LookingAtThings.hyper

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

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

LookingAtThings.hyper

1.1 Looking At Things (Mon Nov 2 20:14:02 1992)

1.2 Looking At Things : Commands used in this tutorial

addstruct	Add structures to the 'stru' list
addtag	Add a tag to the current tag list
attc	The key attachment list
clearstruct	Clear all structures in the 'stru' list
cleartags	Clear all tags in the current tag list
conf	Autoconfig list
crsh	Crash list
dbug	Debug list
devs	Exec device list
dosd	Dos device list
exec	ExecBase structure list
fdfi	Fd file list
fils	File list
for	For each element in list execute a command
font	Font list

func	Function monitor list
gadgets	Show all gadgets in a window
graf	GraphicsBase structure list
hunks	Show all hunks for a process
ihan	Input handler list
info	Give information about an element in a list
interprete	Interprete some memory as a structure (from the 'stru' list)
intb	IntuitionBase structure list
libinfo	Ask information about a library function from a fd-file
intr	Interrupt list
libs	Library list
list	Show a list (tasks, libraries, message ports, ...)
llist	Traverse a list and show all elements in list
loadfd	Load a fd-file
loadtags	Load tags in the current tag list
lock	Lock list
lwin	Logical window list
memory	List memory
memr	Memory node list
mmuregs	Show all registers from mmu
mmureset	Reset the mmu tree
mmutree	Show the mmu tree
mode	Set PowerVisor preferences
moni	Monitor list
owner	Search owner of memory
pathname	Get pathname from lock
port	Message port list
print	Print a string
pubs	Public screen list
pwin	The physical window list
remstruct	Remove a structure from the 'stru' list
remtag	Remove a tag from the current tag list
resm	Resident module list
reso	Resource list
savetags	Save all tags in the current tag list
scrs	Screen list
sema	Semaphore list
specregs	Show all special 68020..68030 registers
speek	Peek from memory (using mmu)
spoke	Poke in memory (using mmu)
stru	Structure list
struct	Make and manage structure definitions
tags	Show all tags in the current tag list
task	Task and process list
tg	Temporarily set another tag list as current
unasm	Disassemble memory
usetag	Set another current tag list number
view	List memory while using tags to determine the type of memory
wins	Window list

1.3 Looking At Things : Functions used in this tutorial

apeek	Peek address from structure
base	Get the first element in the current list
curlist	Get the current list

lastmem	Get last memory used by 'memory', 'unasm' or 'view'
peek	Peek element from structure
stsize	Ask the size of a structure definition
taglist	Ask the current tag list number

1.4 Looking At Things : Introduction

PowerVisor can display memory, disassemble instructions, show structures, give information, ... In short, PowerVisor has a lot of instruction to SHOW you something. This tutorial file describes all these commands. All commands in this tutorial give you information of some kind.

1.5 Looking At Things : Simple memory viewing

The simplest way to look at memory is using the memory command. Simply try it :

```
< memory 0 100 <enter>
```

or

```
< m 0 100 <enter>

> 00000000: 00000000 07E007CC 00F80834 00F80B16              ...........4....
> 00000010: 00F80ADA 00F80ADC 00F80ADE 00F80AE0              ................
> 00000020: 00F80C00 00F80AE4 00F80AE7 00F80AE8              ................
> 00000030: 00F80AEA 00F80AEC 00F80AEE 00F80AF0              ................
> 00000040: 00F80AF2 00F80AF4 00F80AF6 00F80AF8              ................
> 00000050: 00F80AFA 00F80AFC 00F80AFE 00F80B00              ................
> 00000060: 00F80B02                                         ....
```

You will now see 100 bytes or 25 longwords of memory starting on location 0.

You can also use :

```
< m 0 <enter>
> 00000000: 00000000 07E007CC 00F80834 00F80B16              ...........4....
> 00000010: 00F80ADA 00F80ADC 00F80ADE 00F80AE0              ................
> 00000020: 00F80C00 00F80AE4 00F80AE7 00F80AE8              ................
> 00000030: 00F80AEA 00F80AEC 00F80AEE 00F80AF0              ................
> 00000040: 00F80AF2 00F80AF4 00F80AF6 00F80AF8              ................
> 00000050: 00F80AFA 00F80AFC 00F80AFE 00F80B00              ................
> 00000060: 00F80B02 00F810F4 00F81152 00F81188              ...........R....
> 00000070: 00F811E6 00F8127C 00F812C6 00F81310              .......|........
> 00000080: 00F80B70 00F80B72 00F80B74 00F80B76              ...p...r...t...v
> 00000090: 00F80B78 00F80B7A 00F80B7C 00F80B7E              ...x...z...|...~
> 000000A0: 00F80B80 00F80B82 00F80B84 00F80B86              ................
> 000000B0: 00F80B88 00F80B8A 00F80B8C 00F80B8E              ................
> 000000C0: 00F80B90 00F80B92 00F80B94 00F80B96              ................
> 000000D0: 00F80B98 00F80B9A 00F80B9C 00F80B9E              ................
> 000000E0: 00F80BA0 00F80BA2 00F80BA4 00F80BA6              ................
> 000000F0: 00F80BA8 00F80BAA 00F80BAC 00F80BAE             ................
> 00000100: 0030A6FC 00006600 00000610 00000000             .0....f.........
> 00000110: 200066FB 00006600 0000A600 203066FA              .f...f..... 0f.
> 00000120: 003026FB 00006600 00006630 005066F7             .0&...f...f0.Pf.
> 00000130: 001066FB 00006600 00006600 001026F7             ..f...f...f...&.
```

So you don’t have to specify the number of bytes to print. The default number is 320 (Note that PowerVisor remembers the last number of bytes used, so the default number is actually equal to that number)

If you prefer the output of this command in another format, you can do this with the mode command :

```
< mode byte <enter>
< m 0 100 <enter>
> 00000000: 00 00 00 00 07 E0 07 CC 00 F8 08 34 00 F8 0B 16  ...........4....
> 00000010: 00 F8 0A DA 00 F8 0A DC 00 F8 0A DE 00 F8 0A E0  ................
> 00000020: 00 F8 0C 00 00 F8 0A E4 00 F8 0A E7 00 F8 0A E8  ................
> 00000030: 00 F8 0A EA 00 F8 0A EC 00 F8 0A EE 00 F8 0A F0  ................
> 00000040: 00 F8 0A F2 00 F8 0A F4 00 F8 0A F6 00 F8 0A F8  ................
> 00000050: 00 F8 0A FA 00 F8 0A FC 00 F8 0A FE 00 F8 0B 00  ................
> 00000060: 00 F8 0B 02                                      ....
```

Or back to normal with :

```
< mode long <enter>
```

Other formats are: ’mode word’ or ’mode ascii’.

