UMEA UNIVERSITET
Fysiska institutionen
Sverker Johansson

1982-10-01

HÅKAN JOELSON
87130

Elektronik

MIKRODATORTEKNIK

Häfte 13: TSC 6800 Debug Package

TSC 6809 exakt lika (förutom registeruppsättn.m)

INNEHALL

		sid
13.1	Debug tutorial	
	I. Introduction	13-1
	II. The simulated computer	13-1
	III. Whats in memory?	13-3
	IV. Simulating the program	13-6
	V. Breakpointing the program	13-7
	VI. Advanced breakpoints	13-9
	VII. Protect your memory	13-12
	VIII. Trapping those bugs	13-13
	IX. And there is still more	13-14
13.2	Command descriptions	
	I. Introduction	13-16
	II. General system control	13-16
	III. Memory commands	13-21
	IV. Simulation control	13-23
	V. Breakpoints	13-27
	VI. Memory protection	13-31
	VII. Execution traps	13-32
	VIII. Interupt control	13-34
13.3	Message description	13-35
13.4	Command summary	13-37

13 TSC 6800 DEBUG PACKAGE

L871206

13.1 DEBUG TUTORIAL

I. Introduction

Program debugging is usually thought of as work. It should be thought of as an art. There is no reason for a lot of crying while attempting to make a new program do what was intended. This is only true, however, if the program was designed with some forethought and planning. Computer programs are executed in a logical, step by step, fashion. This is the approach both program writing AND debugging should take. So many times a programmer will spend hundreds of hours, carefully planning the flow of a new program but spend only a few minutes thought on a debugging approach. The debugging is usually attempted in some hap hazard, keep your fingers crossed, method. Sometimes this works and sometimes it does not, but in most cases, valuable time is wasted.

By using a debugging tool and by incorporating some logical thinking, program debugging can become very straight forward and sometimes even fun! The purpose of this tutorial is to introduce the reader to the capabilities of the TSC Debug Package and offer some suggestions on how to tackle those program bugs. The following sections give a more detailed description of its capabilities.

II. The Simulated Computer

The TSC Debug Package is more than the name may imply. It is in fact a complete 6800 simulator. A computer simulator is a program which when run, behaves exactly like the computer it is simulating. Given 6800 machine language, the simulator will perform the instructions exactly like the 6800 CPU. There are two major differences, one being an advantage, the second being a disadvantage. First for the good news. The simulator has the ability to keep close account of all internal actions. For example, any 1)illegal opcodes are quickly detected and reported. Such things as 2)stack overflow and underflow are also easily checked. Each byte of memory may have an 3)assigned protection type such as write protection. 4)General conditions may also be spotted such as the occurence of a transfer of address type instruction. Overall, the simulator can keep close watch over the executing program and detect any peculiarities.

This all sounds great, but as stated before, there is a disadvantage in the simulator, namely speed. The simulated program runs somewhere between 100 and 300 times slower than a real 6800 CPU. This means that real time dependent code may not be simulated. This is not a serious drawback since less than one percent of all computer programs are real time dependent.

871207 må [5]

Debugging AVLUSNING

Svårt

Man måste ha fullst kännedom om vad progr.t ska göra (hur flödet ska gå) f a kunna avlusa O sen va så vaken\nollställd a man

TSC 6809 ligger i $5500..$7FFF
simulerar MPU 6809

L> 1% .. 0,3% av maxtiden

W progr. skr.u går
½ tiden åt a skr progr.t
½ tiden åt a avlusa.

Häfte 13 upplagd:
A. Tutorial
B. Kommando-lista

TSC 6800/6809 k

Register dump

Nesting djup
Håller reda på hur många
PCreg s läggs upp på stacken

State Counter
Räknar gmfda maskincykler

Trail

Mode Ø el 1
Ø snabbare, men spolar en del faciliteter
Kör normalt i 1.

Minnesundersökningar
Celler
Stacken

Brytpunkter

Disassembler

Assembler

LEA x -1,X funkar ej
skr ist:
LEA x $FF,X

Gmgg av kommando-listan

x = 1000 ix = FF
y = 3FFF ix = 00

Obs TSC 6809 ligger i
$5500 .. $7FFF start\fr. adr = $5500
stacken ligger nedanför.
Använd därf ej $~5000 .. $7FFF samtidigt c debuggern.

Obs alla siffror skrivs i hexform (xx)
utan $-prefix
utom i CALC

The 6800 simulator incorporated in the TSC Debug Package supports all of the 6800 instructions. All of the user registers are also provided (A, B, CC, X, PC, and SP). To examine the contents of these registers it is only necessary to type R followed by a carriage return. (This is assuming the Debug Package is ready to work indicated by the two asterisk prompt ('**'). Typing the R command will cause the debugger to display a line containing all register names followed by their contents in hex. At the end of the line is the instruction currently being pointed to by the program counter (P register) and it is displayed in disassembled form (standard Motorola mnemonics). A nonstandard register is also displayed, the N register. This register's value represents the subroutine nest depth. Each time a subroutine is called, its value will be incremented, and each time a return from subroutine is executed, its value will be decremented. The contents of any of the displayed registers may also be set by using the SET command. For example:

```
**SET,P=100,A=F3
```

will set the value of the PC to hex 100 and the value of the A register to hex F3. There are several other registers and states of the simulated machine. These can be viewed by typing MACH. The items displayed with this command are primarily the states of various traps which will be described a little later.

There are several other internal machine variables which may be easily examined. One of these is the contents of the stack. Typing STACK will display the top several bytes of the stack. If more stack contents are desired, simply type the number of items desired after the command.

```
**STACK,15
```

This will display the top 15 bytes of the stack. Note that a comma was used as a separator in the command line. It will be used in all examples in this manual but a space is also acceptable and sometimes easier to type. Another command which references the stack is the RET command. This will print the top two bytes of the stack as an address and represents the return address if currently in a subroutine.

The simulated machine always keeps track of where it has been and how much time was spent there. The machine 'states counter' is used to tally the total number of machine states or cycles used so far by the executing program. Each 6800 instruction requires a certain number of machine cycles to execute. If the CPU is running at 1 megahertz, each machine state is equivalent to 1 microsecond. The machine states counter is capable of counting up to 99,999,999 cycles, or roughly 99.99 seconds of actual program execution time. This counter is useful for determining the exact execution time of a routine.

871208 ti 6

Repet.
DEBUG
Kan måste känna progr.t i detalj
f a kunna avlusa

LOGIKANALYS

Instr: Logikanalysator
f a kunna kolla fel i maskinvaran
(Sista utvägen när d ej funkar alls)

SONY TEKTRONIX 308 ~15 000:-
bra a kolla seriell kommunik c tex RS232;
klarar ox 8bitars parallell u

Philips PM 3632 ~50 000:-
klarar 32bitars HP-IB par.

* * X, GET MAXMIN.BIN Hämta i progr.t

Sätta upp skydd
** PROT, 100,200,X

M
S tex i FLEX
W

SOUNCE redovisar skydd
Brytpunkter
CLP [typ]? kopplar bort

Brytpunkter
S START
SET P=??
GO STEP
GO

Tillbehör: Mikroprocessor probe×
Disassembler
Om prod.n ej en instr.buss k man koppla in log.analys.n
HPIB, GPIB IEEE 488

Minne (i log.analys.n) tot 1000
32 bitars bredd

Minesdjup dir korrellerat t antal kanaler.

Pulserna presenteras "helt digitalt"
Gör g a göra flankobservationer el pulslängd (hur säker signalen är)

Apparaten ligger o mäter
tills den får en TRIGG
n TRIGG-villkoret är uppfyllt el triggord kommer

POST (efter)
Man bunkrar data/f-lopp e trigg-ordet

× finns typer s man klämmer dir ö chipet.

The TRAIL command will print the address of the last transfer type instruction. A transfer of address instruction is one which causes the CPU to change its normal course of instruction execution. Normally instructions are executed in a sequential fashion, stepping through memory sequentially. A JMP instruction for example will cause the next instruction to be fetched from the address specified in the instruction, rather than from the next sequential address. In effect, we have a transfer of address. The TRAIL command will print the location of the last transfer type instruction that was executed. This is very handy in determining what caused a program to end up in memory where it did.

The simulated machine is capable of running in two different modes. These are referenced as mode 1 and mode 0. In mode 1 (the default mode), all checking and bookkeeping is performed. In mode 0, several of the features are turned off in order to improve the speed performance of the simulator. (It is recommended that mode 1 always be used since it does the most work for you and will catch more errors.)

III. Whats in Memory?

Now that the simulated CPU has been described we need to look at memory. The TSC Debug Package offers several ways of examining the contents of memory locations, as well as altering them. The simplest form is the MEM command, or M for short. Typing M followed by an address will display that byte of memory. For example:

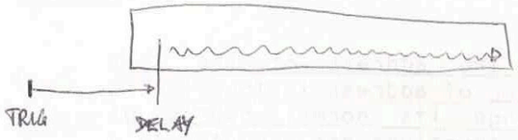
```
**M,100
100 CE
```

shows that memory location hex 100 contains a hex CE. At this time several choices are at hand. If all you wanted to do was

A. check the contents of location 100, simply type a carriage return
B. and the debug prompt will be issued. If you want to change the contents of 100, simply type the new value followed by a 'space'. The 'space' tells the debugger that the new value is ready to be entered. It is only necessary to type the significant digits of the new value to be entered. For example, if 6 was to be entered, simply type 6 followed by a space. It should be noted that only the last two digits will be used so if 'C23A' is typed, '3A' will get entered. If zero is to be entered, simply type a space. After the new value is entered, the next sequential memory
C. location will be displayed. Any time a non hex character is
a) typed (with the exception of space), one of two actions will occur. First if the character is a 'line feed', the previous
b) location will be displayed, with the current location left unchanged. If the character is any other non hex character, the next location will be displayed leaving the current unchanged.

An example will clarify the M command's use.

