Debug.hyper

COLLABORATORS			
	TITLE : Debug.hyper		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		December 6, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Debug.hyper

1.1 Debugging (Tue Nov 3 14:38:07 1992)

```
Contents:
   Introduction
   Loading a program
   Two example sessions
   Tracing
   Breakpoints
   Some theory
   Resident breakpoints
   The 'fdebug' command
   Sourcelevel debugging
   Using the PortPrint feature
   Using the tag system and fd-files
   Summary of all commands

Various:
   Commands used in this tutorial
   Functions used in this tutorial
   Back to main contents
```

1.2 Debugging : Commands used in this tutorial

```
   addtag      Give a type to some memory region
   break       Control breakpoints
   debug       Control debug tasks
   disp        Display integer
   dnexti      scroll to next instruction
   dprevi      scroll to previous instruction
   drefresh    Refresh debug display
   dscroll     Scroll in debug display
   dstart      Set start programcounter in debug display
   duse        Set the default debug task
   dwin        Open/close 'Debug' logical window
   info        Ask information about a structure or node
   list        List structures
   loadfd      Load fd-file
```

```
prefs        Set preferences
source       Load source files for sourcelevel debugger
struct       Make and manage structures
symbol       Control symbols
trace        Control tracing
unasm        Disassemble memory
with         Temporarily set the default debug task
```

1.3 Debugging : Functions used in this tutorial

```
botpc        Get the programcounter at the bottom of the display
toppc        Get the programcounter at the top of the display
```

1.4 Debugging : Introduction

I don’t think that you will be surprised if I tell you that PowerVisor can even debug programs :-) This file explains how you should do this. It also explains how you can make life easier with a fully customized fullscreen debugger. PowerVisor is a very powerful debugger. For example, you can debug multiple tasks at the same time.

Note that PowerVisor is not really a source level debugger, although you can load the source (even for C programs). The source will also follow the current programcounter. In future I plan more support for local and global variables in C. Stack backtracing would also be nice.

Note that the PowerVisor debug system works much better in the AmigaDOS 2.0 version. This is because AmigaDOS 2.0 has some nice features making life a lot easier for the programmer. I’m sorry for all AmigaDOS 1.3 users.
The examples below work on AmigaDOS 1.3, 2.0 and 3.0.

Warning! Only the master instance of PowerVisor can debug programs! All slave instances (instances of PowerVisor started when there was already an instance of PowerVisor running) cannot debug.

1.5 Debugging : Loading a program

There are several ways to load a program. The method you chooses depends on what you really need. The debug command is provided by PowerVisor to control the debug tasks (or debug nodes). All the debug nodes can be found in the ’dbug’ list. With the ’debug’ command you can load a program, you can unload a program and you can do other things as well.

1.6 Debugging : Two example sessions

The following two items are for the first example session. Note that we do not use the fullscreen debugger in this example. Use of the fullscreen debugger is explained in the next session. It is recommended that you type each command as it appears here. Note that the output given here assumes that you have all preferences set to default values (use 'mode shex', 'prefs dmode f' and 'prefs debug 5 1' if you are not sure that the default values are used, see prefs and mode).

Starting session one
Tracing

I have prepared another program so you can see the power of PowerVisor. In this session we are going to make you used to breakpoints and some other advanced features of the trace command. We are also going to use the fullscreen debugger (Note that I will explain later how you can customize this fullscreen debugger to your wishes and how you can use the 'db' script to do this for you).

Starting sessions two
Breakpoints

1.7 Debugging : Starting the first session

'debug n' is the recommended way to load a program (with the debug command). 'debug n' waits for the next program that is started and interrupts it before the first instruction is executed. To do this, 'debug n' patches the Dos LoadSeg function. Example :

```
< debug n <enter>
```

PowerVisor waits for you to start the program you want to debug. You can start this program from the WorkBench (click on the icon) or you can start it from the Cli or Shell. I have provided an example program with a resident breakpoint (see Resident breakpoints). You can find this program in the 'Examples' subdirectory.

```
CLI< examples/buggyprogram <enter>
or
CLI< run examples/buggyprogram <enter>
```

(Note! Only use 'run' when 'run' is resident or a built-in shell command, in other words: don't use 'run' when 'run' itself must be loaded from disk with 'loadseg'. You probably don't want to debug 'run' :-) 'run' is always resident in AmigaDOS 2.0)

(Note! When PowerVisor is waiting for a program you must be careful not to use any other program (that is already running) that might use LoadSeg for some other purpose. Fonts, for example, are loaded using LoadSeg)

(Note! You can interrupt 'debug n' with <esc>)

'debug n' is the best way to load a debug task because the program runs in exactly the same environment as the environment you get when you simply run the program.

Allright, we have now loaded the program in memory.

```
< list dbug <enter>
> Debug task          : Node     Task     InitPC   TD ID Mode   SMode TMode
> ---------------------------------------------------------------------------
> Background CLI      : 07EA7A58 07EF8FA8 07EAA7D8 FF FF NONE  WAIT  NORM
```

Most of this information is rather technical and is not very interesting at this moment. 'InitPC' is interesting though. Let's disassemble some instructions with unasm :

```
< u 07EAA7D8 <enter>
or
< unasm 07EAA7D8 <enter>

> 07EAA7D8: 7200                          MOVEQ.L  #0,D1
> 07EAA7DA: 7064                          MOVEQ.L  #$64,D0
> 07EAA7DC: 5281                          ADDQ.L   #1,D1
> 07EAA7DE: 51C8 FFFC                     DBF      D0,$7EAA7DC
> 07EAA7E2: 6100 0010                     BSR      $7EAA7F4
> 07EAA7E6: 6708                          BEQ      $7EAA7F0
> 07EAA7E8: 6100 0022                     BSR      $7EAA80C
> 07EAA7EC: 6100 0046                     BSR      $7EAA834
> 07EAA7F0: 7000                          MOVEQ.L  #0,D0
> 07EAA7F2: 4E75                          RTS
> 07EAA7F4: 203C 0000 0064                MOVE.L   #$64,D0
> 07EAA7FA: 7200                          MOVEQ.L  #0,D1
> 07EAA7FC: 2C78 0004                     MOVEA.L  (4),A6
> 07EAA800: 4EAE FF3A                     JSR      ($FF3A,A6)
> 07EAA804: 41FA 0042                     LEA      ($7EAA848,PC),A0
> 07EAA808: 2080                          MOVE.L   D0,(A0)
> 07EAA80A: 4E75                          RTS
> 07EAA80C: 7000                          MOVEQ.L  #0,D0
> 07EAA80E: 7201                          MOVEQ.L  #1,D1
> 07EAA810: 7402                          MOVEQ.L  #2,D2
```

