructions) does the obvious, so don't waste your time with this one. Single-stepping quits when IDB has executed n instructions or has encountered a breakpoint that hits. If n has a significance of two, single-stepping quits when the P register is equal to n or a breakpoint hits. Only one register dump is given for each nX command entered. Typing an escape character on the console device will stop the single-stepper, give a register dump, and return to IDB command input mode. An interesting way to say "execute forever" is to enter "yyyyX" where "yyyy" is some address that the program will never execute. The nX command is a very powerful tool for debugging, and it's easier to use than setting breakpoints.

The nU command is used to single step until some condition is true. The value n is treated as the address of a conditional breakpoint test subroutine. The single-stepper is invoked repeatedly, and after each invocation, the conditional test subroutine is called (assuming a breakpoint has not been encountered, or <escape> has not been hit by the programmer). If the conditional test says "false", single-stepping continues, otherwise, IDB stops single-stepping and does a register dump. This command is particularly useful when trying to find out who is storing into a memory location; one sets up a conditional routine that checks to see if the desired location has changed, and turns IDB loose with the U command. It will stop after the instruction that changed the memory location. Single stepping will stop if IDB encounters an illegal instruction. IDB will stop immediately and do a register dump.

The N command is used to single step until the PC is equal to the address of the current instruction plus its length. This is used to quickly single-step through a subroutine called by a BSR or JSR.

The single stepper increments a counter every time it is called. The value of this counter can be used as a value by using a # symbol as an argument; it can be displayed by entering "#=" as a command. The counter is zeroed (and displayed) by entering a ";#" command. This is primarily useful when attempting to build very tight real time code, and an accurate instruction count for some process is needed.

Examples: (this is worth examining carefully!)

```
100;P G         SET P COUNTER TO 100 AND GO

100G            SET P COUNTER TO 100 AND GO

100! 100G       SET BREAKPOINT AND GO, GIVING IMMEDIATE BREAKPOINT
P=0100 A=4E B=4C C=53 X=524E S=9F73 */ 7E0132 0100/ 7E
P               SINGLE-STEP LOCATION 100 AND START REALTIME EXECUTION

100! 100G
P=0100 A=4E B=4C C=53 X=524E S=9F73 */ 7E0132 0100/ 7E
100P            SET ITERATION COUNTER TO 256 AND DO P COMMAND

100;P  SET P COUNTER TO LOC 100 AND SINGLE-STEP
X P=0132 A=00 B=00 C=C0 X=0000 S=00FD */ 8E0032 0100/ 7E
X P=0135 A=00 B=00 C=C0 X=0000 S=0032 */ 2003   0100/ 7E
X P=013A A=00 B=00 C=C0 X=0000 S=0032 */ 86FF   0100/ 7E
X P=013C A=FF B=00 C=C0 X=0000 S=0032 */ 06     0100/ 7E
X P=013D A=FF B=00 C=FF X=0000 S=0032 */ 8601   0100/ 7E
X P=013F A=01 B=00 C=F1 X=0000 S=0032 */ 16     0100/ 7E

100;P           SET P COUNTER TO 100 AND STEP 37 TIMES
25X P=0118 A=D0 B=D0 C=D0 X=0148 S=002E */ 33     0000/ 3F

100;P           SET P COUNTER TO 100 AND STEP UNTIL ADDRESS 915
915X P=0915 A=00 B=FF C=F0 X=003D S=0032 */ 3E     0000/ 3F
```

```
100/ 7E ;D/ 7E030B 4A<LF>       INSERT "INCA"
0101/ ?03 08<LF>                INSERT "INX"
0102/ 0B 7C0005<LF>             INSERT "INC 5"
0105/ 737600 20<LF>             INSERT "BRA $100"
0106/ 76007E 100R<CR>
5000/ 733220 9605<LF>           INSTALL CONDITIONAL BREAKPOINT ROUTINE
5002/ 2053 A103<LF>             FROM PREVIOUS EXAMPLE
5004/ 0A 39<CR>
6F73;S
100;P 0;A 0;X 5/ ?72 0<CR>
?
P=0100 A=00 B=6E C=CD X=0000 S=6F73 */ 4A     0005/ ?00
;H<CR>
?
P=0100 A=00 B=6E C=CD X=0000 S=6F73 */ 4A     0005/ 00
X P=0101 A=FF B=6E C=C9 X=0000 S=6F73 */ 08     0005/ 00
X P=0102 A=FF B=6E C=C9 X=0001 S=6F73 */ 7C0005 0005/ 00
X P=0105 A=FF B=6E C=C1 X=0001 S=6F73 */ 20F9   0005/ 01
100! <CR>
X
P=0100 A=FF B=6E C=C1 X=0001 S=6F73 */ 4A     0005/ 01
X P=0101 A=FE B=6E C=C9 X=0001 S=6F73 */ 08     0005/ 01
X P=0102 A=FE B=6E C=C9 X=0002 S=6F73 */ 7C0005 0005/ 01
X P=0105 A=FE B=6E C=C1 X=0002 S=6F73 */ 20F9   0005/ 02
X
P=0100 A=FE B=6E C=C1 X=0002 S=6F73 */ 4A     0005/ 02
P

P=0100 A=FD B=6E C=C1 X=0003 S=67F3 */ 4A     0005/ 03
3P

P=0100 A=FA B=6E C=C1 X=0006 S=67F3 */ 4A     0005/ 06
100\ 100,5000! G

P=0100 A=80 B=6E C=CB X=0080 S=67F3 */ 4A     0005/ 80

5000/ 9605 <LF>                 INSTALL CONDITIONAL BREAKPOINT...
5002/ 810E <LF>                 TO TEST FOR (5) = HEX 'E'
5004/ 39 <CR>
5/ 80 ;H/ 80  5000U             SINGLE STEP UNTIL (5) = HEX 'E'
P=0105 A=F2 B=21 C=C0 X=010E S=6F73 */ 20F9   0005/ 0E
100/ BD020001<CR>               ENTER A SUBROUTINE CALL
200/ 4C0939<CR>                 SUBROUTINE IS INCA/DEX/RTS
100;P                           GET SET TO SINGLE STEP THRU SUBROUTINE
;# 0047                         RESET STEPPED INSTRUCTION COUNT
N                               SINGLE STEP PAST SUBROUTINE

P=0103 A=F3 B=21 C=C0 X=010D S=6F73 */ 01     0005/ 0E
#=0004                          SHOW NUMBER OF INSTRUCTIONS STEPPED
```