DELAY

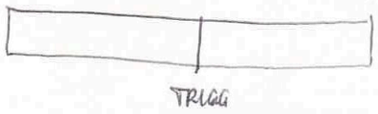


TRIGG PRE (före) - mode

Man samlar in data fe TRIG-punkten fryser



Vanligt är a man samlar data både fe + e



CLOCK intern
Extern



Anvs [bara] f a klocka in data i log.an. minnet. (upp- el ner-flank kvittar)

CLOCK QUALIFIER

Logisk nivå s styr om vi ska acceptera klock-pulsen
beset om kl-pulsen ska komma fram
(en AND-fkn mel CQ + CL)

TRIGGER

TRIGGERORD
32 bitars ord (alla bitar beh q anvs – sätt x på oanvända)

Kombinationer av 4 st triggord k användas A,B,C,D

tex A, B el A, (B el C)

Vi får göra en frivillig lab (fram + jul)

~~Förfarande [exempel]~~ på PM 3632

INKOPPLING
[på ET 3400 :]

0-7	Dataledn.
8-23	Adressledn.
24	UMA
25	R/$\overline{W}$
26-31	NC
CLOCK	ϕ2
QUALIFIER	UMA

INSTÄLLNINGAR

LOGIC 32
Vilken prob.
FORMATET på ledn.na
hex
el binärt ?

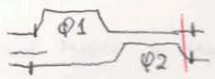
Vilka ledn.r NC

2 presentationssätt

A. Rådata i tabellform.

B. Tidsdiagram.

CLOCK
Intern
el Extern
Vilken ϕ2-flank?
(När både adr- data- + ktl-signr är giltiga



är fallar ϕ2-puls ?
QUAL. always


```
**M,100
0100 CE .
0101 3A 46
0102 4D
```

Location $100 was left unaltered, while location $101 was changed from a $3A to $46. Finally this mode was exitted on the next line by typing a return.

Many times while program debugging it is desirable to examine a large block of memory. The DUMP command is used for exactly that. This command will display 16 lines of data, 16 bytes per line, for a specified memory region. Each byte is displayed as a hex value as well as its ASCII equivalent. All control characters (those bytes having a value less than 20 hex) are displayed as an underscore character '_'. To display 256 bytes starting at memory location $1000, the following command should be typed:

=256by.

```
**DUMP,1000
```

A. At the end of the dumped block the program will stop and wait for a character to be typed. Typing an 'F' will move forward in memory, printing the next sequential 256 bytes. In this example, typing an F would display the block starting at $1100. It is
B. also possible to display the previous block of 256 bytes by typing a 'B', for backward movement. A carriage return will
C. cause the debugger to regain control and the prompt will be reissued. Any other characters will be ignored. It should be noted that any time the debugger is displaying data on the terminal, the display may be stopped at the end of the line by typing an 'escape' character. Once stopped, another 'escape' will resume the display, while a 'return' will give control back to the debugger. This is a very convenient feature.

Another useful memory interogation command is the FIND command which is used to find a specific string of bytes or characters in a selected block of memory. As an example, suppose there was a jump to subroutine instruction somewhere in your program. It is known that the code is BD 34 00, and that it is somewhere between locations $100 and $300. The following command line will find it.

```
**FIND,100,300,BD,34,00
```

This tells the debugger to look between memory locations hex 100 and 300 for the hex string 'BD3400'. All memory locations which contain this string will be displayed on the terminal. The length of the string searched is limited by the maximum command line length which is 80 characters. It is also possible to search for an ASCII string. Suppose it was necessary to find the character string 'ERROR 3' in memory. It should be somewhere between locations $200 and $1000. This can be done in the following way:

DELAY

TRIGGER ORD

A: RC00 (monitorstart)

SEQUENCE (A)

STATUS

Redovisar gjorda inställn.r

Kör igg

När triggordet kommer, fastnar data

K kollas av

A. Scrolla värdeangivelser STAT

B. Tidediagr TIMER

TESTR. 308

Mätsätt:

SIGNATUR analys

Man vet vilken ~~utan~~ respons man
ska få i en viss punkt på ~~stimulans~~
viss stimulans
i en annan p.

PARALLELL

SERIELL

Ställ FORMAT

7 el 8 bitar ?

~~Bit~~ paritet ?

baud rate

Ställ TRIGGERORD

~~För~~ va 2 st, s alltid kommer i
sa kronb följd

256 by minnesdjup x 8 ledar portar

PAR ~~STATE~~ Rådata

SER " ASCII-kod

tidediagr.

Slå av ström
te proj in koppl.r


```
**FIND,200,1000,"ERROR 3
```

The double quote character tells the find command that the following characters are to be considered ASCII characters instead of hex. Otherwise the command works exactly as described above.

So far the memory commands described have been oriented toward hex and ASCII values. Many times during debugging it is necessary to decode these hex values into assembler language instructions. The DIS command does exactly that! This command is a complete program disassembler which allows the user to examine the contents of memory in a higher level form. Each memory location in a specified block will be printed as address, followed by the opcode mnemonic and addressing mode. Standard Motorola mnemonics and addressing mode designators are used. To use the disassembler, simply type the command name (DIS), followed by two address boundaries. For example, to disassemble the memory range between locations 100 and 108, type the following.

```
**DIS,100,108
0100 LDAA $32
0102 STAA $0240
0105 BNE  $0121
0107 DECA
0108 STAA 2,X
```

Remember that at any time the display is being produced, the 'escape' key may be typed to temporarily halt the action. The DIS command is a very useful and powerful command.

Now that we can examine memory in a higher level form it would be nice if we could alter it in the same way, that is, using assembler language mnemonics. The ASM command does exactly that! It acts as a line at a time assembler, allowing standard mnemonics and addressing modes to be typed, while the corresponding hex values are automatically inserted into memory. To start this process simply type the command name followed by the address where the code should be placed. The debugger will respond by printing the address of the location specified followed by a space. At this time, simply type the desired instructions following each with a carriage return. The next available address will then be printed and assembly can continue. Typing a carriage return in response to the address prompt will exit this mode of operation. To show the workings of this command, some code will be assembled at location $200.

```
**ASM,200
0200 LDAA 10
0202 LDAB $10
0204 PSHA
0205 LDAA #M
0207 STAA 0,X
```

imm. el dir?

```
0209 JMP $3000
020C
**
```

Note that numeric values are interpreted as decimal unless preceded by a dollar sign ($) to designate hex. It is also possible to enter an ASCII constant by preceding it with a single quote ('). No spaces are allowed between the register specifiers 'A' or 'B' and the instruction (e.g. LDAA is correct, LDA A is not). The ASM command is a great time saver!

IV. Simulating the Program

Program simulation is very simple. If the test program starts at $100, simply type START,100 to start the simulation process. The program will run exactly as the CPU would run it, just slower. The START command clears several of the machine conditions such as the states counter. To start a program where it left off, the GO command can be used. This will cause the program to start execution at the location pointed to by the program counter (P register). No states will be cleared.

A very valuable feature of the simulator is the 'trace mode'. When trace is enabled, a register dump (exactly like that produced by the R command) will be displayed after each instruction is executed. The simulation may be temporarily halted by typing an 'escape' character anytime during the tracing operation. The simulation may also be stopped by typing a 'control C'. This will cause the debug prompt to be reissued. To enable the trace mode use the TRACE command.

```
**TRACE=10
```

This line will cause the debugger to trace all instructions which are in a subroutine nest level of 9 or lower. The number in the command line specifies the nest level where tracing should be disabled. This allows only the outermost program structure to be traced if desired, while the deeper subroutines will be simulated without the tracing. To disable the trace, use a count of zero (e.g. TRACE=0).

There are several other methods of starting program simulation. One is the SIM command. This command will allow the simulation of a specified number of instructions. Tracing is disabled during the execution of this command.

```
**SIM,100
```

This line will cause 100 instructions to be simulated starting at the address pointed to by the program counter. The TSIM command is identical to the SIM command except trace is automatically set to 256 during the execution of the command.

It is often desirable to step through the execution of a program, one instruction at a time. The STEP command will start simulation at the instruction pointed to by the program counter, execute a specified number of instructions, print a register dump, and then wait for input. At this time, a space will repeat the process, while a return will return control back to the debugger. The usual method of operation is 'single' step which will execute one instruction, then dump the registers. This mode can be entered by:

```
**STEP
```

Multiple instructions can be executed between register dumps by specifying a count. For example;

```
**STEP,25
```

will cause 25 instructions to be simulated at a time. The step mode is a very powerful method for closely following the flow of a program.

During program execution, the simulator keeps track of the last 256 instructions executed. If a program ever goes off on its own, ending up in memory where it should not, the PAST command can be used to examine the instructions executed to get it there. Typing the command,

```
**PAST,20
```

will display the addresses and mnemonic instructions of the last 20 opcodes executed. This feature alone will find a good percentage of program bugs.

V. Breakpointing the Program

So far, methods have been described which allow all or a certain number of instructions to be simulated. Most of the time, the number of instructions to a certain point in the program is not known. It would be helpful if a break in the program simulation could be specified to take place at a particular point in the program, or in other words, breakpoints. A breakpoint is a mechanism for stopping the execution at a specified address in the program. As an example, to set a breakpoint at location $23A, use the following command.

```
**B@23A
```

As the program executes, any time location $23A is reached, simulation will stop and the registers will be dumped to the terminal. After the program has stopped, typing a 'G' will restart execution, starting at address $23A (the breakpoint will be temporarily ignored). It should be noted that the method used to create the breakpoint does not alter the contents of memory in

any way. This means that after setting a breakpoint, the contents of memory at the breakpoint location will be unchanged. This allows breakpoints to be set in ROM as well as RAM!