Well, this is our program. But there are symbol hunks in our program. 'debug n' does not automatically load them ('debug l' does, but this command will be explained later). You can load symbols with the symbol command :

```
< symbol l examples/buggyprogram <enter>

< u 07EAA7D8 <enter>
> StartProgr7200                          MOVEQ.L  #0,D1
> 07EAA7DA: 7064                          MOVEQ.L  #$64,D0
> loop      5281                          ADDQ.L   #1,D1
> 07EAA7DE: 51C8 FFFC                     DBF      D0,loop
> 07EAA7E2: 6100 0010                     BSR      Sub1
> 07EAA7E6: 6708                          BEQ      theend
> 07EAA7E8: 6100 0022                     BSR      Sub2
> 07EAA7EC: 6100 0046                     BSR      Sub3
> theend    7000                          MOVEQ.L  #0,D0
> 07EAA7F2: 4E75                          RTS
> Sub1      203C 0000 0064                MOVE.L   #$64,D0
> 07EAA7FA: 7200                          MOVEQ.L  #0,D1
```

```
> 07EAA7FC: 2C78 0004                    MOVEA.L  (4),A6
> 07EAA800: 4EAE FF3A                    JSR      ($FF3A,A6)
> 07EAA804: 41FA 0042                    LEA      (Block,PC),A0
> 07EAA808: 2080                         MOVE.L   D0,(A0)
> 07EAA80A: 4E75                         RTS
> Sub2      7000                         MOVEQ.L  #0,D0
> 07EAA80E: 7201                         MOVEQ.L  #1,D1
> 07EAA810: 7402                         MOVEQ.L  #2,D2
```

You can show all symbols with the 'symbol' command :

```
< symbol s <enter>
> StartProgram                            : 07EAA7D8 , 132818904
> loop                                    : 07EAA7DC , 132818908
> theend                                  : 07EAA7F0 , 132818928
> Sub1                                    : 07EAA7F4 , 132818932
> Sub2                                    : 07EAA80C , 132818956
> Sub3                                    : 07EAA834 , 132818996
> Block                                   : 07EAA848 , 132819016
```

The two values on the right of each symbol are the same. The only difference is that the left one is hexadecimal and the right one is decimal.

Because we have loaded the symbols for the current debug task we can use the symbols in expressions. Here are some examples :

Disassemble 5 instructions starting with 'StartProgram' (note that symbols are case sensitive) :

```
< u StartProgram 5 <enter>
> StartProgr7200                         MOVEQ.L  #0,D1
> 07EAA7DA: 7064                         MOVEQ.L  #$64,D0
> loop      5281                         ADDQ.L   #1,D1
> 07EAA7DE: 51C8 FFFC                    DBF      D0,loop
> 07EAA7E2: 6100 0010                    BSR      Sub1
```

Show the distance between subroutine 2 and subroutine 1 :

```
< d Sub2-Sub1 <enter>
> 00000018,24
```

You can do many other things with the 'symbol' command but 'symbol l' and 'symbol s' are sufficient at this moment.

There is still one thing we should do :

```
< loadfd exec fd:exec_lib.fd <enter>
```

With the loadfd command PowerVisor loads all the library function definitions in memory. That way PowerVisor will know how to show a library function when one is encountered while tracing. You do not have to load fd-files, but it is certainly very easy. I have the four big fd-files ('exec.library', 'graphics.library', 'intuition.library' and 'dos.library') permanently loaded in memory (I have put four 'loadfd' commands in the s:PowerVisor-startup file).

See the Using the tag system and fd-files section for more
information about fd-files and the VERY useful tag system in combination
with debugging (with this system PowerVisor will show names for offsets
in structures instead of numbers)!

Continue this session : Tracing
Go to session menu   : Two examples sessions

1.8 Debugging : Tracing

Now we can start tracing with trace :

```
< trace i <enter>
or
< tr i <enter>
> -----------------------------------------------------------------------
> D0: 00000001   D1: 01FAA9F5   D2: 00002EE0   D3: 07ED3A1C
> D4: 00000001   D5: 00000001   D6: 01FAA08F   D7: 07EAA7D4
> A0: 07ED3A1C   A1: 07EF9D28   A2: 07E0CEA4   A3: 07EAA7D4
> A4: 07EFCC00   A5: 00F906DE   A6: 00F906D2
> PC: 07EAA7D8   SP: 07EFCBFC   SR: 0010
> 00000000: 0000 0000                     ORI.B    #0,D0
>
> StartProgr7200                          MOVEQ.L  #0,D1
> 07EAA7DA: 7064                          MOVEQ.L  #$64,D0
> loop      5281                          ADDQ.L   #1,D1
> 07EAA7DE: 51C8 FFFC                     DBF      D0,loop
> 07EAA7E2: 6100 0010                     BSR      Sub1
```

(tr i : give ’I’nformation)
This command shows where we are. No actual tracing is done. The registers
are shown and the five first instructions. The program counter points to
the second instruction in this output. The first instruction is always
equal to the previous executed instruction. Initially it is initialized to
address 0. Note that you can change the format of this output with the
’prefs dmode’ and ’prefs debug’ commands (See the prefs command
and the Installing PowerVisor chapter in general).

Now we are really going to trace one instruction :

```
< tr <enter>
> -----------------------------------------------------------------------
> D0: 00000001   D1: 00000000   D2: 00002EE0   D3: 07ED3A1C
> D4: 00000001   D5: 00000001   D6: 01FAA08F   D7: 07EAA7D4
> A0: 07ED3A1C   A1: 07EF9D28   A2: 07E0CEA4   A3: 07EAA7D4
> A4: 07EFCC00   A5: 00F906DE   A6: 00F906D2
> PC: 07EAA7DA   SP: 07EFCBFC   SR: 0014
> StartProgr7200                          MOVEQ.L  #0,D1
>
> 07EAA7DA: 7064                          MOVEQ.L  #$64,D0
> loop      5281                          ADDQ.L   #1,D1
> 07EAA7DE: 51C8 FFFC                     DBF      D0,loop
> 07EAA7E2: 6100 0010                     BSR      Sub1
> 07EAA7E6: 6708                          BEQ      theend
```

```
In the register display you can see that 'd1' now has the value 0.
'StartProgr' is now the previous instruction. The programcounter now points
to the instruction 'moveq.l #$64,d0'.

Trace six instructions at once :

< tr n 6 <enter>
> ---------------------------------------------------------------------------
> D0: 00000062   D1: 00000003   D2: 00002EE0   D3: 07ED3A1C
> D4: 00000001   D5: 00000001   D6: 01FAA08F   D7: 07EAA7D4
> A0: 07ED3A1C   A1: 07EF9D28   A2: 07E0CEA4   A3: 07EAA7D4
> A4: 07EFCC00   A5: 00F906DE   A6: 00F906D2
> PC: 07EAA7DE   SP: 07EFCBFC   SR: 0000
> loop      5281                        ADDQ.L   #1,D1
>
> 07EAA7DE: 51C8 FFFC                   DBF      D0,loop
> 07EAA7E2: 6100 0010                   BSR      Sub1
> 07EAA7E6: 6708                        BEQ      theend
> 07EAA7E8: 6100 0022                   BSR      Sub2
> 07EAA7EC: 6100 0046                   BSR      Sub3

(tr n : trace 'N'umber instruction)
We are now in the loop.

To step over the loop we can use the following instruction :

< tr o <enter>
> ---------------------------------------------------------------------------
> D0: 0000FFFF   D1: 00000065   D2: 00002EE0   D3: 07ED3A1C
> D4: 00000001   D5: 00000001   D6: 01FAA08F   D7: 07EAA7D4
> A0: 07ED3A1C   A1: 07EF9D28   A2: 07E0CEA4   A3: 07EAA7D4
> A4: 07EFCC00   A5: 00F906DE   A6: 00F906D2
> PC: 07EAA7E2   SP: 07EFCBFC   SR: 0000
> 07EAA7DE: 51C8 FFFC                   DBF      D0,loop
>
> 07EAA7E2: 6100 0010                   BSR      Sub1
> 07EAA7E6: 6708                        BEQ      theend
> 07EAA7E8: 6100 0022                   BSR      Sub2
> 07EAA7EC: 6100 0046                   BSR      Sub3
> theend    7000                        MOVEQ.L  #0,D0
> Breakpoint...

(tr o : trace 'O'ver)
'tr o' places a breakpoint after the current instruction and then executes
until the breakpoint is encountered. You can trace over every instruction
with this command, but you can't use it in ROM-code since PowerVisor can't
put a breakpoint in ROM (don't worry ! there are solutions to this problem,
we will see them later on).

We step into the subroutine 'Sub1' with :

< tr <enter>
> ---------------------------------------------------------------------------
> D0: 0000FFFF   D1: 00000065   D2: 00002EE0   D3: 07ED3A1C
> D4: 00000001   D5: 00000001   D6: 01FAA08F   D7: 07EAA7D4
> A0: 07ED3A1C   A1: 07EF9D28   A2: 07E0CEA4   A3: 07EAA7D4
> A4: 07EFCC00   A5: 00F906DE   A6: 00F906D2
```

```
> PC: 07EAA7F4   SP: 07EFCBF8   SR: 0000
> 07EAA7E2: 6100 0010                    BSR      Sub1
>
> Sub1      203C 0000 0064               MOVE.L   #$64,D0
> 07EAA7FA: 7200                         MOVEQ.L  #0,D1
> 07EAA7FC: 2C78 0004                    MOVEA.L  (4),A6
> 07EAA800: 4EAE FF3A                    JSR      ($FF3A,A6)
> 07EAA804: 41FA 0042                    LEA      (Block,PC),A0
```

