

Elektronik

MIKRODATORTEKNIK

Häfte 13: TSC 6800 Debug Package

TSC 6809 exakt lika (förutom registersättning)

INNEHÅLL

13.1	Debug tutorial	sid
	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. Interrupt control	13-34
13.3	Message description	13-35
13.4	Command summary	13-37

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 illegal opcodes are quickly detected and reported. Such things as ²stack overflow and underflow are also easily checked. Each byte of memory may have an ³assigned protection type such as write protection. ⁴General conditions may also be spotted such as the occurrence 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

Minnesundersökningar

celler

stacken

Debugging

AULUSNING

Svart

Brytpunkter

Man måste ha fullt kännedom
om vad progr. + ska göra (hur
flödet ska gå) f a kunna avläsa
O sen va så vaken/nollställd a man

Disassembler

Assembler

TSC 6809

ligger i \$5500..\$7FFF

simulerar MPW

6809

LEA x -1, x

funker g

skr ist:

LEA x \$FF, x

↳ 1% .. 0,3% av maxtiden

W progr. skr u gör

1/2 tiden åt a skr progr. +

1/2 tiden åt a avläsa.

Gångs a kommando-listan

Häfte 13 upplagd:

A. Tutorial

B. Kommando-lista

TSC 6800/6809 k

Register dump

Nesting djup

Håller reda på hur många

Preg s läggs upp på stacken

State Counter

Räknar gmflda maskin cykler

Trail

Mode 0 el 1

0 snabbare, men spolar en
del faciliteter

Kör normalt i 1.

Obs TSC 6809 ligger i

\$5500..\$7FFF start/1fr. adr = \$5500

ligger i stacken ligger nedanför.

Anv därt g \$~5000..\$7FFF samtidigt
c debuggern.

Obs alla siffror skrivs i hexform (xx)

utan \$-prefix

utom i CALC

X = 1000

IX = FF

Y = 3FFF

IX = 00

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

LOGIKANALYS

Repet.

DEBLIG

Man måste känna program i detalj
+ a kunna avläsa

* * X, GET MAX MIN. BIN

Hämta i
program

Sätta upp skydd

* * PROT, 100, 200, X

lust: Logikanalysator

+ a kunna kolla fel i maskinvaran
(sista utvägen när d g funkas alls)

SONY TEXTRONIX 308

~15 000:-

bra a kolla seriell kommunik c kr RS232,
klavar ox 8 bitars parallell u

Philips PM 8632

~50 000:-

klavar 32 bitars 80486 par.

M

S

kr ö FLEX

W

BOUND

redovisar skydd

Brytpunkter

CLP [typ]?

kopplar bort

Tillbehör: Mikroprocessor prober *

Disassembler

Om man har en instr. buss k
man koppla in log. analys. u

HP IB, GPIB IREE 488

Brytpunkter

S START

SET P = ??

GO STEP

GO

Minne (i log. analys. u) tot 1000

32 bitars bredd

Minnesdjup dir korrelerat + antal
kanaler.

Pulsarna presenteras "helt digitalt"

Går g a göra flaukobservationer el
pulshöjd (hur säker signalen är)

Apparaten lägger 2 mäter

ills den får en TRIGG

" TRIGG-villkoret är uppfyllt att trigger
kommer

POST (effekt)

Man brukar data/flopp e triggerordet

* 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.)

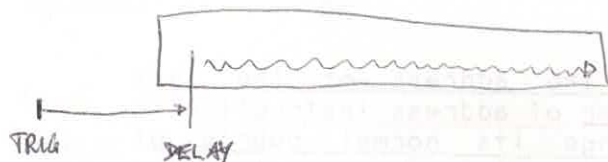
III. Whats in Memory?

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
```

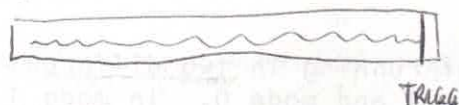
- shows that memory location hex 100 contains a hex CE. At this time several choices are at hand. If all you wanted to do was
- A. check the contents of location 100, simply type a carriage return
 - B. and the debug prompt will be issued. If you want to change the contents of 100, simply type the new 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 chracter is typed (with the exception of space), one of two actions will
 - a) occur. First if the character is a 'line feed', the previous location will be displayed, with the current location left
 - b) 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

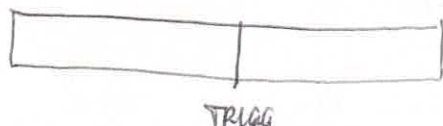


TRIG PRE - mode
(före)

Man samlar in data fr TRIG-punkten
friger



Viktigt är att man samlar
data både fr & e



CLOCK

Inten
Extern



Ans [bara] f a klocka i data
i log.an. minnet. (upp- el ned-flank
kvittar)

CLOCK QUALIFIER

Logisk nivå s styr om vi ska
acceptera klock-pulsen
best om kl-pulsen ska komma fram
(en AND-fkt med CQ & CL)

TRIGGER

TRIGGERORD

32 bitars ord (alla bitar beh g ans
- sätt x på användare)

Kombinationer av 4 st triggerord
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 BT 3400:]

0-7 Dataledn.r

8-23 Adressledn.r

24 UMA

25 R/W

26-31 NC

CLOCK $\phi 2$

QUALIFIER UMA

INSTALLNINGAR

LOGIC 32

Vilken prob.

FORMATET på ledn.ra

hex
el binärt?

Vilka ledn.r NC

2 presentatione sätt

A. Rådata i tabellform.

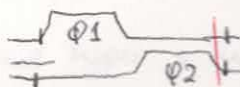
B. Tiedediagram.

CLOCK

Inten
el Extern

Vilken $\phi 2$ -flank?

(När både adr- data- & kl-sig. är giltiga)



az faller $\phi 2$ -puls fr
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 exited 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:

```

**DUMP,1000

```

- 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 also possible to display the previous block of 256 bytes by typing a 'B', for backward movement. A carriage return will 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 interrogation 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

TESTR. 308

TRIGGER ORD

A: PC00 (monitor start)

SEQUENCE (A)

STATUS

Redovisar gjorda inställn.

Måtsätt:

SIGNALR analys

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

PARALLEL

SERIELL

Kör igg

När triggordet kommer, jämför data

K kallas av

A. Scrolla värdeangivelser STAT

B. Tidediagr TIMER

Ställ FORMAT

7 el 8 bitar ?

paritet ?

baud rate

Ställ TRIGGERORD

För va 2 st, s alltid kommer i
sa kombiföljd

256 by minnesdjup $\times 8$ ledar
portar

PAR STATE

Rådata

SER "

ASCII-kod

Tidediagr.

Slå av ström
fe prob i koppl.