In the above example, the breakpoint caused two actions to take place. One was printing the registers, the other was stopping program simulation. These actions are the ones performed by most debugging systems. The TSC Debug Package allows six other actions to be performed upon the execution of a breakpoint. A list of all 8 possible actions follow:

1. R...Print register contents
2. Z...Zero the states counter
3. T...Enable the trace function
4. U...Disable trace (untrace)
5. H...Histogram counter
6. M...Print a message
7. J...Jump to specified address
8. S...Stop simulation

The first breakpoint example shown defaulted to R and S type actions since none were specified. The Z action zeroes the machine states counter. This is useful for program timing. For an example, the states counter may be zeroed upon entry to a subroutine and a stop type breakpoint set at the exit point of the routine. By using the STATES command after the program stops, the exact number of executed machine states for that routine will be displayed.

The T and U actions allow the trace mode to be enabled and disabled at selected points in a program. When enabled, trace will be set to level 255. Many times, tracing is only desired during one routine or selected portion of the program. These actions will permit this sort of program tracing. A few examples will demonstrate action type breakpoints.

```
**B,RZ@1000
**B,T@A16
```

The first command will set a breakpoint at location hex 1000 which when executed will print the registers and zero the states counter, The program will then continue since a stop (S) action was not specified. The second example will cause trace to be turned on at location hex A16.

Another action is the histogram (H). A histogram counter counts the number of times the instruction at that address has been executed. This is useful for determining 'hot spots' or sections of programs which are executed very frequently. By setting a histogram breakpoint at the first instruction of each subroutine in a program, it is possible to find out exactly how many times each routine was called. As an example, suppose there were three subroutines in a program, and they were located at $100, $123, and $1A0. To set histogram counters at these

locations, type the following commands:

```
**B,H@100
**B,H@123
**B,H@1A0
```

After simulating the program, typing HIST will display the totals of the counters at each address. This command is used to examine the histogram counters at any time. The CLH command is used to clear the histogram counters.

```
**CLH,100
**CLH
```

The first command will clear (set to zero) the value of the histogram counter at location 100. The second command will zero all of the counters. The histogram commands allow a very complete profiling of a program, letting the user 'fine tune' it for maximum speed.

The remaining two action codes are special purpose. One permits a selected message to be printed as the action, the second allows transfer of control to a specified address (like a JMP instruction).

```
**B,M@325,SUB 1
**B,J@27C,1000
```

The first line will print the message "SUB 1" each time the instruction at $325 is executed. The second command will cause the instruction at address hex 1000 to be the next instruction executed. The instruction at 27C will not be executed!

Any combination of action codes may be listed for a breakpoint. They are executed in the order they appear in the above list. For example,

```
**B,TRZ@300
```

will cause the registers to be displayed (R), the states counter to be zeroed (Z), and trace to be enabled (T), in that order. This ordering may be important, for in the actions 'RSJ', the stop (S) will never get executed since the J transfers control to another address.

VI. Advanced Breakpoints

Programs containing loops or recursion are often difficult to breakpoint since one particular section of code may be called thousands, or even millions of times. As an example, suppose there is a loop in the program being debugged, and it is necessary to examine the contents of the X register after the 600th time through the loop. One way is to set a breakpoint at

the desired instruction and start the program simulating. Every time the program halts at the breakpoint, type G to restart it. Repeat this process 600 times and you can examine X. You are probably thinking that this would take forever and you are right! The TSC Debug Package allows a pass counter to be associated with a breakpoint. This count determines how many times the instruction at the address of the breakpoint should be executed before the actions specified should be performed. In the above example, assuming the instruction to be breakpointed is at address 300, the following will do exactly what we want.

```
**B@300,>600
   or
**B,SR@300,>600
```

Both commands are identical since the first defaults to SR actions. The '>' is the pass count modifier and should be read as 'after'. The result of this command is to stop and print the registers on the instruction at location 300, after 600 times through it. Once the count reaches 600 (or whatever value was set), the breakpoint actions will always occur. A second similar type of pass count uses a '<' for a modifier and should be read as 'before'. This is used to create a temporary breakpoint.

```
**B,R@300,<100
```

This command will set up a breakpoint at 300 which will print the registers for the first 100 times through. After the 100th time, the breakpoint will be cleared and no longer function. In summary, the pass count value associated with a breakpoint is decremented each time the instruction at the specified address is executed. If the modifier is a '>', no actions will be performed until 'after' the count has reached zero. With the '<' mofifier, actions are only performed 'before' the count reaches zero, and once it is zero, the breakpoint is cleared.

In the above example it was decided that the program should be stopped after 600 times through the loop. While debugging loops, it is not always possible to determine an exact number of times to execute the loop before it should be stopped. Often it is desirable to stop on a certain condition, such as the contents of a register or the state of a particular memory location. Conditional expressions are allowed in breakpoint definitions and yield a great deal of power. The conditional can be determined on the contents of a selected register (A, B, C, X, P, S, or N) being equal (or not equal) to a specified value. A particular memory location may also be tested for zero or not zero. Following are a few examples.

```
**B@1000,IF A=3F
**B,R@320,IF B!=10
**B,T@6A7,IF $20=0
```

The 'IF' statement designates the conditional part of the

breakpoint definition. The first example will stop and print the registers at location hex 1000 but only when the value in the A accumulator is hex 3F. The second example will print the registers at 320 only if the contents of the B register is not hex 10 ('!=' is to be read as 'not equals'). The last example will enable the trace mode at location 6A7 if the contents of memory location hex 20 is zero. The dollar sign '$' is used to designate a memory reference and not a hex value (the value is always interpreted as hex). The value on the right of the equals sign must always be zero when a memory reference has been designated.

The above breakpoint features may be combined in a variety of ways to produce an almost endless variety of breakpoints. As an example:

```
**B,TZ@1000,>100,IF X=100
```

will cause trace to be enabled and the states counter to be zeroed, after executing the instruction at hex 1000, 100 times, but then only if the value of the index register is $100. It should be noted that the H, M, and J action codes will not allow a conditional expression as part of the breakpoint definition, and J will not support a pass counter.

Once breakpoints are set it is possible to examine the loaction of them as well as remove them. To check the locations of breakpoints, use the BP command.

```
**BP
**BP,100
**BP,100-500
```

The first line will print the location of all breakpoints, each one followed by a list of its action codes. No pass counts or conditionals are displayed. The second example will display the action codes of the breakpoint at location hex 100 (if one exists). The last command line will display all breakpoints between location 100 and 500, inclusive. The CLB command is similar in syntax but is used to clear or remove a breakpoint. CLB by itself will clear all breakpoints. If it is followed by an address, the breakpoint at that address will be removed. If two addresses are specified, then all breakpoints in their range will be cleared.

While debugging very large programs, it may become quite time consuming to simulate the program up to a desired address. For example, a program which requires a minute to execute in real time may require over an hour if simulated. To get around this problem, it is possible to set a 'real time' breakpoint. This is entirely different from the previously described breakpoints in that it does modify the contents of memory (by substituting a JMP instruction) and no pass counting or conditionals are permitted. The only action performed is to stop and print the registers. An

example of use follows:

```
**RT,5A00
```

This command will cause the CPU to start executing the program (NOT the simulator) at the current address of the program counter. When the program reaches the specified address (5A00), the program will stop, print the registers, and restore the contents of RAM at that location (remove the breakpoint). Since the program is being executed in real time and not being simulated, no other breakpoints, illegal condition checking, states counting, or record keeping is performed. This type of execution is not recommended for this reason and should only be used where the simulation time gets tremendously long.

VII. Protect Your Memory

Perhaps the most aggravating aspect of program debugging is having your program destroy itself in memory. Too many times, programs 'run away', writing garbage in memory, usually exactly where it is not wanted. In these instances, it would be nice to be able to 'write protect' memory, or at least certain portions of it. The TSC Debug Package will allow exactly that! In fact, any section of memory, right down to a single byte, may be write, execute, memory, or simulate protected! Write protecting memory will prohibit any stores or writes into it. Execute protection prohibits opcodes from being fetched from memory. In other words, the program counter (PC) will not be permitted to point to a location of memory which is execute protected. Memory protect is a brute force type of protection. By memory protecting a region, you are in effect saying that no memory exists in this region and that nothing should be allowed to reference it in any way. Any memory referenced in conflict with its protection will cause the simulation to stop and an appropriate message will be printed. Finally, simulate protection is slightly different from the rest. It is used to tell the simulator to execute any code in a simulate protected region in real time, or in other words, not simulated. A restriction requires the code in a simulate protected region to be called as a subroutine (JSR or BSR) from the non-simulate protected code. This is very convenient for I/O operations. All I/O routines can be simulate protected (such as TTY and disk routines) allowing them to be executed by the CPU (real time) and not the simulator. It is often convenient to simulate protect the entire region of memory containing the monitor and/or operating system since this code is known functional. Keep in mind that code in simulate protected memory may only be accessed via a subroutine call.

The command used to set protection is PROT. A few examples will demonstrate its use.

```
**PROT,100-3FF,X
**PROT,2E0,W
**PROT,500-6FF,M,1200-1FFF,W
```

The first example will execute (X) protect the memory between locations $100 and $3FF. The second line write protects (W) location $2E0. The last example will memory protect (M) locations $500 through $6FF and write protect $1200 through $1FFF. There are some general rules to follow when protecting memory. Memory protection should be used on all sections of memory not referenced or used by the program being debugged, especially the area of memory containing the Debug Package. This will keep a runaway program from clobbering something it should not. Sections of memory which are used for register storage or flags should be execute protected. Memory containing the actual program code should be write protected for obvious reasons. Finally, as mentioned above, the memory locations where the monitor and/or operating system reside should be simulate protected.

Once the protection has been defined it may be checked by using the BOUNDS command. This command will allow the examination of the boundaries of each type of protection. Either all types or selected ones may be displayed.

```
**BOUNDS
**BOUNDS,W,M
```

The first example will display all types while the second will show only the defined boundaries for write and memory protection. Memory protection can be cleared in a similar fashion.

```
**CLP
**CLP,X,W
```

The first command will clear all protection while the second will only clear the defined execute and write protected regions.