Pressing enter with an empty commandline will cause the memory listing to continue (if the last command was a ’memory’ command). (This is also the case if the last command was a view or an unasm (see later)).

Typing ’memory’ with no arguments will also cause a continued memory listing.

You can use the lastmem() function to see where Powervisor will continue the memory listing :

```
< disp lastmem() <enter>
> 00000064 , 100
```

Note that when you are debugging a program, this command will also show the 9 first characters of a symbol when there is one on some address.

1.6 Looking At Things : Disassembling memory

If you want to disassemble memory, you can use the unasm command. This command disassembles 68000, 68010, 68020, 68030, 68040, 68881, 68882 and 68851 code.

```
< unasm 0 <enter>
```

or

```
< u 0 <enter>

> 00000000: 0000 0000                     ORI.B    #0,D0
> 00000004: 07E0                          BSET     D3,-(A0)
> 00000006: 07CC 00F8                     MOVEP.L  D3,($F8,A4)
> 0000000A: 0834 00F8 0B16 00F8           BTST     #$F8,([A0],D0.L*2,$F8)
> 00000012: 0ADA 00F8                     CAS      D0,D3,(A2)+
> 00000016: 0ADC 00F8                     CAS      D0,D3,(A4)+
> 0000001A: 0ADE 00F8                     CAS      D0,D3,(A6)+
> 0000001E: 0AE0 00F8                     CAS      D0,D3,-(A0)
> 00000022: 0C00 00F8                     CMPI.B   #$F8,D0
> 00000026: 0AE4 00F8                     CAS      D0,D3,-(A4)
> 0000002A: 0AE7 00F8                     CAS      D0,D3,-(A7)
> 0000002E: 0AE8 00F8 0AEA                CAS      D0,D3,($AEA,A0)
> 00000034: 00F8 0AEC                     ORI.?    #$F8,($AEC)
> 00000038: 00F8 0AEE                     ORI.?    #$F8,($AEE)
> 0000003C: 00F8 0AF0                     ORI.?    #$F8,($AF0)
> 00000040: 00F8 0AF2                     ORI.?    #$F8,($AF2)
> 00000044: 00F8 0AF4                     ORI.?    #$F8,($AF4)
> 00000048: 00F8 0AF6                     ORI.?    #$F8,($AF6)
> 0000004C: 00F8 0AF8                     ORI.?    #$F8,($AF8)
> 00000050: 00F8 0AFA                     ORI.?    #$F8,($AFA)
```

The default number of instructions to disassemble is 20, but you can choose another number after the address.

```
< u 0 5 <enter>
> 00000000: 0000 0000                     ORI.B    #0,D0
> 00000004: 07E0                          BSET     D3,-(A0)
> 00000006: 07CC 00F8                     MOVEP.L  D3,($F8,A4)
> 0000000A: 0834 00F8 0B16 00F8           BTST     #$F8,([A0],D0.L*2,$F8)
> 00000012: 0ADA 00F8                     CAS      D0,D3,(A2)+
```

If you do not like the words in this output you can disable them with the mode command :

```
< mode no shex <enter>

< u 0 5 <enter>
> 00000000:                               ORI.B    #0,D0
> 00000004:                               BSET     D3,-(A0)
> 00000006:                               MOVEP.L  D3,($F8,A4)
> 0000000A:                               BTST     #$F8,([A0],D0.L*2,$F8)
> 00000012:                               CAS      D0,D3,(A2)+
```

Or enable them :

```
< mode shex <enter>
```

When you are debugging a program, this command shows all labels and symbols present in this program (Therefor it can be useful to disable the words in the output, that way PowerVisor can show longer labels).

Note that the disassembler used by the debugger is a lot smarter than the normal disassembler with the ’unasm’ command. The disassembler makes use of all loaded fd-files so that library functions are disassembled with their name instead of their number. In combination with the tag system and structures PowerVisor will even show names for structures offsets instead of numbers (see Debugging for more info).

Pressing enter with an empty commandline will cause the disassembly to continue (if the last command was a 'unasm' command).

Typing 'unasm' with no arguments will also cause a continued disassembly.

1.7 Looking At Things : Listing things

You can also list a lot of things in PowerVisor. The current list concept was already explained in the Getting Started chapter. I assume you have read that chapter.

The following lists are available at this moment :
(All lists with a '*' have more information in the AmigaDOS 2.0 version, this (extra) information can be viewed with the info command or the list command (the 'info' command also works on the AmigaDOS 1.3 version but gives less information))

Big structures :	
Exec *	the listing of the ExecBase structure
Intb	IntuitionBase structure
Graf *	Graphics base structure
Exec/dos/graphics and intuition things :	
Task *	The listing of the tasks in the system (default list)
Libs	Exec-Libraries
Devs	Exec-devices
Reso	Exec-Resources
INTR	Exec-Interrupts
Memr	Memory list
Port	Message ports
Wins *	All windows
Scrs	Screens
Font	Fonts currently in memory
DOsd	Dos-devices
SEma	Semaphores
RESM	Resident modules
FIls	Open files
LOck	Locks
IHan	Input handlers
COnf	AutoConfigs
MOni *	Monitors (AmigaDOS 2.0 only)
PUbs *	Public Screens (AmigaDOS 2.0 only)
PowerVisor things :	
FUnc	All Function monitor nodes (see addfunc command)
FDfi	All fdfiles loaded (see loadfd command)
Attc	All attached keys (see attach command)
Crsh	All crashed programs
DBug	All debug nodes (see the Debugging chapter)
STru	All structure defines (see addstruct command)
LWin	All logical windows for PowerVisor
PWin	All physical windows for PowerVisor

Some examples :

```
< list exec <enter>
> SoftVer      : 012F     | LowMemChkSum : 0000     | ChkBase      : F81FF833
> ColdCapture  : 00000000 | CoolCapture  : 00000000 | WarmCapture  : 00000000
> SysStkUpper  : 07E02230 | SysStkLower  : 07E00A30 | MaxLocMem    : 00200000
> DebugEntry   : 00F82E88 | DebugData    : 00000000 | AlertData    : 00000000
> MaxExtMem    : 00000000 | ChkSum       : A2BE     | ThisTask     : 07EA0B08
> IdleCount    : 000045BE | DispCount    : 00005039 | Quantum      : 0004
> Elapsed      : 0004     | SysFlags     : 0000     | IDNestCnt    : FF
> TDNestCnt    : FF       | AttnFlags    : 0017     | AttnResched  : 0000
> ResModules   : 07E00410 | TaskTrapCode : 07EA6924 | TaskExceptCod: 00F83AEC
> TaskExitCode : 00F8242C | TaskSigAlloc : 0000FFFF | TaskTrapAlloc: 8000
> VBlankFreq   : 32       | PowerSupplyFr: 32       | KickTagPtr   : 00000000
> KickCheckSum : 00000000 | RamLibPrivate: 07E1E528 | EClockFreq   : 000AD303
> CacheCtrl    : 00002919 | TaskID       : 00000001 | PuddleSize   : 00000000
> MMULock      : 00000000 |
```