```
**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

It is often desired to have the program, one instruction at a time, to be executed at the rate of one instruction per second. This is accomplished by the use of a clock. The clock is a device which generates a series of pulses at a constant rate. Each pulse is used to advance the program counter by one instruction. The program counter is a register which holds the address of the instruction to be executed next. The clock is connected to the program counter so that each pulse advances the counter by one instruction.

The clock is connected to the program counter so that each pulse advances the counter by one instruction. The clock is a device which generates a series of pulses at a constant rate. Each pulse is used to advance the program counter by one instruction. The program counter is a register which holds the address of the instruction to be executed next. The clock is connected to the program counter so that each pulse advances the counter by one instruction.

The clock is connected to the program counter so that each pulse advances the counter by one instruction. The clock is a device which generates a series of pulses at a constant rate. Each pulse is used to advance the program counter by one instruction. The program counter is a register which holds the address of the instruction to be executed next. The clock is connected to the program counter so that each pulse advances the counter by one instruction.

First, 20

The clock is connected to the program counter so that each pulse advances the counter by one instruction. The clock is a device which generates a series of pulses at a constant rate. Each pulse is used to advance the program counter by one instruction. The program counter is a register which holds the address of the instruction to be executed next. The clock is connected to the program counter so that each pulse advances the counter by one instruction.

Programmed the program

The clock is connected to the program counter so that each pulse advances the counter by one instruction. The clock is a device which generates a series of pulses at a constant rate. Each pulse is used to advance the program counter by one instruction. The program counter is a register which holds the address of the instruction to be executed next. The clock is connected to the program counter so that each pulse advances the counter by one instruction.

20

The clock is connected to the program counter so that each pulse advances the counter by one instruction. The clock is a device which generates a series of pulses at a constant rate. Each pulse is used to advance the program counter by one instruction. The program counter is a register which holds the address of the instruction to be executed next. The clock is connected to the program counter so that each pulse advances the counter by one instruction.

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

② 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.

③④ 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.

⑤ 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

...of the ...
...
...

...
...
...

...
...
...
...
...

...
...
...

...
...
...

...
...
...
...

...
...
...

...
...
...
...
...

...
...

...
...
...
...
...
...
...

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 '<' modifier, 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

the Bureau of the Census, Department of Commerce, Washington, D.C. 20540, and the Bureau of Economic Analysis, Department of Commerce, Washington, D.C. 20540. The Bureau of the Census is the primary agency for the collection, analysis, and dissemination of statistical information. The Bureau of Economic Analysis is the primary agency for the collection, analysis, and dissemination of economic information. The Bureau of the Census is the primary agency for the collection, analysis, and dissemination of statistical information. The Bureau of Economic Analysis is the primary agency for the collection, analysis, and dissemination of economic information.

Page 10 of 10

Page 10 of 10

Both agencies are authorized to collect, analyze, and disseminate statistical and economic information. The Bureau of the Census is the primary agency for the collection, analysis, and dissemination of statistical information. The Bureau of Economic Analysis is the primary agency for the collection, analysis, and dissemination of economic information. The Bureau of the Census is the primary agency for the collection, analysis, and dissemination of statistical information. The Bureau of Economic Analysis is the primary agency for the collection, analysis, and dissemination of economic information.

Page 10 of 10

The Bureau of the Census is the primary agency for the collection, analysis, and dissemination of statistical information. The Bureau of Economic Analysis is the primary agency for the collection, analysis, and dissemination of economic information. The Bureau of the Census is the primary agency for the collection, analysis, and dissemination of statistical information. The Bureau of Economic Analysis is the primary agency for the collection, analysis, and dissemination of economic information.

The Bureau of the Census is the primary agency for the collection, analysis, and dissemination of statistical information. The Bureau of Economic Analysis is the primary agency for the collection, analysis, and dissemination of economic information. The Bureau of the Census is the primary agency for the collection, analysis, and dissemination of statistical information. The Bureau of Economic Analysis is the primary agency for the collection, analysis, and dissemination of economic information.

The Bureau of the Census is the primary agency for the collection, analysis, and dissemination of statistical information. The Bureau of Economic Analysis is the primary agency for the collection, analysis, and dissemination of economic information. The Bureau of the Census is the primary agency for the collection, analysis, and dissemination of statistical information. The Bureau of Economic Analysis is the primary agency for the collection, analysis, and dissemination of economic information.

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 location 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.

The first example is a simple program that prints out the numbers 1 through 10. This is a very basic program, but it illustrates the basic structure of a program. The program starts with a loop that iterates from 1 to 10. Inside the loop, there is a print statement that outputs the current value of the counter. The program ends with a return statement that exits the program.

The second example is a program that calculates the sum of the numbers 1 through 10. This program is slightly more complex than the first one, but it follows the same basic structure. It uses a loop to iterate from 1 to 10, and inside the loop, it adds the current value of the counter to a running total. The program ends with a print statement that outputs the final sum.

The third example is a program that prints out the numbers 1 through 10, but only the even numbers. This program is more complex than the previous ones, but it still follows the same basic structure. It uses a loop to iterate from 1 to 10, and inside the loop, it checks if the current value of the counter is even. If it is, it prints the value. If it is not, it skips the print statement and continues the loop.

The fourth example is a program that prints out the numbers 1 through 10, but only the odd numbers. This program is similar to the third one, but it checks if the current value of the counter is odd instead of even. The program ends with a print statement that outputs the final sum of the odd numbers.

The fifth example is a program that prints out the numbers 1 through 10, but only the numbers that are divisible by 3. This program is more complex than the previous ones, but it still follows the same basic structure. It uses a loop to iterate from 1 to 10, and inside the loop, it checks if the current value of the counter is divisible by 3. If it is, it prints the value. If it is not, it skips the print statement and continues the loop.

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

The first step in the process of debugging is to identify the problem. This is often done by looking at the error messages and the behavior of the program. Once the problem is identified, the next step is to isolate the cause. This is often done by using a debugger to step through the code and watch the values of variables and the execution of instructions.

There are several different types of debuggers available. Some are built into the compiler, while others are separate programs. The choice of debugger depends on the language and the operating system. Once a debugger is chosen, the next step is to learn how to use it. This is often done by reading the documentation and by practicing with simple programs.