VIII. Trapping Those Bugs

The previously described breakpointing feature allows programs to be stopped at specific locations and on specific conditions. It is often desirable to 'trap' a program on some general condition such as every time a transfer of address instruction is encountered. The memory protection described above is a form of trap in that the program will stop if a protection violation is detected (e.g. writing into write protected memory). There is address information associated with this protection which makes it different from the general traps available in the Debug Package. The general traps cause programs to stop on a general condition which is not address dependent.

One of these traps is the illegal opcode trap which is always enabled. Any time an illegal opcode is encountered during the course of program simulation, the program will stop and report its occurrence. A second, always enabled trap will stop the program if an RTS instruction is encountered and the current nest level is 0.

There are several user controlled traps which may be enabled and disabled at will. The transfer trap is enabled with the XFR command. When enabled, the program will stop each time a transfer of address is encountered. These instructions are JMP, BRA, and all conditional branches such as BCC. The subroutine calls and returns are not trapped out.

```
**XFR=ON
**XFR=OFF
```

These two commands will enable and disable this trap respectively. Once a program has stopped because of a transfer trap, typing G will restart it, allowing the current transfer to be executed. This is very useful for quickly following the major flow of a program. Another one of the general traps allows halting the program if the subroutine nest counts reaches a specified level.

```
**NEST=20
```

This will cause a trap if the nest level ever reaches 20. To disable the nest trap, use NEST=0.

The last general trap to be discussed is the ITRAP. This command allows activation of the interrupt trap and will cause the simulating program to stop if an interrupt type instruction is encountered (SWI, RTI, and WAI). Since these instructions are not used in the majority of programs it is a good idea to use this feature. An example will demonstrate its use.

```
**ITRAP=ON
**ITRAP=OFF
```

These two commands will enable and disable the interrupt trap respectively.

IX. And There is Still More!

There are still many undescribed features of the TSC Debug Package. One of these is the handy little CALC command which acts as a hex calculator. Typing CALC followed by a return will cause the debugger to output an equals sign (=) for a prompt. At this time hex and decimal addition and subtraction may be performed. To add two numbers simply type them in separated by a plus sign. If the number is hex precede it with a dollar sign, otherwise the debugger will interpret it as decimal. Use a minus sign for

subtraction. It is also possible to do base conversions. This can be accomplished by entering just one number after the prompt (hex or decimal) followed by a return. All answers are displayed in both hex and decimal. An example follows.

```
**CALC
 =$1A+10
$0024  36
 =256
$0100  256
 =
```

After entering the calculator mode, the numbers hex 1A and decimal 10 were added to give the result hex 24 or decimal 36. The second entry is a base conversion of the decimal number 256. The result shows its hex equivalent is $100. The calculator mode can be left by typing a return in response to the prompt.

There are still many other features in the Debug Package, such as interrupt simulation, which have not been described. It is not the intention of this tutorial to teach all there is to know about the debugger, but to teach enough to make the user feel comfortable with the majority of its features. Once the material in this section is thoroughly understood, the following detailed command description should be studied in depth.

Now that the basic mechanics of the Debug Package are understood they should be put to good use. Keep in mind that a logical and planned approach should be taken when debugging a program. Use the available tools such as memory protection and breakpoints. When first starting the debug process on a new program, start at the beginning, working your way through the flow of the program. Let the program be the guide. If you pay close attention, it will definitely point out the bugs. Above all, have patience. Great bugs are not killed overnight!

13.2 COMMAND DESCRIPTIONS

I. Introduction

This section of the manual contains a detailed description of each Debug command. Each command is shown with a few examples. The syntax definitions show optional items in square brackets ([]). All command parameters are shown separated by commas for clarity in the syntax definitions and examples. Any place a comma is shown, a space may also be used. The following definitions apply throughout this document:

Item	Meaning
<address>	1-4 digit hex value
<value>	decimal number (max = 255)
<count>	decimal number (max = 65,000)

The Debug Package is ready to accept a command anytime the '**' prompt is present on the line. When typing commands, a 'control H' will cause a backspace, and delete the last character typed. A 'control X' will cause the entire line to be deleted and a new prompt of '??' will be output to show the deletion of the line. Any time text is being output to the terminal, display may be stopped at the end of a line by typing an 'escape' character. Once stopped, another 'escape' will restart the output while a 'return' will give control back to the debugger and the '**' prompt will be output.

II. General System Control

The general system control commands allow a variety of general actions to be performed. Register examination and changing is supported by use of the REG and SET commands. The status of several machine control registers can be obtained through the MACH command. Commands to view the stack contents, set simulation speed, reset machine parameters, enter a calculator mode, examine the 'machine states counter', and exit the debugger are all described in this section.

CALC (C or CALC) CALC

PURPOSE:

The calculator mode will be entered and a '=' prompt will be printed. The calculator will allow addition or subtraction of two numbers. The numbers may be hex (designated by a '$' prefix) or decimal. If two numbers are typed, they must be separated by a '-' or '+' and the appropriate result will be displayed. The answer is shown in both hex and decimal. It is possible to enter

only one number (hex or decimal) followed by a return. The answer will be this number printed in both hex and decimal, thus allowing base conversions. After each calculation, a new '=' prompt will be output. To exit this mode, type a 'return' as a response to the prompt.

EXAMPLES:

```
CALC            Enter calculator mode
 =$A+10         Add hex A and 10
$14  20         The result is printed
```

DELAY=<value> (DEL or DELAY) DELAY

PURPOSE:
This will set the simulation delay (the amount of delay after each instruction is executed) to an amount proportional to <value>. The higher the number (max = 255) the longer the delay. A delay of zero will result in the delay being turned off.

EXAMPLES:

```
DELAY=100       Set delay to 100
DELAY=0         Disable the delay
```

DEPTH (DEP or DEPTH) DEPTH

PURPOSE:
The depth command will print the deepest value of the stack pointer (the lowest memory address at which the stack was extended during program simulation). To initialize this pointer, it is necessary to set the stack pointer using the SET command. The depth value will be set to the same value as the stack pointer. This command is useful for determining the amount of stack space required by a program.

EXAMPLES:

```
DEPTH           Print the deepest stack location
```

EXIT (E or EXIT) EXIT

PURPOSE:
Exit the debug program. Use this command when finished with the Debug Package.

EXAMPLES:

```
EXIT            Exit the debugger
```

FLAG[=<adress>] (FL or FLAG) FLAG

PURPOSE:
The Flag register is a 2 byte word at the specified memory location which will be displayed on a REG command or during tracing, as the 'F' register. The memory location for the flag will be set to the address specified. If no address is given, the flag register will be disabled. This is useful for tracking flags in memory during program tracing. See the REG command.

EXAMPLES:

FLAG=1A85	Set flag register to $1A85
FLAG	Disable flag register printout

IND=ON or OFF IND

PURPOSE:
Used to enable or disable the indirection printout in a register dump (see REG). If IND is ON, the register dump will show a register called 'I' which is the value of the memory location pointed to by the index (X) register. If this feature is off, the I register will not be displayed.

EXAMPLES:

IND=ON	Turn indirection on
IND=OFF	Turn it off

MACH (MA or MACH) MACH

PURPOSE:
The MACH command will print the current status of the simulated machine. Values displayed are for mode (M), trace (T), instruction count trap (I), nest trap (N), stop address (S), interrupt trap (IT), transfer trap (XT), IRQ count (IRQ), and NMI count (NMI). The description of these appear elsewhere in this manual.

EXAMPLES:

MACH	Print the machine status

MODE=1 or 0 (MO or MODE) MODE

PURPOSE:
The debugger has two modes of operation, mode 0 and mode 1. The system comes up in mode 1. Mode 1 offers all debug features allowing the simulated program to run approximately 250 times slower than real time. In mode 0, the program will run approximately 100 times slower than real time, but the following features are not supported; nest count checking, all traps, states counting, memory protection, past instruction bookkeeping, and automatic interrupts. Mode 1 should be used most of the time to take full advantage of the debugger.

EXAMPLES:

MODE=1	Set mode to 1
MO=0	Set mode to 0

REG (R or REG) REG

PURPOSE:
Print the contents of the machine registers. All values are shown in hex. Besides the condition codes (C), A, B, and X registers, program counter (P) and stack pointer (S), the nest level, N, is displayed (shows how deep in subroutine calls) as well as two optional registers. One is enabled by the IND command and displays the byte of memory being pointed to by the index register. This is shown as 'I' in the REG dump. The second option is enabled by the FLAG command and will display the selected two bytes of memory. This is shown as 'F' in the dump.

EXAMPLES:

REG	Display all registers
R	Display all registers also

RESET (RES or RESET) RESET

PURPOSE:
The RESET command is used to reset all machine states. All registers will be set to zero, the stack pointer will be set to $A07F, all breakpoints and memory protection will be cleared, and the mode will be set to 1. This will set up the machine exactly the same as initializing the debugger upon first entry.

EXAMPLES:

RESET	Reset the machine

RET RET

PURPOSE:
Print the top two items on the stack. If the system is currently in a subroutine, these bytes will represent the return address from this routine. If the nest level is currently zero (N=0), the message "NEST LEVEL IS 0" will be displayed.

EXAMPLES:

RET	Print the return address

SET,<register list> (S or SET) SET

PURPOSE:
The SET command is used to set or assign values to registers. The <register list> is a list of register names (C,A,B,X,S,P,N) followed by an equals sign, followed by the hex value. Setting the stack pointer will also set the depth value to the same amount.

EXAMPLES:

SET,P=100,A=C3	Set PC to $100 and A to $C3
S B=20 X=1FFF	Set B to $20 and X to $1FFF

STACK[,<value>] STACK

PURPOSE:
Print the contents of the stack. The number of bytes specified by <value> will be printed. If <value> is not specified, the top 6* bytes will be printed. The stack is printed from high address to low address, so the top of stack will be the last item printed.