Trace another three instructions :

```
< tr n 3 <enter>
> -----------------------------------------------------------------------------
> D0: 00000064   D1: 00000000   D2: 00002EE0   D3: 07ED3A1C
> D4: 00000001   D5: 00000001   D6: 01FAA08F   D7: 07EAA7D4
> A0: 07ED3A1C   A1: 07EF9D28   A2: 07E0CEA4   A3: 07EAA7D4
> A4: 07EFCC00   A5: 00F906DE   A6: 07E007D8
> PC: 07EAA800   SP: 07EFCBF8   SR: 0004
> 07EAA7FC: 2C78 0004                    MOVEA.L  (4),A6
>
> 07EAA800: 4EAE FF3A                    JSR      (AllocMem,A6)
> 07EAA804: 41FA 0042                    LEA      (Block,PC),A0
> 07EAA808: 2080                         MOVE.L   D0,(A0)
> 07EAA80A: 4E75                         RTS
> Sub2      7000                         MOVEQ.L  #0,D0
```

Thanks to the loaded fd-file you can now see that this function is actually the Exec AllocMem. We do not want to run through the complete rom function so we trace over the call with :

```
< tr t <enter>
> -----------------------------------------------------------------------------
> D0: 07EFCE90   D1: 00002F48   D2: 00002EE0   D3: 07ED3A1C
> D4: 00000001   D5: 00000001   D6: 01FAA08F   D7: 07EAA7D4
> A0: 07E00000   A1: 07EFCE90   A2: 07E0CEA4   A3: 07EAA7D4
> A4: 07EFCC00   A5: 00F906DE   A6: 07E007D8
> PC: 07EAA804   SP: 07EFCBF8   SR: 0010
> 07EAA800: 4EAE FF3A                    JSR      ($FF3A,A6)
>
> 07EAA804: 41FA 0042                    LEA      (Block,PC),A0
> 07EAA808: 2080                         MOVE.L   D0,(A0)
> 07EAA80A: 4E75                         RTS
> Sub2      7000                         MOVEQ.L  #0,D0
> 07EAA80E: 7201                         MOVEQ.L  #1,D1
> Breakpoint...
```

(tr t : 'T'race over BSR or JSR   sorry, couldn't find a better character)
'tr t' looks similar to 'tr o'. The big difference is that 'tr t' works only for 'BSR' and 'JSR' instructions. And what is more important : 'tr t' works in ROM-code. If 'tr t' is used for an instruction other than 'BSR' or 'JSR' it is analogous to 'tr' (simple singlestep).

We can see that the AllocMem function had success (I hope this is really the case) because 'd0' contains the address of the newly allocated memory.

We continue tracing until the next change of program flow happens :

```
< tr b <enter>
> ----------------------------------------------------------------------------
> D0: 07EFCE90   D1: 00002F48   D2: 00002EE0   D3: 07ED3A1C
> D4: 00000001   D5: 00000001   D6: 01FAA08F   D7: 07EAA7D4
> A0: 07EAA848   A1: 07EFCE90   A2: 07E0CEA4   A3: 07EAA7D4
> A4: 07EFCC00   A5: 00F906DE   A6: 07E007D8
> PC: 07EAA80A   SP: 07EFCBF8   SR: 0010
> 07EAA808: 2080                          MOVE.L   D0,(A0)
>
> 07EAA80A: 4E75                          RTS
> Sub2      7000                          MOVEQ.L  #0,D0
> 07EAA80E: 7201                          MOVEQ.L  #1,D1
> 07EAA810: 7402                          MOVEQ.L  #2,D2
> 07EAA812: 7603                          MOVEQ.L  #3,D3

(tr b : trace until 'B'ranch)
'tr b' traces until a change of program control happens. This means that
tracing will stop always at the following instructions :
   JMP
   JSR
   BRA
   BSR
   RTE
   RTD
   RTR
   RTS
   TRAP
and tracing will stop at the following instructions if the brach would
succeed :
   Bcc
   DBcc

Go out this subroutine :

< tr <enter>
> ----------------------------------------------------------------------------
> D0: 07EFCE90   D1: 00002F48   D2: 00002EE0   D3: 07ED3A1C
> D4: 00000001   D5: 00000001   D6: 01FAA08F   D7: 07EAA7D4
> A0: 07EAA848   A1: 07EFCE90   A2: 07E0CEA4   A3: 07EAA7D4
> A4: 07EFCC00   A5: 00F906DE   A6: 07E007D8
> PC: 07EAA7E6   SP: 07EFCBFC   SR: 0010
> 07EAA80A: 4E75                          RTS
>
> 07EAA7E6: 6708                          BEQ      theend
> 07EAA7E8: 6100 0022                     BSR      Sub2
> 07EAA7EC: 6100 0046                     BSR      Sub3
> theend    7000                          MOVEQ.L  #0,D0
> 07EAA7F2: 4E75                          RTS

< tr <enter>
> ----------------------------------------------------------------------------
> D0: 07EFCE90   D1: 00002F48   D2: 00002EE0   D3: 07ED3A1C
> D4: 00000001   D5: 00000001   D6: 01FAA08F   D7: 07EAA7D4
> A0: 07EAA848   A1: 07EFCE90   A2: 07E0CEA4   A3: 07EAA7D4
> A4: 07EFCC00   A5: 00F906DE   A6: 07E007D8
> PC: 07EAA7E8   SP: 07EFCBFC   SR: 0010
```

```
> 07EAA7E6: 6708                         BEQ      theend
>
> 07EAA7E8: 6100 0022                    BSR      Sub2
> 07EAA7EC: 6100 0046                    BSR      Sub3
> theend    7000                         MOVEQ.L  #0,D0
> 07EAA7F2: 4E75                         RTS
> Sub1      203C 0000 0064               MOVE.L   #$64,D0
```