See the Expressions chapter for what you can do with the ':' operator (the list operator) for this list and the two other lists : 'graf' and 'intb'. Note that the '&' unary operator can only be used with these three lists. The ':' operator can be used for almost any list except 'lock' and 'file'.

```
< list wins <enter>
> Window name          : Address  Left  Top Width Height WScreen
> ----------------------------------------------------------------------------
>                      : 07EA7568    0   12   692    430 07EA6760
>                      : 07E45E38    0    0   704    456 07E46110
> My Shell             : 07E1FD48    0  568   692    456 07E2D258
>                      : 07E3B398    0   16   692   1008 07E2D258
```

You can use the curlist() function to see in which list we are. This function returns a pointer to the curlist string (in ARexx this function returns a string). You can use the print command to look at the current list :

```
< print \(curlist(),%s)\0a
> task
```

(Since there is no newline in the current list string, there will be no newline printed on the screen).

If you want the pointer to the first element in the list you can use the base() function :

```
< disp base() <enter>
> 07E28330 , 132285232
```

When you want to execute a specific command on each element in a list, you can use the for command. This command is especially useful when using tags (see The tag system and 'view').
The command you give as an argument to the 'for' command is executed once for each element in the list. The command can find the pointer to the element in the list in the 'rc' variable.

For example, to display all elements in a list :

```
< for task disp rc <enter>
> 07E28330 , 132285232
> 07E51458 , 132453464
> 07E5B258 , 132493912
> 07E609A8 , 132516264
> 07E53F28 , 132464424
> 07E1E6F0 , 132245232
> 07E1EFE0 , 132247520
> 07E51DC8 , 132455880
> 07E0D992 , 132176274
> 07E43418 , 132396056
> 07E6E5C8 , 132572616
> 07EA8348 , 132809544
> 07E0A7C0 , 132163520
> 07E0A428 , 132162600
> 07E104E8 , 132187368
> 07E16278 , 132211320
> 07E189B0 , 132221360
> 07E34200 , 132334080
> 07E0F1B4 , 132182452
> 07E08B22 , 132156194
> 07E23BF8 , 132267000
> 07EA9648 , 132814408
```

More information about each list can be found in the List Reference chapter. In that chapter you can also find all the variables printed by the info command.
(Also see Asking more 'info' about something).

1.8 Looking At Things : Asking more 'info' about something

If you want more information about something that is in a list, you can use the info command :

Make the window list current :

```
< wins <enter>

< list <enter>
> Window name          : Address  Left  Top Width Height WScreen
> ---------------------------------------------------------------------------
>                      : 07EA7568    0   12   692    430 07EA6760
>                      : 07E45E38    0    0   704    456 07E46110
> My Shell             : 07E1FD48    0  568   692    456 07E2D258
>                      : 07E3B398    0   16   692   1008 07E2D258
```

You can now ask more info about 'My Shell' for example :

```
< info my <enter>

> Window name          : Address  Left  Top Width Height WScreen
> ---------------------------------------------------------------------------
> My Shell            : 07E1FD48    0  568   692    456 07E2D258
>
> MinWidth     : 0050     | MinHeight    : 0032     | MaxWidth     : FFFF
> MaxHeight    : FFFF     | Flags        : 2800104F | MenuStrip    : 00000000
> ScreenTitle  : Workbench Screen
> FirstReques  : 00000000 | DMRequest    : 00000000 | ReqCount     : 0000
> RPort        : 07E20068 | Pointer      : 00000000 | PtrHeight    : 00
> PtrWidth     : 00       | XOffset      : 00       | YOffset      : 00
> IDCMPFlags   : 00000000 | UserPort     : 00000000 | WindowPort   : 00000000
> MessageKey   : 00000000 | DetailPen    : 00       | BlockPen     : 01
> CheckMark    : 07E0B960 | ExtData      : 00000000 | UserData     : 00000000
> BorderLeft   : 04       | BorderTop    : 10       | BorderRight  : 12
> BorderBottom : 02       | BorderRPort  : 00000000 | Parent       : 07E3B398
> Descendant   : 07EA7568 | GZZMouseX    : 005D     | GZZMouseY    : 00D6
> GZZWidth     : 029E     | GZZHeight    : 01B6     | IFont        : 07E083F0
> MoreFlags    : 00000000 |
>
> Flags: WINDOWSIZING WINDOWDRAG WINDOWDEPTH WINDOWCLOSE SIMPLEREFRESH ACTIVATE
> VISITOR HASZOOM
> IDCMP:
```

You get a lot of information. Basically this is the window structure.

Not all lists have that much extra information. Some lists give no extra information at all. Only the header is dumped.

IMPORTANT ! If 'wins' wasn't the current list you should ask information as follows :

First go to another current list :

```
< task <enter>
```

Ask information about 'My Shell' in the window list.

```
< info wins:my wins <enter>
> ...
```

Especially the last 'wins' argument is very important. If you omit it PowerVisor will try to interprete the 'My Shell' window as a task or process. This can cause crashes. In general it is safest to always supply this extra argument. You may add it to the command even if the current list is already good.

You must also be careful using name expansion (don't type this) :

```
< info my wins <enter>
```

will NOT work when 'wins' is not the current list. This command can even crash. What happens is that PowerVisor first searches the current list for something that starts with 'my'. If you are so unlucky to really have a task starting with 'my' PowerVisor will then try to interprete that task as a window.

So you should really be careful when you use the 'info' command. Nasty things can happen when you are not careful enough about the current list and the arguments you give to 'info'. If you are cautious however, the 'info' command is really useful and can safe you lots of debugging time.

Using the for command, you can ask information about all items in a list.

For example, to dump info about each task in the system to a file (not to the screen), use :

```
< to ram:Info -for task {info rc task;print \0a\0a} <enter>
```

This is a rather complex example. I will explain it in detail.

The to command redirects the output of the following command to the file 'ram:Info' (see the Screens and Windows chapter for more info about the 'to' command).

The for command is the command whose output is redirected (it is an argument for the 'to' command). Because there is a '-' in front of the 'for' no output is printed on the PowerVisor window.

The 'for' command executes the following command for each element in the 'task' list.

The command that is executed for each element in the task list is a group of commands.

