Enforcer

COLLABORATORS			
	TITLE : Enforcer		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 16, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Enforcer **1**

1.1 main . . . 1
1.2 credits . . . 2
1.3 credits_testers . . . 2
1.4 enforcer . . . 3
1.5 findhit . . . 6
1.6 lawbreaker . . . 7
1.7 mmu . . . 9
1.8 move4k . . . 11
1.9 segtracker . . . 12
1.10 rebootoff . . . 14
1.11 debuggers1 . . . 15
1.12 debuggers2 . . . 17
1.13 notes1 . . . 18
1.14 notes2 . . . 19
1.15 notes3 . . . 20
1.16 notes4 . . . 20
1.17 notes6 . . . 22
1.18 cpu_library . . . 23
1.19 cpu_patches . . . 25
1.20 copyback_dma . . . 36
1.21 option_quiet . . . 39
1.22 option_tiny . . . 39
1.23 option_small . . . 40
1.24 option_showpc . . . 40
1.25 option_stacklines . . . 40
1.26 option_stackcheck . . . 40
1.27 option_aregcheck . . . 41
1.28 option_dregcheck . . . 41
1.29 option_datestamp . . . 41

1.30 option_deadly . . . 42
1.31 option_fspace . . . 42
1.32 option_verbose . . . 42
1.33 option_led . . . 42
1.34 option_parallel . . . 43
1.35 option_rawio . . . 43
1.36 option_file . . . 43
1.37 option_stdio . . . 44
1.38 option_buffersize . . . 44
1.39 option_intro . . . 44
1.40 option_priority . . . 45
1.41 option_noalertpatch . . . 45
1.42 option_on . . . 45
1.43 option_quit . . . 45
1.44 output . . . 45
1.45 findseg . . . 47
1.46 quotes . . . 50
1.47 copyright . . . 51
1.48 detailexample . . . 52
1.49 output_datestamp . . . 57
1.50 output_write . . . 57
1.51 output_address . . . 57
1.52 output_writedata . . . 57
1.53 output_pc . . . 58
1.54 output_buserror . . . 58
1.55 output_sr . . . 58
1.56 output_sw . . . 58
1.57 output_decode . . . 59
1.58 output_tcb . . . 59
1.59 output_dataregs . . . 59
1.60 output_d0 . . . 59
1.61 output_d1 . . . 59
1.62 output_d2 . . . 60
1.63 output_d3 . . . 60
1.64 output_d4 . . . 60
1.65 output_d5 . . . 60
1.66 output_d6 . . . 60
1.67 output_d7 . . . 60
1.68 output_addrregs . . . 61

1.69 output_a0 . . . 61
1.70 output_a1 . . . 61
1.71 output_a2 . . . 61
1.72 output_a3 . . . 61
1.73 output_a4 . . . 61
1.74 output_a5 . . . 62
1.75 output_a6 . . . 62
1.76 output_a7 . . . 62
1.77 output_stack . . . 62
1.78 output_stackword . . . 62
1.79 output_segtracker . . . 63
1.80 output_segtrackeraddress . . . 63
1.81 output_segtrackername . . . 63
1.82 output_segtrackerhunk . . . 63
1.83 output_segtrackeroffset . . . 63
1.84 output_name . . . 64
1.85 output_taskname . . . 64
1.86 output_cliname . . . 64
1.87 output_alert . . . 64
1.88 output_alertnum . . . 64
1.89 output_showpc . . . 64
1.90 output_showpc_m8 . . . 65
1.91 output_showpc_m7 . . . 65
1.92 output_showpc_m6 . . . 65
1.93 output_showpc_m5 . . . 65
1.94 output_showpc_m4 . . . 65
1.95 output_showpc_m3 . . . 66
1.96 output_showpc_m2 . . . 66
1.97 output_showpc_m1 . . . 66
1.98 output_showpc_p0 . . . 66
1.99 output_showpc_p1 . . . 66
1.100 output_showpc_p2 . . . 67
1.101 output_showpc_p3 . . . 67
1.102 output_showpc_p4 . . . 67
1.103 output_showpc_p5 . . . 67
1.104 output_showpc_p6 . . . 67
1.105 output_showpc_p7 . . . 68
1.106 sourcecode . . . 68
1.107 orderform . . . 69
1.108 index . . . 69

Chapter 1

Enforcer

1.1 main

```
                                    Enforcer  by
                          Michael Sinz

                       Copyright 1992-1998
                       All Rights Reserved


                    Enforcer

                    SegTracker

                    FindHit

                    LawBreaker

                    RebootOff

                    Move4K

                    MMU

                    Copyright
                    Credits
                ************************************************************************

*                                                                              *
* Permission is hereby granted to distribute the Enforcer archive             *
* containing the executables and documentation for non-commercial purposes    *
* so long as the archive and its contents are not modified in any way.        *
*                                                                              *
* Enforcer and related tools may not be distributed for a profit.             *
*                                                                              *
* Enforcer and related tools are not in the public domain.                    *
*                                                                              *
********************************************************************************
```

```
              ----> Support Enforcer - Register today and get a source code  ↩
                 license! <-----
                      +-----------------------------------------------------+ ↩

        | Michael Sinz                                                      |
        |                  I-NET:  Enforcer@sinz.org                        |
        | BIX:  msinz      or      msinz@bix.com                            |
        | "Can't I just bend one of the rules?" said the student.           |
        | The Master just looked back at him with a sad expression.         |
        +-------------------------------------------------------------------+
```

1.2 credits

```
                I would like to thank those nice people at ATL to letting me have  ↩
                   a
68060 based Amiga to work with.  Without it, the 68060 support would
not have been possible.

I would like to thank Bryce Nesbitt for coming up with the original
Enforcer idea.  Enforcer has helped the Amiga more than any other
debugging tool.

The Enforcer shield in the icon was designed by David "talin" Joiner.

I would also like to thank
                the people
                 who stayed with me during
all the long testing and the many beta releases Enforcer had.

However, I want to thank most the Amiga developers who use Enforcer
every day.  Like any other tool, Enforcer can not help the quality
of Amiga software if it is not used.  Running Enforcer all the time
makes it easier to notice bugs that happen during regular use of
the Amiga.

Thank you for making your software better!  It really does help the
Amiga when the software for it works well.

                   -- Michael Sinz

PS - To those people who still say that Enforcer causes working software
to have problems:  Enforcer just points out actions in software that
are already a problem and could cause major problems in some cases.
Enforcer does *not* cause any problems for software that does not
access invalid addresses.  Enforcer is 100% benign to software that
follows the rules.
```

1.3 credits_testers

```
The following are some of the people who helped test
various versions of Enforcer V37:
```

```
Peter Cherna                    peter.cherna@scala.com
Dave Haynie                     dave.haynie@scala.com
Erik Quackenbush                erik.quackenbush@scala.com
Martin Taillefer                vertex@bix.com
Brian Gontowski                 bgontowski@bix.com
Toby Simpson                    toby@bix
Benjamin Fuller                 benfuller@bix.com
David Joiner                    talin@bix.com
James M. Barkley, Jr            jim.barkley@scala.com
Chris Green                     c_green@bix.com
David N. Junod                  djunod@bix.com
Joanne Dow                      jdow@bix.com
Jim Cooper                      jcooper@bix.com
Doug Walker                     djwalker@bix.com
Steve Krueger                   skrueger@bix.com
Steve Tibbett                   s.tibbett@bix.com
Heinz Wrobel                    heinz@hwg.muc.de
Kenneth T. Spoor                metadigm@bix.com
Victor A. Wagner                metadigm@bix.com
Sebastiano Vigna                svigna@bix.com
Tomas Rokicki                   radical.eye@bix.com
Redmond Simonsen                rsimonsen@bix.com
Willem Langeveld                langeveld@bix.com
Marvin Weinstein                mweinstein@bix.com
Lamonte Koop                    lkoop@bix.com
Allan M. Purtle                 snapper@mgl.ca
Gregory B Tibbs                 gbtibbs@bix.com
Robert Chapman                  rchapman@bix.com
```


1.4 enforcer

```
                Enforcer V37 - An advanced version of Enforcer - Requires V37

SYNOPSIS
     Enforcer - A tool to watch for illegal memory accesses

FUNCTION
     Enforcer will use the MMU in the advanced 680x0 processors
     to set up MMU tables to watch for illegal accesses to memory
     such as the low-page and non-existent pages.

     To use, run Enforcer (plus any options you may wish)
     If you wish to detach, just use RUN >NIL: <NIL: to start it.
     You can also start it from the Workbench.  When started from Workbench,
     Enforcer will read the tooltypes of its icon or selected project icon
     for its options.  (See the sample project icons)

     Enforcer should only be run *after* SetPatch.

     If
                SegTracker
                 is running in the system when Enforcer is started,
     Enforcer will use the public
                SegTracker
                 seglist tracking for
```

identifying the hits.

INPUTS

The options for Enforcer are as follows:

QUIET

DATESTAMP

STDIO

TINY

DEADLY

BUFFERSIZE

SMALL

FSPACE

INTRO

SHOWPC

VERBOSE

PRIORITY

STACKLINES

LED

NOALERTPATCH

STACKCHECK

PARALLEL

ON

AREGCHECK

RAWIO

QUIT

DREGCHECK

FILE

RESULTS

When run, a set of MMU tables that map addresses that are not in the system's address map as invalid are installed. Enforcer will then trap invalid access attempts and generate a diagnostic message as to what the illegal access was. The first memory page (the one starting at location 0) is also marked as invalid as many

programming errors cause invalid access to these addresses. Invalid addresses are completely off limits to applications.

When an access violation happens, a report such as the following is output.

Output Example

Detail Example

WARNING

Enforcer is for software testing. In this role it is vital. Software that causes Enforcer hits may not be able to run on newer hardware. (Enforcer hits of high addresses on systems not running Enforcer but with a 68040 will most likely crash the system) Future systems and hardware will make this even more important. The system can NOT survive software that causes Enforcer hits.

However, Enforcer is NOT a system protector. As a side effect, it may well keep a system from crashing when Enforcer hits happen, but it may just as well make the software crash earlier. Enforcer is mainly a development and testing tool.

Enforcer causes no ill effects with correctly working software. If a program fails to work while Enforcer is active, you should contact the developer of that program.

NOTES

General Notes

68020 Notes

68030 Notes

68040 Notes

68060 Notes

BridgeBoard

Important 680x0.library developer notes

WRITING DEBUGGERS

If you wish to make a debugger that works with Enforcer to help pinpoint Enforcer hits in the application and not cause Enforcer hits itself, here are some simple tips and a bit of code.

Debuggers: Trapping a hit

Debuggers: Not causing a hit

SEE ALSO

"A master's secrets are only as good as the
master's ability to explain them to others." - Michael Sinz

1.5 findhit

```
               FindHit - A tool that can locate the source file and line number
          that a
               SegTracker
                report happened at.

SYNOPSIS
     FindHit will read the executable file and if there is debugging
     information in it, will try to locate the source file and line
     number that correspond to the
               Enforcer
                hit HUNK/OFFSET.

FUNCTION
     FindHit uses the Lattice/SAS/MetaScope standard 'LINE'
     debug hunk to locate the closest line to the hunk/offset given.
     Note that this can only happen if the executable has the
     LINE debugging turned on.  (The
               LawBreaker
                program has this
     such that you can test this yourself.)

     In SAS/C 6.x, you need to compile with DEBUG=LINE or better
     and do not use the link option of NODEBUG.

     In SAS/C 5.x, you need to compile with -d1 or better.
     Note that FindHit works with the old SAS/C 5.x 'SRC '
     debugging information too.  This is required for -d2 or
     higher debugging support.  However, I do not have 'SRC ' hunk
     documentation and thus FindHit may be very specific to the
     SAS/C 5.x version of this hunk.

     In DICE (2.07 registered being the one I tried) the -d1
     debug switch also supports the 'LINE' debug hunk and
     works with FindHit.

     In HX68 and CAPE, you need to add the DEBUG directive to
     the assembly code program.  (See
               LawBreaker
                source)

     For other languages, or other versions of the above, please
     see the documentation that comes with the language.

INPUTS
     FILE/A      - The executable file, with debugging information.

     OFFSETS/A/M - The HEX offset (with or without leading $)
                   If a hunk number other than the default
                   is needed, it is expressed as hunk:offset.
                   The default hunk is that of the last argument
                   or hunk 0 if no hunk number has been given.
                   For example:  12 $22 $3:12 22 4:$12 32 $0:$32
                   will find information for:
                   hunk $0, offset $12
```

```
                    hunk $0, offset $22
                    hunk $3, offset $12
                    hunk $3, offset $22
                    hunk $4, offset $12
                    hunk $4, offset $32
                    hunk $0, offset $32

EXAMPLE
     FindHit FooBar $0342 $1:4F2 3:$1A 2C
     badcode.c : Line 184
     No line number information for Hunk $1, Offset $4F2
     badcode2.c : Line 12
     badcode2.c : Line 14

     See the
                Enforcer
                 documentation about issues dealing with the
     exact location of the
                Enforcer
                 hit.  The line given may
     not be exactly where the hit happened.

     The way I use this is to always have line debugging turned on
     when I compile.  This does not change the quality of the code
     and takes only a small amount of extra disk space.  However,
     what I do is to link the program twice:  Once to a file called
     program.ld which contains all of the debugging information.
     Then, I link program.ld to program, stripping debug information.
     The command line for SLINK or BLINK is as follows:

         BLINK program.ld TO program NODEBUG

     I keep both of these on hand; with program being the one I
     distribute and use.  When a hit happens, I can just use program.ld
     with FindHit to get the line number and source file that it happened
     in.  This way you can distribute your software without the debugging
     information and still be able to use FindHit on the actual code.
     (After all, that link command does nothing but strip symbol and
     debug hunks)

NOTES
     Note that this program does nothing when run from the Workbench
     and thus does not have an icon.

SEE ALSO
     "Quantum Physics:  The Dreams that Stuff is made of." - Michael Sinz
```