We suspect nothing bad in 'Sub2' so we simply trace over it :

```
< tr t <enter>
> ---------------------------------------------------------------------------
> D0: 00000000   D1: 00000001   D2: 00000002   D3: 00000003
> D4: 00000004   D5: 00000005   D6: 00000006   D7: 00000007
> A0: 07EFCE90   A1: 07EFCE90   A2: 07E0CEA4   A3: 07EAA7D4
> A4: 07EFCC00   A5: 00F906DE   A6: 07E007D8
> PC: 07EAA820   SP: 07EFCBF8   SR: 0010
> 07EAA7E8: 6100 0022                    BSR      Sub2
>
> 07EAA820: 4AFC                         ILLEGAL
> 07EAA822: 20C0                         MOVE.L   D0,(A0)+
> 07EAA824: 20C1                         MOVE.L   D1,(A0)+
> 07EAA826: 20C2                         MOVE.L   D2,(A0)+
> 07EAA828: 20C3                         MOVE.L   D3,(A0)+
> Illegal instruction !
```

There is something wrong ! This is called a resident breakpoint. You can put resident breakpoints in a program using the 'ILLEGAL' instruction. PowerVisor will automatically stop at such places (See
 Resident breakpoints for more info).

Skip over the instruction with :

```
< tr s <enter>
> ---------------------------------------------------------------------------
> D0: 00000000   D1: 00000001   D2: 00000002   D3: 00000003
> D4: 00000004   D5: 00000005   D6: 00000006   D7: 00000007
> A0: 07EFCE90   A1: 07EFCE90   A2: 07E0CEA4   A3: 07EAA7D4
> A4: 07EFCC00   A5: 00F906DE   A6: 07E007D8
> PC: 07EAA820   SP: 07EFCBF8   SR: 0010
> 07EAA820: 4AFC                         ILLEGAL
>
> 07EAA822: 20C0                         MOVE.L   D0,(A0)+
> 07EAA824: 20C1                         MOVE.L   D1,(A0)+
> 07EAA826: 20C2                         MOVE.L   D2,(A0)+
> 07EAA828: 20C3                         MOVE.L   D3,(A0)+
> 07EAA82A: 20C4                         MOVE.L   D4,(A0)+
```

('tr s' : 'S'kip instruction)
Now we have something special. Since we used the 'tr t' command to trace over the subroutine 'Sub2' we have created a breakpoint after the 'BSR Sub2' instruction. But if we would look after the 'BSR Sub2' instruction we would find no breakpoint (we will see later how PowerVisor shows breakpoints in the disassembly display). This is because the 'tr t' command works in a special way to make sure that you can use it in ROM-code too. Here follows an explanation of what has happened :

You typed ’tr t’ to skip ’BSR Sub2’ some time ago.
PowerVisor performs a ’tr’ to trace the ’BSR’ instruction.
Now the top of the stack contains the returnaddress for the ’BSR’ instruction, this is the address of the instruction after ’BSR Sub2’.
PowerVisor replaces the address on the stack with another address.
This address points to a private breakpoint. Since this private breakpoint is always in RAM, there is no problem setting this breakpoint. When the subroutine returns (with ’RTS’) later on (this has not happened at this moment), it will not return to the instruction after the ’BSR’ but to the breakpoint in RAM. PowerVisor will trap this and set the programcounter of the task to the right address: this is the instruction after the ’BSR Sub2’.

It would be different if you hade used ’tr o’ instead of ’tr t’.
’tr o’ would put a breakpoint directly after the ’BSR Sub2’. This will ofcourse not work if the ’BSR’ is in ROM since a breakpoint is in fact an ILLEGAL instruction.

But since the routine ’Sub2’ was interrupted (the ’ILLEGAL’ instruction caused this). The private breakpoint has not been encountered yet and the value on the stack is still the wrong value. We can make use of this feature and simply continue the ’tr t’ where it left of with :

```
< tr g <enter>
> -------------------------------------------------------------------------
> D0: 00000000   D1: 00000001   D2: 00000002   D3: 00000003
> D4: 00000004   D5: 00000005   D6: 00000006   D7: 00000007
> A0: 07EFCEB0   A1: 07EFCE90   A2: 07E0CEA4   A3: 07EAA7D4
> A4: 07EFCC00   A5: 00F906DE   A6: 07E007D8
> PC: 07EAA7EC   SP: 07EFCBFC   SR: 0010
> 07EAA822: 20C0                          MOVE.L   D0,(A0)+
>
> 07EAA7EC: 6100 0046                     BSR      Sub3
> theend    7000                          MOVEQ.L  #0,D0
> 07EAA7F2: 4E75                          RTS
> Sub1      203C 0000 0064                MOVE.L   #$64,D0
> 07EAA7FA: 7200                          MOVEQ.L  #0,D1
> Breakpoint...
```

(’tr g’ : trace ’G’o)
The ’tr g’ command simply executes the program until a breakpoint is encountered.
Note that it would make no difference if you would trace the program step by step. At one moment you would encounter the private breakpoint. Simply tracing over this breakpoint will return to the correct place in the program.

We step into ’Sub3’ :

```
< tr <enter>
> -------------------------------------------------------------------------
> D0: 00000000   D1: 00000001   D2: 00000002   D3: 00000003
> D4: 00000004   D5: 00000005   D6: 00000006   D7: 00000007
> A0: 07EFCEB0   A1: 07EFCE90   A2: 07E0CEA4   A3: 07EAA7D4
> A4: 07EFCC00   A5: 00F906DE   A6: 07E007D8
> PC: 07EAA834   SP: 07EFCBF8   SR: 0010
> 07EAA7EC: 6100 0046                     BSR      Sub3
```

```
>
> Sub3      203C 0000 0040                MOVE.L   #$40,D0
> 07EAA83A: 227A 000C                     MOVEA.L  (Block,PC),A1
> 07EAA83E: 2C78 0004                     MOVEA.L  (4),A6
> 07EAA842: 4EAE FF2E                     JSR      ($FF2E,A6)
> 07EAA846: 4E75                          RTS

< tr <enter>
> -----------------------------------------------------------------------------
> D0: 00000040   D1: 00000001   D2: 00000002   D3: 00000003
> D4: 00000004   D5: 00000005   D6: 00000006   D7: 00000007
> A0: 07EFCEB0   A1: 07EFCE90   A2: 07E0CEA4   A3: 07EAA7D4
> A4: 07EFCC00   A5: 00F906DE   A6: 07E007D8
> PC: 07EAA83A   SP: 07EFCBF8   SR: 0010
> Sub3      203C 0000 0040                MOVE.L   #$40,D0
>
> 07EAA83A: 227A 000C                     MOVEA.L  (Block,PC),A1
> 07EAA83E: 2C78 0004                     MOVEA.L  (4),A6
> 07EAA842: 4EAE FF2E                     JSR      ($FF2E,A6)
> 07EAA846: 4E75                          RTS
> Block     07EF CE90                     BSET     D3,($CE90,A7)
```

We see that something is wrong. We have allocated 100 bytes of memory ($64) but we are only going to free 64 bytes ($40). This is clearly a bug and should be fixed. But to prevent memory loss we are going to continue anyway. We simply change the 'd0' register :

```
< d @d0 <enter>
> 00000040 , 64

< @d0=100 <enter>
```

You see how we can look at registers and change their values.