The first command in this group is the info command. Its argument is the 'rc' variable which contains the pointer to the element currently processed by the 'for' command. We add the 'task' argument since we could as well execute this command with another current list.

The second command in this group is the print command. This command prints two newlines after each info block.

Since the 'for' command remembers all output in memory and only starts printing after the list is traversed, you need not worry about the list becoming corrupt after a long time (This is especially true for the task list since this is a very busy list).

You could also have typed :

```
< -to ram:Info for task {info rc task;print \0a\0a} <enter>
```

But not :

```
< to ram:Info for task -{info rc task;print \0a\0a} <enter>
```

Since the 'for' command remembers all output even if the output is hidden.

1.9 Looking At Things : Viewing structures

PowerVisor also has the ability to view structures. These structure are defined in ascii files and converted to 'pvsd'-files (PowerVisor Structure Definition files) by the 'MStruct' utility (see Making structures (MStruct) for more information about how to make structures). These ascii files look a bit like machinelanguage include files (see the examples in the Source subdirectory). You can also use the struct command to make structures from within PowerVisor.

Note that structures can also be used for debugging. In combination with the tag system (see The tag system and 'view') the debug disassembly (either the fullscreen debugger or the 'trace i' output) will automatically use names for the offsets in a structure instead of numbers (see Debugging for more info).

On the PowerVisor disk there is a file called 'Exec.pvsd'. This file contains all definitions for the structures in Exec 2.0. You can load all structures from this file using the addstruct command :

```
< addstruct exec.pvsd <enter>
> UNIT
> IS
> IV
> IO
> IOSTD
> LIB
> LH
> MLH
> ML
> ME
> MH
> MC
> LN
> MLN
> MP
> MN
> RT
> SSR
> SS
> SM
> TC
> ETask
> StackSwapStruct
```

This command adds all structures to the 'stru' list. You can list this list to see all structures in memory :

```
< list stru <enter>
> Struct node name    : Node     Pri InfoBlock Strings  Length
> ------------------------------------------------------------------------
> IS                  : 07EBA5B0 FD  07EBA5D2  07E5A4D2     22
> IV                  : 07EBA5F0 FD  07EBA612  07EBA63A     12
> IO                  : 07EBA650 FD  07EBA672  07EBA6AA     32
> LH                  : 07EBA888 FD  07EBA8AA  07EBA8E2     14
> ML                  : 07EBA948 FD  07EBA96A  07E622AA     16
> ME                  : 07EBA980 FD  07EBA9A2  07E622CA      8
> MH                  : 07EBA9C0 FD  07EBA9E2  07EBAA1A     32
> MC                  : 07EBAA50 FD  07EBAA72  07E706EA      8
> LN                  : 07EBAA90 FD  07EBAAB2  07EBAAEA     14
> MP                  : 07EBAB48 FD  07EBAB6A  07EBAB9A     34
> MN                  : 07EBABB8 FD  07EBABDA  07EBABFA     20
> RT                  : 07EBAC10 FD  07EBAC32  07EBAC92     26
> SS                  : 07EBAD10 FD  07EBAD32  07EBAD6A     46
> SM                  : 07EBADA0 FD  07EBADC2  07E5A602     36
> TC                  : 07EBADD8 FD  07EBADFA  07EBAE9A     84
> LIB                 : 07EBA788 FC  07EBA7EA  07EBA842     34
> MLH                 : 07EBA900 FC  07EBA922  07EBA7AA     12
> MLN                 : 07EBAB08 FC  07EBAB2A  07E73452      8
> SSR                 : 07EBACD8 FC  07EBACFA  07E761FA     12
> _cli                : 07EB97D8 FB  0008055C  00000000      0
> UNIT                : 07E5A5A8 FB  07EBA53A  07EB7BCA     38
> _exec               : 07E805C8 FA  00080C9E  00000000      0
> _intb               : 07E836E0 FA  00080E30  00000000      0
> _libs               : 07E86598 FA  000804EC  00000000      0
> _devs               : 07E86990 FA  000804EC  00000000      0
> _reso               : 07E86BB0 FA  000804EC  00000000      0
> _font               : 07E86BD0 FA  0008050C  00000000      0
> _graf               : 07E8A478 FA  0008111E  00000000      0
> _pubs               : 07E9A7D8 FA  000802A4  00000000      0
> _moni               : 07EB6D00 FA  000802BC  00000000      0
> _lwin               : 07EB8620 FA  00080404  00000000      0
> _pwin               : 07EB8ED8 FA  0008049C  00000000      0
> _wins               : 07EB8EF8 FA  0008074C  00000000      0
> _scrs               : 07EB90F8 FA  00080694  00000000      0
> _proc               : 07EB97B8 FA  000805DC  00000000      0
> _task               : 07EB97F8 FA  00080374  00000000      0
> _conf               : 07EB9838 FA  00080884  00000000      0
> IOSTD               : 07EBA6D0 FA  07EBA6F2  07EBA74A     48
> ETask               : 07EBAF38 FA  07EBAF5A  07EBAFAA     86
> _ioreq              : 07EB9818 F9  00080214  00000000      0
> StackSwapStruct     : 07EBAFF8 F0  07EBB032  07EBB05A     12
```

Note that all structure definitions starting with an underscore ('_') are standard structure definitions which are always defined. They all correspond with the structures you get to see with the info command.

You can then use the remstruct and clearstruct commands to remove one structure or all structures from memory.

Now we interprete an element of the task list as a task with the interprete command :

```
< task <enter>

< list task <enter>
> Task node name      : Node     Pri   StackU   StackS Stat Command         Acc
> -----------------------------------------------------------------------------
> ConClip Process     : 07E60410 00       242     4000 Wait sys:c/ConCl(02) -
> RexxMaster          : 07E6AA48 04       162     2048 Wait            (00) -
> ...
> trackdisk.device    : 07E0E714 05        98      512 Wait            TASK -
> input.device        : 07E07F12 14        86     4096 Wait            TASK -
> RAM                 : 07E31220 0A       678     1200 Wait            PROC -
> pv                  : 07F62FC0 04       438    16000 Run  pv         (01) -
```

```
< interprete 'input' tc <enter>
> Pri          : 14       |
> Name         : input.device
> Flags        : 00       | State        : 04       | IDNestCnt    : FF       |
> TDNestCnt    : 00       | SigAlloc     : C000FFFF | SigWait      : C0000000 |
> SigRecvd     : 00000004 | SigExcept    : 00000000 | ETask        : 80000000 |
> ExceptData   : 00000000 | ExceptCode   : 00F83068 | TrapData     : 00000000 |
> TrapCode     : 00F83068 | SPReg        : 07E08F1A | SPLower      : 07E07F70 |
> SPUpper      : 07E08F70 | Switch       : 00000000 | Launch       : 00000000 |
> MemEntry     : 07E07F5C | Userdata     : 00000000 |
```

This command dumps the structure defined in the 'stru' list. ('tc' is the task structure).