*6809: 12

EXAMPLES:

STACK	Print the top 5 stack bytes
STACK,10	Print the top 10 stack bytes

STATES (STAT or STATES) Kolla kvalitén på progr.t STATES

PURPOSE:
Display the current value of the states counter. This value represents the number of actual machine cycles (micro seconds on a 1 megahertz computer) which have been executed since the last START or RESET command. It is also possible to set this counter to zero using breakpoints.

EXAMPLES:

STATES	Print the current states count

TRAIL TRAIL

PURPOSE:
Print the address of the last executed instruction which caused a transfer of address (e.g. JMP instruction). This is useful when attempting to find how a program ended up where it did.

EXAMPLES:

TRAIL	Print the last transfer address

X,<operating system command>

X

PURPOSE:
The X command is only operational on disk systems (see Adaptions). It allows the execution of any DOS command from within the debugger.

EXAMPLES:
X,CAT,1 Catalog drive 1

L871207

III. Memory Commands

The memory commands allow examining and altering the contents of memory in a variety of ways. The assembler allows simple, direct insertion of object code by using standard opcode mnemonics and addressing mode designators. The disassembler provides an opposite type of convenience, in that the contents of memory may be displayed as assembler language mnemonics and operands. A single byte memory examine and change function is also available (the MEM command). Commands for viewing large blocks of memory, finding specific hex or ASCII strings, and filling a section of memory with a selected character are all available in this group.

ASM[,<adress>] (A or ASM)

ASM

PURPOSE:
Enter the line at a time assembly mode. Assembly will start at the address specified or at the location of the program counter if no address is specified. No labels are permitted. All standard Motorola opcode mnemonics are accepted (no pseudo ops). When instructions contain a register specifier, there should be no space between the mnemonic and the specifier (e.g. LDAB, not LDA B). All standard addressing modes are accepted. All page zero references will be assembled as extended addresses. Three types of constants are permitted, decimal, hex (precede the number with '$'), and ASCII (precede the ASCII letter with a single quote ('')). The PC is automatically advanced to the next location after the line is assembled. To exit this mode, type a return in response to the address prompt.

EXAMPLES:

```
ASM,100          Start assembly at $100
 100 LDAA #10    Load A with 10
 102 LDAB #'1    Load B with ASCII 1
 104 BRA $100    Loop forever
 106             Exit with return
```

måste kugga rätt (peka på en verklig opkod)

DIS,<start address>,<stop address> (D or DIS) DIS

PURPOSE:
Disassemble memory between the addresses specified. The address, mnemonic, and addressing mode will be printed out for each instruction in the range. If an illegal opcode is found, three stars (***) will be displayed instead of a mnemonic, followed by the hex value found at that address.

EXAMPLES:
DIS,100,1A0 Disassemble from 100 to 1A0

DUMP,<address> (DU or DUMP) DUMP

PURPOSE:
Dump 256 byte blocks of memory starting at the address specified. The memory is displayed 16 bytes per line, followed by the ASCII values of the hex numbers. After each block is dumped, typing an 'F' will move Forward and display the next 256 bytes, typing a 'B' will move Back and display the previous 256 bytes. Typing a 'return' will exit this mode.

EXAMPLES:
DUMP,A00 Dump memory at $A00

FILL ,<start address>,<stop address>[,<byte>] (FIL or FILL) FILL

PURPOSE:
This command will fill memory with the <byte> (hex) specified starting at the first address, filling through the second address. If <byte> is not specified, zero will be used.

EXAMPLES:
FILL,100,300,FF Fill with FF from 100 to 300
FILL,0,100 Clear from 0 to 100

FIND,<start address>,<stop address>,<string> (FIN or FIND) FIND

PURPOSE;
Find the specified string in memory. The search will start at the <start address> and continue through the <stop address>. The address of each location where the string is found will be displayed. The <string> can be entered in one of two ways. The first can be a string of hex digits separated by spaces or commas. The second is an ASCII string preceded by a double quote character. The limit on string length is the input buffer (72 characters).

EXAMPLES:
FIND,0,60,7E,33,A2 Find the hex value 7E33A2
FIND,0,1000,"TEST Find TEST in memory

MEM,<address> (M or MEM)

MEM

PURPOSE:
Examine and alter memory. The address specifies the first location to be examined. Upon entering this command, the address specified and its contents will be displayed on a new line. At this time, typing any non hex printing character will cause a move to the next location and display its contents. Typing a 'line feed' will move to the previous location. A carriage return will exit this mode. To change the contents of a location, type the new hex value immediately following the one displayed. After the value, type a space. The new value will be entered and the next memory location will be displayed. It should be noted that it is only necessary to type the number of significant digits and only the last two digits are used. For example, typing a 1 would enter 01, typing 1A2 would enter A2, etc. If only a space is typed (no number) a zero will be entered. Any time a non-hex character is typed (besides a space), the next location will be displayed, leaving the current location unchanged.

EXAMPLES:

MEM,540	Examine memory at $540
M,200	Examine location $200

L871207

IV. Simulation Control

This group of commands is used to control the program simulator. Code in RAM or ROM may be simulated. There are several methods of initiating simulation. Programs may be executed with 'trace' on or off. While trace is on, each instruction will be displayed prior to its execution, along with the current state of the CPU (all register contents are displayed). Trace provides a very powerful tool for following program flow. Several keyboard commands may be invoked during actual program simulation. These commands allow the speeding up or slowing down of simulation, as well as ways to halt the execution of the program. The PAST command is a powerful bookkeeper which keeps track of where your program has been.

GO (G or GO) GO

PURPOSE:
Start the program executing at the location currently pointed to by the program counter. No machine values are altered with this command.

EXAMPLES:

GO	Start the simulation at the PC
G	Does the same thing

JUMP,<address> (J or JUMP) JUMP

PURPOSE:
This command is exactly like GO except execution will begin at the address specified. No machine values are altered with this command, except the program counter which is set to <address>.

EXAMPLES:

JUMP,322	Start simulation at $322
J,80	Start simulation at $80

PAST[,<value>] (PA or PAST) PAST

PURPOSE:
Display the past several instructions executed by the simulated program. If <value> is not specified, the past 255 instructions will be printed (oldest to most recent), otherwise <value> sets the number of instructions to be displayed. Each instruction is shown in a disassembled form, with its address.

EXAMPLES:

PAST	Display the past 255 instructions
PAST,10	Display the past 10 instructions

SIM[,<count>] SIM

PURPOSE:
Simulate the number of instructions specified by <count> with the trace disabled. If the count is not specified, one instruction will be executed. Execution starts at the current PC. No machine values are altered prior to simulation. Trace will be reset to its original value following SIM's termination.

EXAMPLES:

SIM	Simulate one instruction
SIM,100	Simulate 100 instructions

TSC default sätter Stop = $0000

START,<address> (ST or START)

START

PURPOSE:
Start program simulation at the specified address. The PC will be set to the address specified, the states counter will be zeroed, and the nest count will be cleared.

EXAMPLES:

START,1000	Start simulation at $1000
ST,2A	Start simulation at $002A

STEP[,<count>]

STEP

PURPOSE:
This command will cause the debugger to enter the 'step' mode. The <count> specifies how many instructions should be executed at a time in this mode and defaults to one (single step). Upon entering the STEP command, the system will immediately execute the number of instructions specified by <count>, then print a register dump. The execution will begin at the location pointed to by the P register (program counter). After the register dump, typing a 'space' will cause execution of the next <count> instructions and produce another register dump. Typing a 'return' will exit the step mode. Any other character will be ignored. It should be noted that while in the step mode, breakpoints and tracing are inoperable.

EXAMPLES:

STEP	Enter 'single step' mode
STEP,10	Execute 10 instructions at a time

TRACE=<value> (T or TRACE) kopplad + REG

TRACE

PURPOSE:
Set the trace depth. If value is set to zero, trace mode will be disabled. Setting trace to a non-zero value will enable tracing up to but not including the subroutine nest level indicated by <value>. For example, if TRACE=2 is entered, tracing will occur at nest level 0 and 1 but will be disabled at nest levels of 2 and higher. The nest level is displayed as 'N' in a REGister dump.

EXAMPLES:

TRACE=255	Enable trace at all levels
T=0	Disable trace mode

Dumpen tystnar i o.c.d. nesting djup man specificerat.

TSM[,<count>] (TS or TSM) TSM

PURPOSE:
This command is similar to SIM except trace mode is enabled (Trace=255) and the registers will be dumped after each instruction simulated. The count will default to 1 if not specified. Trace will be reset to its original value following TSIM's termination.

EXAMPLES:

TSIM	Trace and simulate 1 instruction
TSIM,20	Trace 20 instructions

'Control C' 'Control C'

PURPOSE:
Anytime a program is being simulated, a 'control C' will cause the execution to halt and the message 'OP HALT AT XXXX' to be displayed at the terminal. This means 'Operator Halt' and the XXXX will be replaced by the actual address where the program was halted.

'Escape Character' 'Esc'

PURPOSE:
During program tracing, typing an 'escape' will cause the program to pause at the end of the next displayed line. At this time, typing another 'escape' will enable the trace to restart, while typing a 'return' will return control back to the command entry mode.

'Control F' 'Control F'

PURPOSE:
During program simulation, the delay value (see DELAY) may be dynamically changed. Each time a 'control F' is typed (only during program simulation) the delay value will be decremented by one, thus making the program run faster. If the delay is zero, the 'control F' will be ignored. It should be noted that for large delays, many 'control F' functions will need to be typed to see the increase in speed.

'Control S' 'Control S'

PURPOSE:
This is similar to the 'control F' key but makes the simulation run slower. If the delay is already at its maximum value, the 'control S' will be ignored.

LSI1207

V. Breakpoints

Breakpoints allow the insertion of check points into a program. A breakpoint always has an address associated with it. The address specifies where in the program the breakpoint action should occur. These actions range from printing the machine registers to controlling trace mode. Each breakpoint may also have a pass counter which determines the amount of time until it becomes active, or the amount of time it should remain active. The actions are also dependent on the result of a conditional expression involving a CPU register or memory location. Breakpoints are decoded with th following precedence. ①If the address of the current PC matches the address of a breakpoint, then the pass count is checked. ②If the counter is in a state to allow continuing, then the condition is checked (if present). Finally the ③actions specified for the breakpoint are performed. The other commands in this group allow clearing breakpoints (removing them), printing histogram counter values, print breakpoint location and type, and clear histogram counters.