We are not interested in the rest of the program. We simply let it go :

```
< tr g <enter>
> Program quits !
```

The program has stopped.

Some important 'trace' commands have been explained. There are a lot more. Some of the other 'trace' commands will be used in the following example. Refer to the documentation for trace for the other features.

Go to session menu   : Two examples sessions

1.9 Debugging : Starting the second session

We are now going to load the program using 'debug l' (see debug . Normally this is not the recommended way since this instruction does not perfectly emulate a Cli or WorkBench. But this does not matter for our little program. Note that the AmigaDOS 2.0 version of PowerVisor perfectly creates a CLI, so 'debug l' is a perfectly good way to load a program if you have AmigaDOS 2.0 and you want a CLI environment for your program.

```
< debug l examples/buggyprogram2 <enter>

The symbols are automatically loaded by 'debug l' :

< symbol s <enter>
> StartProgram                          : 07EADCC0 , 132832448
> Long                                  : 07EADCCE , 132832462
> recur                                 : 07EADCE0 , 132832480
> theend                                : 07EADCEC , 132832492

Open the fullscreen debugger display with  dwin
and  prefs  :

< dwin <enter>
< prefs dmode n <enter>

The 'prefs dmode' command is used to disable the output on the 'Main'
logical window you normally get after each trace. All the output goes
automatically to the 'Debug' logical window if it is open (but if you
set 'prefs dmode f' as it is default you will get output in the 'Debug'
logical window AND on the 'Main' logical window. This is probably not
as intended).

Drag the horizontal bar between the 'Main' logical window and the
'Debug' logical window so that all the five instructions of the disassembly
are visible.

The following keys can be used :

      <ctrl>+<NumPad Up>      (attempt) to scroll to the previous
                              instruction
      <ctrl>+<NumPad Down>    to scroll to the next instruction
      <ctrl>+<NumPad Left>    to decrease the top visible instruction
                              address with 2
      <ctrl>+<NumPad Right>   to increase this address with 2
      <ctrl>+<NumPad PgUp>    to decrease this address with 20
      <ctrl>+<NumPad PgDn>    to increase this address with 20
      <ctrl>+<NumPad 5>       to set this address equal to the program-
                              counter

Using these keys you can scroll through your code (try it).

Press :

< <ctrl>+<NumPad 5>

To go back to the programcounter.
(Note that you can also use the  dscroll  and  dstart  commands to scroll
through your program).

The fullscreen debugger display looks almost the same as the output from
the  trace  command in the earlier section. The differences are :

   - There is an indicator of what the task is doing.
         NONE     the task is waiting for PowerVisor instructions
         TRACE    the task is tracing
```

```
        FLOWT    the task is flow-tracing ('trace qf', 'trace rf',
                 'trace cf', ...) (only for 68020 or higher)
        ROUT     the task is in routine trace mode ('trace qr',
                 'trace rr', ...)
        EXEC     the task is executing

   - The top instruction (except for the previous instruction indicator)
     is not always equal to the instruction at the programcounter. The
     programcounter is indicated by the hilighted line.

   - The 'previous instruction' is only updated when the programcounter
     makes a jump out of the current displayed instructions.
```

Continue this session : Breakpoints
Go to session menu   : Two examples sessions

1.10 Debugging : Breakpoints

```
First a simple breakpoint :

Lets put a breakpoint in the 'Long' subroutine with  break  :

< u Long <enter>
> Long      7000                         MOVEQ.L  #0,D0
> 07EADCD0: 7201                         MOVEQ.L  #1,D1
> 07EADCD2: 7402                         MOVEQ.L  #2,D2
> 07EADCD4: 7603                         MOVEQ.L  #3,D3
> 07EADCD6: 7804                         MOVEQ.L  #4,D4
> 07EADCD8: 7A05                         MOVEQ.L  #5,D5
> 07EADCDA: 7C06                         MOVEQ.L  #6,D6
> 07EADCDC: 7E07                         MOVEQ.L  #7,D7
> 07EADCDE: 4E75                         RTS
> recur     5280                         ADDQ.L   #1,D0
> 07EADCE2: 0C80 0000 00C8               CMPI.L   #$C8,D0
> 07EADCE8: 6E02                         BGT      theend
> 07EADCEA: 61F4                         BSR      recur
> theend    4E75                         RTS
> 07EADCEE: 0000 07EA                    ORI.B    #$EA,D0
> 07EADCF2: DD08                         ADDX.B   -(A0),-(A6)
> 07EADCF4: 0000 0000                    ORI.B    #0,D0
> 07EADCF8: 07E2                         BSET     D3,-(A2)
> 07EADCFA: 68A0                         BVC      $7EADC9C
> 07EADCFC: 0002 004C                    ORI.B    #$4C,D2

< break n 07EADCD2 <enter>       (Note ! Use the equivalent address!)
or
< b n 07EADCD2 <enter>

> 00000001,1

('b n' : 'N'ormal breakpoint)
The output from this command is the breakpoint number. PowerVisor can have
as many breakpoints as memory permits. Breakpoints are always refered to
with their number.
```

With the info command you can now ask more information about the breakpoints :

```
< l dbug <enter>
> Debug task          : Node     Task     InitPC   TD ID Mode  SMode TMode
> ----------------------------------------------------------------------------
> examples/buggyprogra: 07EADB90 07ED5840 07EADCC0 FF FF NONE  WAIT  NORM

< info dbug:'examples/buggyprogram2' dbug <enter>
or
< i db:examp db <enter>

> Debug task          : Node     Task     InitPC   TD ID Mode  SMode TMode
> ----------------------------------------------------------------------------
> examples/buggyprogra: 07EADB90 07ED5840 07EADCC0 FF FF NONE  WAIT  NORM
>
> Node     Number Where    UsageCnt Type Condition
> ----------------------------------------------------------------------------
> 07EBA168     1  07EADCD2        0   N
```

We can see that there is one breakpoint defined with number 1 and position 07EA77DA. It has not been used yet and it is a normal (N) breakpoint. ('Condition' is explained later).