1.6 lawbreaker

```
                LawBreaker - A quicky test of
                Enforcer
                SYNOPSIS
     This is a quick test of
                Enforcer
                 and its reporting abilities.
```

```
FUNCTION
     This program is used to make sure that
               Enforcer
                is correctly
     installed and operating.  LawBreaker works from either the CLI
     or Workbench.  It will try to read and write certain memory
     areas that will cause an
               Enforcer
                hit or four.

     LawBreaker will also do an Alert to show how
               Enforcer
                reports
     an Alert.

     Note that the LawBreaker executable has debugging information
     in it (standard LINE format debug hunk) such that you can
     try the
               FindHit
                program to find the line that causes the hit.

INPUTS
     Just run it...

RESULTS
     When running
               Enforcer
               , you will see some output from
               Enforcer
               .
     Output on a 68030 machine would look something like this:

25-Jul-93  17:15:04
WORD-WRITE to  00000000        data=0000       PC: 0763857C
USP:  07657C14 SR: 0004 SW: 04C1  (U0)(-)(-)  TCB: 07642F70
Data: DDDD0000 DDDD1111 DDDD2222 DDDD3333 0763852A DDDD5555 DDDD6666 DDDD7777
Addr: AAAA0000 AAAA1111 AAAA2222 AAAA3333 AAAA4444 0763852A 07400810 --------
Stck: 00000000 0752EE9A 00002800 07643994 00000000 076786D8 000208B0 2EAC80EE
Stck: 487AFD12 486C82C4 4EBA3D50 4EBAEA28 4FEF0014 52ACE2E4 204D43EC 88BC203C
----> 0763857C - "lawbreaker"  Hunk 0000 Offset 00000074
Name: "Shell"  CLI: "LawBreaker"  Hunk 0000 Offset 00000074

25-Jul-93  17:15:04
LONG-READ from AAAA4444                       PC: 07638580
USP:  07657C14 SR: 0015 SW: 0501  (U0)(F)(-)  TCB: 07642F70
Data: DDDD0000 DDDD1111 DDDD2222 DDDD3333 0763852A DDDD5555 DDDD6666 DDDD7777
Addr: AAAA0000 AAAA1111 AAAA2222 AAAA3333 AAAA4444 0763852A 07400810 --------
Stck: 00000000 0752EE9A 00002800 07643994 00000000 076786D8 000208B0 2EAC80EE
Stck: 487AFD12 486C82C4 4EBA3D50 4EBAEA28 4FEF0014 52ACE2E4 204D43EC 88BC203C
----> 07638580 - "lawbreaker"  Hunk 0000 Offset 00000078
Name: "Shell"  CLI: "LawBreaker"  Hunk 0000 Offset 00000078

25-Jul-93  17:15:04
BYTE-WRITE to  00000101        data=11         PC: 0763858A
USP:  07657C14 SR: 0010 SW: 04A1  (U0)(F)(D)  TCB: 07642F70
Data: 00000000 DDDD1111 DDDD2222 DDDD3333 0763852A DDDD5555 DDDD6666 DDDD7777
```

```
Addr: AAAA0000 AAAA1111 AAAA2222 AAAA3333 AAAA4444 0763852A 07400810 --------
Stck: 00000000 0752EE9A 00002800 07643994 00000000 076786D8 000208B0 2EAC80EE
Stck: 487AFD12 486C82C4 4EBA3D50 4EBAEA28 4FEF0014 52ACE2E4 204D43EC 88BC203C
----> 0763858A - "lawbreaker"  Hunk 0000 Offset 00000082
Name: "Shell"  CLI: "LawBreaker"  Hunk 0000 Offset 00000082

25-Jul-93  17:15:04
LONG-WRITE to  00000102        data=00000000   PC: 07638592
USP:  07657C14 SR: 0014 SW: 0481  (U0)(-)(D)  TCB: 07642F70
Data: 00000000 DDDD1111 DDDD2222 DDDD3333 0763852A DDDD5555 DDDD6666 DDDD7777
Addr: AAAA0000 AAAA1111 AAAA2222 AAAA3333 AAAA4444 0763852A 07400810 --------
Stck: 00000000 0752EE9A 00002800 07643994 00000000 076786D8 000208B0 2EAC80EE
Stck: 487AFD12 486C82C4 4EBA3D50 4EBAEA28 4FEF0014 52ACE2E4 204D43EC 88BC203C
----> 07638592 - "lawbreaker"  Hunk 0000 Offset 0000008A
Name: "Shell"  CLI: "LawBreaker"  Hunk 0000 Offset 0000008A

25-Jul-93  17:15:06
Alert !! Alert 35000000     TCB: 07642F70     USP: 07657C10
Data: 00000000 DDDD1111 DDDD2222 DDDD3333 0763852A DDDD5555 DDDD6666 35000000
Addr: AAAA0000 AAAA1111 AAAA2222 AAAA3333 AAAA4444 0763852A 07400810 --------
Stck: 076385A0 00000000 0752EE9A 00002800 07643994 00000000 0762F710 076305F0
----> 076385A0 - "lawbreaker"  Hunk 0000 Offset 00000098
```

Now, using
FindHit
, you would type:

```
FindHit LawBreaker 0:82
```

and it will tell you the source file name and the line number where the hit happened. See the
FindHit
documentation.

NOTES

If
Enforcer
is not running, the program should not cause the system to crash. It will, however, write to certain areas of low memory. Also, it will cause read access of some addresses that may not exist. This may cause bus faults.

SEE ALSO

"Quantum Physics: The Dreams that Stuff is made of." - Michael Sinz

BUGS

There are 4 known
Enforcer
hits in this code and 1 alert, however, they will not be fixed. ;^)

1.7 mmu

MMU - A 68040/68060 MMU Mapping Tool

```
SYNOPSIS
     MMU is a tool for 68040/060 systems to display and/or change the MMU
     configuration contained within.  MMU will display the tables as
     compressed as possible, noting address ranges that are mapped the
     same and only displaying one line for the whole range.

INPUTS
     ADDRESS/K       - Hex address to display (if not given, all)
     SIZE/K          - Hex address size of address range
     READWRITE       - Change address range to ReadWrite
     READONLY        - Change address range to ReadOnly
     VALID           - Change address range to VALID
     INVALID         - Change address range to INVALID
     CACHE           - Change address range to CACHE enable
     NOCACHE         - Change address range to CACHE disable
     COPYBACK        - Change address range to non-serialized
     NOCOPYBACK      - Change address range to serialized
     GLOBAL          - Change address range to GLOBAL
     LOCAL           - Change address range to LOCAL
     SUPERVISOR      - Change address range to SUPERVISOR-only
     USER            - Change address range to USER access
     VERBOSE         - Show the changes as they are made
     NOSHOW          - Do not show the whole table

RESULTS
     On my A3000 with a 68040 installed and
                Enforcer
                 running, the
     MMU output looks as follows:  (No options)

Current 68040 MMU table setup:
$00000000-$00000FFF->$00000000:  Local, User, Invalid, Read-Only,  Serialized
$00001000-$001FFFFF->$00001000: Global, User,   Valid, Read/Write, Serialized
$00200000-$00BBFFFF->$07542000:  Local, User, Invalid, Read-Only,  Serialized
$00BC0000-$00BFFFFF->$00BC0000: Global, User,   Valid, Read/Write, Serialized
$00C00000-$00D7FFFF->$07542000:  Local, User, Invalid, Read-Only,  Serialized
$00D80000-$00DFFFFF->$00D80000: Global, User,   Valid, Read/Write, Serialized
$00E00000-$00E8FFFF->$07542000:  Local, User, Invalid, Read-Only,  Serialized
$00E90000-$00EAFFFF->$00E90000: Global, User,   Valid, Read/Write, Serialized
$00EB0000-$00EFFFFF->$07542000:  Local, User, Invalid, Read-Only,  Serialized
$00F00000-$00FFFFFF->$00F00000: Global, User,   Valid, Read-Only,    Copyback
$01000000-$073FFFFF->$07542000:  Local, User, Invalid, Read-Only,  Serialized
$07400000-$07F7FFFF->$07400000: Global, User,   Valid, Read/Write,   Copyback
$07F80000-$FFFFFFFF->$07542000:  Local, User, Invalid, Read-Only,  Serialized

WARNING
     This tool is a hack and does not have any safeguards on the options.
     Use at your own risk.

SEE ALSO
     "A master's secrets are only as good as the
      master's ability to explain them to others."  -  Michael Sinz

BUGS
     Misuse can cause major problems on your system.  Be *VERY* careful!
     MMU will not check that what you are doing is safe.
```

1.8 move4k

```
                Move4K - Moves as much out of the lower 4K of RAM as possible

SYNOPSIS
     On 68040 systems, as much of the lower 4K of CHIP RAM as possible
     is removed from system use.

FUNCTION
     On 68040 systems the MMU page sizes are 4K and 8K.
               Enforcer
                    uses the 4K page size.  Since watching for hits of low memory
     is a vital part of
               Enforcer
               , this means that the first 4K
     of RAM will be marked invalid.  On current systems, only
     the first 1K of RAM is invalid and thus 3K of RAM in that
     first 4K will end up needing to be emulated in
               Enforcer
               .
     In order to reduce the overhead that this causes (and the
     major performance loss) this program will try to move as much
     from that first 4K as possible and make any of the free
     memory within the first 4K inaccessible.


               Enforcer
                itself also has this logic, but it may be useful
     to be able to run this program as the first program in
     the Startup-Sequence (*AFTER* SetPatch) to try to limit
     the number of things that may use the lower 4K of RAM.

INPUTS
     Just run it...  Can be run from CLI or Workbench

RESULTS
     Any available memory in the lower 4K of CHIP RAM is removed
     plus a special graphics buffer is moved if it needs to be.
     After running this program you may have a bit less CHIP RAM
     than before.  You can run this program as many times as you
     wish since it only moves things if it needs to.

NOTES
     This program will do nothing on systems without a 68040.
     It does not, however, check for the MMU and thus it will
     move the lower 4K even if the CPU is not able to run
               Enforcer
               .

     V39 of the operating system already does have the lowest
     MMU page empty and thus this program will effectively do
     nothing under V39.

SEE ALSO
     "Eloquence is vehement simplicity"
```

```
BUGS
     None.
```

1.9 segtracker

```
                SegTracker - A global SegList tracking utility

SYNOPSIS
     A global tracking utility for disk loaded files including
     libraries and devices.  If placed in the startup-sequence
     right after SetPatch, it will track all disk loaded segments
     (other than those loaded by SetPatch)

FUNCTION
     SegTracker will patch the DOS LoadSeg(), NewLoadSeg(), and UnLoadSeg()
     functions in order to track the SegLists that are loaded.
     SegTracker keeps these seglist stored in a "safe" manner and
     even handles programs which SegList split.

     The first time the program is run, it installs the patches
     and semaphore.  After that point, it just finds the semaphore
     and uses it.

     When SegTracker is installed, it will scan the ROM for ROM modules
     and add their locations to the tracking list such that addresses
     within those modules can be identified.  Note that the offsets
     from the module is based on the location of the module's ROMTAG.
     The NOROM option will prevent this feature from being installed.

     By using SegTracker, it will be possible to better identify
     where
               Enforcer
                hits come from when dealing with libraries
     and devices.  Basically, it is a system-global Hunk-o-matic.

     External programs can then pass in an address to SegTracker
     either via the command line or via the given function pointer
     in the SegTracker semaphore and get back results as to what
     hunk and offset the address is at.

     To work with the function directly, you need to find the
     the semaphore of "SegTracker" using FindSemaphore().
     The structure found will be the following:

     struct  SegSem
     {
     struct  SignalSemaphore seg_Semaphore;
             SegTrack        *seg_Find;
     };

     The function pointer points to a routine that takes an address
     and two pointers to long words for returning the Segment number
     and Offset within the segment.  The function returns the name
     of the file loaded.  Note that you must call this function
     while in Forbid() and then copy the name as the seglist may
```

```
     be UnLoadSeg'ed at any moment and the name string will then
     no longer be in memory.

     typedef char (* __asm SegTrack(register __a0 ULONG Address,
                                    register __a1 ULONG *SegNum,
                                    register __a2 ULONG *Offset));

     The above is for use in C code function pointer prototype
     in SAS/C 5 and 6.

INPUTS
     SHOW/S  - Shows all of the segments being tracked.

     DUMP/S  - Displays all of the segment elements being tracked.

     NOROM/S - Tells segtracker not to scan ROM when it is
               installed, thus not adding ROM addresses to the
               tracking list.

     FIND/M  - Find the hex (in $xxxxx format) address in
               the tracked segments.  Multiple addresses
               can be given.

     Options are not available from Workbench as they require
     the CLI.  However, you can run SegTracker from Workbench
     to install it.

EXAMPLE USAGE


                 Example program
               NOTES
     The earlier this command is run, the better off it will be in
     tracking disk loaded segments.  Under debug usage, you may
     wish to run the command right *AFTER* SetPatch.

     Some things may not call UnLoadSeg() to free their seglists.
     There is no way SegTracker can follow a seglist that is not
     unloaded via the dos.library call to UnLoadSeg().  For this
     reason, SegTracker adds new LoadSeg() segments to the top
     of its list.  This way, if any old segments are still on
     the list but have been unloaded via some other method
     they will not clash with newer segments during the find operation.

     Note that the resident list is one such place where
     UnLoadSeg() is not called to free the seglist.  Thus,
     if something is made resident and then later unloaded
     it will still be listed as tracked by SegTracker.

     In order to support a new feature in CPR, the SegTracker function
     got a "kludge" added to it.  If a segment is found, you can then
     call the function again with the same address but with having
     both pointers point to the same longword of storage.  By doing
     this, the function will now return (in that longword) the
     SegList pointer (CPTR not BPTR) of the file that contains
     the address.  The reason this method was used was so it
     would be compatible with older SegTracker versions.  In older
```

```
     versions you would not get the result you wanted but you would
     also not crash.  See the example above for more details on how
     to use this feature.  The SegTracker FIND option has been
     expanded to include this information.

     Due to the fact that I am working on a design of a new set of
     debugging tools (Enforcer/SegTracker/etc)  I do not wish to
     expand the current SegTracker model in too many ways.

SEE ALSO
     "Quantum Physics:  The Dreams that Stuff is made of." - Michael Sinz
```