You can also peek a certain value from this list with the peek() function :

```
< disp peek('input',tc,spupper) <enter>
> 07E08F70 , 132157296
```

Or you can change a value (do not execute this command!) with apeek() :

```
< *apeek('input',tc,spupper).l=5 <enter>
```

You can use the stsize function to ask the size of a structure :

```
< d stsize(ln) <enter>
> 0000000E , 14
```

Structure definitions can also be used with the view command.
(Also see The tag system and 'view').

1.10 Looking At Things : Making structures (MStruct)

In the previous section Viewing structures we saw how you can interprete some region of memory as a structure and look at the contents of all fields. In this section we will see how you can make your own structures using the 'MStruct' utility and the struct command.

'MStruct' takes an input file containing the description of the structure and converts this input file to a PVSD (PowerVisor Structure Definition) file. The addstruct command reads all structures from PVSD files.
The input file looks a bit like a machinelanguage include file. The best way to see how you can use 'MStruct' is to look at the example input files in the 'Source' subdirectory ('intuition.struct' and 'exec.struct'). These contain the definition for the most common structures in Intuition and Exec respectively.

Each input file may contain as many structures as you wish. Each structure starts with the following line (note that case for the keywords is not important) :

```
  STRUCTURE <structure name>,<offset to skip>
```

This line gives the name of the structure that you will later use in PowerVisor. <offset to skip> gives the number of bytes to skip before the real fields in the structure start.

There is no explicit command to end a structure. The start of a new structure will end the definition of the previous one.

The following keywords are also supported and correspond with the standard Amiga types :

```
  APTR      <fieldname>
  BPTR      <fieldname>
  CHAR      <fieldname>
  BYTE      <fieldname>
  WORD      <fieldname>
  LONG      <fieldname>
  UBYTE     <fieldname>
  UWORD     <fieldname>
  ULONG     <fieldname>
  CSTR      <fieldname>
  BSTR      <fieldname>
  STRUCT    <fieldname>,<size of structure>
```

The following keywords can be used to skip some space in the structure. These commands do not cause an entry in the definition of the structure (they only advance the offset in the structure) so <fieldname> is ignored :

```
  PADBYTE   <fieldname>
  PADWORD   <fieldname>
  PADLONG   <fieldname>
  PADSTRUCT <fieldname>,<size of structure>
```

All illegal keywords are ignored by the utility (also the ’LABEL’ keyword, which is used in the example files).

Instead of the ’mstruct’ utility you can also use the struct command. With this command you can make and change structures in an interactive way. This command is VERY useful in combination with debugging and the addtag command.
You can also save structures to a file with the ’struct’ command.

1.11 Looking At Things : The tag system and ’view’

The most powerful command to view memory is the view command. This command uses tags. A tag is a definition for a range of memory. Using tags you can define a region of memory to be code, or full ascii, ... The ’view’ command displays all memory according to its type.

In combination with structures (see Viewing structures), this command has even more power (see Using tags and structures).

Note that the tag system is also used by the memory protection system (see watch).

When you first start PowerVisor the 'view' command works exactly like the memory command. This is because the default memory type for all memory that is not defined by a tag is Long/Ascii.

Lets explain all this with an example :

First we define the memory starting on location 0 as a range of longwords with the addtag command :

```
< addtag 0 50 la <enter>
```

This 'addtag' command adds a definition for a range of memory. A memory range with 50 bytes starting from address 0 is defined as LA. This is Long/Ascii. This is the default, so you won't see anything special when you view that memory.

```
< addtag 50 50 wa <enter>
```

The next 50 bytes of memory (starting on address 50) are defined as WA or Word/Ascii. We can use the 'view' command to see what we have done :

```
< view 0 <enter>
```

(Note that the 'view' command has the same sort of arguments as the 'memory' command).

```
> 00000000: 00000000 07E007CC 00F80834 00F80B16              ...........4....
> 00000010: 00F80ADA 00F80ADC 00F80ADE 00F80AE0              ................
> 00000020: 00F80C00 00F80AE4 00F80AE7 00F80AE8              ................
> 00000030: 00F8                                             ..
> 00000032: 0AEA 00F8 0AEC 00F8 0AEE 00F8 0AF0 00F8          ................
> 00000042: 0AF2 00F8 0AF4 00F8 0AF6 00F8 0AF8 00F8          ................
> 00000052: 0AFA 00F8 0AFC 00F8 0AFE 00F8 0B00 00F8          ................
> 00000062: 0B02                                             ..
> 00000064: 00F810F4 00F81152 00F81188 00F811E6              .......R........
> 00000074: 00F8127C 00F812C6 00F81310 00F80B70              ...|...........p
> 00000084: 00F80B72 00F80B74 00F80B76 00F80B78              ...r...t...v...x
> 00000094: 00F80B7A 00F80B7C 00F80B7E 00F80B80              ...z...|...~....
> 000000A4: 00F80B82 00F80B84 00F80B86 00F80B88              ................
> 000000B4: 00F80B8A 00F80B8C 00F80B8E 00F80B90              ................
> 000000C4: 00F80B92 00F80B94 00F80B96 00F80B98              ................
> 000000D4: 00F80B9A 00F80B9C 00F80B9E 00F80BA0              ................
> 000000E4: 00F80BA2 00F80BA4 00F80BA6 00F80BA8              ................
> 000000F4: 00F80BAA 00F80BAC 00F80BAE 00000000              ................
> 00000104: 00000000 00000000 00000000 00000000              ................
> 00000114: 00000000 00000000 00000000 00000000              ................
> 00000124: 00000000 00000000 00000000 00000000              ................
> 00000134: 00000000 00000000 00000000                       ............
```

You can see that the memory starting at location 50 is listed in Word/Ascii format.

```
< addtag 100 50 ba <enter>
```