Lets have a look at the disassembly with unasm :

```
< u Long 20 <enter>
> Long      7000                            MOVEQ.L  #0,D0
> 07EADCD0: 7201                            MOVEQ.L  #1,D1
> 07EADCD2: 4AFC                            MOVEQ.L  #2,D2  >1
> 07EADCD4: 7603                            MOVEQ.L  #3,D3
> 07EADCD6: 7804                            MOVEQ.L  #4,D4
> 07EADCD8: 7A05                            MOVEQ.L  #5,D5
> 07EADCDA: 7C06                            MOVEQ.L  #6,D6
> 07EADCDC: 7E07                            MOVEQ.L  #7,D7
> 07EADCDE: 4E75                            RTS
> recur     5280                            ADDQ.L   #1,D0
> 07EADCE2: 0C80 0000 00C8                  CMPI.L   #$C8,D0
> 07EADCE8: 6E02                            BGT      theend
> 07EADCEA: 61F4                            BSR      recur
> theend    4E75                            RTS
> 07EADCEE: 0000 07EA                       ORI.B    #$EA,D0
> 07EADCF2: DD08                            ADDX.B   -(A0),-(A6)
> 07EADCF4: 0000 0000                       ORI.B    #0,D0
> 07EADCF8: 07E2                            BSET     D3,-(A2)
> 07EADCFA: 68A0                            BVC      $7EADC9C
> 07EADCFC: 0002 004C                       ORI.B    #$4C,D2
```

The breakpoint is the instruction with the '>1' appended.

Now we start the program and see where it ends with trace :

```
< tr g <enter>
> Breakpoint...
```

(Notice that we no longer get the complete output on 'Main'. All output is in the 'Debug' logical window)

```
The breakpoint has been encountered. Since it is a normal breakpoint it
is not removed.

< i db:examp db <enter>
> Debug task          : Node     Task     InitPC   TD ID Mode  SMode TMode
> ---------------------------------------------------------------------------
> examples/buggyprogra: 07EADB90 07ED5840 07EADCC0 FF FF NONE  WAIT  NORM
>
> Node     Number Where    UsageCnt Type Condition
> ---------------------------------------------------------------------------
> 07EBA168     1  07EADCD2        1   N

Now we see that the usage counter has incremented.

We make two new breakpoints :

< b t 07EADCDA <enter>

< b c recur '@d0==100' <enter>

< i db:exam db <enter>
> Debug task          : Node     Task     InitPC   TD ID Mode  SMode TMode
> ---------------------------------------------------------------------------
> examples/buggyprogra: 07EADB90 07ED5840 07EADCC0 FF FF NONE  WAIT  STEP
>
> Node     Number Where    UsageCnt Type Condition
> ---------------------------------------------------------------------------
> 07EBA288     3  07EADCE0        0   C  @d0==100
> 07EB5B60     2  07EADCDA        0   T
> 07EBA168     1  07EADCD2        1   N

('b t' : 'T'emporary breakpoint)
('b c' : 'C'onditional breakpoint)
'b t' makes a temporary breakpoint. This is a breakpoint that only breaks
once. 'b c' makes a conditional breakpoint. Conditional breakpoints are
very powerful as you will see in the following demonstration.

< tr g <enter>
> Breakpoint...

The breakpoint breaks and is immediately removed.

< tr g <enter>
> Breakpoint...

The conditional breakpoint breaks because 'd0' is equal to 100. A
conditional breakpoint is a very powerful way to control your program. The
breakpoint condition can be as complex as you wish (with the exception that
you can't use the group operator) and you can refer to all registers
like @pc, @sr, @sp, @d0 to @d7 and @a0 to @a7.

We remove the breakpoint with :

< b r 3 <enter>

('b r' : 'R'emove breakpoint)
```

Now we are going to put a breakpoint just after the 'BSR' instruction :

```
< u StartProgram <enter>
> StartProgr6100 000C                     BSR      Long
> 07EADCC4: 7000                          MOVEQ.L  #0,D0
> 07EADCC6: 6100 0018                     BSR      recur
> 07EADCCA: 7000                          MOVEQ.L  #0,D0
> 07EADCCC: 4E75                          RTS
> Long      7000                          MOVEQ.L  #0,D0
> 07EADCD0: 7201                          MOVEQ.L  #1,D1
> 07EADCD2: 4AFC                          MOVEQ.L  #2,D2  >1
> 07EADCD4: 7603                          MOVEQ.L  #3,D3
> 07EADCD6: 7804                          MOVEQ.L  #4,D4
> 07EADCD8: 7A05                          MOVEQ.L  #5,D5
> 07EADCDA: 7C06                          MOVEQ.L  #6,D6
> 07EADCDC: 7E07                          MOVEQ.L  #7,D7
> 07EADCDE: 4E75                          RTS
> recur     5280                          ADDQ.L   #1,D0
> 07EADCE2: 0C80 0000 00C8                CMPI.L   #$C8,D0
> 07EADCE8: 6E02                          BGT      theend
> 07EADCEA: 61F4                          BSR      recur
> theend    4E75                          RTS
> 07EADCEE: 0000 07EA                     ORI.B    #$EA,D0
```

We see that there is still another breakpoint present in the 'Long' subroutine. Remove it with :

```
< b r 1 <enter>
```

We make the new breakpoint :

```
< b n 07EADCCA <enter>
> 00000001,1
```

Now we execute until we reach that breakpoint :

```
< tr g <enter>
> Breakpoint...
```

And we start all over again by setting the programcounter back to the start of the program :

```
< @pc=StartProgram <enter>
```

Now we are ready to demonstrate yet another powerful feature which looks a bit like conditional breakpoints : conditional tracing.

```
< tr c '@d0==100' <enter>
```

('tr c' : 'C'onditional tracing)
'tr c' singlesteps the program until the condition is true. The difference with the conditional breakpoint is that the breakpoint only checks the condition when the breakpoint is passed. With conditional tracing the condition is checked after each instruction. Conditional tracing is of course much slower.

Note that for this simple expression you could have used the following
command :

< tr q '@d0==100' <enter>

('tr q' : 'Q'uick conditional tracing)
'tr q' works almost the same as 'tr c'. The difference is that it is faster (because it compiles the expression to machinecode) but you are more limited with the conditional expression. See the trace command for more info about this feature.

If you have a 68020 or higher you can also use the 'tr cf' or 'tr qf' commands. These commands are a lot faster but less accurate.

Remove the debug task from memory with the debug command :

< debug u <enter>

This command removes all breakpoints and unloads the program. It is best to always use this command in conjunction with 'debug l'. You can also use 'debug r' to remove all breakpoints and stop debugging. After 'debug r' the debug program will simply continue as if nothing has happened. This has two disadvantages : It is possible that the program is buggy and will crash. In that case it is not wise to use 'debug r'. PowerVisor will also not be able to unload the program from memory. This means that you will loose some memory (you == your Amiga :-)
'debug r' is more useful in conjunction with the 'debug n' command (and also with the 'debug c' command which can be used to catch a task).
You can also use 'debug f' (see the 'CommandRef' file for more info).

Close the debug logical window with :

< dwin <enter>

Go to session menu : Two examples sessions

1.11 Debugging : Some theory

When you issue a trace command to PowerVisor, the trace command will return immediately. This means that when the trace could take a long time, you will still be able to use PowerVisor for other commands. For example, when you are tracing conditionally, PowerVisor will do absolutely nothing. The debug task does everything until the condition becomes true. The debug task will then send a signal to PowerVisor and PowerVisor will update the debug display.