1.10 rebootoff

```
                RebootOff - A keyboard reset handler to turn off Enforcer

SYNOPSIS
     This is a simple utility that will turn off
                Enforcer
                 when a
     keyboard reset happens.

FUNCTION
     This utility uses the feature of the A1000/A2000/A3000/A4000
     Amiga systems to turn off
                Enforcer
                 when the user does a
     keyboard reset (ctrl-Amiga-Amiga).  This utility requires that
     your Amiga supports (in hardware) the keyboard reset system.

     The reason this was written was so that
                Enforcer
                 could be
     "quit" just before you reboot your Amiga 3000.  This way
     it will let the kickstart not need to be reloaded and
     thus let utilities such as RAD: work across reboots.  Note
     that this does *not* help in the case where the Amiga reboots
     under software conditions.  It is only for keyboard resets.

INPUTS
     Just run it from either the CLI or Workbench.  It installs
     a handler and exits.  On a keyboard reset, it will turn
                Enforcer
                     off before it lets the reset continue...  (max time of 10  ↩
                        seconds)

RESULTS
     Installs a small reset handler object and task into the system.
     About 3700 bytes needed the first time it is run.

NOTES
     If
                Enforcer
                 is not running, nothing will happen at Reset time.
     If
```

```
                Enforcer
                 can not quit, the reset system will continue to try
     to quit
                Enforcer
                 until the hardware timeout happens...

SEE ALSO
     From the home of the imaginary deadlines:
     "It will take 2i weeks to do that project." - Michael Sinz
```

1.11 debuggers1

```
                To trap a hit requires a number of things to work.

First, the debugger itself must never cause an
                Enforcer
                 hit.
For help on that, see the "DEBUGGERS: NOT CAUSING A HIT"

Second, the debugger must be global.  That is, you must be
able to deal with a task getting a hit that is not the task
under test.  There are a number of simple ways to deal with
this, and I will leave this up to the debugger writer.
(One method will be shown below)

Third, the debugger must start *AFTER*
                Enforcer
                 starts.
If it is started before
                Enforcer
                , the hits will not be
trapped.  (Note that this is not a problem)

A very important point:  The code needs to be fast for
the special case of location 4.  This is shown in the
code below.  It is very important that this be fast.

Note that it is much prefered that debuggers use the
method described below for trapping hits.  It should
be much more supportable this way as any of the tricky
work that may need to be done in the hit processing
will be handled by
                Enforcer
                 itself.  If you wish the
hit decoded, you can capture the
                Enforcer
                 output via a
pipe or some other method (such as
                RAWIO
                ) or you can
leave that issue up to the user.

Now, given the above, the following bits of code can be
used to get the debugger to switch into single-step mode
at the point of the
```

```
                Enforcer
                 hit.  You can also set some
data value here to tell your debugger about this.

;
; The following code is inserted into the bus error vector.
; Make sure you follow the VBR to find the vector.
; Store the old vector in the address OldVector
; Make sure you already have the single-step trap vector
; installed before you install this.  Note that any extra
; code you add in the comment area *MUST NOT* cause a bus
; fault of any kind, including reading of location 4.
;
; This is the common part...
;
EnforcerHit:    ds.l    1                       ; Some private flag
MyTask:         ds.l    1                       ; Task under test
MyExecBase:     ds.l    1                       ; The local copy
OldVector:      ds.l    1                       ; One long word
                ;
                ; Now, if you wish to only trap a specific task,
                ; do the check at this point.  For example, a
                ; simple single-task debugger would do something
                ; like this:
Common:         move.l  a0,-(sp)                ; Save this...
                move.l  MyExecBase(pc),a0       ; Get ExecBase...
                move.l  ThisTask(a0),a0         ; Get ThisTask
                cmp.l   MyTask(pc),a0           ; Are they the same?
                move.l  (sp)+,a0                ; Restore A0 (no flags)
                bne.s   TraceSkip               ; If not my task, skip
                ;
                bset.b  #7,(sp)                 ; Set trace bit...
                ; If you have any other data to set, do it now...
                ; Set as setting the EnforcerHit bit in your data...
                addq.l  #1,EnforcerHit          : Count the hit...
                ;
TraceSkip:      move.l  OldVector(pc),-(sp)     ; Ready to return
                rts
;
; This is the 68020/68030 version...
;
NewVector030:   cmp.l   #4,$10(sp)              ; 68020 and 68030
                beq.s   TraceSkip               ; If AbsExecBase, OK
                bra.s Common      ; Do the common stuff
;
; This is the 68040 version...
;
NewVector040:   cmp.l   #4,$14(sp)              ; 68040
                beq.s   TraceSkip               ; If AbsExecBase, OK
                bra.s Common      ; Do the common stuff
;
; This is the 68060 version...
;
NewVector060:   cmp.l   #4,$08(sp)              ; 68060
                beq.s   TraceSkip               ; If AbsExecBase, OK
                bra.s Common      ; Do the common stuff
```

1.12 debuggers2

```
In order not to cause Enforcer hits, you can do a number
of things.  The easiest is to test the address with the TypeOfMem()
EXEC function.  If TypeOfMem() returns 0, the address is not
in the memory lists.  However, this does not mean it is not a
valid address in all cases.  (ROM, chip registers, I/O boards)
For those cases, you can build a "valid memory access table"
much like Enforcer does.  Here is the code from Enforcer for
the base memory tables:

/*
 * Mark_Address(mmu,start address,length,type)
 */

/*
 * Special case the first page of CHIP RAM
 */
mmu=Mark_Address(mmu,0,0x1000,INVALID | NONCACHEABLE);

/*
 * Map in the free memory
 */
Forbid();
mem=(struct MemHeader *)SysBase->MemList.lh_Head;
while (mem->mh_Node.ln_Succ)
{
  mmu=Mark_Address(mmu,
                   (ULONG)(mem->mh_Lower),
                   (ULONG)(mem->mh_Upper)-(ULONG)(mem->mh_Lower),
                   ((MEMF_CHIP & TypeOfMem(mem->mh_Lower)) ?
                     (NONCACHEABLE | VALID) : (CACHEABLE | VALID)));
  mem=(struct MemHeader *)(mem->mh_Node.ln_Succ);
}
Permit();

/*
 * Map in the autoconfig boards
 */
if (ExpansionBase=OpenLibrary("expansion.library",0))
{
struct  ConfigDev       *cd=NULL;

  while (cd=FindConfigDev(cd,-1L,-1L))
  {
    /* Skip memory boards... */
    if (!(cd->cd_Rom.er_Type & ERTF_MEMLIST))
    {
      mmu=Mark_Address(mmu,
                       (ULONG)(cd->cd_BoardAddr),
                       cd->cd_BoardSize,
                       VALID | NONCACHEABLE);
    }
  }
  CloseLibrary(ExpansionBase);
}
```

```
/*
 * Now for the control areas...
 */
mmu=Mark_Address(mmu,0x00BC0000,0x00040000,VALID | NONCACHEABLE);
mmu=Mark_Address(mmu,0x00D80000,0x00080000,VALID | NONCACHEABLE);

/*
 * and the ROM...
 */
mmu=Mark_Address(mmu,
                 0x00F80000,
                 0x00080000,
                 VALID | CACHEABLE | WRITEPROTECT);

/*
 * If the credit card resource, make the addresses valid...
 */
if (OpenResource("card.resource"))
{
  mmu=Mark_Address(mmu,0x00600000,0x00440002,VALID | NONCACHEABLE);
}

/*
 * If CD-based Amiga (CDTV, A570, etc.)
 */
if (FindResident("cdstrap"))
{
  mmu=Mark_Address(mmu,0x00E00000,0x00080000,VALID | NONCACHEABLE);
  mmu=Mark_Address(mmu,0x00B80000,0x00040000,VALID | NONCACHEABLE);
}

/*
 * Check for ReKick/ZKick/KickIt
 */
if ((((ULONG)(SysBase->LibNode.lib_Node.ln_Name)) >> 16) == 0x20)
{
  mmu=Mark_Address(mmu,
                   0x00200000,
                   0x00080000,
                   VALID | CACHEABLE | WRITEPROTECT);
}
```

1.13 notes1

```
                This is Enforcer V37.  Bryce Nesbitt came up with the original
"Enforcer" that has been instrumental to the improvement in the
quality of software on the Amiga.  The Amiga users and developers
owe him a great deal for this.  Thank you Bryce!  Enforcer V37,
however, is a greatly enhanced and more advanced tool.

Enforcer V37 came about due to a number of needs.  These included
the need for more output options and better performance.  It also
marks the removal of all kludges that were in the older versions.
Also, some future plans required some of these changes...
```

In addition, the complete redesign was needed in order to support the 68040. The internal design of Enforcer is now set up such that CPU/MMU specific code can be cleanly accessed from the general house keeping aspect of the code. The MMU bus error handling is, however, 100% CPU specific.

Since AbsExecBase is in low memory, reads of this address are slower with Enforcer running. Caching AbsExecBase locally is highly recommended since it is in CHIP memory and on systems with FAST memory, it will be faster to access the local cached value. (In addition to the performance increase when running Enforcer) Note that doing many reads of location 4 will hurt interrupt performance.

When the Amiga produces an ALERT, EXEC places some magic numbers into some special locations in low memory. The exact pattern changes between versions of the operating system.

Enforcer will patch the EXEC function ColdReboot() in an attempt to "get out of the way" when someone tries to reboot the system. Enforcer will clean up as much as possible the MMU tables and then call the original LVO. When Enforcer is asked to quit, it will check to make sure it can remove itself from this LVO. If it can not, it will not quit at that time. If run from the shell, it will display a message saying that it tried but could not exit. Enforcer will continue to be active and you can try later to deactivate it.

Enforcer will also patch the EXEC function Alert() in an attempt to provide better tracking of other events in the system. It is also patched such that dead-end alerts will correctly reset the system and be displayed. With this patch in place, the normal alerts will not be seen but will be replaced by the Enforcer output shown above. See

LawBreaker

for a more complete example of this.

Other notes:

68020 Notes

68030 Notes

68040 Notes

68060 Notes

BridgeBoard

1.14 notes2

The 68020 does not have a built-in MMU but has a co-processor feature that lets an external MMU be connected. Enforcer MMU code is designed for use with 68851 MMU. This is the some-what 68030 compatible MMU by Motorola. Enforcer uses the same code for both the 68030 and the 68020/68851. For this reason, 68020/68851 users

should see the
68030 NOTES
section.

1.15 notes3

The 68030 uses cycle/instruction continuation and will supply the data on reads and ignore writes during an access fault rather than let the real bus cycle happen. This means that on a fault caused by MMU tables, no bus cycle to the fault address will be generated. (For those of you with analyzers)

In some cases, the 68030 will have advanced the Program Counter past the instruction by the time the access fault happens. This is usually only on WRITE faults. For this reason, the PC may either point at the instruction that caused the fault or just after the instruction that caused the fault. (Which could mean that it is pointing to the middle of the instruction that caused the fault.)

Note that there is a processor called 68EC030. This processor has a disabled or defective MMU. However, it may function well enough for Enforcer to think it has a fully functional MMU and thus Enforcer will attempt to run. However, even if it looks like the MMU is functioning, it is not fully operational and thus may cause strange system activity and even crashes. Do not assume that Enforcer is safe to use on 68EC030 systems.

1.16 notes4

Enforcer, on the 68040, *requires* that the
68040.library
be
installed and it requires an MMU 68040 CPU. The 68EC040 does not have a MMU. The 68LC040 does have an MMU and is supported. Enforcer will work best in a system with the
68040.library
37.10 or better
but it does know how to deal with systems that do not have that version.