B,<actions>@<address>[,<modifier><count>][,IF<condition>]
   or
B@<address>[,<modifier><count>][,IF<condition>]

B

PURPOSE:

The B command is used to set breakpoints. These breakpoints are nondestructive in that they do not alter the contents of memory at the breakpoint location. Two forms of the command exist. The first is the general form of the command and allows user definable breakpoint actions. The <actions> may be any one or combination of the following:

R...Print register contents
Z...Zero the states counter
T...Trace mode on
U...Trace mode off (untrace)
H...Histogram counter
M...Print message
J...Jump to new address
S...Stop simulation

The above actions are executed in the order shown. A histogram action causes a counter to be set up such that each time the instruction at the address specified is executed, the counter will be incremented by one. By later requesting a HISTogram, all of the counters and their associated counts will be displayed. The second form of the B command is a special case of the first. In this form, no actions are specified, and they default to S and R (just as if S and R were used in form one). The <count> part of the syntax is optional and acts as a pass counter. The <modifier> shown in the command description represents either a '>', used to mean 'after', and '<' to

represent 'before'. A count preceded by '>' will cause the breakpoint defined on the line to remain inactive until <count> number of times through that address. A count preceded by '<' will cause the breakpoint defined to be active for only the <count> number of times through that address, at which time it will be automatically removed. The <count> in either case must not exceed 32,000. The next part of the syntax is the optional <conditional>. This allows the breakpoint action to be dependent on some condition. The condition can be the contents of any machine register being equal or not equal to a hex value ('=' and '!=' respectively), or the contents of a specified memory location being zero or not zero. If a register is used, simply state the register name, followed by the relational, followed by the hex value (e.g. A=23, or B!=E2). To use a memory location, a dollar sign '$' must precede the address. For example, $100=0 would check if the byte at location hex 100 was zero, and $A20!=0 would check if the byte at location hex A20 was not zero. If a memory address is specified, the only allowed value to the right of the relational is zero, and if any other value is used, it will be ignored. NOTE: The conditional part of the breakpoint definition may not be used with H,M, or J action codes. Two of the breakpoint actions require special syntax. These are the M (message) and J (jump) types. The M action is used to print a specified message to the terminal upon execution of the breakpoint. The J action is used to transfer control to another address (like a JMP instruction). Any breakpoint containing M may not contain J and vice versa. A breakpoint containing M should have an ASCII string following the <count> (or following the address if no count is specified). This string is the message which will be printed on the terminal each time the instruction is to be executed. Messages should be kept short (under 5 letters if possible). For the J type action, the hex address of the location of transfer should be provided after the <count> field. The examples below will help clarify the syntax.

EXAMPLES:

B@100	Stop and print registers at $100
B,SR@100	Same as above
B,H@A100	Set histogram at $A100
B,ZR@300 >100	Zero states and print registers after 100 times through $300
B@200,IF A=3C	Stop & print registers at $200 only if acc. A = $3C
B,M@210,SUB 1	Print message 'SUB 1' every time through location $210
B,J@100,1000	Transfer control to location $1000 when reach instruction at $100
B,TZ@400,<25,IF $20=0	For the first 25 times through location $400, turn trace on and zero the states counter, but only if location $20 is zero.

BP[,<address>[-<address>]]

BP

PURPOSE:
The BP command is used to print the location of breakpoints and their associated action codes. The two address specifications are used to define the region of memory for checking breakpoints (beginning and ending, respectively). If no addresses are specified, all breakpoints will be listed. If only one address is given, then only the breakpoint at that address will be displayed (if one exists). Only the action codes are listed with each address.

EXAMPLES:

BP,10-C00	List breakpoints between $10 & $C00
BP	List all breakpoints

CLB[,<address>[-<address>]]

CLB

PURPOSE:
Clear breakpoints in specified memory region. The addresses define the region of memory. If only one address is listed then only the breakpoint at that location will be cleared. If no addresses are specified, all breakpoints will be cleared.

EXAMPLES:

CLB	Clear all breakpoints
CLB,0-100	Clear breakpoints between $0 & $100
CLB,22A	Clear breakpoint at $22A

CLH[,<address>[-<address>]]

CLH

PURPOSE:
Clear histogram counters in the specified memory region. The addresses define the region of memory. If only one address is listed then only the histogram counter at that location will be cleared. If no addresses are specified, all counters will be declared. NOTE: This command does not remove the histogram breakpoints, but clears its associated counter to zero in preparation for a new run.

EXAMPLES:

CLH	Clear all histogram counters
CLH,25-200	Clear counters between $25 & $200

CLM CLM

PURPOSE:
Clear all messages in the breakpoint message table (used by the M action code, see the B command). This table is a fixed size and can be filled up. When deleting message type breakpoints using the CLB command, the associated space in the message table does not get freed. It is recommended that whenever all M type breakpoints have been cleared, also use the CLM command. Do not use this command if there are any active M type breakpoints. Their message strings will be destroyed!

EXAMPLES:
CLM — Clear all messages

HIST[,<address>[-<address>]] (H or HIST) HIST

PURPOSE:
Print the histogram counter totals for the section of memory specified. The addresses define the region of memory. If only one address is listed then only the counter at that location is displayed. If no addresses are specified, all counter contents will be displayed. Each counter is shown preceded by its address. The counter value shows the number of times the instruction at that address has been executed.

EXAMPLES:
HIST — Display all histogram counters
H,0-200 — Display counters between 0 & $200

RT[,<address>] tex när progr.t ska RT

PURPOSE:
Start real time program execution (not simulated) at the current PC location. Program execution will halt at the <address> specified. This is similar to the standard breakpoint most users are familiar with in that memory is actually altered at the address specified (with a JMP instruction). Entering RT without an address will clear any real time breakpoint which may have been previously entered. This type of breakpoint and program execution is not recommended since no protection or checking is performed. When the program reaches the break address specified, the breakpoint is automatically cleared and the original code restored in memory. ROM may not be breakpointed with this command.

EXAMPLES:
RT,600 — Start at PC, end at $600
RT — Clear an existing RT breakpoint

Stäppa DEBUG + låta CPU:n köra fram t angiven adr. (e måste va 1a byten i en instrukt. e måste ligga i RAM)
(Innebär a DMP1 kör e får ett software-interpt vi angiven adr.n.

L871207

VI. Memory Protection

The memory protection commands are a very powerful feature of the program debugger. The PROT command allows selected areas of memory to be write, execute, memory, or simulate protected. ③Write protected memory will cause a trap on any attempt to write ② to it. ②Execute protect will not allow opcodes to be fetched. ① Memory protect will not permit any type of reference; read, ④ write, or execute. ④Simulate protect is used to protect sections of code which should not be simulated (executed in real time). It is important that only code called as a subroutine from non-simulate protected memory be contained in the area(s) of memory designated as simulate protected. An example would be to simulate protect the section of memory where a DOS resides. All subroutine calls to the DOS would then be executed in real time. Code which is simulate protected and does not follow this convention will usually cause the CPU to take over the execution of the program resulting in a loss of control. NOTE: To protect the memory around the machine stack (upper and lower bounds), use the 'memory' protection. This is the only type checked on stack references. Other commands in this group allow examination of protected memory regions or bounds, as well as the clearing of protection types.

s 13-32

BOUNDS[,<types>] (BO or BOUNDS)

BOUNDS

PURPOSE:
Display the bounds of protected memory. Each <type> specified will list all regions of memory protected by that type. <type> may be W, M, X, or S for write, memory, execute, and simulate, respectively. Multiple types may be displayed by listing the types on the command line separated by a comma or space. If no type is specified, all types of protection will be listed.

EXAMPLES:

BOUNDS	Display all memory protection
BO,M,X	Display memory and execute protection bounds

CLP[,<type>]

CLP

PURPOSE:
Clear all protected regions for a specified type of protection. The <type> is specified by the same letters described in BOUNDS. Only one type may be listed per command line. If type is not specified, all protection will be cleared.

EXAMPLES:

CLP	Clear all protection
CLP,X	Clear execute protection

① M ≈ där finns inget

② X k tex läggas ö data-area
där inget finns a exekvera

③ W läggas lämpl ö programmets minnes-area

④ S u tex u FLEX
→ processen går ö i realtid

tex. ① ,300,BFFF,M

② ,100,1FF,X

③ ,200,226,W

L871207

PROT ,<address>[-<address>],<type> (P or PROT) PROT

se ox s13-31

PURPOSE:
The PROT command is used to assign protection to a region of memory. The two <address> specifiers designate the beginning and ending addresses of the selected region. If only one address is specified, only the byte at that location will be protected. The <type> designator may either be M①, X②, W③, or S④ for memory, execute, write, and simulate protection respectively. Only one type may appear with each address range. Multiple protection may be performed on one line by separating the range-type specifiers by a comma or a space.

EXAMPLES:

```
PROT,0-100,M          Memory prot 0-$100
P,100,W,A100-A600,S
                      Write prot $100 and simulate
                      protect $A100-A600
```

VII. Execution Traps

Execution traps allow program stopping on certain general conditions. Several traps are always enabled. These include; trap on illegal opcode and trap on RTS if nest count=0. The user may enable and disable several other traps. These traps are for interrupt type instructions, transfer of address type instructions, trap on a selected subroutine depth (nest count), an instruction count timeout, and a general 'stop' address.

INST=<count> INST

PURPOSE:
Set the instruction count timer to the value of count. If set to zero, this trap will be disabled. This timer is used to count the number of simulated instructions. Each time this counter reaches zero, the program will halt and print 'IC TIMEOUT AT XXXX', where XXXX is the address where the program stopped, and the counter will be reset to the value it started at (the value specified by <count>).