One of the most common types of debugger is the command-line debugger. This type of debugger is used by typing commands at a prompt. The commands are used to set breakpoints, step through the code, and watch the values of variables. The command-line debugger is often used for debugging small programs or for debugging specific parts of a larger program.

Another type of debugger is the graphical debugger. This type of debugger has a graphical user interface (GUI) that allows the user to interact with the debugger in a more intuitive way. The GUI typically includes a window for the source code, a window for the call stack, and a window for the watch list. Graphical debuggers are often used for debugging larger programs or for debugging complex data structures.

The last type of debugger is the hardware debugger. This type of debugger is used to debug hardware devices, such as microprocessors and memory modules. Hardware debuggers are often used in the development of embedded systems and in the diagnosis of hardware problems. They typically consist of a hardware interface unit and a software component that runs on a host computer.

Once a debugger is chosen and the user is familiar with its basic operations, the next step is to apply the debugger to the program. This is often done by setting breakpoints at the locations where the problem is suspected to occur. The user then steps through the code, watching the values of variables and the execution of instructions. When the program reaches a breakpoint, the debugger pauses execution, allowing the user to inspect the state of the program.

There are several different techniques for debugging. One technique is to use a debugger to step through the code line by line. This is often used for debugging small programs or for debugging specific parts of a larger program. Another technique is to use a debugger to set breakpoints at the locations where the problem is suspected to occur. This is often used for debugging larger programs or for debugging complex data structures.

IX. And finally, the last step in the process of debugging is to verify the solution. This is often done by running the program again and watching the behavior. If the problem is solved, the program should run correctly. If the problem is not solved, the user should return to the previous step and try to isolate the cause again. Debugging is often an iterative process, and it may take several attempts to solve a problem.

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!

...the ... of the ...
...the ... of the ...
...the ... of the ...

...
...
...
...
...

...the ... of the ...
...the ... of the ...
...the ... of the ...

...the ... of the ...
...the ... of the ...
...the ... of the ...

...the ... of the ...
...the ... of the ...
...the ... of the ...

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:

<u>Item</u>	<u>Meaning</u>
<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

1. Introduction

This document is a report on the results of a study conducted by the author. The study was designed to investigate the effects of a certain treatment on a specific population. The results of the study are presented in the following sections. The study was conducted over a period of six months, and the results are presented in the following sections. The study was conducted over a period of six months, and the results are presented in the following sections.

2. Methodology

The study was conducted using a randomized controlled trial design. The participants were randomly assigned to two groups: a control group and an experimental group. The control group received a placebo, while the experimental group received the treatment. The results of the study are presented in the following sections. The study was conducted over a period of six months, and the results are presented in the following sections. The study was conducted over a period of six months, and the results are presented in the following sections.

3. Results

The results of the study are presented in the following sections. The study was conducted over a period of six months, and the results are presented in the following sections. The study was conducted over a period of six months, and the results are presented in the following sections.

4. Conclusion

The results of the study suggest that the treatment has a significant effect on the population. The study was conducted over a period of six months, and the results are presented in the following sections. The study was conducted over a period of six months, and the results are presented in the following sections.

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
= \$A+10
\$14 20

Enter calculator mode
Add hex A and 10
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
DELAY=0

Set delay to 100
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

1/1 sätt flagga på cell
✓ se anropas f a se vad s blev result i cellen

FLAG[=<address>] (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
FLAG

Set flag register to \$1A85
Disable flag register printout

IND=ON or OFF

ON: Visar vad indexregistra pekar på

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
IND=OFF

Turn indirection on
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
MO=0

Set mode to 1
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

1. The purpose of this document is to provide a detailed description of the system and its components. The system is designed to provide a secure and reliable means of communication and data storage. It is intended for use by authorized personnel only. The system consists of a central processing unit, a memory unit, and a communication unit. The central processing unit is responsible for the execution of the program and the management of the system resources. The memory unit is used to store the program and the data. The communication unit is used to transmit and receive data. The system is designed to be highly secure and reliable. It is intended to be used in a secure environment. The system is designed to be used by authorized personnel only. The system is designed to be used in a secure environment. The system is designed to be used by authorized personnel only. The system is designed to be used in a secure environment.

EXAMPLE 1: Display of system status

2. The purpose of this document is to provide a detailed description of the system and its components. The system is designed to provide a secure and reliable means of communication and data storage. It is intended for use by authorized personnel only. The system consists of a central processing unit, a memory unit, and a communication unit. The central processing unit is responsible for the execution of the program and the management of the system resources. The memory unit is used to store the program and the data. The communication unit is used to transmit and receive data. The system is designed to be highly secure and reliable. It is intended to be used in a secure environment. The system is designed to be used by authorized personnel only. The system is designed to be used in a secure environment. The system is designed to be used by authorized personnel only. The system is designed to be used in a secure environment.

EXAMPLE 2: Display of system status

3. The purpose of this document is to provide a detailed description of the system and its components. The system is designed to provide a secure and reliable means of communication and data storage. It is intended for use by authorized personnel only. The system consists of a central processing unit, a memory unit, and a communication unit. The central processing unit is responsible for the execution of the program and the management of the system resources. The memory unit is used to store the program and the data. The communication unit is used to transmit and receive data. The system is designed to be highly secure and reliable. It is intended to be used in a secure environment. The system is designed to be used by authorized personnel only. The system is designed to be used in a secure environment. The system is designed to be used by authorized personnel only. The system is designed to be used in a secure environment.

EXAMPLE 3: Display of system status

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^x 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.

EXAMPLES:

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

STATES (STAT or STATES) *Kolla kvalitén på program*

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
-------	---------------------------------

1. The first step is to determine the value of the variable 'x' at the start of the loop. This is done by setting 'x' equal to the value of the variable 'y' at the start of the loop.

2. The second step is to calculate the value of the variable 'x' at the end of the loop. This is done by adding the value of 'x' to the value of 'y' at the end of the loop.

3. The third step is to calculate the value of the variable 'x' at the start of the loop. This is done by setting 'x' equal to the value of the variable 'y' at the start of the loop.

4. The fourth step is to calculate the value of the variable 'x' at the end of the loop. This is done by adding the value of 'x' to the value of 'y' at the end of the loop.

5. The fifth step is to calculate the value of the variable 'x' at the start of the loop. This is done by setting 'x' equal to the value of the variable 'y' at the start of the loop.