We define the next 50 bytes of memory as Byte/Ascii and :

```
< addtag 150 50 as <enter>
```

the next 50 bytes of memory as full Ascii and :

```
< addtag 200 50 co <enter>
```

the next 50 bytes of memory as code.

```
< view 0 <enter>
> 00000000: 00000000 07E007CC 00F80834 00F80B16              ...........4....
> 00000010: 00F80ADA 00F80ADC 00F80ADE 00F80AE0              ................
> 00000020: 00F80C00 00F80AE4 00F80AE7 00F80AE8              ................
> 00000030: 00F8                                             ..
> 00000032: 0AEA 00F8 0AEC 00F8 0AEE 00F8 0AF0 00F8          ................
> 00000042: 0AF2 00F8 0AF4 00F8 0AF6 00F8 0AF8 00F8          ................
> 00000052: 0AFA 00F8 0AFC 00F8 0AFE 00F8 0B00 00F8          ................
> 00000062: 0B02                                             ..
> 00000064: 00 F8 10 F4 00 F8 11 52 00 F8 11 88 00 F8 11 E6  .......R........
> 00000074: 00 F8 12 7C 00 F8 12 C6 00 F8 13 10 00 F8 0B 70  ...|...........p
> 00000084: 00 F8 0B 72 00 F8 0B 74 00 F8 0B 76 00 F8 0B 78  ...r...t...v...x
> 00000094: 00 F8                                            ..
> 00000096: .z...|...~........................................
> 000000C8: 00F8 0B94                     ORI.?    #$F8,($B94)
> 000000CC: 00F8 0B96                     ORI.?    #$F8,($B96)
> 000000D0: 00F8 0B98                     ORI.?    #$F8,($B98)
> 000000D4: 00F8 0B9A                     ORI.?    #$F8,($B9A)
> 000000D8: 00F8 0B9C                     ORI.?    #$F8,($B9C)
> 000000DC: 00F8 0B9E                     ORI.?    #$F8,($B9E)
> 000000E0: 00F8 0BA0                     ORI.?    #$F8,($BA0)
> 000000E4: 00F8 0BA2                     ORI.?    #$F8,($BA2)
> 000000E8: 00F8 0BA4                     ORI.?    #$F8,($BA4)
> 000000EC: 00F8 0BA6                     ORI.?    #$F8,($BA6)
> 000000F0: 00F8 0BA8                     ORI.?    #$F8,($BA8)
> 000000F4: 00F8 0BAA                     ORI.?    #$F8,($BAA)
> 000000F8: 00F8 0BAC                     ORI.?    #$F8,($BAC)
> 000000FA: 0BAC00F8 0BAE0000 00000000 00000000              ................
> 0000010A: 00000000 00000000 00000000 00000000              ................
> 0000011A: 00000000 00000000 00000000 00000000              ................
> 0000012A: 00000000 00000000 00000000 00000000              ................
> 0000013A: 00000000 0000                                    ......
```

(The code example is useless in this case since that memory clearly isn’t code).
You can see that tags are very versatile. They can be very useful when you are debugging and do not want to loose track of all the different types of memory. If you still want to look at memory in a uniform way (either data or code) you can still use the memory and unasm commands. These commands ignore the tags.

You can see which tags are defined with tags :

```
< tags <enter>
> 00000000 : 00000032 LA
> 00000032 : 00000032 WA
> 00000064 : 00000032 BA
> 00000096 : 00000032 AS
> 000000C8 : 00000032 CO
```

(All values in this output are hexadecimal).

Note that it is possible to create overlapping tags. This is not encouraged since the search order of these tags is not defined. If you have an address that is defined in two different tags, you can never be sure which tag is taken as the correct one.
However, PowerVisor will automatically detect overlapping tags when the new tag is not completely in another tag or when the new tag does not completely redefine another tag. In that case the other tag is made smaller.

You can remove a tag using the remtag command :

```
< remtag 100 <enter>
```

will remove the definition for the range starting at address 100.

You can remove all tags at once with the cleartags command.

```
< cleartags <enter>
< tags <enter>
```

All tags are gone.

You can load and save tags using the loadtags and savetags commands.

If you want different tag lists for different applications you can use any of the other 15 tag lists. PowerVisor has 16 tag lists numbered from 0 to 15. The default tag list is 0.

You can change the current tag list using the usetag command :

```
< usetag 1 <enter>
```

will use tag list 1.

```
< usetag 0 <enter>
```

Back to tag list 0.

All commands on tags (addtag , remtag , loadtags , savetags , cleartags , view , ...) only look at the current tag list.

You can temporarily set the current tag list using the tg command :

```
< tg 1 view 0 <enter>
```

will view the memory starting at 0 using tag list 1. After the operation it will restore the current tag list.

Use the taglist() function to see the current tag list.

```
< disp taglist() <enter>
```

```
> 00000000 , 0
```

1.12 Looking At Things : Using tags and structures

In The tag system and 'view' we saw five different tag types :

BA	Byte/Ascii
WA	Word/Ascii
LA	Long/Ascii
AS	Full Ascii
CO	Code

There is a sixth tag type :

ST	Structure

We explain structure tags with an example :

Clear all structures and tags in memory with the clearstruct and cleartags commands :

```
< clearstruct <enter>
< cleartags <enter>
```

Load the exec structure file with addstruct :

```
< addstruct exec.pvsd <enter>
> UNIT
> IS
> IV
> IO
> IOSTD
> LIB
> LH
> MLH
> ML
> ME
> MH
> MC
> LN
> MLN
> MP
> MN
> RT
> SSR
> SS
> SM
> TC
> ETask
> StackSwapStruct
```

(See Viewing structures for more info about these commands).

Now we can use these structures to define structure tags with addtag :

```
< task <enter>

< list <enter>
> Task node name      : Node     Pri   StackU   StackS Stat Command         Acc
> -----------------------------------------------------------------------------
> ConClip Process     : 07E60410 00       242     4000 Wait sys:c/ConCl(02) -
> RexxMaster          : 07E6AA48 04       162     2048 Wait            (00) -
> « IPrefs »          : 07E59568 00       862     3500 Wait            PROC -
> ...
> trackdisk.device    : 07E0E714 05        98      512 Wait            TASK -
> input.device        : 07E07F12 14        86     4096 Wait            TASK -
> ramlib              : 07E29048 00       230     2048 Wait            PROC -
> RAM                 : 07E31220 0A       678     1200 Wait            PROC -
> pv                  : 07F62FC0 04       438    16000 Run  pv         (01) -

< addtag ramlib stsize(tc) st tc <enter>
```

What have we done? We have defined a new tag starting with the address of the ’ramlib’ task. This tag defines a region of memory that is ’stsize(tc)’ bytes big. stsize() is a function that returns the size of a structure. The structure is the ’TC’ structure (task structure). The tag we define has type ’ST’ (structure tag). When you use the ’ST’ type for a tag you need another argument to ’addtag’: the pointer to the structure definition. This is ’TC’ (note that you can also use ’_task’ which is the builtin task structure definition).