The conditional trace command is one of the trace commands that uses singlestep mode for tracing. This is slow but sometimes the only way to trace something. The 'go' trace command ('tr g') is another trace command. This trace command uses execute mode for tracing. The task runs at full speed until a breakpoint is encountered. It is possible that you want singlestep mode for the 'tr g' command too. For example, you could use this to see how a program runs. Since the program runs a bit slower you will be able to see much better what happens at each step. To use singlestep mode with the 'tr g' command you must use 'tr gt' ('t' for trace). Many tracing commands have these two versions.

If you have a 68020 or higher you can also use flow mode for tracing. In this mode the task is stopped everytime a change of programflow occurs. This is a lot faster (compared with singlestep mode) but less accurate because there are fewer samplepoints. In general this is not so bad. 'tr r', 'tr g', 'tr c' and 'tr q' can use this mode (append a 'f' after the command letter).

You can also use routine trace mode instead of singlestep mode or flow mode. In this mode PowerVisor will not leave the current routine. All instructions in the routine are singlestepped while BSR and JSR calls are executed at full speed. Append 'r' to the trace command if you want routine trace mode.

See the trace command for more info about all possible trace commands and modes.

Note that you can interrupt the tracing if you like with 'tr h' or 'tr f'.

Some commands (like 'tr u' and 'tr o' (explained later)) make a private breakpoint. A private breakpoint is a breakpoint with number 0. This breakpoint is automatically cleared when another breakpoint with number 0 is about to be created, or when the breakpoint breaks.

1.12 Debugging : Resident breakpoints

You can set resident breakpoints in your programs by including an 'ILLEGAL' instruction at the right place. When you want to use them you must make sure that PowerVisor is started and that you use 'mode patch' (see mode). Otherwise the results will not be very satisfactory. After that you simply start your program (from the 'Shell' or 'Workbench') (Note! Don't use 'debug n' in PowerVisor). When the program collides with the resident breakpoint, PowerVisor will trap the crash. You have now made a crash node. You can than use 'debug t' with the crash node or with the crashed task to start debugging at the 'ILLEGAL' instruction.

1.13 Debugging : The 'fdebug' command

To make life easier s/PowerVisor-startup defines an alias that you can use to initialize the fullscreen debugger. This alias uses the 'db' script to open the debug logical window and to initialize some keys. See the Alias Reference chapter for more information about the fdebug alias.

1.14 Debugging : Sourcelevel debugging

If you want you can load the source for the debug task you are tracing.
PowerVisor will automatically follow this source, even when you switch
to a routine in another file. See the source command
for more information.

1.15 Debugging : Using the PortPrint feature

You can use the powervisor.library in your own programs to put
several sorts of information on the PowerVisor screen. Note that
the output from these library functions appears on the master PowerVisor
screen (the slaves are ignored).

Look at 'pptest.asm' for an example.
The following library functions are available :

```
   PP_InitPortPrint()
      This function initializes the msgport for you. You need only
      do this once. The result you get in d0 is the pointer to the
      replyport (or null if no success). Use this pointer in all
      following commands.
   PP_StopPortPrint(a0)
      Clear the msgports for portprint. You need only do this once.
      a0 is the pointer to the replyport (the result from InitPortPrint).
   PP_ExecCommand(a0,a1,a2,d0)
      This routine is provided for the use of the  addfunc  command, but
      you are free to use it for your own purposes.
      a0 is the pointer to the replyport. a1 is a pointer to data (may
      be 0), a2 is a pointer to a commandstring that you want to execute.
      d0 is the size of the data (may be 0). When you call this routine
      PowerVisor will first make a copy of your data. PowerVisor will
      then execute the command (note ! PowerVisor will execute it, the
      calling task will only wait until PowerVisor is ready). The command
      that is executed will get the pointer to the copy of the data in
      the 'rc' variable. You can return a result from this command (using
      the  void  command for example). This result will be returned
      in d0.
   PP_DumpRegs(a0)
      Dump all registers on the PowerVisor screen.
      a0 is the pointer to the replyport.
   PP_Print(a0,a1)
      Print one line of text on the PowerVisor screen.
      a0 is the pointer to the replyport.
      a1 is the pointer to the text to print.
      Note that the replyport may be null for this call. Also note that
      this means that your program will not wait for PowerVisor to answer
      the message. This means that if you use this function again soon
      after the first call you will only see the results of the last call
   PP_PrintNum(a0,d0)
      Print a number on the PowerVisor screen.
      a0 is the pointer to the replyport.
      d0 is the number to print.
      Note that the replyport may be null for this call. Also note that
      this means that your program will not wait for PowerVisor to answer
      the message. This means that if you use this function again soon
```

```
      after the first call you will only see the results of the last call
   PP_SignalPowerVisor(a0,d0)
      Internal function to send a signal to PowerVisor. At this moment
      this function is only used by the memory protection system when a
      bus error occurs.
      a0 is the pointer to the replyport.
      d0 is the signal number (1 for bus error, 2 for bus error with
         freeze (not supported yet))
      Note that the replyport may be null for this call. Also note that
      this means that your program will not wait for PowerVisor to answer
      the message. This means that if you use this function again soon
      after the first call you will only see the results of the last call
```

1.16 Debugging : Using the tag system and fd-files

The disassembly used for the debug display (either with the 'trace i' command or with the fullscreen debugger) is a bit smarter than the normal disassembly with the unasm command.

First you have fd-files.
PowerVisor will use all loaded fd-files so that JSR and JMP instructions relative to a6 will be disassembled with the correct library name instead of a number. This feature makes debugging a LOT easier. Use the loadfd command to load fd-files.

The second feature makes use of the tag system and structures. In the Looking At Things chapter you can find all information about this system. Especially the addtag and struct commands are very useful. With these two commands you can add and define structures so that PowerVisor will be able to use the name of an offset in a structure instead of a number when disassembling for the debug task (it needs a debug task because the registers must have a contents so that PowerVisor knows to which structure the instruction points).