6. The sixth step is to calculate the value of the variable 'x' at the end of the loop. This is done by adding the value of 'x' to the value of 'y' at the end of the loop.

7. The seventh step is to calculate the value of the variable 'x' at the start of the loop. This is done by setting 'x' equal to the value of the variable 'y' at the start of the loop.

8. The eighth step is to calculate the value of the variable 'x' at the end of the loop. This is done by adding the value of 'x' to the value of 'y' at the end of the loop.

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 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[,<address>] : (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

1. The first part of the report is a general introduction to the subject of the study. It discusses the importance of the study and the objectives of the research.

2. The second part of the report is a detailed description of the methodology used in the study. It includes information about the sample, the data collection methods, and the statistical analysis.

3. The third part of the report is a discussion of the results of the study. It presents the findings of the research and discusses their implications for the field of study.

4. The fourth part of the report is a conclusion and a list of references. The conclusion summarizes the main findings of the study, and the references list the sources of information used in the research.

5. The fifth part of the report is a list of appendices. These appendices contain additional information that is relevant to the study but is not included in the main body of the report.

6. The sixth part of the report is a list of figures and tables. These figures and tables provide a visual representation of the data and results of the study.

7. The seventh part of the report is a list of footnotes. These footnotes provide additional information about the study and its findings.

8. The eighth part of the report is a list of acknowledgments. These acknowledgments thank the individuals and organizations that provided support and assistance during the study.

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
ST, 2A

Start simulation at \$1000
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
STEP, 10

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

TRACE=<value> (T or TRACE) *Koplad + R[EG]*

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
T=0

Enable trace at all levels
Disable trace mode

Dumper tyebar i o c. d nesting dump man specificat.

SECRET

CLASSIFICATION: CONFIDENTIAL

1. The purpose of this document is to provide information regarding the activities of the [redacted] and the [redacted] in the [redacted] area. This information is being provided to you for your information only and is not to be distributed outside of your organization.

2. The information contained in this document is classified as [redacted] and is to be handled accordingly. It is the responsibility of the recipient to ensure that this information is not disclosed to unauthorized personnel.

SECRET

CLASSIFICATION: CONFIDENTIAL

3. The information contained in this document is classified as [redacted] and is to be handled accordingly. It is the responsibility of the recipient to ensure that this information is not disclosed to unauthorized personnel. The information is being provided to you for your information only and is not to be distributed outside of your organization.

4. The information contained in this document is classified as [redacted] and is to be handled accordingly. It is the responsibility of the recipient to ensure that this information is not disclosed to unauthorized personnel.

SECRET

CLASSIFICATION: CONFIDENTIAL

5. The information contained in this document is classified as [redacted] and is to be handled accordingly. It is the responsibility of the recipient to ensure that this information is not disclosed to unauthorized personnel. The information is being provided to you for your information only and is not to be distributed outside of your organization.

6. The information contained in this document is classified as [redacted] and is to be handled accordingly. It is the responsibility of the recipient to ensure that this information is not disclosed to unauthorized personnel.

7. The information contained in this document is classified as [redacted] and is to be handled accordingly. It is the responsibility of the recipient to ensure that this information is not disclosed to unauthorized personnel.

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.

LE71207

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 the following precedence. ① If the address of the current PC matches the address of a breakpoint, then the pass count is checked. ② If the counter is in a state to allow continuing, then the condition is checked (if present). Finally the ③ actions 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>]

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
BP

List breakpoints between \$10 & \$C00
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
CLB,0-100
CLB,22A

Clear all breakpoints
Clear breakpoints between \$0 & \$100
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
CLH,25-200

Clear all histogram counters
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. 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

*Släppa DEBUG & låta CPU:n köra fram & angiven adr. (s måste va 1a byten i en instrukt. & måste ligga i RAM)
(Innebär a JMP:n kör & får ett software-intropt w angiven adress.)*

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.

- ② Write protected memory will cause a trap on any attempt to write to it. ② Execute protect will not allow opcodes to be fetched. ④ Memory protect will not permit any type of reference; read, write, or execute. ④ Simulate 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.

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
BO,M,X

Display all memory protection
Display memory and execute
protection bounds

CLPI,<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
CLP,X

Clear all protection
Clear execute protection

- ① M $\approx$ där finns inget
- ② X k tex läggs ö data-areor
där inget finns a exekvera
- ③ W läggs lämpl ö programmets minnes-are
- ④ S u tex " 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

PURPOSE:

se ox s13-31

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 instruction counter

The first number of the series is 1, and the second is 1. The third number is 2, and the fourth is 3. The fifth number is 5, and the sixth is 8. The seventh number is 13, and the eighth is 21. The ninth number is 34, and the tenth is 55. The eleventh number is 89, and the twelfth is 144. The thirteenth number is 233, and the fourteenth is 377. The fifteenth number is 610, and the sixteenth is 987. The seventeenth number is 1597, and the eighteenth is 2584. The nineteenth number is 4181, and the twentieth is 6765. The twenty-first number is 10946, and the twenty-second is 17711. The twenty-third number is 46368, and the twenty-fourth is 75025. The twenty-fifth number is 196410, and the twenty-sixth is 317811. The twenty-seventh number is 806515, and the twenty-eighth is 1313953. The twenty-ninth number is 3355536, and the thirtieth is 5497554. The thirty-first number is 14473934, and the thirty-second is 23781571. The thirty-third number is 61091290, and the thirty-fourth is 99194853. The thirty-fifth number is 259695496, and the thirty-sixth is 421303539. The thirty-seventh number is 1677613923, and the thirty-eighth is 2748381301. The thirty-ninth number is 10675103660, and the fortieth is 17340306000. The forty-first number is 67659132365, and the forty-second is 110112379514. The forty-third number is 423354225648, and the forty-fourth is 691497142162. The forty-fifth number is 2679142562443, and the forty-sixth is 4348907007421. The forty-seventh number is 16977091427031, and the forty-eighth is 27777795528712. The forty-ninth number is 109463554685442, and the fiftieth is 177114869538461. The fifty-first number is 1123752223306577, and the fifty-second is 1836311943022191. The fifty-third number is 7447566022189161, and the fifty-fourth is 12139339202007771. The fifty-fifth number is 47539866607587000, and the fifty-sixth is 77056046196117419. The fifty-seventh number is 309321546176669936, and the fifty-eighth is 501457142686640673. The fifty-ninth number is 3293838464004983696, and the sixtieth is 5349203054947574421. The sixty-first number is 21447869468819432000, and the sixty-second is 34732813744371320557. The sixty-third number is 141872530854062771140, and the sixty-fourth is 230478838165246911239. The sixty-fifth number is 922476670435977326328, and the sixty-sixth is 1498506000721470921927. The sixty-seventh number is 5912864999156288800962, and the sixty-eighth is 9613855475366205055761. The sixty-ninth number is 61305790721816562586809, and the seventieth is 100007702639166854811360. The seventy-first number is 622702081402137376425421, and the seventy-second is 1010334378754947961971771. The seventy-third number is 6324655077576465965616660, and the seventy-fourth is 10210688166304427581689431. The seventy-fifth number is 64234898917081101441921871, and the seventy-sixth is 10321807045054375740660402. The seventy-seventh number is 65235324584891211658118042, and the seventy-eighth is 104367988967597932177764453. The seventy-ninth number is 66247924676280321874266063, and the eightieth is 105556637864157659855430884. The eighty-first number is 67272798252469432090414084, and the eighty-second is 106784116750717387533101715. The eighty-third number is 68319948404758542306562105, and the eighty-fourth is 108050535637277115210772546. The eighty-fifth number is 69389480257047652522710126, and the eighty-sixth is 109355994523836842888443377. The eighty-seventh number is 70481518109336762738858147, and the eighty-eighth is 110700604410396570566114208. The eighty-ninth number is 71596280961625872955006168, and the ninetieth is 112084474296956298243785039. The ninety-first number is 72733898813914983171154189, and the ninety-second is 113507724183516025921455870. The ninety-third number is 73894500666204093387302210, and the ninety-fourth is 114970474070075753599126701. The ninety-fifth number is 75078220518493203603450231, and the ninety-sixth is 116472723956635481276797532. The ninety-seventh number is 76285190370782313819598252, and the ninety-eighth is 118014573843195208954468363. The ninety-ninth number is 77515540223071424035746273, and the hundredth is 119596023729754936632139194. The hundred-first number is 78769410075360534251894294, and the hundred-second is 121217273616314664309810025. The hundred-third number is 80046939927649644468042315, and the hundred-fourth is 122878523502874391987480856. The hundred-fifth number is 81348279779938754684190336, and the hundred-sixth is 124579773389434119665151687. The hundred-seventh number is 82673579632227864900338357, and the hundred-eighth is 126321023275993847342822518. The hundred-ninth number is 84023999484516975116486378, and the hundred-tieth is 128103273162553575020493349. The hundred-first number is 85399709336806085332634399, and the hundred-second is 129927523049113302698164180. The hundred-third number is 86799879189095195548782420, and the hundred-fourth is 131794772935673030375835011. The hundred-fifth number is 88224679041384305764930441, and the hundred-sixth is 133705022822232758053505842. The hundred-seventh number is 89674278893673415981078462, and the hundred-eighth is 135659272708792485731176673. The hundred-ninth number is 91148848745962526197226483, and the hundred-tieth is 137658522595352213408847504. The hundred-first number is 92648558598251636413374504, and the hundred-second is 139703772481911941086518335. The hundred-third number is 94173488450540746629522525, and the hundred-fourth is 141796022368471668764189166. The hundred-fifth number is 95723808302829856845670546, and the hundred-sixth is 143936272255031396441860000. The hundred-seventh number is 97299788155118967061818567, and the hundred-eighth is 146124522141591124119530835. The hundred-ninth number is 98901688007408077277966588, and the hundred-tieth is 148361772028150851797201670. The hundred-first number is 100529767859697187494114609, and the hundred-second is 150648021914710579474872505. The hundred-third number is 102184107711986297710262630, and the hundred-fourth is 152983271801270307152543340. The hundred-fifth number is 103864987564275407926410651, and the hundred-sixth is 155367521687830034830214175. The hundred-seventh number is 105572667416564518142558672, and the hundred-eighth is 157800771574389762507885010. The hundred-ninth number is 107307407268853628358706693, and the hundred-tieth is 160283021460949490185555845. The hundred-first number is 109069867121142738574854714, and the hundred-second is 162815271347509217863226680. The hundred-third number is 110860306973431848791002735, and the hundred-fourth is 165397521234068945540897515. The hundred-fifth number is 112679096825720959007150756, and the hundred-sixth is 168030771120628673218568350. The hundred-seventh number is 114526606678010069223298777, and the hundred-eighth is 170715021007188400896239185. The hundred-ninth number is 116403316530299179439446798, and the hundred-tieth is 173451270893748128573910020. The hundred-first number is 118309626382588289655594819, and the hundred-second is 176240520780307856251580855. The hundred-third number is 120245936234877399871742840, and the hundred-fourth is 179083770666867583929251690. The hundred-fifth number is 122212646087166509987890861, and the hundred-sixth is 181981020553427311606922525. The hundred-seventh number is 124210165939455620104038882, and the hundred-eighth is 184933270440000000000000000. The hundred-ninth number is 126238915791744730220186903, and the hundred-tieth is 187941520326569727677670835. The hundred-first number is 128299335644033840336334924, and the hundred-second is 191006770213129455355341670. The hundred-third number is 130391875496322950452482945, and the hundred-fourth is 194139020100000000000000000. The hundred-fifth number is 132516985348612060568630966, and the hundred-sixth is 197339270000000000000000000. The hundred-seventh number is 134675115200901170684778987, and the hundred-eighth is 200607520000000000000000000. The hundred-ninth number is 136866715053190280799926998, and the hundred-tieth is 203943770000000000000000000. The hundred-first number is 139092254905479390915075019, and the hundred-second is 207348020000000000000000000. The hundred-third number is 141352294757768501030223040, and the hundred-fourth is 210821270000000000000000000. The hundred-fifth number is 143647294610057611145371061, and the hundred-sixth is 214363520000000000000000000. The hundred-seventh number is 145977714462346721260519082, and the hundred-eighth is 217974770000000000000000000. The hundred-ninth number is 148344034314635831375667103, and the hundred-tieth is 221655020000000000000000000. The hundred-first number is 150746714166924941490815124, and the hundred-second is 225405270000000000000000000. The hundred-third number is 153185314019214051605963145, and the hundred-fourth is 229225520000000000000000000. The hundred-fifth number is 155659393871503161721111166, and the hundred-sixth is 233115770000000000000000000. The hundred-seventh number is 158169513723792271836259187, and the hundred-eighth is 237076020000000000000000000. The hundred-ninth number is 160716233576081381951407208, and the hundred-tieth is 241107270000000000000000000. The hundred-first number is 163299993428370492066555229, and the hundred-second is 245209520000000000000000000. The hundred-third number is 165921233280659602181703250, and the hundred-fourth is 249383770000000000000000000. The hundred-fifth number is 168579473132948712296851271, and the hundred-sixth is 253629020000000000000000000. The hundred-seventh number is 171275152985237822412000000, and the hundred-eighth is 257945270000000000000000000. The hundred-ninth number is 174008732837526932527148000, and the hundred-tieth is 262332520000000000000000000. The hundred-first number is 176780672689816042642296000, and the hundred-second is 266790770000000000000000000. The hundred-third number is 179591432542105152757444000, and the hundred-fourth is 271319020000000000000000000. The hundred-fifth number is 182441552394394262872592000, and the hundred-sixth is 275917270000000000000000000. The hundred-seventh number is 185331572246683372987740000, and the hundred-eighth is 