NON-MASKABLE INTERRUPTS

IDB traps non-maskable interrupts, gives a register dump, and goes into command input mode. This is normally used to stop an undebugged program that is not hitting any breakpoints. Using the non-maskable interrupt entry point will cause IDB to remove any BKPT instructions and restore the user's code. The P register will point to the next instruction to execute. P commands are not valid, but X, N and G commands are. See the section on the I/O interface table to see how to re-direct (in effect, override) the non-maskable entry point jump. A NMI can be used to stop a dump or a search display, but this will destroy the user program's context block (see "?" command).

THE I/O INTERFACE TABLE

This table contains jumps to the IDB entry point and non-maskable interrupt entry point, and jumps to the entry points of all the I/O routines. The I/O is channel-oriented; that is, IDB does all control I/O on one channel, loading on a second, punching on a third, and dumping on a fourth. By plugging in jumps to new I/O routines, IDB can be customized to perform in virtually any environment. All routines must return with interrupts disabled. If interrupts are enabled, switching to a stack with space for the interrupts is required, and the stack must be restored when the return is made. All registers except those specified can be trashed. All entry point jumps are relative to the first address of IDB, which is usually on a 4K boundary. Let's say that n represents the first address of IDB, then we have the following descriptions:

Sacred space (n+$0) through (n+$4) - don't touch!! This is the program runaway entry point.

DEBUG (n+$5) contains a jump to the first instruction of IDB. The restart vector should be aimed here. Sacred space, don't touch!!

DEBNMI (n+$8) contains a jump to the non-maskable entry point. If the non-maskable vector is aimed at this point, then IDB will handle the interrupt. If this is the case, then this location can be plugged with another jump to override this if desired. If the non-maskable vector is aimed somewhere else, and it is desired that IDB handle the interrupt, then someone must jump to this entry point.

Sometimes it is convenient to build a context block in software and transfer control to this point. If this is the case, interrupts must be disabled before transferring to DEBNMI.

DEBRESET (n+$B) contains a jump to an IDB internal RESET routine. This is used by power-up reset code to make sure that the debugger has been initialized (i.e., is ready to take an NMI or a runaway) without transferring control to the debugger. If control is not passed to DEBUG at power up, this subroutine *must* be called by the reset logic.