EXAMPLES:

```
INST=400              Set counter to 400
INST=0                Disable the intruction counter
```

ITRAP=ON or OFF (IT or ITRAP) ITRAP

PURPOSE:
Turning the ITRAP on will cause the simulator to treat interrupt type instructions similar to illegal opcodes. Any time a RTI, SWI, or WAI instruction is found, the message 'I TRAP AT XXXX' will be displayed. The address of the instruction will be printed in place of the XXXX shown.

EXAMPLES:

ITRAP=ON	Enable the interrupt trap
IT=OFF	Turn off the trap

NEST=<value> (N or NEST) NEST

PURPOSE:
Set the nest trap at the level specified by <value>. The simulator will trap execution if a subroutine call instruction is found which will cause the nest level to equal or exceed that set by NEST. Setting the <value> to zero will disable this trap.

EXAMPLES:

NEST=6	Set nest trap to level 6
N=0	Disable nest trap

STOP=<address> Default: = $0000 STOP

PURPOSE:
The STOP trap is a general 'stop at address X' trap. It is useful for trapping returns to monitor type programs or operating systems. The trap is set at the address specified. Avsedd a sättas på sista instruktionen i progr.t.

EXAMPLES:

STOP=100	Set stop trap at $100
STOP=E0D0	Set trap at MIKBUG entry

XFR=ON or OFF XFR

PURPOSE:
Enabling the XFR trap will cause a trap each time a transfer of address type instruction is found (JMP, BRA, or BXX). This is useful for following major program flow. Typing a 'G' command after this trap will cause the program to start executing again.

EXAMPLES:

XFR=ON	Enable the transfer trap
XFR=OFF	Turn the trap off

L881207

VIII. Interrupt Control

Both NMI and IRQ type interrupts may be simulated. Two modes of operation are possible. The first is automatic, periodic interrupt generation. This mode allows interrupts to be generated every N instructions. The second allows random interrupt generation from the keyboard. When these keys are typed during program simulation, the appropriate interrupt will be issued.

IRQ=<count> IRQ

PURPOSE:
Cause an IRQ type interrupt to be generated every <count> instructions. If count is set to zero, IRQ interrupts will be shut off.

EXAMPLES:

IRQ=5000	Generate IRQ every 5000 instructions
IRQ=0	Turn off automatic IRQs

NMI=<count> NMI

PURPOSE:
Cause an NMI type interrupt to be generated every <count> instructions. If <count> is zero, automatic NMI interrupts will be turned off.

EXAMPLES:

NMI=300	Generate NMI every 300 instructions
NMI=0	Turn off automatic NMIs

'Control I' 'Control I'

PURPOSE:
Typing a 'control I' during program simulation will cause an IRQ type interrupt to be generated.

'Control N' 'Control N'

PURPOSE:
Typing a 'control N' during program simulation will cause an NMI type interrupt to be generated.

13.3 MESSAGE DESCRIPTION

The following is a list of all Debug generated messages and their respective meanings.

WHAT? = This is the general error message reported when an invalid input command has been entered.

"STOP" AT = The address set by the STOP trap command has been reached.

IC TIMEOUT AT = The number of instructions specified by the INST trap command have been executed.

ILLEGAL OPCODE AT = The instruction pointed to by the PC is an illegal opcode.

I TRAP AT = An SWI, RTI, or WAI instruction has been encountered and the ITRAP command has been used to enable the interrupt trap.

LAST XFR FROM = Displayed by request using the TRAIL command. The address gives the location of the last transfer of address type instruction which was executed.

SYNTAX ERROR = The command just entered does not follow the syntax rules for that command.

EP TRAP AT = An Execution Protect trap at the specified location resulting from an attempt to execute code in execute protected memory.

WP TRAP AT = A Write Protect trap at the specified location resulting from an attempt to write into write protected memory.

EX - MP TRAP AT = An attempt to execute code residing in memory protected memory has been detected at the specified address.

REF - MP TRAP AT = An attempt to reference (read or write) a byte in memory protected memory has been detected at the specified address.

SP TRAP AT = A Stack Pointer reference (PSH, JSR, etc.) was attempted in a section of memory which is memory protected.

TABLE OVERFLOW = The last command entered caused an internal table to overflow. The command dis not get executed.

NC TRAP AT = A Nest Count trap occurred as a result of the nest level reaching the level specified in a NEST command.

RTS IN LEVEL 0 AT = An RTS instruction was encountered while the nest level was 0 (no previous call to subroutine had been executed).

NEST LEVEL IS 0 = There is no return address on the stack so the RET command can not display an address.

XFR TRAP AT = A transfer of address type instruction has been encounterd with the transfer trap enabled (from XFR=ON).

MON XFR AT = The program being simulated tried to pass control to the monitor address which is used by the EXIT command.

OP HALT AT = An operator halt signal (control C character) was detected by the simulator.

command	cmd	page
ASN[,W=<drive>][,S=<drive>]	ASN	10-9
CAT[,<drive list>][,<match list>]	CAT	10-11
COPY,<filespec>,<file spec>	COPY	10-13
COPY,<file spec>,<drive>		
COPY,<drive>,<drive>[,<match list>]		
DATE[,<mm,dd,yy>]	DATE	10-15
DELETE,<file spec>[,<file list>]	DELETE	10-16
GET,<file spec>[,<file list>]	GET	10-7
default extension : .BIN		
LIST,<file spec>[,<line range>][,N]	LIST	10-17
MON	MON	10-7
O,<file spec>,<command>	O	10-18
default extension : .OUT		
P,<command>	P	10-19
PROT,<file spec>[,<options>]	PROT	10-20
RENAME,<file spec 1>,<file spec 2>	RENAME	10-21
default extension : .TXT		
SAVE,<file spec>,<begin adr>,<end adr>[,<transfer adr>]		
default extension : .BIN	SAVE	10-22
SAVE.LOW	SAVE.LOW	10-22
STARTUP	STARTUP	10-23
TTYSET[,<parameter list>]	TTYSET	10-25
VERSION,<file spec>	VERSION	10-29
default extension : .CMD		

GETCHR	Get character	$AD15	10-33
PUTCHR	Put character	$AD18	10-33
PSTRNG	Print string	$AD1E	10-33
INBUFF	Input to line buffer	$AD1B	10-34
PCRLF	Print CR + LF	$AD24	10-34
OUTADR	Output hexadecimal address	$AD45	10-34
OUTDEC	Output decimal number	$AD39	10-35
OUTHEX	Output hexadecimal number	$AD3C	10-35
GETHEX	Get hexadecimal number	$AD42	10-35
INDEC	Input decimal number	$AD48	10-35
Line buffer		$A080 - $A0FF	10-32
System date registers		$AC0E - $AC10	10-32
Line buffer pointer		$AC14 - $AC15	10-32
Escape return register		$AC16 - $AC17	10-32

OmegaSoft Screen Editor

SCREEN EDITOR COMMANDS

Line syntax : "." = current, "$" = last, digits = offset or relative line number, + and - allowed.

Buffered Commands ('E' prompt)

Edit <filename> _____ Clear buffer and read file.
New <filename> _____ Clear buffer and set filename.
Get <filename> _____ Get file past cursor.
Save [<filename>] _____ Save entire file.
Save and Quit _____ Save entire file and quit.
Save Partial <line>,<line> <filename> _____ Save part of file.
Move <line>[,<line>] _____ will move lines to after cursor position.
Copy <line>[,<line>] _____ will copy lines to after cursor position.
DElete <line>[,<line>] _____ will delete lines.
Position <line> _____ positions cursor to line.
Tabs Repetitive <incr> _____ Sets tabs at columns 0, incr, 2 * incr, 3 * incr, etc.
Tabs Non-repetitive (<column>) _ Sets tab at entered columns. (Columns separated by spaces).
Tabs _____ Clear all tabs.
T+ _____ Enable display of tab markers.
T- _____ Disable display of tab markers.
? _____ Display version, filename, and line and memory status. Enter any key to exit.

When loading a new file or exiting the editor you will be prompted to save the current buffer if it has been modified since last saved. Any response other than an 'N' or 'n' will save the old buffer. A response of 'C' or 'c' will allow you to change the file name before saving.

Non-buffered commands (see page 19-9)

		original version
	Quit _____ Exit editor.	
	4 _____ move cursor left one character.	J
	6 _____ move cursor right one character.	L
H-19	8 _____ move cursor up one line.	I
	2 _____ move cursor down one line.	,
	5 _____ alternate between left and right of line.	K
	0 _____ move cursor 1/2 screen up	O (letter)
	. _____ move cursor 1/1 screen down	
	space _____ move cursor to next identifier.	
	HT or cntrl T _ move cursor to next tab.	
	cntrl R _____ move cursor backwards to previous tab.	
	cntrl H _____ single character delete. deleting the carriage return will merge lines, with multiple leading spaces compressed into one space.	
	cntrl X _____ delete current line.	original version
	9 _____ Overwrite one character.	1 (digit one)
	5 _____ overwrite characters until escape.	V
H-19	1 _____ enter append mode.	A
	7 _____ enter insert mode.	;

Text search commands

Find _____ will position cursor, response is
CR = stop command
Space = continue
N = multiple occurrences on line

Replace _____ will position cursor, response is
CR = stop command
Space = continue
R = Replace and continue
N = multiple occurrences on line
E = entire buffer

Note : When the 'F' prompt appears enter the find pattern - if you enter just a carriage return then this will terminate the command. When the 'R' prompt appears enter the Replace pattern. A '^' is used as a don't care in the find and Replace patterns. The entire buffer option will enable the replacement throughout the rest of the buffer. The multiple occurrences option will allow a match on each occurence of the find pattern on any one line, else only the first match on a line is recognized. The search will start on the current line, continue to the end of the buffer, start back at the top, and stop on the current line. The cursor will be positioned at the last match when the command terminates.