Due to the design of the 68040, Enforcer is required to do a number of things differently. For example, the MMU page size can only be either 8K or 4K. This means that to protect the low 1K of memory, Enforcer will end up having to mark the first 4K of memory as invalid and emulate the access to the 3K of that memory that is valid. For this reason Enforcer moves a number of possible structures from the first 4K of memory to higher addresses. This means that the system will continue to run at a reasonable speed. The first time Enforcer is run it may need to allocate memory for these structures that it will move. Enforcer can never return this memory to the system.

In addition to the fact that the 68040 MMU table size is different, the address fault handling is also different. Namely, the 68040 can only rerun the cycle and not continue it like the 68030. This means that on a 68040, the page must be made available first and then made unavailable. To make this work, Enforcer will switch the instruction that caused the error into trace mode and let it run with a special MMU setup. When the trace exception comes in, the MMU is set back to the way it was. Enforcer does its best to keep debuggers working. Note, however, that the interrupt level during a trace of a READ will end up being set to 7. This is to prevent interrupts from changing the order of trace/MMU table execution. The level will be restored to the original state before continuing. Since T0 mode tracing is also supported, there are also some changes in the way it operates. T0 mode tracing is defined, on the 68040, to cause a trace whenever the instruction pipeline needed to be reloaded. While on the 68020/030 processors this was normally only for the branch instructions, in the 68040 this includes a large number of other instructions. (Including NOP!) Anyway, if an Enforcer hit happens while in T0 tracing mode, the trace will happen even on instructions that normally would not cause a T0 mode trace. Since this may actually help in debugging and because it was not possible to do anything else, this method of operation is deemed acceptable.

Another issue with the 68040 is that WRITE faults happen *after* the instruction has executed. (Except for MOVEM) In fact, it is common for the 68040 to execute one or more extra instructions before the WRITE fault is executed. This design makes the 68040 much faster, but it also makes the Program Counter value that Enforcer can report for the fault much less likely to be pointing to the instruction that caused it. The worst cases are sequences such as a write fault followed by a branch instruction. In these cases, the branch is usually already executed before the write fault happens and thus the PC will be pointing to the target of the branch. There is nothing that can be done within Enforcer to help out here. You will just need to be aware of this and deal with it as best as possible.

Along with the above issue, is the fact that since a write fault may be delayed, a read fault may happen before the write fault shows up. Internally, enforcer does not do special processing for these and they will not show up. Since another hit was happening anyway, it is felt that it is best to just not report the hit. Along the same lines, the hit generated from a MOVEM instruction may only show as a single hit rather than 1 for each register moved.

On the Amiga, MOVE16 is not supported 100%. Causing an Enforcer hit with a MOVE16 will cause major problems and maybe cause Enforcer or your task to lock. Since MOVE16 is not supported, this is not a major issue. Just watch out if you are using this 68040 instruction. (Also, watch out for the 68040 CPU bug with MOVE16)

The functions CachePreDMA(), CachePostDMA(), and CacheControl() are patched when the 68040 MMU is turned on by Enforcer. These functions are patched such the issues with DMA and the 68040 COPYBACK data caches are addressed. The

68040.library

normally

deals with this, however since Enforcer turns on the MMU, the method
of dealing with it in the older
                68040.library
                 will not work. For this
reason, Enforcer will patch these and implement the required fix for
when the MMU is on. When Enforcer is asked to exit, it will check
if it can remove itself from these functions. If it can not, it
will ignore the request to exit. If Enforcer was run from the CLI,
it will print a message saying that it can not exit when the attempt
is made.

These patches are not needed in V37.30 or better of
                68040.library
                .

1.17 notes6

                Enforcer, on the 68060, *requires* that the
                68060.library
                 be
installed. Due to the fact that various possible
                68060.library
                versions may exist, Enforcer tries to not second guess it.
Thus, Enforcer assumes that the
                68060.library
                 has all of the
same functionality as V37.30 or better of the
                68040.library
                .

It turns out that some of the 68060 libraries do not have the
same functionality of the
                68060.library
                . One common library
has elected not to handle Pre/Post DMA MMU table operations
when Enforcer installs its MMU table. This results in some
DMA/Cache interactions. Enforcer can not work around this
problem safely. If you happen to have a 68060.library with
version 2.1 (19.07.96) you may be able to patch it to not
have this problem. At offset $09BE there should be the
4-byte sequence $20 6D 00 04 Changing this to $4E 7A 88 06
will let it handle Enforcer's MMU tables too. (The same
patch may work in other versions of the library)

For implementers of
                68060.library
                , see my notes as to what
had to be done in
                68040.library
                 for correct operation.
Note that this does not mean that Enforcer needs this. The Amiga
system needs this to operate correctly. Enforcer just may
cause these problems to become more evident. The notes are only
in the AmigaGuide version of the Enforcer documentation.

The 68060 exception model is full-restart, which means that
all instructions are re-run. Both reads *and* writes.
This means that Enforcer can not tell you what the data
that would be written is, unlike the 68040 and earlier CPUs.
So, the output for a write will not include the data that
was to be written. This does mean that faults happen
before the instruction is executed (usually) and thus the
reported PC will be more exact. This restart model also
means that if an real bus-fault happens, Enforcer will be
unable to do much other than let it happen. (The same is
true for reads on the 68040) Enforcer maps all addresses
as either valid based on system configuration or invalid.
This is so that no address should cause a bus fault unless
the system configuration is incorrect and an address that
was marked valid actually causes a fault.

Be sure to read the
                68040 notes
                 as the 68060 is a superset
of much of these notes.

Due to the complexity of emulating access to lower memory and
the fact that the 68060 was introduced well after V39 kickstart,
it is highly recommended that you use V39 or better with 68060 CPUs.
This mainly has to deal with lower 4K of memory. As of V38 of
exec.library, 68040/68060 processors would map out the lower 4K
of RAM rather than just 1K. This was required since the newer
CPUs did not have page sizes less than 4K.

It turns out that some 68060 CPU cards also have other hardware
on them. This is not a problem, unless this hardware does not
autoconfigure. Enforcer needs to know about hardware in the
system so it can map the MMU to that hardware. If the hardware
is not true Amiga AutoConfig (as in no expansion.library entry)
then Enforcer has no way of knowing it exists.

A common location for such hardware control registers to be
placed is in the reserved $00F00000 address range (known as
F-Space). This 512K space was reserved for future Kickstart
growth. It also has some magic in it so that you can wedge
special startup routines there for things like 68060 cards.
At least one vendor is doing all of this correctly and
has even made sure that expansion.library knows about the
hardware that is located in the $00F00000 address range.

If, when running Enforcer, your machine does lock up *and*
when you run Enforcer with the VERBOSE option it does not
say that the $00F00000 address range is a "board address"
then you may wish to try the FSPACE option to see if this
is the reason. If this does not fix the problem, you more
than likely have either a real bug or some other non-AutoConfig
hardware in your system.

1.18 cpu_library

The original concept behind the 68040.library was to move certain system support routines into CPU specific libraries. The goal was to have the Kickstart ROM make the system boot reliably but not to worry about instruction emulation, complex cache control, or address space manipulation.

This became an issue back when I was first working on prototype 68040 CPUs at Commodore. We already knew that the FPU (floating point unit) was going to cause problems since many of the more complex instructions were now going to need to be emulated in software. Plus, the complexity of a copy-back cache meant that some interesting cache control issues were going to come up.

The first cut at the 68040.library that became public was V37.10. (Not counting version sent to developers during beta) This library had the optimized FPU routines (updated from those that Motorola wrote) plus a number of critical patches to the system to enable more stable operation with the caches turned on and with copyback mode enabled.

These
patches
were mainly to cover cases where code had not been flushing caches after having generated or loaded new code. The cases were found by looking at example source and shipping products that we needed to keep running and finding the safest and least impact way of supporting them. If you are writing a new CPU library, for example 68060.library, please read about these
patches
along with the section on
COPYBACK mode and DMA
.

After working the V37.10 version of the library for a while we noticed a few very rare crashes when doing DMA and CPU processing of related data. In fact, it was when SAS/C started to do ASYNC source file loading that it became most noticeable. (When I say that, I mean that once a week or so a 68040 Amiga with DMA SCSI would crash while compiling.)

After thinking about what is really going on, I determined that there was a rare but very dangerous interaction between DMA and COPYBACK caches. The interaction and the fix is described in detail in the section on
COPYBACK mode and DMA
. The fix was implemented in V37.30 of 68040.library and Enforcer does a crude version of this fix if a pre-37.30 of 68040.library is found.

In addition, V37.30 also implemented a fix for problems related to hardware devices that live in Zorro-III space. This fix is directly related to the
COPYBACK mode and DMA
fix.
Details of why this is needed will not be described here. I wrote a number of AmigaMail articles on the issues of hardware

```
and 68040 systems.  The end result is that the MMU must be used
to make 68040 (and 68060) systems operate correctly.  68040.library
does this.  (And 68060.library will also need to do this.)
The reason is that non-cacheable hardware addresses must be
mapped by the MMU as non-cached otherwise very bad things
will happen.

The 68060.library concept continues this design but Commodore
was falling apart and I left before prototype 68060 CPUs were
available.  As such, other people have produced 68060.library
versions that hopefully have incorporated all of the issues
learned in the development of the 68040.library.  Since there
is more than one 68060.library, Enforcer can not know if to
patch it or not.  As such, Enforcer assumes that the 68060.library
has correctly handled the issues that V37.30 of 68040.library.
```