GETC (n+$E) contains a jump to the I/O routine responsible for reading a character into register A from the control device (normally a terminal). All input routines must ignore nulls and strip the parity bit off the resulting character.

ECHO (n+$11) contains a jump to the I/O routine responsible for outputting a character from register A to the control device (use a "RTS" here for MIKBUG or any half-duplex device). This routine is used for echoing input characters, obviously.

PUTC (n+$14) contains a jump to the I/O routine responsible for outputting a character from register A to the control device.

OPENL (n+$17) contains a jump to the I/O routine responsible for opening the load file (send XON for some devices, or whatever is required).

READL (n+$1A) contains a jump to the I/O routine responsible for reading a character from the load file into register A.

CLOSEL (n+$1D) contains a jump to the I/O routine responsible for closing the load file (send XOFF for some devices, or whatever is required).

CREATP (n+$20) contains a jump to the I/O routine responsible for creating an output file for the punch channel. In an operating system environment, this may mean to open a file which is reserved for punching, or whatever is appropriate.

WRITEP (n+$23) contains a jump to the I/O routine responsible for outputting a character to the punch file from register A.

CLOSEP (n+$26) contains a jump to the I/O routine responsible for closing the punch file (whatever is appropriate).

CREATD (n+$29) contains a jump to the I/O routine responsible for creating an output file for the dump channel.

WRITED (n+$2C) contains a jump to the I/O routine responsible for outputting a character to the dump file from register A.

CLOSED (n+$2F) contains a jump to the I/O routine responsible for closing the dump file (whatever is appropriate).

ESCAPE (n+$32) contains a jump to the I/O routine responsible for checking for the occurrence of an escape character on the control device. Does immediate return with Z bit set if yes, reset if no. Does not echo the character. If you are replacing MIKBUG, then this feature won't work, so place a "LDAA #1", "RTS" here.

INITZ (n+$35) contains a jump to the I/O routine responsible for all initialization functions, such as resetting ACIA's or whatever is appropriate for your configuration. INITZ is called only once for each transfer to DEBUG entry point. Note that DEBRESET also calls INITZ. On exit from INITZ, the X register must contain the default user program Stack pointer (the INITZ routine can set up the context block so the registers contain default values). IDB uses this value once at the DEBUG entry time as though an n;S was typed in as the first command. (Some systems set up an initial stack pointer in such a way that typing "G" immediately after starting up IDB causes a transfer to a disk bootstrap program). The first 7 (9 for 6811) bytes of the 128 bytes of RAM scratch storage allocated to IDB are set aside to be used as this default stack.

INTDS (n+$38) contains a (jump to a) subroutine which disables all interrupts. For most 6800 systems, these three bytes can be set to NOP,SEI,RTS.

INTRTI (n+$3B) contains a (jump to a) routine that conditionally enables interrupts and then does an RTI. The "I" bit in the condition code register on top of the stack (I=0 means "enable") does an RTI. Most 6800 systems can simply place an "RTI" here.

FETCHBYTE (n+$3E) contains a (jump to a) routine that fetches a byte to the A register from the location specified by the contents of the X register, and advances the X register by one. This is used to allow IDB to access a user ROM that normally lives where IDB is in the address space. Normally this contains the code "LDAA 0,X\INX\RTS".

STOREBYTE (n+$42) contains a (jump to a) routine that stores the content of the A register in the location specified by the X register, and advances the X register by one. This is used to allow IDB to store into RAM that normally occupies the space used by IDB during debugging. Normally contains the code "STAA 0,X\INX\RTS".

BREAKPOINTINST (n+$46) contains the instruction to use for a breakpoint. Changing the first byte to SWI causes IDB to store only a SWI; otherwise, it stores 3 byte JSR for breakpoints. Normally contains "JSR BREAKPOINTENTRY".

BREAKPOINTENTRY (n+$49) is the entry point into IDB where a breakpoint must go after pushing a context block on the stack and advancing the PC past the breakpoint instruction. If BREAKPOINTINST contains a SWI, the SWI vector must be configured to (eventually) transfer control to this location.