1.17 Debugging : Summary of all commands

Here follows a summary of what you can do with all debug commands :
(the following commands are used : break , debug ,
drefresh , dscroll , dstart , duse ,
dwin , symbol , source , trace , dnexti ,
dprevi and with)

```
   break n <address>    Set 'N'ormal breakpoint.
                        The breakpoint is not removed after breaking
   break t <address>    Set 'T'emporary breakpoint.
                        The breakpoint is removed after breaking
   break p <address>    Set 'P'rofile breakpoint
                        This breakpoint never breaks. It only increments
                        the usagecounter. You can use it to see if a
                        certain routine is much used
   break a <address> <timeout>
```

```
                     Break 'A'fter <timeout> passes.
                     The breakpoint is removed after breaking
break c <address> <condition>
                     'C'onditional breakpoint. This breakpoint breaks
                     when the condition is true. The breakpoint is not
                     removed after breaking
break r <breakpoint number>
                     Remove a breakpoint
debug n              Wait for 'N'ext prorgram
debug c              Wait for next task
debug l <filename>   'L'oad a program and load symbols
                     This command also creates a CLI structure if you
                     use the AmigaDOS 2.0 version of PowerVisor
debug t <task>|<crash node>
                     Take an existing task or crash node and make
                     a debug node for it. With this command you can
                     in theory debug any task in the system (be
                     careful though)
debug f              Remove the current debug node and freeze the
                     debug task
debug f <debug node> Remove the specified debug node and freeze the
                     corresponding task. Use this command if you are
                     debugging multiple programs at the same time.
                     You can find all debug nodes in the 'dbug' list
debug r              Remove the current debug node. The debug task
                     will continue executing at the programcounter
debug r <debug node> Remove the specified debug node
debug u              Remove the current debug node. The debug task
                     will be stopped and the program will be unloaded.
debug u <debug node> Same as 'debug u' but for a specified debug node.
debug d <name>       Create a dummy debug node with name <name>
                     You can't use this node for debugging but you
                     can use it to create symbols
debug q 0            Cleanup everything when the current debug task
                     quits. This is default
debug q 1            Prevent the current debug task from quiting.
                     This is useful in combination with the profiler
                     (see  prof ) and the resource tracker (see
                      track )
drefresh             Refresh the debug display
dscroll <offset>     Scroll <offset> bytes up in the fullscreen
                     debugger. Negative values are allowed. <offset>
                     will be made a multiple of two
dstart <address>     Set the start of the debug logical window
dnexti               Scroll to the next instruction
dprevi               Attempt to scroll to the previous instruction
duse <debug node>    Set the default debug node. This is useful when
                     you are debugging multiple tasks at the same time.
dwin                 Open/Close 'Debug' logical window
symbol l <filename> [<hunkaddress>]
                     Load the symbols for the current debug task.
                     If you give <hunkaddress>, PowerVisor will load the
                     symbols for the given hunks. This is extremely
                     useful when you have created a dummy debug task.
                     Note that <hunkaddress> is 4 more than the number
                     given in the hunklist with the  hunks  command.
                     Note that <hunkaddress> is not optional when you
```

```
                          are loading symbols for a dummy debug task.
   symbol c               Clear all symbols for the current debug node
   symbol t               Remove all temporary symbols for the current debug
                          node (temporary symbols start with a dot '.' or
                          contain only digits and end with a '$')
   symbol a <symbolname> <value>
                          Add a symbol to the list of symbols
   symbol r <symbolname>
                          Remove a symbol from the list of symbols
   symbol s               List all symbols for the current debug node
   source l <filename> [<hunkaddress>]
                          Load the source for the current debug task.
                          If you give <hunkaddress>, PowerVisor will load the
                          source for the given hunks. This is extremely
                          useful when you have created a dummy debug task.
                          Note that <hunkaddress> is 4 more than the number
                          given in the hunklist with the  hunks  command.
                          Note that <hunkaddress> is not optional when you
                          are loading the source for a dummy debug task
   source w <address>     Use this command to see in which source file and
                          on which line a specific address is located
   source t <tab size>    Set the tab size used for the source display. The
                          default tab value is 8
   source s               Show all sources for the current debug task
   source r               Redisplay the source in the 'Source' logical window
   source c               Clear all sources and unload them
   source g <line>        Move the source to a specific line
   source a <address>     Display the right source at the right linenumber
                          for <address>
   source h <hold mode>   Set (1) or unset (0) hold mode for the source
                          logical window. The source logical window will not
                          follow the program counter in hold mode
   trace                  Trace one instruction (singlestep mode)
   trace n <number>       Trace <number> instructions (singlestep mode)
   trace nr <number>      The same as 'trace n' but use routine trace mode
   trace nf <number>      Trace <number> changes of program flow. This
                          command uses flow mode (only 68020 or higher)
   trace b                Trace until the next change of program flow
                          (singlestep mode)
   trace t                Trace over JSR or BSR. IF the instruction is
                          not a BSR or JSR this command is analogous to
                          'trace' (execute mode)
   trace j                Trace until a library ROM function is about
                          to be called with JMP ...(a6) or JSR ...(a6).
                          (singlestep mode)
   trace jf               Like 'trace j' but use flow mode (68020 or higher
                          only)
   trace jr               Like 'trace r' but use routine trace mode
   trace r <register>     Trace until a specified register is changed.
                          Register can be d0-d7, a0-a6 or sp.
                          (singlestep mode)
   trace rf <register>    Same as 'trace r' but use flow mode (68020
                          or higher only).
                          (flow mode)
   trace rr <register>    Same as 'trace r' but use routine trace mode
   trace u <address>      Trace until programcounter is equal to <address>.
                          This command works by setting a private
```

```
                        breakpoint (number 0) at <address>. This command
                        only works when <address> is not in ROM
                        (execute mode)
   trace ut <address>   Trace until programcounter is equal to <address>.
                        No breakpoint is set by this command. <address>
                        can be in ROM
                        (singlestep mode)
   trace o              Trace over the current instruction. This command
                        is analogous to 'trace u' with <address> equal
                        to the instruction following the current
                        instruction
                        (execute mode)
   trace ot             Trace over the current instruction.
                        This version can be used in ROM
                        (singlestep mode)
   trace c <condition>  Trace until <condition> is true
                        (singlestep mode)
   trace cf <condition> Same as 'trace c' but use flow mode (68020
                        or higher only)
                        (flow mode)
   trace cr <condition> Same as 'trace c' but use routine trace mode
   trace q <condition>  Trace until <condition> is true
                        This command is faster (compared with 'trace c')
                        but the condition string is more limited (see
                         trace )
                        (singlestep mode)
   trace qf <condition> Same as 'trace q' but use flow mode (68020
                        or higher only)
                        (flow mode)
   trace qr <condition> Same as 'trace q' but use routine trace mode
   trace s              Skip instruction
   trace i              Do not trace. Show the current registers and
                        instructions (obsolete in the fullscreen debugger)
   trace g              Trace until a breakpoint is encountered (note that
                        all previous trace commands also stop when a
                        breakpoint is encountered)
                        (execute mode)
   trace gt             Trace until a breakpoint is encountered
                        (singlestep mode)
   trace gf             Same as 'trace g' but use flow mode (68020
                        or higher only)
                        (flow mode)
   trace gr             Same as 'trace g' but use routine trace mode
   trace p              Profile tracing
                        (singlestep mode)
   trace pf             Profile tracing (68020 or higher only)
                        (flow mode)
   trace pr             Same as 'trace p' but use routine trace mode
   trace z <adr> <len>  Trace until the checksum for the given range
                        changes. <adr> and <len> are converted to longword
                        alligned values
                        (singlestep mode)
   trace zr <adr> <len> Like 'trace z' but use routine trace mode
   trace zf <adr> <len> Like 'trace z' but use flow mode
   trace h              Interrupt the tracing or executing of the
                        current debug task
   trace f              Interrupt the tracing or executing of the
```

```
                     current debug task as soon as this task
                     is in ready state
with <debug node> <command>
                     Temporarily set the current debug node and execute
                     <command>. This is useful for example, if you are
                     debugging with multiple programs at the same time
                     and you want to have a look at the symbols or
                     registers of the other program
```