1.19 cpu_patches

```
               The following patches are made by
               68040.library
                in order to
help keep software working.  The descriptions and the code
to the patch are given here, however the code is not 100%
complete and requires that the programmer put this into the
correct places in the CPU library initialization code,
after the correct checks as to if the library should initialize.

Note that on the CachePreDMA and CachePostDMA patches, these
functions were design specifically for this purpose (along
with virtual memory mapping, which I did not have the chance
to complete before Commodore started to fall to pieces.)

Note also that the CachePreDMA/CachePostDMA patches are those
that are described in the
               COPYBACK mode and DMA
                link.
```

```
        ;
        ; CachePreDMA
        ;
        ; This adds the special patch to make 68040 and DMA devices
        ; work with CopyBack modes turned on...
        ;
                PRINTF  <'Installing CachePreDMA() patch'>
                lea     NewCachePreDMA(pc),a0
                move.l  a0,d0                   ; Get pointer to new routine
                move.l  a6,a1                   ; Get library to be patched
                move.w  #_LVOCachePreDMA,a0     ; Get LVO offset...
                CALLSYS SetFunction             ; Install new code...
        ;
        ; CachePostDMA
        ;
        ; This adds the special patch to make 68040 and DMA devices
        ; work with CopyBack modes turned on...
        ;
```

```
        PRINTF  <'Installing CachePostDMA() patch'>
        lea     NewCachePostDMA(pc),a0
        move.l  a0,d0                   ; Get pointer to new routine
        move.l  a6,a1                   ; Get library to be patched
        move.w  #_LVOCachePostDMA,a0    ; Get LVO offset...
        CALLSYS SetFunction             ; Install new code...
;
; CacheControl
;
; It fixes the return values from CacheControl to correctly return
; the BURST ENABLE bit if the cache bit is on.  (68040 bursts all
; caches)  It also deals with the cache settings vs DMA.
;
        PRINTF  <'Installing CacheControl() patch'>
        lea     NewCacheControl(pc),a0
        move.l  a0,d0                   ; Get pointer to new routine
        move.l  a6,a1                   ; Get library to be patched
        move.w  #_LVOCacheControl,a0    ; Get LVO offset...
        CALLSYS SetFunction             ; Install new code...
;
; AddLibrary
;
; This fixes programs/libraries that do not use
; MakeLibrary() to generate the library structure.
;
        PRINTF  <'Installing AddLibrary() patch'>
        lea     NewAddLibrary(pc),a0
        move.l  a0,d0                   ; Get pointer to new routine
        move.l  a6,a1                   ; Get library to be patched
        move.w  #_LVOAddLibrary,a0      ; Get LVO offset...
        CALLSYS SetFunction             ; Install new code...
        lea     OldAddLibrary(pc),a0    ; Get storage slot...
        move.l  d0,(a0)                 ; Save old code address...
;
; CloseLibrary
;
; This fixes arp.library on 68040 machines since it
; places some code onto the stack and then runs it to close
; a library...
;
        PRINTF  <'Installing CloseLibrary() patch'>
        lea     NewCloseLibrary(pc),a0
        move.l  a0,d0                   ; Get pointer to new routine
        move.l  a6,a1                   ; Get library to be patched
        move.w  #_LVOCloseLibrary,a0    ; Get LVO offset...
        CALLSYS SetFunction             ; Install new code...
        lea     OldCloseLibrary(pc),a0  ; Get storage slot...
        move.l  d0,(a0)                 ; Save old code address...
;
; AddDevice
;
; This fixes programs/libraries that do not use
; MakeLibrary() to generate the library structure.
;
        PRINTF  <'Installing AddDevice() patch'>
        lea     NewAddDevice(pc),a0
        move.l  a0,d0                   ; Get pointer to new routine
```

```
        move.l  a6,a1                   ; Get library to be patched
        move.w  #_LVOAddDevice,a0       ; Get LVO offset...
        CALLSYS SetFunction             ; Install new code...
        lea     OldAddDevice(pc),a0     ; Get storage slot...
        move.l  d0,(a0)                 ; Save old code address...
;
; AddResource
;
; This fixes programs/libraries that do not use
; CacheClearU() after generating the resource.
;
        PRINTF  <'Installing AddResource() patch'>
        lea     NewAddResource(pc),a0
        move.l  a0,d0                   ; Get pointer to new routine
        move.l  a6,a1                   ; Get library to be patched
        move.w  #_LVOAddResource,a0     ; Get LVO offset...
        CALLSYS SetFunction             ; Install new code...
        lea     OldAddResource(pc),a0   ; Get storage slot...
        move.l  d0,(a0)                 ; Save old code address...
;
; AddTask
;
; This fixes programs that install the code into memory
; without flushing the caches.  This happens to also fix
; the most common problem like this:  Fake seglist generation
; for calls to CreateProc()  (A trick needed in pre-2.0 days)
;
        PRINTF  <'Installing AddTask() patch'>
        lea     NewAddTask(pc),a0
        move.l  a0,d0                   ; Get pointer to new routine
        move.l  a6,a1                   ; Get library to be patched
        move.w  #_LVOAddTask,a0         ; Get LVO offset...
        CALLSYS SetFunction             ; Install new code...
        lea     OldAddTask(pc),a0       ; Get storage slot...
        move.l  d0,(a0)                 ; Save old code address...
;
; AddIntServer
;
; Once again, people had generated code that was then
; installed as a server for the interrupts.  This should
; be a very minor hit since very few call AddIntServer()
;
        PRINTF  <'Installing AddIntServer() patch'>
        lea     NewAddIntServer(pc),a0
        move.l  a0,d0                   ; Get pointer to new routine
        move.l  a6,a1                   ; Get library to be patched
        move.w  #_LVOAddIntServer,a0    ; Get LVO offset...
        CALLSYS SetFunction             ; Install new code...
        lea     OldAddIntServer(pc),a0  ; Get storage slot...
        move.l  d0,(a0)                 ; Save old code address...
;
; SetIntVector
;
; Same issues as AddIntServer above...
;
        PRINTF  <'Installing SetIntVector() patch'>
        lea     NewSetIntVector(pc),a0
```

```
                move.l  a0,d0                   ; Get pointer to new routine
                move.l  a6,a1                   ; Get library to be patched
                move.w  #_LVOSetIntVector,a0    ; Get LVO offset...
                CALLSYS SetFunction             ; Install new code...
                lea     OldSetIntVector(pc),a0  ; Get storage slot...
                move.l  d0,(a0)                 ; Save old code address...
        ;
        ; Now, patch input.device so that a IND_ADDHANDLER will flush
        ; the caches.  (Arg!  But this is a big payoff)
        ;
                PRINTF  <'Installing input.device/IND_ADDHANDLER patch'>
        ; First, we need to find input.device on the list
                lea     DeviceList(a6),a0       ; Get list structure
                lea     InputName(pc),a1        ; Get input.device string
                CALLSYS FindName                ; Find it on the list
                move.l  d0,a1                   ; This is the device we patch
                tst.l   d0                      ; Check if NULL
                beq.s   NoINDPatch              ; If NULL, no Patch...
        ;
        ; We patch BeginIO in input.device to check for ADDHANDLER
        ; as the command.  Since many tools copy up code for use as
        ; input handlers and just ADDHANDLER them, this will fix
        ; all of those caching issues.
        ;
                lea     NewBeginIO(pc),a0       ; Get new code
                move.l  a0,d0                   ; address for SetPatch...
                move.w  #DEV_BEGINIO,a0         ; LVO offset for BeginIO...
                CALLSYS SetFunction             ; Install it...
                lea     OldBeginIO(pc),a0       ; Save old code address
                move.l  d0,(a0)                 ; ...for the patch.

        ; continue with our work...
*
*****************************************************************************
*
********** Now for the implementation:
*
InputName:      dc.b    'input.device',0        ; For patching input.device
                CNOP    0,2
*****************************************************************************
*
* This is the MMU frame.  NULL on systems without MMU setup.
*
MMUFrame:       dc.l    0       ; MMU frame...
*
*****************************************************************************
*
* AddLibrary patch code
*
OldAddLibrary:  dc.l    0                       ; Storage for old
NewAddLibrary:  move.l  OldAddLibrary(pc),-(sp) ; Set so RTS to old code
                move.l  a1,-(sp)                ; Only A1 is needed...
                CALLSYS CacheClearU             ; Clear the caches
                move.l  (sp)+,a1                ; Restore
                rts
*
*****************************************************************************
```

```
*
        IFNE    ARP_FIX
*
* CloseLibrary patch code
*
OldCloseLibrary:
                dc.l    0                       ; Storage for old
NewCloseLibrary:
                move.l  OldCloseLibrary(pc),-(sp)       ; Set so RTS to old
                move.l  a1,-(sp)                ; Only A1 is needed...
                CALLSYS CacheClearU             ; Clear the caches
                move.l  (sp)+,a1                ; Restore
                rts
*
        ENDC
*
******************************************************************************
*
* AddDevice patch code
*
OldAddDevice:   dc.l    0                       ; Storage for old
NewAddDevice:   move.l  OldAddDevice(pc),-(sp)  ; Set so RTS to old code
                move.l  a1,-(sp)                ; Only A1 is needed...
                CALLSYS CacheClearU             ; Clear the caches
                move.l  (sp)+,a1                ; Restore
                rts
*
******************************************************************************
*
* AddResource patch code
*
OldAddResource: dc.l    0                       ; Storage for old
NewAddResource: move.l  OldAddResource(pc),-(sp) ;Set so RTS to old code
                move.l  a1,-(sp)                ; Only A1 is needed...
                CALLSYS CacheClearU             ; Clear the caches
                move.l  (sp)+,a1                ; Restore
                rts
*
******************************************************************************
*
* AddTask patch code
*
OldAddTask:     dc.l    0                       ; Storage for old
NewAddTask:     move.l  OldAddTask(pc),-(sp)    ; Set so RTS to old code
                move.l  a1,-(sp)                ; Only A1 is needed...
                CALLSYS CacheClearU             ; Clear the caches
                move.l  (sp)+,a1                ; Restore a1
                rts
*
******************************************************************************
*
* AddIntServer patch code
*
OldAddIntServer:
                dc.l    0                       ; Storage for old
NewAddIntServer:
                move.l  OldAddIntServer(pc),-(sp)       ; Set so RTS to old code
```

```
                movem.l d0/a1,-(sp)             ; Only D0/A1 are needed...
                CALLSYS CacheClearU             ; Clear the caches
                movem.l (sp)+,d0/a1             ; Restore a1
                rts
*
******************************************************************************
*
* SetIntVector patch code
*
OldSetIntVector:
                dc.l    0                       ; Storage for old
NewSetIntVector:
                move.l  OldSetIntVector(pc),-(sp)       ; Set so RTS to old code
                movem.l d0/a1,-(sp)             ; Only D0/A1 are needed...
                CALLSYS CacheClearU             ; Clear the caches
                movem.l (sp)+,d0/a1             ; Restore a1
                rts
*
******************************************************************************
*
* input.device BeginIO patch code to trap/flush on IND_ADDHANDLER
*
OldBeginIO:     dc.l    0                       ; Storage for old
NewBeginIO:     move.l  OldBeginIO(pc),-(sp)    ; Set so RTS to old code
                ; Now, check if it is IND_ADDHANDLER
                cmp.w   #IND_ADDHANDLER,IO_COMMAND(a1)
                bne.s   Not_ADDHANDLER          ; If not ADDHANDLER, skip...
                movem.l a1/a6,-(sp)             ; save these
                move.l  _AbsExecBase,a6         ; Get EXECBASE
                CALLSYS CacheClearU             ; Clear the caches
                movem.l (sp)+,a1/a6             ; Restore...
Not_ADDHANDLER: rts
*
******************************************************************************
*
*   NAME
*       CachePostDMA - Take actions after to hardware DMA  (V37)
*
*   SYNOPSIS
*       CachePostDMA(vaddress,&length,flags)
*                    a0       a1      d0
*
*       CachePostDMA(APTR,LONG *,ULONG);
*
*   FUNCTION
*       Take all appropriate steps after Direct Memory Access (DMA).  This
*       function is primarily intended for writers of DMA device drivers.  The
*       action will depend on the CPU type installed, caching modes, and the
*       state of any Memory Management Unit (MMU) activity.
*
*       As implemented
*               68000 - Do nothing
*               68010 - Do nothing
*               68020 - Do nothing
*               68030 - Flush the data cache
*               68040 - Flush matching areas of the data cache
*               ????? - External cache boards, Virtual Memory Systems, or
```

```
*                       future hardware may patch this vector to best emulate
*                       the intended behavior.
*                       With a Bus-Snooping CPU, this function my end up
*                       doing nothing.
*
*   INPUTS
*       address - Same as initially passed to CachePreDMA
*       length  - Same as initially passed to CachePreDMA
*       flags   - Values:
*                       DMA_NoModify - If the area was not modified (and
*                       thus there is no reason to flush the cache) set
*                       this bit.
*
*   SEE ALSO
*       exec/execbase.i, CachePreDMA, CacheClearU, CacheClearE
*
******************************************************************************
*
* Replace CachePostDMA to handle the 68040 CopyBack vs DMA problem...
*
* This is a real nasty problem:  We have to watch out for DMA to memory
* while the CPU is accessing memory within the same cache line.
* This all mixes in with the CacheControl function since what we
* will do is to have PreDMA turn off CopyBack mode and PostDMA
* turn it back on...  (only if needed as CacheControl() may have
* been called too...  arg!!!)  If we have an MMU we will play
* with the MMU tables...
*
NewCachePostDMA:
                btst.l  #DMAB_ReadFromRAM,d0    ; Check if READ DMA
                bne.s   dma_Caches              ; If so, skip...
                move.l  a0,d1                   ; Get address...
                or.l    (a1),d1                 ; or in length...
                and.b   #$0F,d1                 ; Check for non-aligned...
                beq.s   dma_Caches              ; Don't count if aligned...
*
* Now, we check if we can do the MMU trick...
*
                move.l  MMUFrame(pc),d1         ; Get MMU frame
                bne.s   On_MMU_Way              ; Do it the MMU way...
*
                lea     Nest_Count(pc),a1       ; We trash a1...
                subq.l  #1,(a1)                 ; Subtract the nest count...
                bra.s   dma_Caches              ; Do the DMA work...
*
* Ok, so we have an MMU and need to deal with turning on the pages
*
On_MMU_Way:     move.l  a0,-(sp)                ; (result, fake)
                move.l  a4,-(sp)                ; Save a4
                lea     On_MMU_Page(pc),a4      ; Address of Cache ON code
                bra.s   MMU_Way                 ; Do the common code...
*
******************************************************************************
*
*   NAME
*       CachePreDMA - Take actions prior to hardware DMA  (V37)
*
```

```
*   SYNOPSIS
*       paddress = CachePreDMA(vaddress,&length,flags)
*       d0                     a0       a1      d0
*
*       APTR CachePreDMA(APTR,LONG *,ULONG);
*
*
*   INPUTS
*       address - Base address to start the action.
*       length  - Pointer to a longword with a length.
*       flags   - Values:
*                       DMA_Continue - Indicates this call is to complete
*                       a prior request that was broken up.
*
*   RESULTS
*       paddress- Physical address that coresponds to the input virtual
*                 address.
*       &length - This length value will be updated to reflect the contiguous
*                 length of physical memory present at paddress.  This may
*                 be smaller than the requested length.  To get the mapping
*                 for the next chunk of memory, call the function again with
*                 a new address, length, and the DMA_Continue flag.
*
******************************************************************************
*
* Replace CachePreDMA to handle the 68040 CopyBack vs DMA problem...
*
NewCachePreDMA:
                btst.l  #DMAB_Continue,d0       ; Check if we are continue mode
                bne.s   ncp_Continue            ; Skip the Continue case...
                btst.l  #DMAB_ReadFromRAM,d0    ; Check if READ DMA
                bne.s   ncp_Continue            ; Skip if read...
                move.l  a0,d1                   ; Get address...
                or.l    (a1),d1                 ; or in length...
                and.b   #$0F,d1                 ; Check of non-alignment
                beq.s   ncp_Continue            ; Don't count if aligned
*
* Now, we check if we can do the MMU trick...
*
                move.l  MMUFrame(pc),d1         ; Get MMU frame...
                bne.s   Off_MMU_Way             ; If so, do MMU way...
*
                lea     Nest_Count(pc),a1       ; Get a1...
                addq.l  #1,(a1)                 ; Nest this...
ncp_Continue:   move.l  a0,d0                   ; Get result...
dma_Caches:     move.l  d0,-(sp)                ; Save result...
ncp_DoWork:     moveq.l #0,d0                   ; Clear bits
                moveq.l #0,d1                   ; Clear mask
                bsr.s   NewCacheControl         ; Do the cache setting/clear
                move.l  (sp)+,d0                ; Restore d0
                rts                             ; Return...
*
* Ok, so we have an MMU and need to deal with turning off the pages
* given...
*
Off_MMU_Way:    move.l  a0,-(sp)                ; Save result
                move.l  a4,-(sp)                ; Save a4
```

```
                lea     Off_MMU_Page(pc),a4     ; Address of Cache OFF code
*
MMU_Way:        move.l  a5,-(sp)                ; Save a5
                lea     Do_MMU_Way(pc),a5       ; Get addres of code
                CALLSYS Supervisor              ; Do it...
                move.l  (sp)+,a5                ; Restore a5
                move.l  (sp)+,a4                ; Restore a4
                bra.s   ncp_DoWork              ; Return with result on stack
*
******************************************************************************
*
*   NAME
*       CacheControl - Instruction & data cache control
*
*   SYNOPSIS
*       oldBits = CacheControl(cacheBits,cacheMask)
*       D0                     D0        D1
*
******************************************************************************
*
* This new cache control completely replaces the ROM version.
* There is no reason to support the other chips here.
* If there was an external cache, it would be handled here...
*
NewCacheControl:        movem.l d2/d3,-(sp)             ; Save...
                        and.l   d1,d0           ; Destroy irrelevant bits
                        not.l   d1              ; Change mask to preserve bits
                        move.l  a5,a1           ; Save a5...
                        lea.l   ncc_Sup(pc),a5  ; Code that runs in supervisor
                        CALLSYS Supervisor      ; Do it...
                        move.l  d3,d0           ; Set return value...
                        movem.l (sp)+,d2/d3     ; Restore...
                        rts                     ; Done...
*
* Some storage for these features...
*
                cnop    0,4                     ; Long align them...
Base_Cache:     dc.l    0                       ; Base cache settings...
Nest_Count:     dc.l    0                       ; Nest count of the cache...
*
*       d0-mask d1-bits d2-scratch d3-result
*       a1-Saved a5...
*
ncc_Sup:        or.w    #$0700,SR       ;DISABLE
                movec   CACR,d2         ; Get cache control register
                and.l   #CACRF_040_ICache!CACRF_040_DCache,d2 ;!BIT ASUMPTIONS!
*                               ;10987654321098765432109876543210
*                               ;D00000000000000I000000000000000
                swap    d2      ;I00000000000000D000000000000000 CACRF_040
                ror.w   #8,d2   ;I00000000000000000000000D0000000 CACRF_040
                rol.l   #1,d2   ;000000000000000000000000D0000000I CACRF_040
*
* Add in the "ghost" cache setting...
*
                or.l    Base_Cache(pc),d2       ; Base cache mode...
*
* Now, set the burst modes too...  (040 always bursts the cache)
```

```
*
                move.l  d2,d3           ; Move it over...
                rol.l   #4,d3           ; Shift cache info into burst info
                or.l    d3,d2           ; Store with the burst bits as needed
*
* Mirror the Data Cache into the CopyBack bit...
*
                btst.l  #CACRB_EnableD,d2
                beq.s   ncc_NoCB        ; If no data cache, no copyback...
                bset.l  #CACRB_CopyBack,d2
ncc_NoCB:       move.l  d2,d3           ; Set result: old cache settings
*
* Now, mask out what we want to change and change it...
*
                and.l   d1,d2           ; Mask out what we want to change...
                or.l    d0,d2           ; Change those...
*
* Now store the "asked for" new setting in Base_Cache...
*
                move.l  #CACRF_EnableD,d0       ; Get data cache...
                and.l   d2,d0                   ; Mask it...
                move.l  d0,Base_Cache-ncc_Sup(a5)       ; Store it...
*
* Now, check if data cache should be off due to DMA...
*
                tst.l   Nest_Count(pc)          ; Check for PreDMA nest
                beq.s   ncc_Normal              ; If not, we just do it...
                bclr.l  #CACRB_EnableD,d2       ; If set, we don't do DCache
ncc_Normal:
*
* Now, take the 68030 settings and go back to 68040 settings...
*
*                               ;10987654321098765432109876543210
*                               ;XXXXXXXXXXXXXXXXXXXXXXXXDXXXXXXXI
                ror.l   #1,d2   ;IXXXXXXXXXXXXXXXXXXXXXXXDXXXXXXX CACRF_040
                rol.w   #8,d2   ;IXXXXXXXXXXXXXXXDXXXXXXXXXXXXXXX CACRF_040
                swap    d2      ;DXXXXXXXXXXXXXXXIXXXXXXXXXXXXXXX CACRF_040
                and.l   #CACRF_040_ICache!CACRF_040_DCache,d2 ;!BIT ASUMPTIONS!
*
* All we need to do is play with the internal cache settings...
*
ncc_NoECache:   nop                     ;68040 BUG KLUDGE. Mask 14D43B
                cpusha  BC              ; Push data and instruction cache...
                nop                     ;68040 BUG KLUDGE. Mask 14D43B
                movec   d2,CACR         ; Set the new cache control reg bits
                nop                     ;68040 BUG KLUDGE. Mask 14D43B
                move.l  a1,a5           ; Restore a5...
                rte                     ;rte restores SR
*
*******************************************************************************
*
* The magic for MMU based Pre/PostDMA calls...
*
* This routine is the general page manager.  It will deal with the
* start and end pages as needed.
* Input:        a4 - Routine to manipulate the page
*               d1 - MMU Frame
```

```
*               a0 - Start address
*               *a1- Size
*               a5 - Scrap...
*               d0 - SCrap...
*               a6 - ExecBase
*
Do_MMU_Way:     move.l  d1,a5                   ; Get MMU Frame into a5...
                move.l  a0,d0                   ; Get start address...
                move.l  d0,-(sp)                ; Save start address...
                add.l   (a1),d0                 ; Calculate end address...
                bsr.s   Do_MMU_d0               ; d0 is address; do it...
                move.l  (sp)+,d0                ; Get start again...
                bsr.s   Do_MMU_d0               ; d0 is address; do it...
                rte                             ; We be done...
*
* Ok, so now we are called as follows:
*
*       a6 - ExecBase
*       a5 - MMU Frame pointer
*       a4 - Routine to manipulate the page
*       d0 - Address which needs protection
*       d1 - Scrap
*       a0 - Scrap
*       a1 - Scrap
*
*       a0/a1/d0/d1 may all be trashed :-)
*
Do_MMU_d0:      moveq.l #$0F,d1                 ; Mask...
                and.l   d0,d1                   ; Check for cache line address
                beq.s   Do_MMU_RTS              ; If on line address, no-op.
*
                move.l  d0,-(sp)                ; Save address...
                bfextu  d0{1:19},d0             ; Get page number
                move.l  mmu_NestCounts(a5),d1   ; Point at list head
dmd_Loop:       move.l  d1,a0                   ; Get into address register
                move.l  (a0),d1                 ; Get Next pointer
                beq.s   dmd_NoFind              ; Did not find it...
                cmp.l   nc_Low(a0),d0           ; Are we above low?
                bcs.s   dmd_Loop                ; Not this one...
                cmp.l   nc_High(a0),d0          ; Are we below limit?
                bhi.s   dmd_Loop                ; Not this one...
                sub.l   nc_Low(a0),d0           ; Subtract low...
                lea     nc_Count(a0),a1         ; Point at start of space
                add.l   d0,a1                   ; Adjust for page offset
                add.l   d0,a1                   ; (*2 since they are words)
                move.l  (sp)+,d0                ; Restore address...
*
                movec.l urp,a0                  ; Get ROOT pointer...
                bfextu  d0{0:7},d1              ; Get the root index...
                asl.l   #2,d1                   ; *4
                add.l   d1,a0                   ; Add to root pointer...
                move.l  (a0),d1                 ; Get page entry
                and.w   #$FE00,d1               ; Mask into the page table
                move.l  d1,a0                   ; Store pointer...
                bfextu  d0{7:7},d1              ; Get the pointer index...
                asl.l   #2,d1                   ; *4
                add.l   d1,a0                   ; Add to table pointer...
```

```
                move.l  (a0),d1                 ; Get page entry...
                and.w   #$FF00,d1               ; Mask to the pointer...
                move.l  d1,a0                   ; Put into address register...
                bfextu  d0{14:6},d1             ; Get index into page table
                asl.l   #2,d1                   ; *4
                add.l   d1,a0                   ; a0 now points at the page...
                move.l  (a0),d1                 ; Get page entry...
                btst.l  #0,d1                   ; Check if bit 0 is set...
                bne.s   dmd_skip                ; If set, we are valid...
                bclr.l  #1,d1                   ; Check if indirect...
                beq.s   dmd_skip                ; If not indirect, A0 is valid
                move.l  d1,a0                   ; a0 is now the page entry...
dmd_skip:       jmp     (a4)                    ; Ok, so now do the page work
*
dmd_NoFind:     move.l  (sp)+,d0                ; Restore d0...
Do_MMU_RTS:     rts                             ; Done...
*
* At this point we are being called as follows:
*       a0 - Points to the page entry in the MMU table for the address
*       a1 - Points at the WORD size nest count for this page in the MMU
*       d0 - Scrap
*       d1 - Scrap
*       STACK - Ready to RTS...
*
Off_MMU_Page:   move.w  (a1),d0                 ; Get the count...
                addq.w  #1,(a1)                 ; Bump the count...
                tst.w   d0                      ; Are we 0?
                bne.s   Do_MMU_RTS              ; If not, we already are nested
                addq.l  #3,a0                   ; Point at last byte of long
                cpusha  dc                      ; Push the data cache before ATC
                pflusha                         ; Flush the ATC...
                bclr.b  #5,(a0)                 ; Clear the copyback bit...
                cpushl  dc,(a0)                 ; Push the cache...
                rts
*
* This routine is called just like Off_MMU_Page is...
*
On_MMU_Page:    subq.w  #1,(a1)                 ; Drop count...
                move.w  (a1),d0                 ; Get count...
                bne.s   Do_MMU_RTS              ; If not 0, still nested...
                addq.l  #3,a0                   ; Point at last byte of long
                pflusha                         ; Flush the ATC...
                bset.b  #5,(a0)                 ; Set the copyback bit...
                cpushl  dc,(a0)                 ; Push the cache...
                rts
*
*******************************************************************************
```