280585520000000000000000000. The hundred-ninth number is 188262052098972483102888000, and the hundred-tieth is 285323770000000000000000000. The hundred-first number is 191233531951261593218036000, and the hundred-second is 290132020000000000000000000. The hundred-third number is 194246511803550703333184000, and the hundred-fourth is 295010270000000000000000000. The hundred-fifth number is 197301491655839813448332000, and the hundred-sixth is 300000000000000000000000000. The hundred-seventh number is 200398971508128923563480000, and the hundred-eighth is 305100000000000000000000000. The hundred-ninth number is 203539451360418033678628000, and the hundred-tieth is 310310000000000000000000000. The hundred-first number is 206723431212707143793776000, and the hundred-second is 315630000000000000000000000. The hundred-third number is 209951411065000000000000000, and the hundred-fourth is 321060000000000000000000000. The hundred-fifth number is 213223990917289153908864000, and the hundred-sixth is 326600000000000000000000000. The hundred-seventh number is 216541670769578264018016000, and the hundred-eighth is 332250000000000000000000000. The hundred-ninth number is 219904850621867374127168000, and the hundred-tieth is 338010000000000000000000000. The hundred-first number is 223313030474156484236320000, and the hundred-second is 343880000000000000000000000. The hundred-third number is 226766610326445594345472000, and the hundred-fourth is 349860000000000000000000000. The hundred-fifth number is 230266090178734704454624000, and the hundred-sixth is 355950000000000000000000000. The hundred-seventh number is 233811970031023814563776000, and the hundred-eighth is 362150000000000000000000000. The hundred-ninth number is 237404750883312924672928000, and the hundred-tieth is 368460000000000000000000000. The hundred-first number is 241045030735602034782080000, and the hundred-second is 374880000000000000000000000. The hundred-third number is 244733310587891144891232000, and the hundred-fourth is 381410000000000000000000000. The hundred-fifth number is 248469990440180254999384000, and the hundred-sixth is 388050000000000000000000000. The hundred-seventh number is 252255570292469365108536000, and the hundred-eighth is 394800000000000000000000000. The hundred-ninth number is 256090550144758475217688000, and the hundred-tieth is 401660000000000000000000000. The hundred-first number is 259975330000000000000000000, and the hundred-second is 408630000000000000000000000. The hundred-third number is 263910310852289585326840000, and the hundred-fourth is 415710000000000000000000000. The hundred-fifth number is 267895990704578695435992000, and the hundred-sixth is 422900000000000000000000000. The hundred-seventh number is 271932870556867805545144000, and the hundred-eighth is 430200000000000000000000000. The hundred-ninth number is 276021450409156915654296000, and the hundred-tieth is 437610000000000000000000000. The hundred-first number is 280162230261446025763448000, and the hundred-second is 445130000000000000000000000. The hundred-third number is 284355710113735135872600000, and the hundred-fourth is 452760000000000000000000000. The hundred-fifth number is 288602490000000000000000000, and the hundred-sixth is 460500000000000000000000000. The hundred-seventh number is 292903170852289585326840000, and the hundred-eighth is 468350000000000000000000000. The hundred-ninth number is 297258350704578695435992000, and the hundred-tieth is 476310000000000000000000000. The hundred-first number is 301668530556867805545144000, and the hundred-second is 484380000000000000000000000. The hundred-third number is 306134310409156915654296000, and the hundred-fourth is 492560000000000000000000000. The hundred-fifth number is 310656290261446025763448000, and the hundred-sixth is 500850000000000000000000000. The hundred-seventh number is 315234970113735135872600000, and the hundred-eighth is 509250000000000000000000000. The hundred-ninth number is 319869950000000000000000000, and the hundred-tieth is 517760000000000000000000000. The hundred-first number is 324561830852289585326840000, and the hundred-second is 526380000000000000000000000. The hundred-third number is 329311210704578695435992000, and the hundred-fourth is 535110000000000000000000000. The hundred-fifth number is 334118690556867805545144000, and the hundred-sixth is 543950000000000000000000000. The hundred-seventh number is 338984870409156915654296000, and the hundred-eighth is 552900000000000000000000000. The hundred-ninth number is 343910250261446025763448000, and the hundred-tieth is 561960000000000000000000000. The hundred-first number is 348895430113735135872600000, and the hundred-second is 571130000000000000000000000. The hundred-third number is 353940910000000000000000000, and the hundred-fourth is 580410000000000000000000000. The hundred-fifth number is 359047290852289585326840000, and the hundred-sixth is 589800000000000000000000000. The hundred-seventh number is 364215170704578695435992000, and the hundred-eighth is 599300000000000000000000000. The hundred-ninth number is 369445050556867805545144000, and the hundred-tieth is 608910000000000000000000000. The hundred-first number is 374737530409156915654296000, and the hundred-second is 618630000000000000000000000. The hundred-third number is 380093210261446025763448000, and the hundred-fourth is 628460000000000000000000000. The hundred-fifth number is 385512690113735135872600000, and the hundred-sixth is 638400000000000000000000000. The hundred-seventh number is 390996470000000000000000000, and the hundred-eighth is 648450000000000000000000000. The hundred-ninth number is 396545150852289585326840000, and the hundred-tieth is 658610000000000000000000000. The hundred-first number is 402159330704578695435992000, and the hundred-second is 668880000000000000000000000. The hundred-third number is 407839510556867805545144000, and the hundred-fourth is 679260000000000000000000000. The hundred-fifth number is 413586290409156915654296000, and the hundred-sixth is 689750000000000000000000000. The hundred-seventh number is 419399270261446025763448000, and the hundred-eighth is 699350000000000000000000000. The hundred-ninth number is 425279050113735135872600000, and the hundred-tieth is 709060000000000000000000000. The hundred-first number is 431225330000000000000000000, and the hundred-second is 718880000000000000000000000. The hundred-third number is 437238710852289585326840000, and the hundred-fourth is 728810000000000000000000000. The hundred-fifth number is 443319690704578695435992000, and the hundred-sixth is 738850000000000000000000000. The hundred-seventh number is 449468870556867805545144000, and the hundred-eighth is 748990000000000000000000000. The hundred-ninth number is 455686750409156915654296000, and the hundred-tieth is 759230000000000000000000000. The hundred-first number is 461973930261446025763448000, and the hundred-second is 769570000000000000000000000. The hundred-third number is 468330910113735135872600000, and the hundred-fourth is 779910000000000000000000000. The hundred-fifth number is 474758290000000000000000000, and the hundred-sixth is 790350000000000000000000000. The hundred-seventh number is 481256670852289585326840000, and the hundred-eighth is 800890000000000000000000000. The hundred-ninth number is 487826650704578695435992000, and the hundred-tieth is 811530000000000000000000000. The hundred-first number is 494468830556867805545144000, and the hundred-second is 822270000000000000000000000. The hundred-third number is 501183610409156915654296000, and the hundred-fourth is 833110000000000000000000000. The hundred-fifth number is 507971490261446025763448000, and the hundred-sixth is 844050000000000000000000000. The hundred-seventh number is 514833070113735135872600000, and the hundred-eighth is 855090000000000000000000000. The hundred-ninth number is 521768850000000000000000000, and the hundred-tieth is 866230000000000000000000000. The hundred-first number is 528779430852289585326840000, and the hundred-second is 877470000000000000000000000. The hundred-third number is 535865410704578695435992000, and the hundred-fourth is 888810000000000000000000000. The hundred-fifth number is 543027390556867805545144000, and the hundred-sixth is 899250000000000000000000000. The hundred-seventh number is 550265970409156915654296000, and the hundred-eighth is 909790000000000000000000000. The hundred-ninth number is 557580750261446025763448000, and the hundred-tieth is 920430000000000000000000000. The hundred-first number is 564972330113735135872600000, and the hundred-second is 931170000000000000000000000. The hundred-third number is 572441310000000000000000000, and the hundred-fourth is 942010000000000000000000000. The hundred-fifth number is 579988290852289585326840000, and the hundred-sixth is 952950000000000000000000000. The hundred-seventh number is 587613870704578695435992000, and the hundred-eighth is 963990000000000000000000000. The hundred-ninth number is 595318650556867805545144000, and the hundred-tieth is 975130000000000000000000000. The hundred-first number is 603103230409156915654296000, and the hundred-second is 986370000000000000000000000. The hundred-third number is 610968410261446025763448000, and the hundred-fourth is 997710000000000000000000000. The hundred-fifth number is 618914790113735135872600000, and the hundred-sixth is 1009150000000000000000000000. The hundred-seventh number is 626943070000000000000000000, and the hundred-eighth is 1020690000000000000000000000. The hundred-ninth number is 635054850852289585326840000, and the hundred-tieth is 1032330000000000000000000000. The hundred-first number is 643249630704578695435992000, and the hundred-second is 1044070000000000000000000000. The hundred-third number is 651528010556867805545144000, and the hundred-fourth is 1055910000000000000000000000. The hundred-fifth number is 659890690409156915654296000, and the hundred-sixth is 1067850000000000000000000000. The hundred-seventh number is 668338270261446025763448000, and the hundred-eighth is 1079890000000000000000000000. The hundred-ninth number is 676871450113735135872600000, and the hundred-tieth is 1092030000000000000000000000. The hundred-first number is 6854908300000