Insert or append mode

Character entry until an escape is entered. Insert mode will insert characters before the current cursor position. If a carriage return is entered during insert mode then append mode will be entered with the last part of the line carried over until append mode is exited. Append mode will start new lines past the current line with no partial line carried over. In addition the following special characters are recognized :

BS or DEL _____ backspaces over character.
HT or cntrl T __ forward tab.
cntrl R _____ reverse tab.
cntrl A _____ copies remainder of last line entered.
cntrl X _____ removes all characters entered so far.

DIRECTIVES

END	end of source program	12-12
EQU	equate a symbol to operand	12-12
FCB	form constant byte	12-13
FDB	form double constant byte	12-14
FCC	form constant characters	12-13
LIB	process library file	12-14
NAM	specify name or title	12-14
ORG	assign origin of program counter	12-15
PAG	skip to next page	12-16
RMB	reserve memory bytes	12-16
SPC	insert spaces of output listing	12-16
TTL	specify name or title	12-14
OPT	assembler control options	12-15
	PAG enable page formatting	

COMMAND

[P,]ASMB,<filename1>[,<filename2>[,+<optionlist>]

optionlist

B	no binary file
D	no date on list
G	print only one line for FCB, FDB and FCC
L	no list
S	no symbol table
Y	delete binary file

13.4 COMMAND SUMMARY

I. General System Control

		Description page
CALC	CALC	13-16
DELAY=<value>	DELAY	13-17
DEPTH	DEPTH	13-17
EXIT	EXIT	13-17
FLAG[=<address>]	FLAG	13-18
IND=ON or OFF	IND	13-18
MACH	MACH	13-18
MODE=0 or 1	MODE	13-18
REG	REG	13-19
RESET	RESET	13-19
RET	RET	13-19
SET,<register list>	SET	13-20
STACK[,<value>]	STACK	13-20
STATES	STATES	13-20
TRAIL	TRAIL	13-20
X,<operating system command>	X	13-21

II. Memory Commands

ASM[,<address>]	ASM	13-21
DIS,<start address>,<stop address>	DIS	13-22
DUMP,<address>	DUMP	13-22
FILL,<start address>,<stop address>[,<byte>]	FILL	13-22
FIND,<start address>,<stop address>,<string>	FIND	13-22
MEM,<address>	MEM	13-23

III. Simulation control

GO	GO	13-24
JUMP,<address>	JUMP	13-24
PAST[,<value>]	PAST	13-24
SIM[,<count>]	SIM	13-24
START,<address>	START	13-25
STEP[,<count>]	STEP	13-25
TRACE=<value>	TRACE	13-25
TSM[,<value>]	TSM	13-26
'Control C'	Cont. C	13-26
'Escape Character'	Esc.	13-26
'Control F'	Cont. F	13-26
'Control S'	Cont. S	13-26

* need ann. here and Wolf Brandstetter

DEBUG COMMAND SUMMARY

General system control

CALC	*Calc*	*13-16*
DELAY=<value>	*DELay*	*13-17*
DEPTH	*DEPth*	*13-17*
EXIT	*Exit*	*13-17*
FLAG[=<adress>]	*FLag*	*13-18*
IND=ON or OFF	*IND*	*13-18*
MACH	*MAch*	*13-18*
MODE=1 or 0	*MOde*	*13-18*
REG	*Reg*	*13-19*
RESET	*RESet*	*13-19*
RET	*RET*	*13-19*
SET,<register list>	*Set*	*13-20*
STACK[,<value>]	*STACK*	*13-20*
STATES	*STATes*	*13-20*
TRAIL	*TRAIL*	*13-20*
X,<FLEX command>	*X*	*13-21*

Memory commands

ASM[,<adress>]	*Asm*	*13-21*
DIS,<start address>,<stop adress>	*Dis*	*13-22*
DUMP,<address>	*DUmp*	*13-22*
FILL,<start address>,<stop adress>[,<byte>]	*FILl*	*13-22*
FIND,<start adress>,<stop adress>,<string>	*FINd*	*13-22*
MEM,<adress>	*Mem*	*13-23*

Simulation control

GO	*Go*	*13-24*
JUMP,<address>	*Jump*	*13-24*
PAST[,<value>]	*PAst*	*13-24*
SIM[,<count>]	*SIM*	*13-24*
START,<address>	*STart*	*13-25*
STEP[<count>]	*STEP*	*13-25*
TRACE=<value>	*Trace*	*13-25*
TSM[,<count>]	*TSm*	*13-26*
'ctrl'C	*operator halt*	*13-26*
escape	*halt on EOL*	*13-26*
'ctrl'F	*faster*	*13-26*
'ctrl'S	*slower*	*13-26*

Breakpoints

B,<actions>â<address>[,<modifier><count>][,IF<condition>]

B *13-27*

Bâ<address>[,<modifier><count>][,IF<condition>]

R...Print register contents
Z...Zero the states counter
T...Trace mode on
U...Trace mode off (untrace)
H...Histogram counter
M...Print message
J...Jump to new address
S...Stop simulation

BP[,<address>[-<address>]]	*BP*	*13-29*
CLB[,<address>[-<address>]]	*CLB*	*13-29*
CLH[,<address>[-<address>]]	*CLH*	*13-29*
CLM	*CLM*	*13-29*
HIST[,<address>[->address>]]	*Hist*	*13-30*
RT[,<address>]	*RT*	*13-30*

Memory protection

BOUNDS[,<types>]	*BOunds*	*13-31*
CLP[,<type>]	*CLP*	*13-31*
PROT,<address>[-<address>],<type>	*Prot*	*13-32*

Execution traps

INST=<count>	*INST*	*13-32*
ITRAP=ON or OFF	*ITtrap*	*13-33*
NEST=<value>	*Nest*	*13-33*
STOP=<address>	*STOP*	*13-33*
XFR=ON or OFF	*XFR*	*13-33*

Interrupt control

IRQ=<count>	*IRQ*	*13-34*
NMI=<count>	*NMI*	*13-34*
'Ctrl'I	*'Ctrl'I*	*13-34*
'Ctrl'N	*'Ctrl'N*	*13-34*

IV. Breakpoints

	B,<action>@<address>[,<modifier><count>][,IF<condition>]	B	13-27
xx	B@<address>[,<modifier><count>][,IF<condition>]	B	13-27
	BP[,<address>[-<address>]]	BP	13-29
	CLB[,<address>[-<address>]]	CLB	13-29
	CLH[,<address>[-<address>]]	CLH	13-29
	CLM	CLM	13-30
	HIST[,<address>[-<address>]]	HIST	13-30
	RT[,<address>]	RT	13-30

V. Memory Protection

	BOUNDS[,<types>]	BOUNDS	13-31
	CLP[,<type>]	CLP	13-31
	PROT,<address>[-<address>],<type>	PROT	13-32

VI. Execution Traps

	INST=<count>	INST	13-32
	ITRAP=ON or OFF	ITRAP	13-33
	NEST=<value>	NEST	13-33
\	STOP=<address>	STOP	13-33
	XFR=ON or OFF	XFR	13-33

VII. Interrupt Control

	IRQ=<count>	IRQ	13-34
	NMI=<count>	NMI	13-34
	'Control I'	Cont. I	13-34
	'Control N'	Cont. N	13-34

HÅKAN JOËLSON

DEBUG COMMAND SUMMARY

General system control

CALC	Calc	13-16
DELAY=<value>	DELay	13-17
DEPTH	DEPth	13-17
EXIT	Exit	13-17
FLAG[=<adress>]	FLag	13-18
IND=ON or OFF	IND	13-18
MACH	MAch	13-18
MODE=1 or 0	MOde	13-18
REG	Reg	13-19
RESET	RESet	13-19
RET	RET	13-19
SET,<register list>	Set	13-20
STACK[,<value>]	STACK	13-20
STATES	STATes	13-20
TRAIL	TRAIL	13-20
X,<FLEX command>	X	13-21

Memory commands

ASM[,<adress>]	Asm	13-21
DIS,<start address>,<stop adress>	Dis	13-22
DUMP,<address>	DUmp	13-22
FILL, start address>,<stop adress>[,<byte>]	FILl	13-22
FIND,<start adress>,<stop adress>,<string>	FINd	13-22
MEM,<adress>	Mem	13-23

Simulation control

GO	Go	13-24
JUMP,<address>	Jump	13-24
PAST[,<value>]	PAst	13-24
SIM[,<count>]	SIM	13-24
START,<address>	STart	13-25
STEP[<count>]	STEP	13-25
TRACE=<value>	Trace	13-25
TSM[,<count>]	TSm	13-26
'ctrl'C	operator halt	13-26
escape	halt on EOL	13-26
'ctrl'F	faster	13-26
'ctrl'S	slower	13-26

Breakpoints

B,<actions>â<address>[,<modifier><count>][,IF<condition>]

B	B	13-27

Bâ<address>[,<modifier><count>][,IF<condition>]

R...Print register contents
Z...Zero the states counter
T...Trace mode on
U...Trace mode off (untrace)
H...Histogram counter
M...Print message
J...Jump to new address
S...Stop simulation

BP[,<address>[-<address>]]	BP	13-29
CLB[,<address>[-<address>]]	CLB	13-29
CLH[,<address>[-<address>]]	CLH	13-29
CLM	CLM	13-29
HIST[,<address>[->address>]]	Hist	13-30
RT[,<address>]	RT	13-30

Memory protection

BOUNDS[,<types>]	BOunds	13-31
CLP[,<type>]	CLP	13-31
PROT,<address>[-<address>],<type>	Prot	13-32

Execution traps

INST=<count>	INST	13-32
ITRAP=ON or OFF	ITtrap	13-33
NEST=<value>	Nest	13-33
STOP=<address>	STOP	13-33
XFR=ON or OFF	XFR	13-33

Interrupt control

IRQ=<count>	IRQ	13-34
NMI=<count>	NMI	13-34
'Ctrl'I	'Ctrl'I	13-34
'Ctrl'N	'Ctrl'N	13-34

S-hacken startar [DEBUG] på $C07F