1.20 copyback_dma

```
                Copyback caches are wonderful things for CPU performance.
The main reason is that they let code modify data multiple
times before the CPU has to do the slow operation of writing
the data back into physical memory.  The difference in
performance can be significant.
```

It does, however, mean that data is not written out to physical RAM until the CPU really must. The longer the CPU can keep the data in the cache, the more likely another access to that data will happen and thus higher performance.

This all sounds fine until you get to DMA devices. That is, devices that do work without bothering the CPU. These devices are very good for performance since the CPU can continue to deal with the programs that are running and let someone else deal with simplistic tasks such as copying bytes to or from disk drives, network cards, etc.

The first problem is easy to spot. Before starting a DMA operation, the CPU's caches need to be flushed. The reason being that you want to make sure that the DMA device can see the current/correct values in memory.

But, there is an even more evil problem lurking. This problem is so evil that it many times goes unnoticed until just the right set of conditions arrise.

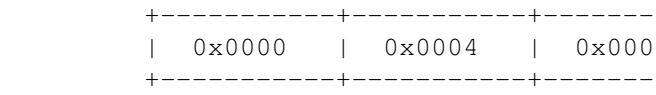
The conditions are complex and yet devilishly simple...

First, a quick understanding of how the cache works will be required. Caches (68030/68040/68060) have what are called cache lines. These are 4 longwords that are controlled as a set. The cache line corresponds to the size of the data fetch burst size. Burst access to RAM happens, in the 680x0 family, 4 longwords at a time. This organization also significantly reduces the complexity of the cache controller and thus makes the cache even faster.

Now, starting with the 68040, caches changed in three ways. First, they became much larger. From 256 bytes to 4096 bytes. Second, in order to handle the larger caches, the cache controller was changed to have only a single "dirty" flag for a line rather than one per longword in the line. (Note that this also fixed a bug that the 68030 had...)
Third, the caches grew a new mode called COPYBACK. This new mode would mark the cache as dirty when a write happens and then when the cache line was needed to cache some other address, the CPU would write back (copyback) the data from the cache line to RAM before caching the new data.

Ok, now the stange is set for the problem... You may already see it, but if not, don't feel bad as I did not catch it right away either.

Let us look at a very simple (and silly) example. I will draw some pictures to help make it easier...



The above is a cache line address 0x0000 to 0x000F. I drew

it with long-words due to resolution limits of ASCII :-)

Anyway, let us say that we started some DMA hardware that
was going to write to addresses 0x0000 and 0x0004. That
is, the first 2 longwords of this cache line. Think of this
as a "read" from, say, a DMA SCSI disk drive.

Now, I started this hardware working, and before doing
so, I call CachePreDMA() with the address and length
and it did the simple "flush" cache. Then I "Wait()"
until the hardware signals me that it is done.

In the mean time, some other code gets a chance to run and
it happens to have memory address 0x0008 and 0x000C allocated
to it. When it wakes up, it writes something to 0x0008.
The CPU will notice that this is not in the cache and read
in a cache line which contains the address 0x0008. It then
writes the data to the cache line and marks the cache line
as dirty.

Meanwhile, the DMA device has now found the data and started
to DMA it to physical RAM...

Ah, but what is this? Are not addresses 0x0000 and 0x0004
in that newly created cache line? And even worse, is not the
cache line marked as dirty and in need of being written out?

So, now, the data changed in the physical memory. A cache
flush is needed to make sure that any changes to physical
memory is noticed by the cache. This is normal so far...

But here is where Copyback and sign single dirty bit come
in. The team up and cause something bad to happen.