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. *Avsadd a sättes på sista instruktion i program.*

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

SECRET

STANDARD OF 100

The first 100 items on which the standard is based are those which are of the highest quality. These items are selected from the total population of items. The standard is then applied to the remaining items. The items which are of the highest quality are those which are of the highest quality. The items which are of the highest quality are those which are of the highest quality.

STANDARD OF 100
100 items
100 items

SECRET

STANDARD OF 100

The first 100 items on which the standard is based are those which are of the highest quality. These items are selected from the total population of items. The standard is then applied to the remaining items. The items which are of the highest quality are those which are of the highest quality. The items which are of the highest quality are those which are of the highest quality.

STANDARD OF 100
100 items
100 items

SECRET

STANDARD OF 100

The first 100 items on which the standard is based are those which are of the highest quality. These items are selected from the total population of items. The standard is then applied to the remaining items. The items which are of the highest quality are those which are of the highest quality. The items which are of the highest quality are those which are of the highest quality.

STANDARD OF 100
100 items
100 items

SECRET

STANDARD OF 100

The first 100 items on which the standard is based are those which are of the highest quality. These items are selected from the total population of items. The standard is then applied to the remaining items. The items which are of the highest quality are those which are of the highest quality. The items which are of the highest quality are those which are of the highest quality.

STANDARD OF 100
100 items
100 items

L871207

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
IRQ=0

Generate IRQ every 5000 instructions
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
NMI=0

Generate NMI every 300 instructions
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 does 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 encountered 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.

THE HISTORY OF THE UNITED STATES OF AMERICA
FROM 1776 TO 1876
BY CHARLES A. BEAN

VOLUME I
FROM 1776 TO 1800

NEW YORK: PUBLISHED BY THE
AUTHOR, 15 NASSAU ST.

1876

THE HISTORY OF THE UNITED STATES OF AMERICA
FROM 1776 TO 1876
BY CHARLES A. BEAN

VOLUME I
FROM 1776 TO 1800

NEW YORK: PUBLISHED BY THE
AUTHOR, 15 NASSAU ST.

1876

THE HISTORY OF THE UNITED STATES OF AMERICA
FROM 1776 TO 1876
BY CHARLES A. BEAN

VOLUME I
FROM 1776 TO 1800

NEW YORK: PUBLISHED BY THE
AUTHOR, 15 NASSAU ST.