With the tags we can see all defined tags :

```
< tags <enter>
> 07E29048 : 0000005C ST TC
```

Now we view the memory surrounding this task structure with view :

```
< view ramlib-50 <enter>
> 07E29016: 000001F8 8D060000 000207E2 37CC0000               ............7...
> 07E29026: 00000000 136C0000 02C20000 090E0000               .....l..........
> 07E29036: 00000000 00000000 000001F8 54DA0000               ............T...
> 07E29046: 0000                                              ..
> 07E29048: TC
> Pri          : 00       |
> Name         : ramlib
> Flags        : 00       | State        : 04       | IDNestCnt    : FF       |
> TDNestCnt    : 00       | SigAlloc     : 0000FFFF | SigWait      : 00000010 |
> SigRecvd     : 00000100 | SigExcept    : 00000000 | ETask        : 80000000 |
> ExceptData   : 00000000 | ExceptCode   : 00F83068 | TrapData     : 00000000 |
> TrapCode     : 00F92A46 | SPReg        : 07E29846 | SPLower      : 07E2912C |
> SPUpper      : 07E2992C | Switch       : 00000000 | Launch       : 00000000 |
> MemEntry     : 07E29092 | Userdata     : 00000000 |
> 07E290A4: 00000000 00000000 00000000 00000008               ................
> 07E290B4: 07E29048 07E290BC 00000000 07E290B8               ...H............
> 07E290C4: 00000000 01F8A3A9 00000800 07E0EEE4               ................
> 07E290D4: 00000000 01F8A44B 00000000 00000000               .......K........
> 07E290E4: 00000000 00000000 00000000 07E0FF94               ................
> 07E290F4: 00000000 07E29928 00000000 00000000               .......(........
> 07E29104: 00000000 00000000 00000000 00000000               ................
> 07E29114: 00000000 07E2911C 00000000 07E29118               ................
> 07E29124: 00000000 00000000 00000000 00000000               ................
> 07E29134: 00000000 00000000 00000000 00000000               ................
> 07E29144: 00000000 00000000 00000000 00000000               ................
> 07E29154: 0000                                              ..
```

The output is the same as with the interprete command.

Of course it would be cumbersome if you hade to repeat this procedure for each task in the task list. You can use the for command to automate this process (also see Listing things) :

```
< for task addtag rc stsize(tc) st tc <enter>
```

This command will define a tag for each task in the task list.

1.13 Looking At Things : Some miscellanious viewing commands

PowerVisor also has a lot of other smaller view commands. These are all explained in this section.

You can list all gadgets in a window with the gadgets command :

```
< list wins <enter>
> Window name         : Address  Left  Top Width Height WScreen
> ---------------------------------------------------------------------------
>                     : 07EA69D8    0   12   692    430 07EA6378
>                     : 07E45E38    0    0   704    456 07E46110
> My Shell            : 07E1FD48    0  568   692    456 07E2D258
>                     : 07E3B398    0   16   692   1008 07E2D258

< gadgets my <enter>
> Gadget ptr : left right width height Render   Text     SpecInfo ID
>
> 07E100D4   :  -22     0    24     16 07E4687C 00000000 00000000     0
> Flags      : GADGHCOMP GADGIMAGE GRELRIGHT LABELITEXT
> Activation : RELVERIFY BORDERSNIFF
> Type       : SYSGADGET WUPFRONT CUSTOMGADGET
>
> 07E10114   :  -45     0    24     16 07E489C4 00000000 00000000     0
> Flags      : GADGHCOMP GADGIMAGE GRELRIGHT LABELITEXT
> Activation : RELVERIFY BORDERSNIFF
> Type       : SYSGADGET WDOWNBACK CUSTOMGADGET
>
> 07E1FDFC   :  -17    -9    18     10 07E48DF4 00000000 00000000     0
> Flags      : GADGHCOMP GADGIMAGE GRELBOTTOM GRELRIGHT LABELITEXT
> Activation : RELVERIFY BORDERSNIFF
> Type       : SYSGADGET SIZING CUSTOMGADGET
>
> 07E1FE3C   :    0     0    20     16 07E58E0C 00000000 00000000     0
> Flags      : GADGHCOMP GADGIMAGE LABELITEXT
> Activation : RELVERIFY BORDERSNIFF
> Type       : SYSGADGET CLOSE CUSTOMGADGET
>
> 07E1FE7C   :    0     0     0     15 00000000 00000000 00000000     0
> Flags      : GADGHCOMP GADGIMAGE GRELWIDTH LABELITEXT
> Activation : BORDERSNIFF
> Type       : SYSGADGET WDRAGGING CUSTOMGADGET
```

You can list all hunks for a process with the hunks command :

```
< list task <enter>
> Task node name      : Node     Pri   StackU   StackS Stat Command         Acc
> -----------------------------------------------------------------------------
> ConClip Process     : 07E60410 00       242     4000 Wait sys:c/ConCl(02) -
> RexxMaster          : 07E6AA48 04       162     2048 Wait            (00) -
> ClickToFront        : 07E75210 15       398     4096 Wait            PROC -
> ...
> Workbench           : 07E6C340 01       166     6000 Wait Workbench  (03) -
> input.device        : 07E07F12 14        86     4096 Wait            TASK -
> RAM                 : 07E31220 0A       678     1200 Wait            PROC -
> pv                  : 07F62FC0 04       438    16000 Run  pv         (01) -

< hunks 'pv' <enter>
> Nr    Hunk     Data         Size
> -----------------------------------------------------------------------------
>     0 07EB9A7C 07EB9A80    14112
>     1 07E55014 07E55018     2472
>     2 07E559BC 07E559C0     1724
>     3 07E87A6C 07E87A70     3860
>     4 07F4D6E4 07F4D6E8    71276
>     5 07EBD19C 07EBD1A0     7960
>     6 07E2B31C 07E2B320      156
>     7 07E2A28C 07E2A290       28
>     8 07E2B3BC 07E2B3C0      212
>     9 07E2ABAC 07E2ABB0      136
```

You can ask the pathname for a lock with the pathname command. Note that you MUST use normal pointers for the 'pathname' command. The result from the AmigaDOS 'Lock' function is a BPTR. You must convert this BPTR to an APTR.

You can use the libinfo command to ask information about a library function in an fd-file you have loaded.

Use the llist command to traverse a list with nodes. The argument to this command is a node. 'llist' will then follow the ln_Succ field in this node for all other nodes. It will display the addresses to these nodes :

```
< task <enter>

< llist df0 <enter>
> Node name           : Node     Pri
> ---------------------------------------------------------------------------
> Work                : 07E189B0 0A
> Workbench           : 07E34018 01
> input.device        : 07E08B22 14
> RAM                 : 07E23BF8 0A
```

Use the owner command if you want to know the owner of a piece of memory. This command tries the best it can to find the owner. At this moment only the 'task' list is searched.

```
< list task <enter>
> Task node name      : Node     Pri   StackU   StackS Stat Command         Acc
> ------------------------------------------------------------------------------
> ConClip Process     : 07E60410 00       242     4000 Wait sys:c/ConCl(02) -
> RexxMaster          : 07E6AA48 04       162     2048 Wait            (00) -
> ClickToFront        : 07E75210 15       398     4096 Wait            PROC -
> CpuBlit             : 07E7BA18 00       266     2048 Wait            PROC -
> ...

< owner 07E6B28A <enter>
> Found in stack
> RexxMaster          : 07E6AA48 04       162     2048 Wait            (00) -
```

1.14 Looking At Things : Commands for MMU and other processors

If you have an 68020, 68030 or 68040 you can use some extra commands.