PRESINGLESTEP (n+$4C) contains (a jump to) code to enable the user space and then do an RTI, which sets the registers to the values of the user program. This is used by the single stepper just before it executes a user program instruction, so that the stepped instruction sees user ROM/RAM where IDB is located, rather than IDB. Normally contains "RTI/SWI/SWI".

POSTSINGLESTEP (n+$4F) contains the address (FDB) of the re-entry point into IDB after executing a single instruction. The registers will be saved by IDB.

POSTSINGLESTEPDONE (n+$51) contains the entry point for re-entry into IDB after single-stepping. A context block, storing the machine state after the stepped instruction, must be pushed onto the stack before transferring control to this point. Used only if IDB is bank-switched.

RAM-based IDB for SDOS

A version of IDB that is loadable under single-user SDOS 1.1 is available. It operates identically to standard IDB with the exception of the commands listed below. Fundamentally, IDB for SDOS uses SDOS system calls so that IDB can access any user files. Thus, it is possible to load a file, make patches, and save the final result.

To invoke IDB from SDOS, type:

```
        .IDB
```

IDB will respond,

```
        IDB V1.2
```

At this time, IDB will semi-permanently allocate about 4K bytes at the top of the user space for its own use. This space will not be available for use by programs being debugged. The top of user space pointer ($FC, $FD) will be adjusted appropriately.

If G is typed immediately after loading, IDB will exit back to the command interpreter, but is still available for debugging via a ^D and the SDOS command DEBUG.

To load a file for debugging, type:

```
        :L
```

In response to the load file request, the name of the file to be loaded is entered and terminated by a return key. An example:

```
         :Load filename (<CR> to exit IDB): D2:MYNEWPROGRAM<CR>.
```

Responding with an empty line causes IDB to release its block of allocated space, and to exit back to SDOS. IDB is then not available via ^D or DEBUG.

To send a memory dump to a file, enter:

```
        :T
```

The user must supply a file name in response to IDB's request for a dump file name.

The user will have to supply a filename for each dump requested.

Example:

```
        1T
        100,50/
        Punch/Dump file name? LPT:
```

To send a MIKBUG punch file to a file, the user must type

```
        n,mP
```

and give a filename in response to the request for a dump file.

```
        Punch/Dump filename? = MYFILE.FIXED<CR>
```

Successive punches will go to the same file until a 0,0P command is used.

Warning: Don't use a dump command before a 0,0P is issued after a sequence of punches as the dump and punch files are the same.

Note: IDB uses the highest available channel number for its file operations; this may conflict with the program being debugged.

COMMAND SUMMARY

;A ;H ;X ;O	Set Display Mode to ASCII/Hex/Index/Opcode
l/	Open Location l and Display in Current Mode
(LF)	Display Next
n(LF)	Deposit and Display Next
^	Display Previous
n^	Deposit and Display Previous
(CR)	Close This Location
n(CR)	Deposit and Close Location
"text"	Deposit Text Into Memory
l:	Open Location l
?	Display Registers, Instruction, and Last Opened Location
-n	Find Negative of n
n-m n+m	Find Sum/Difference
n=	Print Value
n;A n;B n;C	Set 8 bit register register to n
n;D n;X n;Y	Set 16 bit register to n
n;P	Set P Register to n
n;S	Set S Register to n
nM	Set Mask
M	Show Mask
nS	Set Search Target
S	Show Search Target
n,mS	Search Using Mask Between n and m
n,mZ	Zero (Copy Mask to Memory) Between n and m
n,m?	Compute checksum over range and display
l,n/	Dump Memory to Dump Device
nT	Switch Dump/Search Output Channel
;L	Load From Load Device
l,nP	Punch to Punch Device
l!	Set Unconditional Breakpoint on Location l
l,c!	Set Conditional Breakpoint c on Location l
!	Show Breakpoints
l\	Remove Breakpoint From Location l
K	Kill All Breakpoints
G	Start Realtime Execution (GO)
nG	Set P Register and GO
P	Start Realtime Execution from Breakpoint (Proceed)
nP	Proceed from Breakpoint and Set Iteration Counter
X	Single-Step One Instruction
nX	Single-Step Multiple Instructions / Until Address
nU	Single-Step Until Condition Occurs
N	Single-step until Next instruction
;#	Reset and display single-step count
nR	Compute Relative Displacement
.	Value Equal to Last Location Examined
*	Value of P-Counter Displayed in Register Dump
#	Value of number of single-stepped instructions