1876

THE HISTORY OF THE UNITED STATES OF AMERICA
FROM 1776 TO 1876
BY CHARLES A. BEAN

VOLUME I
FROM 1776 TO 1800

NEW YORK: PUBLISHED BY THE
AUTHOR, 15 NASSAU ST.

1876

THE HISTORY OF THE UNITED STATES OF AMERICA
FROM 1776 TO 1876
BY CHARLES A. BEAN

VOLUME I
FROM 1776 TO 1800

NEW YORK: PUBLISHED BY THE
AUTHOR, 15 NASSAU ST.

1876

1. **CONDUCTED**
 2. **THE**
 3. **CONFERENCE**
 4. **AND**
 5. **CONCLUDED**
 6. **REMARKS**

1. **CONDUCTED**
 2. **THE**
 3. **CONFERENCE**
 4. **AND**
 5. **CONCLUDED**
 6. **REMARKS**

```

Line #yntax : "*" = current, "r" = last, digits = offset or
relative line number, "&" - allowed.

Buffered Commands ('F' prompt)

Edit <filename>          Clear buffer and read file.
Get <filename>           Clear buffer and set filename.
Set <filename>           Get full path cursor.
Save <filename>          Save entire file.
Save Partail <line>, <line> Save entire file and quit.
                        <filename>
Move <line>!, <line>     Save part of file.
Copy <line>!, <line>)    Will move lines to after
                        cursor position.
                        Will continue to alter
                        positions cursor to line.
Delete <line>!, <line>)  Will delete lines between
                        positions cursor to line.
Tabn Repetitive <incr>  Sets tabs at columns 0, incr,
                        2 * incr, 3 * incr, etc.
                        (Columns separated by spaces).
Tabn Non-repetitive (<column>) Clears all tabs.
                          Enable display of tab markers.
                          Disable display of tab markers.
                          Display version, filename, and
                          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
the previous session. If you respond with 'Y' or 'y' 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 18-9)

Quit      Exit editor.
Buit      move cursor left one character.
H         move cursor right one character.
L         move cursor up one line.
M         move cursor down one line.
N         move cursor 1/2 screen up.
O         move cursor 1/2 screen down.
P         move cursor to next tab.
R         move cursor to next carriage
        return character deleted.
Ctrl X   Return will merge lines, with multiple leading
        spaces compressed into one space.
        deletes current line.
        Overwrites one character.
        enter append mode.
        enter insert mode.

Text search commands

Find _____ Will position cursor, response is
        CR = stop command
        Space = continue
        M = multiple occurrences on line
        Replace _____ Will position cursor, response is
        CR = stop command
        Space = continue
        M = multiple occurrences on line
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 'F' prompt appears, press 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 occurrence of the find pattern on any one
line. This only changes 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 entered during the current cursor position. If a
carriage return is entered during insert mode, the cursor will
append mode is entered. Append mode will start new lines past the
current line with no partial line carried over. In addition the
following special characters are recognized !
CR or DEL      backwards over character.
Ctrl R         reverse tab.
Ctrl T         reverse end.
Ctrl A         copies remainder of last line entered.
Ctrl X         removes all characters entered so far.
```

PAG enable page formatting

COMMAND

[P, JASMB, <filename1>[, <filename2>[, +<optionalist>]]

optionlist

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

Insert or append mode

Note 1: When the 'r' prompt appears enter the find pattern - if you enter just the find pattern this will terminate the command. When the 'n' prompt appears enter the replace pattern. A '-' is used as a don't care in the find and replace patterns. The 's' option allows you to search across multiple lines. At the end of the first of the multiple occurrences option will allow a match on each occurrence of the find pattern on any one line, also 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 of the buffer and continue until it is positioned at the last match when the command terminates.

Insert or append mode

Character. Unless an escape is entered, insert mode will insert character before the current cursor position. If a carriage return is entered during insert mode, the operation will be terminated and the cursor will be positioned on the line after the current line. Append mode will add the new lines past the current line with no effect on the current line. In addition, the following special characters are recognized:

BS or DEL	backspace over character.
^	toggle case.
RT or DEL	forward tab.
END	end of line.
END	reverse end of line entered.
END	reverse all characters entered so far.

DEBUG COMMAND SUMMARY

General system control

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=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

ASML,<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

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
ctrlC	operator halt	13-26
escape	halt on EOL	13-26
ctrlF	faster	13-26
ctrlS	slower	13-26

Breakpoints

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

B 13-27

Ba,<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

BOUND[,<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	ITrap	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
ctrlI	ctrlI	13-34
ctrlIN	ctrlIN	13-34

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

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

1. The first part of the report is a general introduction to the subject of the study. It discusses the importance of the study and the objectives of the research. It also provides a brief overview of the methodology used in the study.

2. The second part of the report is a detailed description of the study area. It includes information about the location of the study area, the population of the study area, and the characteristics of the study area. It also discusses the data sources used in the study.

3. The third part of the report is a detailed description of the study results. It includes information about the findings of the study, the conclusions drawn from the findings, and the implications of the findings. It also discusses the limitations of the study and the need for further research.

4. The fourth part of the report is a detailed description of the study conclusions. It includes information about the overall findings of the study, the conclusions drawn from the findings, and the implications of the findings. It also discusses the limitations of the study and the need for further research.

5. The fifth part of the report is a detailed description of the study recommendations. It includes information about the recommendations made by the study, the reasons for the recommendations, and the implications of the recommendations. It also discusses the limitations of the study and the need for further research.

6. The sixth part of the report is a detailed description of the study references. It includes information about the references used in the study, the sources of the references, and the relevance of the references. It also discusses the limitations of the study and the need for further research.

7. The seventh part of the report is a detailed description of the study appendices. It includes information about the appendices used in the study, the contents of the appendices, and the relevance of the appendices. It also discusses the limitations of the study and the need for further research.

8. The eighth part of the report is a detailed description of the study acknowledgments. It includes information about the individuals and organizations that provided support for the study, the reasons for the support, and the implications of the support. It also discusses the limitations of the study and the need for further research.

9. The ninth part of the report is a detailed description of the study index. It includes information about the index used in the study, the contents of the index, and the relevance of the index. It also discusses the limitations of the study and the need for further research.

DEBUG COMMAND SUMMARY

General system control

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=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[,<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

Simulation control

GO	Go	13-24
JUMP,<address>	Jump	13-24
PAS[,<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	ITrap	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-stacken startar i DEBUG på \$COFF