When flushing the cache, the cache controller notices that
there is some data in this cache line that needs to be placed
into physical memory. Namely the data that the task wrote to
location 0x0008. However, the cache controller has no idea
what part of the line is "dirty" nor does it care since it
always does "burst" memory access. So, the cache controller
writes the cache line back to RAM... and right over the top
of the new data that was placed there by the DMA device!

Oh the horror of it! New data that was killed by the CPU
before anyone could have seen it.

The solution:

This was rather fun to find and even more fun to solve.

First, any DMA that started and ended at line bounderies
would never have this problem. That is, if the DMA was
always started at addresses that were multiples of 16
and were a multiple of 16 in length would not have a
problem with this. (Unless they played with the memory

```
as the DMA was happening, but that would cause problems
even without copyback.)

So, the fix does not need to be done unless the DMA starts
on a non-quad-longword address or is not a multiple of 16
bytes long.

The simple fix would have been to turn off all caches during
DMA operation.  In fact, I tried that fix just to see if it
would work.  However, the performance impact on software
that does lots of DMA (such as programs that do disk I/O)
was rather dramatic.  (Try it yourself - turn off your caches
and see what happens to performance)

So, the trick is to minimise the area of disabled caches.
With the MMU turned on, I was able to disable the caches
for only those 4K pages where the DMA started and ended.
That is, the page where the DMA starts is cache disabled
and the page where the DMA ends is cache disabled.  When the
DMA is completed, the pages are returned to normal.

Since multiple DMA devices can be in operation at the same
time, there is a need to keep a "nest count" for each page.
This nest count is kept independant of the MMU table since
tools such as Enforcer and Virtual Memory Systems would
need to move the MMU tables around and the fix would then
not work with the correct MMU table.

The code shown in the
                patches
                 section implements most of
this support.  The setup of some of the tables is left
out as it depends on the form of the MMU.  The code deals
with 68040 MMU table structure and should work as is on
the 68060 CPU.
```

1.21 option_quiet

```
QUIET/S

This tells Enforcer not to complain about any invalid
access and to just build MMU tables for cache setting
reasons -- mainly used in conjunction with an
Amiga BridgeBoard in a 68030 environment so that
the system can run with the data cache turned on.
In this case,
               RUN >NIL: Enforcer QUIET
should be placed into the startup-sequence right
after SetPatch.
```

1.22 option_tiny

```
TINY/S

This tells Enforcer to output a minimal hit.  The
output is basically the first line of the Enforcer
hit.
```

1.23 option_small

```
SMALL/S

This tells Enforcer to output the hit line, the
USP: line, and the Name: line.  (This means that
no register or stack display will be output)
```

1.24 option_showpc

```
                SHOWPC/S

This tells Enforcer to also output the two lines
that contain the memory area around the PC where
the hit happened.  Useful for disassembly.
This option will not do anything if
                QUIET
                ,
                SMALL
                 or

                TINY
                 output modes are selected.
```

1.25 option_stacklines

```
STACKLINES/K/N

This lets you pick the number of lines of stack
backtrace to display.  The default is 2.  If set
to 0, no stack backtrace will be displayed.  There
is NO ENFORCED LIMIT on the number of lines.
```

1.26 option_stackcheck

```
                STACKCHECK/S

This option tells Enforcer that you wish all of
the long words displayed in the stack to be checked
against the global seglists via
                SegTracker
```

```
                .
This will tell you what seglist various return
addresses are on the stack.  If you are not
displaying stack information in the Enforcer hit
then STACKCHECK will have nothing to check.
If you are displaying stack information, then
each long word will be check and only those which
are in one of the tracked seglists will be
displayed in a
                SegTracker
                 line.
The output will show the PC address first and
then work its way back on the stack such that you
can read it from bottom up as the order of calling
or from top down as the stack-frame backtrace.
```

1.27 option_aregcheck

```
                AREGCHECK/S

This option tells Enforcer that you wish all of
the values in the Address Registers checked via

                SegTracker
                , much like
                STACKCHECK
                .
```

1.28 option_dregcheck

```
                DREGCHECK/S

This option tells Enforcer that you wish all of
the values in the Data Registers checked via

                SegTracker
                , much like
                STACKCHECK
                .
```

1.29 option_datestamp

```
DATESTAMP/S

This makes Enforcer output a date and time with each
hit.  Due to the nature of the way Enforcer must
work, the time can not be read during the Enforcer
hit itself so the time output will be the last time
value the main Enforcer task set up.  Enforcer will
update this value every second as to try to not
```

```
use any real CPU time.  The time displayed in the
hit will thus be exact.
(Assuming the system clock is correct.)
The date is output before anything from the hit
other than the optional introduction string.
```

1.30 option_deadly

```
DEADLY/S

This makes Enforcer a bit nasty.  Normally,
when an illegal read happens, Enforcer returns 0
as the result of this read.  With this option,
Enforcer will return $ABADFEED as the read data.
This option can make programs with Enforcer hits
cause even more hits.
```

1.31 option_fspace

```
FSPACE/S

This option will make the special $00F00000 address
space available for writing to.  This is useful for
those people with $00F00000 boards.  Mainly Commodore
internal development work -- should only be used
in that enviroment.
```

1.32 option_verbose

```
VERBOSE/S

This option will make Enforcer display information
as to the mapping of the I/O boards and other
technical information.  This information maybe useful
in specialized debugging.
```

1.33 option_led

```
LED/K/N

This option lets you specify the speed at which
the LED will be toggled for each Enforcer hit.
The default is 1 (which is like it always was)
Setting it to 0 will make Enforcer not touch
the LED.  Using a larger value will make the
flash take longer (such that it can be noticed
when doing I/O models other than the default
```

```
serial output)  The time that the flash will
take is a bit more than 1.3 microseconds times
the number.  So 1000 will be a bit more than
1.3 milliseconds.  (Or 1000000 is a bit more than
1.3 seconds.)
```

1.34 option_parallel

```
PARALLEL/S

This option will make Enforcer use the parallel port
hardware rather than the serial port for output.
```

1.35 option_rawio

```
RAWIO/S

This option will make Enforcer stuff the hit report
into an internal buffer and then from the main
Enforcer process output the results via the
RawPutChar() EXEC debugging LVO.  Since the output
happens on the Enforcer task it is possible for a
hit that ends in a system crash to not be able to
be reported.  This option is here such that tools
which can redirect debugging output can redirect
the Enforcer output too.
```

1.36 option_file

```
                FILE/K

This option will make Enforcer output the hit report
but to a file insted of sending it to the hardware
directly or using the
                RAWIO
                 LVO.  A good example of
such a file is CON:0/0/640/100/HIT/AUTO/WAIT.
Another thing that can be done is to have a program
sit on a named pipe and have Enforcer output to it.
This program can then do whatever it feels like with
the Enforcer hits.  (Such as decode them, etc.)
*NOTE*  It is not a good idea to have Enforcer hits
go to a file on a disk as if the system crashes
during/after the Enforcer hit, the disk may
become corrupt.
```

1.37 option_stdio

```
STDIO/S

This option will make Enforcer output the hit report
to STDOUT.  This option only works from the CLI as it
requires STDOUT.  It is best used with redirection or
pipes.
```

1.38 option_buffersize

```
                BUFFERSIZE/K/N

This lets you set Enforcer's internal output buffer
for the special I/O options.  This option is only
valid with the
                RAWIO
                ,
                FILE
                , or
                STDIO
                 options.
The minimum setting is 8000.  The default is 8000.
Having the right amount of buffer is rather
important for the special I/O modes.  The reason
is due to the fact that no operating system calls
can be made from a bus error.  Thus, in the
special I/O mode, Enforcer must store the output
in this buffer and, via some special magic,
wake up the Enforcer task to read the buffer and
write it out as needed.  However, if a task is
in Forbid() or Disable() when the Enforcer hit
happens, the Enforcer task will not be able to
output the results of the hit.  This buffer lets
a number of hits happen even if the Enforcer task
was unable to do the I/O.  If the number of
hits that happen before the I/O was able to
run gets too large, the last few hits will either
be cut off completely or contain only partial
information.
```

1.39 option_intro

```
INTRO/K

This optional introduction string will be output
at the start of every Enforcer hit.  For example:
INTRO="*NBad Program!"   The default is no string.
```

1.40 option_priority

```
                PRIORITY/K/N

This lets you set Enforcer's I/O task priority.
The default for this priority is 99.  In some
special cases, you may wish to adjust this.
It is, however, recommended that if you are using
one of the special I/O options (
                RAWIO
                ,
                FILE
                , or

                STDIO
                ) that you keep the priority rather high.
If the priority you supply is outside of the
valid task priority range (-127 to 127) Enforcer
will use the default priority.
```

1.41 option_noalertpatch

```
NOALERTPATCH/S

This option disables the patching of the EXEC
Alert() function.  Normally Enforcer will patch
this function to provide information as to what
called Alert() and to prevent the Enforcer hits
that a call to Alert() would cause.
```

1.42 option_on

```
ON/S

Mainly for completeness.  If not specified, it
is assumed you want to turn ON Enforcer.
```

1.43 option_quit

```
QUIT=OFF/S

Tells Enforcer to turn off.  Enforcer can also be
stopped by sending a CTRL-C to its process.
```

1.44 output

Example Enforcer output

```
03-Apr-93  21:26:18
WORD-WRITE to  00000000        data=4444       PC: 07895CA4
USP:  078D692C SR: 0000 SW: 0729  (U0)(-)(-)  TCB: 078A2690
Data: DDDD0000 DDDD1111 DDDD2222 DDDD3333 DDDD4444 DDDD5555 DDDD6666 DDDD7777
Addr: AAAA0000 AAAA1111 AAAA2222 AAAA3333 AAAA4444 AAAA5555 07800804 --------
Stck: 00000000 07848E1C 00009C40 078A30B4 BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB
Stck: BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB 078E9048 00011DA8 DEADBEEF
----> 07895CA4 - "lawbreaker"  Hunk 0000 Offset 0000007C
PC-8: AAAA1111 247CAAAA 2222267C AAAA3333 287CAAAA 44442A7C AAAA5555 31C40000
PC *: 522E0127 201433FC 400000DF F09A522E 012611C7 00CE4EAE FF7642B8 0324532E
Name: "New_Shell"  CLI: "lawbreaker"  Hunk 0000 Offset 0000007C

LONG-READ from AAAA4444                        PC: 07895CA8
USP:  078D692C SR: 0015 SW: 0749  (U0)(F)(-)  TCB: 078A2690
Data: DDDD0000 DDDD1111 DDDD2222 DDDD3333 DDDD4444 DDDD5555 DDDD6666 DDDD7777
Addr: AAAA0000 AAAA1111 AAAA2222 AAAA3333 AAAA4444 AAAA5555 07800804 --------
Stck: 00000000 07848E1C 00009C40 078A30B4 BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB
Stck: BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB 078E9048 00011DA8 DEADBEEF
----> 07895CA8 - "lawbreaker"  Hunk 0000 Offset 00000080
PC-8: 247CAAAA 2222267C AAAA3333 287CAAAA 44442A7C AAAA5555 31C40000 522E0127
PC *: 201433FC 400000DF F09A522E 012611C7 00CE4EAE FF7642B8 0324532E 01266C08
Name: "New_Shell"  CLI: "lawbreaker"  Hunk 0000 Offset 00000080
```

Here is a breakdown of what these reports are saying:

In the first report, the first line is the date stamp.

The first line of each report describes the access violation and where it happened from. In the case of a WRITE, the data that was being written will be displayed as well. If an instruction mode access caused the fault, there will be an (INST) in the line.

The first line may also contain the BUS ERROR message. This will be displayed when an address that is valid in the system lists causes a physical bus fault during the access. This usually will happen with plug-in cards or when a hardware problem causes some form of system fault. Watch out, if this does show up, your system may be unstable and/or unreliable.

The second line (starts USP:) displays the USER stack pointer (USP), the status register (SR:), the special status word (SW:). It then displays the supervisor/user state and the interrupt level. This will be from (U0) to (U7) or (S0) to (S7) (S=Supervisor) Next is the forbid state (F=forbid, -=not) and the disable state (D or -) of the task that was running when the access fault took place. Finally, the task control block address is displayed (TCB:)

The next two lines contain the data and address register dumps from when the access fault happened. Note that A7 is not listed here. It is the stack pointer and is listed as USP: in the line above.