You can use the specregs command to view all special 680x0 registers :

```
< specregs <enter>
> MSP  : 560F5B16
> ISP  : 07E02250
> USP  : 07F368D0
> SFC  : 00000007
> DFC  : 00000007
> VBR  : 07EFFB00
> CACR : 00002111
>    Write Allocate : set
>    Data Burst : disabled
>    Clear Data Cache : not set
>    Clear Entry in Data Cache : not set
>    Freeze Data Cache : not set
>    Data Cache : enabled
>    Instruction Burst : enabled
>    Clear Instruction Cache : not set
>    Clear Entry in Instruction Cache : not set
>    Freeze Instruction Cache : not set
>    Instruction Cache : enabled
> CAAR : F8F76BED
```

For all following commands you need a MMU. This means that you either must have an 68851 or an 68030.
At this moment I have not tested PowerVisor on an 68040 processor. I suspect there could be some problems. Especially the mmutree and mmureset commands can cause problems on this new processor.

Also the 'mmureset' and 'mmutree' commands do not support everything from the 68030 mmu.

I have also not been able to test these commands on a computer other than the Amiga 3000.

Use the mmuregs command to view all mmu registers :

```
< mmuregs <enter>
> DRP  : (na)
> CRP  : 80000002   07F5C000
>    L/U bit : set
>    LIMIT = 00000000
>    DT    = Valid 4 byte
>    Table address = 07F5C000
> SRP  : 80000001   00000000
>    L/U bit : set
>    LIMIT = 00000000
>    DT    = Page descriptor
>    Table address = 00000000
> TC   : 80C08660
>    Address translation : enabled
>    Supervisor Root Pointer (SRP) : disabled
>    Function Code Lookup (FCL) : disabled
>    System page size    = 00001000
>    Initial shift       = 00000000
>    Table Index A (TIA) = 00000008
>    Table Index B (TIB) = 00000006
>    Table Index C (TIC) = 00000006
>    Table Index D (TID) = 00000000
> TT0  : 04038207
>    Log Address Base = 00000004
>    Log Address Mask = 00000003
>    TT register : enabled
>    Cache Inhibit : no
>    R/W : set
>    RWM : not set
>    FC value for TT block = 00000000
>    FC bits to be ignored = 00000007
> TT1  : 403F8107
>    Log Address Base = 00000040
>    Log Address Mask = 0000003F
>    TT register : enabled
>    Cache Inhibit : no
>    R/W : not set
>    RWM : set
>    FC value for TT block = 00000000
>    FC bits to be ignored = 00000007
```

With the mmutree command you can view the current mmu tree :

```
< mmutree <enter>
> 00000000   4 BYTE (imuw)  Log: 00000000 # 00000000
> 07FFF140      4 BYTE (imUw)  Log: 00000000 # 01000000
> 07FFF180         PAGE   (IMUw)  Log: 00000000 # 00040000  -> 00000000
> ...
> 07FFF274         PAGE   (iMUw)  Log: 00F40000 # 00040000  -> 00F40000
> 07FFF278         PAGE   (iMUW)  Log: 00F80000 # 00040000  -> 07F80000
> 07FFF27C         PAGE   (iMUW)  Log: 00FC0000 # 00040000  -> 07FC0000
> 07FFF144      PAGE   (iMUw)  Log: 01000000 # 01000000  -> 01000000
> 07FFF148      PAGE   (iMUw)  Log: 02000000 # 01000000  -> 02000000
> 07FFF14C      PAGE   (iMUw)  Log: 03000000 # 01000000  -> 03000000
> 07FFF150      PAGE   (iMUw)  Log: 04000000 # 01000000  -> 04000000
> 07FFF154      PAGE   (iMUw)  Log: 05000000 # 01000000  -> 05000000
> 07FFF158      PAGE   (iMUw)  Log: 06000000 # 01000000  -> 06000000
> 07FFF15C      4 BYTE (imUw)  Log: 07000000 # 01000000
> 07FFF280         INV    (imuw)  Log: 07000000 # 00040000
> ...
> 07FFF34C         INV    (imuw)  Log: 07CC0000 # 00040000
> 07FFF350         PAGE   (iMUw)  Log: 07D00000 # 00040000  -> 07D00000
> 07FFF354         INV    (imuw)  Log: 07D40000 # 00040000
> 07FFF358         INV    (imuw)  Log: 07D80000 # 00040000
> 07FFF35C         INV    (imuw)  Log: 07DC0000 # 00040000
> 07FFF360         PAGE   (iMUw)  Log: 07E00000 # 00040000  -> 07E00000
> 07FFF364         PAGE   (iMUw)  Log: 07E40000 # 00040000  -> 07E40000
> 07FFF368         PAGE   (iMUw)  Log: 07E80000 # 00040000  -> 07E80000
> 07FFF36C         PAGE   (iMUw)  Log: 07EC0000 # 00040000  -> 07EC0000
> 07FFF370         PAGE   (iMUw)  Log: 07F00000 # 00040000  -> 07F00000
> 07FFF374         PAGE   (iMUw)  Log: 07F40000 # 00040000  -> 07F40000
> 07FFF378         PAGE   (iMUW)  Log: 07F80000 # 00040000  -> 07F80000
> 07FFF37C         PAGE   (iMUW)  Log: 07FC0000 # 00040000  -> 07FC0000
> 07FFF160      PAGE   (iMUw)  Log: 08000000 # 01000000  -> 08000000
> 07FFF164      PAGE   (iMUw)  Log: 09000000 # 01000000  -> 09000000
> 07FFF168      PAGE   (iMUw)  Log: 0A000000 # 01000000  -> 0A000000
> 07FFF16C      PAGE   (iMUw)  Log: 0B000000 # 01000000  -> 0B000000
> 07FFF170      PAGE   (iMUw)  Log: 0C000000 # 01000000  -> 0C000000
> 07FFF174      PAGE   (iMUw)  Log: 0D000000 # 01000000  -> 0D000000
> 07FFF178      PAGE   (iMUw)  Log: 0E000000 # 01000000  -> 0E000000
```

With the mmureset command you can reset the ’M’ and ’U’ bits in this tree. So you can see which pages are used and modified.