Then come the lines of stack backtrace. These lines show the data on the stack. If the stack is in invalid memory, Enforcer will display a message to that fact.

```
If
                SegTracker
                 was installed before Enforcer, the "---->" lines
will display in which seglist the given addresses are in based on the
global tracking that
                SegTracker
                 does.  (See docs on
                SegTracker
                )
If no seglist match is found, no lines will be displayed.
One line will be displayed for each of the stack longwords asked
for (see the STACKCHECK option) and one line for the PC address of
the Enforcer hit.  (The PC line is always checked for is
                SegTracker
                is installed.)  The lines are in order: hit, first stack find,
second stack find, etc.  This is useful for tracking down who
called the routine that caused the Enforcer hit.

Next, optionally, comes the data around the program counter when the
access fault happened.  The first line (PC-8:) is the 8 long-words
before the program counter.  The second line starts at the program
counter and goes for 8 long words.

The last line displays the name of the task that was running when
the access fault took place.  If the task was a CLI, it will display
the name of the CLI command that was running.  If the access fault
was found to have happened within the seglist of a loaded program,
the segment number and the offset from the start of the segment will
be displayed.  (Note that this works for any LoadSeg()'ed process)

Note that the name will display as "Processor Interrupt Level x"
if the access happened in an interrupt.

25-Jul-93  17:15:06
Alert !! Alert 35000000     TCB: 07642F70     USP: 07657C10
Data: 00000000 DDDD1111 DDDD2222 DDDD3333 0763852A DDDD5555 DDDD6666 35000000
Addr: AAAA0000 AAAA1111 AAAA2222 AAAA3333 AAAA4444 0763852A 07400810 --------
Stck: 076385A0 00000000 0752EE9A 00002800 07643994 00000000 0762F710 076305F0
----> 076385A0 - "lawbreaker"  Hunk 0000 Offset 00000098

This output happens when a program or the OS calls the EXEC Alert
function.  Enforcer catches these calls and will display the alert
information as seen above.  (With the date and time as needed)

See also the
                Detail Example
                 for information.
```

1.45 findseg

```
/*
 * A simple program that will "find" given addresses in the SegLists
 * This program has been compiled with SAS/C 6.3 without errors or
 * warnings.
```

```
 *
 * Compiler options:
 * DATA=FARONLY PARAMETERS=REGISTER NOSTACKCHECK
 * NOMULTIPLEINCLUDES STRINGMERGE STRUCTUREEQUIVALENCE
 * MULTIPLECHARACTERCONSTANTS DEBUG=LINE NOVERSION
 * OPTIMIZE OPTIMIZERINLOCAL NOICONS
 *
 * Linker options:
 * FindSeg.o TO FindSeg SMALLCODE SMALLDATA NODEBUG LIB LIB:sc.lib
 */
#include <exec/types.h>
#include <exec/execbase.h>
#include <exec/libraries.h>
#include <exec/semaphores.h>
#include <dos/dos.h>
#include <dos/dosextens.h>
#include <dos/rdargs.h>

#include <clib/exec_protos.h>
#include <pragmas/exec_sysbase_pragmas.h>

#include <clib/dos_protos.h>
#include <pragmas/dos_pragmas.h>

#include <string.h>

#include "FindSeg_rev.h"

#define EXECBASE (*(struct ExecBase **)4)

typedef char (* __asm SegTrack(register __a0 ULONG,
                               register __a1 ULONG *,
                               register __a2 ULONG *));

struct SegSem
{
struct SignalSemaphore seg_Semaphore;
       SegTrack        *seg_Find;
};

#define SEG_SEM "SegTracker"

#define TEMPLATE "FIND/M" VERSTAG

#define OPT_FIND  0
#define OPT_COUNT 1

ULONG cmd(void)
{
struct ExecBase *SysBase;
struct Library  *DOSBase;
struct RDArgs   *rdargs;
       ULONG    rc=RETURN_FAIL;
struct SegSem   *segSem;
       char     **hex;
       LONG     opts[OPT_COUNT];
```

```
SysBase = EXECBASE;
if (DOSBase = OpenLibrary("dos.library",37))
{
  memset((char *)opts, 0, sizeof(opts));

  if (!(rdargs = ReadArgs(TEMPLATE, opts, NULL)))
  {
    PrintFault(IoErr(),NULL);
  }
  else if (CheckSignal(SIGBREAKF_CTRL_C))
  {
    PrintFault(ERROR_BREAK,NULL);
  }
  else if (segSem=(struct SegSem *)FindSemaphore(SEG_SEM))
  {
    rc=RETURN_OK;
    if (opts[OPT_FIND])
    {
      for (hex=(char **)opts[OPT_FIND];(*hex);hex++)
      {
      char  *p;
      ULONG val;
      ULONG tmp[4];
      ULONG c;

        val=0;
        p=*hex;
        if (*p=='$') p++; /* Support $hex */
        while (*p)
        {
          c=(ULONG)*p;
          if ((c>='a') && (c<='f')) c-=32;
          c-='0';
          if (c>9)
          {
            c-=7;
            if (c<10) c=16;
          }

          if (c<16)
          {
            val=(val << 4) + c;
            p++;
          }
          else
          {
            val=0;
            p=&p[strlen(p)];
          }
        }

        /*
         * Ok, we need to do this within Forbid()
         * as segments can unload at ANY time, including
         * during AllocMem(), so we use a stack buffer...
         */
        Forbid();
```

```
          if (p=(*segSem->seg_Find)(tmp[0]=val,&tmp[2],&tmp[3]))
          {
          char Buffer[200];

            stccpy(Buffer,p,200);
            tmp[1]=(ULONG)Buffer;
            VPrintf("$%08lx - %s : Hunk %ld, Offset $%08lx",tmp);

            /*
             * Now get the SegList address by passing the
             * same pointer for both hunk & offset.  Note
             * that this is only in the newer SegTracker
             * To test if this worked, check if the result
             * of this call is either a hunk or an offset.
             */
            (*segSem->seg_Find)(val,&tmp[0],&tmp[0]);
            /*
             * This "kludge" is for compatibility reasons
             * Check if result is the same as either the hunk
             * or the offset.  If so, do not print it...
             */
            if ((tmp[0]!=tmp[2]) && (tmp[0]!=tmp[3]))
            {
              VPrintf(", SegList $%08lx",tmp);
            }

            PutStr("\n");
          }
          else VPrintf("$%08lx - Not found\n",tmp);
          Permit();
        }
      }
    }
    else PutStr("Could not find SegTracker semaphore.\n");

    if (rdargs) FreeArgs(rdargs);
    CloseLibrary(DOSBase);
  }
  else if (DOSBase=OpenLibrary("dos.library",0))
  {
    Write(Output(),"Requires Kickstart 2.04 (37.175) or later.\n",43);
    CloseLibrary(DOSBase);
  }

  return(rc);
}
```

1.46 quotes

```
Some of my quotes that have been in my signatures in the past:

Quantum Physics:  The Dreams that Stuff is made of. - Michael Sinz

"A master's secrets are only as good as the
 master's ability to explain them to others" - Michael Sinz
```

"Can't I just bend one of the rules?" said the student.
The Master just looked back at him with a sad expression. - Michael Sinz

From the home of the imaginary deadlines:
"It will take 2i weeks to do that project." - Michael Sinz

By doing the impossible one just proves the point
that one can not do the impossible. - Michael Sinz

Some other quotes that I have used but did not come up with:

When one does business in the vicinity of a gorilla, you
spend much of your time muttering, "Nice gorilla..."

HELP! I am starting to like it here...

Eloquence is vehement simplicity

Programming is like sex:
One mistake and you have to support it for life.

I multitask, therefor we are.

1.47 copyright

The original Enforcer was written by Bryce Nesbitt. It was instrumental
to the development of 2.04 and to the improvement in the quality of
software on the Amiga. It is Copyright © 1991 - Commodore-Amiga, Inc.

Enforcer V37 is a completely new set of code designed to provide even
more debugging capabilities across more hardware configurations and
with more options. Michael Sinz designed and developed Enforcer V37.

+---+
| Michael Sinz |
| I-NET: Enforcer@sinz.org |

```
| BIX:  msinz      or      msinz@bix.com                    |
| "Can't I just bend one of the rules?" said the student.   |
| The Master just looked back at him with a sad expression. |
+-----------------------------------------------------------+
```

1.48 detailexample

```
Example
Enforcer
 Hit:  Click on the field for explaination.


25-Jul-93  17:15:04

WORD-WRITE
 to
00000000

data=0000

PC: 0763857C

----BUS ERROR----

USP:
07657C14
SR: 0004
SW: 04C1

(U0)(-)(-)

TCB: 07642F70

Data:
DDDD0000
DDDD1111
DDDD2222
DDDD3333
0763852A
DDDD5555
DDDD6666
DDDD7777

Addr:
AAAA0000
AAAA1111
AAAA2222
AAAA3333
AAAA4444
0763852A
07400810
--------

Stck:
00000000
```

```
0752EE9A
00002800
07643994
00000000
076786D8
000208B0
2EAC80EE

Stck:
487AFD12
486C82C4
4EBA3D50
4EBAEA28
4FEF0014
52ACE2E4
204D43EC
88BC203C

---->
0763857C
 _ "
lawbreaker
"
Hunk 0000

Offset 00000074

PC-8:
2222263C
DDDD3333
280D2A3C
DDDD5555
2C3CDDDD
66662E3C
DDDD7777
31C00000

PC *:
4EAEFF7C
20144EAE
FF8811C1
01014EAE
FF7621C0
01024EAE
FF822E3C
35000000

Name:
"
Shell
" CLI: "
LawBreaker
"
Hunk 0000

Offset 00000074
And, for Alert hits:
```

```
                25-Jul-93  17:15:06

                Alert
                !!
                Alert

                35000000

                TCB: 07642F70

                USP: 07657C10

                Data:
                DDDD0000
                DDDD1111
                DDDD2222
                DDDD3333
                0763852A
                DDDD5555
                DDDD6666
                35000000

                Addr:
                AAAA0000
                AAAA1111
                AAAA2222
                AAAA3333
                AAAA4444
                0763852A
                07400810
                --------

                Stck:
                076385A0
                00000000
                0752EE9A
                00002800
                07643994
                00000000
                0762F710
                076305F0

                ---->
                076385A0
                 _ "
                lawbreaker
                "
                Hunk 0000

                Offset 00000098
                Note that
                Enforcer
                 hit output is very configurable.  The above example hit
was produced with options:
                SHOWPC
```

```
                DATESTAMP

                STACKCHECK

                STACKLINES=2
                Here are some examples of different output configurations:

Enforcer output with the TINY option:   (Commandline: ENFORCER
                TINY
                )


                WORD-WRITE
                 to
                00000000

                data=0000

                PC: 0763857C
                Enforcer output with the SMALL option:  (Commandline: ENFORCER
                SMALL
                )


                WORD-WRITE
                 to
                00000000

                data=0000

                PC: 0763857C

                USP:
                07657C14
                SR: 0004
                SW: 04C1

                (U0)(-)(-)

                TCB: 07642F70

                Name:
                "
                Shell
                " CLI: "
                LawBreaker
                "
                Hunk 0000

                Offset 00000074
                Enforcer output with DEFAULT options:   (Commandline: ENFORCER)


                WORD-WRITE
                 to
                00000000
```

```
data=0000

PC: 0763857C

USP:
07657C14
SR: 0004
SW: 04C1

(U0)(-)(-)

TCB: 07642F70

Data:
DDDD0000
DDDD1111
DDDD2222
DDDD3333
0763852A
DDDD5555
DDDD6666
DDDD7777

Addr:
AAAA0000
AAAA1111
AAAA2222
AAAA3333
AAAA4444
0763852A
07400810
--------

Stck:
00000000
0752EE9A
00002800
07643994
00000000
076786D8
000208B0
2EAC80EE

Stck:
487AFD12
486C82C4
4EBA3D50
4EBAEA28
4FEF0014
52ACE2E4
204D43EC
88BC203C

---->
0763857C
 _ "
```

```
                lawbreaker
                "
                Hunk 0000

                Offset 00000074

                Name:
                "
                Shell
                " CLI: "
                LawBreaker
                "
                Hunk 0000

                Offset 00000074
```

1.49 output_datestamp

```
                The date stamp field, if enabled, is at the start of the
                Enforcer
                 hit.
The time is only exact to +/- 1 second.
```

1.50 output_write

```
                This tells you that the
                Enforcer
                 Hit was a READ from or WRITE to memory.
The possible writes are:

        -- WRITE -                  - READ --
        BYTE-WRITE -  8-bit write - BYTE-READ
        WORD-WRITE - 16-bit write - WORD-READ
        LONG-WRITE - 32-bit write - LONG-READ
        LINE-WRITE - 68040 only   - LINE-READ
```

1.51 output_address

```
                This field in the output shows the illegal address that was
accessed which triggered the
                Enforcer
                 report.
```

1.52 output_writedata

On an illegal WRITE to memory, the value that was attempted to
be written will be displayed here. The size of this field
changes to match the size of the write. 68040 LINE writes are
not supported in this field. The 68060 does not support
this field.

1.53 output_pc

This field displays the program counter at the time of the MMU
trap of the invalid access. Note that this address is not always
the exact instruction that caused the hit. See the various
notes for your processor for more details.

General Notes

68020 Notes

68030 Notes

68040 Notes

68060 Notes

1.54 output_buserror

This field normally would never be seen by most people. It is generated
when a legal memory address causes a physical bus fault. This usually
can only happen when designing hardware or a part of the system hardware
has become unreliable. Watch out, if this does show up, your system may
be unstable and/or unreliable.

For more information on bus faults, see the Motorola CPU Hardware
Design handbook.

1.55 output_sr

This is the CPU status register as found on the MMU trap
stack frame. It contains the condition flags and the
current mode/etc.

1.56 output_sw

This is the special status word that is part of the MMU
trap frame. Check your CPU manuals for more details as
to what this word contains. Note that it is different
for the different versions of the 680x0 family.

Note that on the 68060 this is the upper WORD of the FSLW.

1.57 output_decode

This field contains special task information. This is
useful for determining what is going on at the time of the hit.

```
     (U0)(-)(-)
      ^^  ^  ^
      ||  |  |
      ||  |  +-- This will have a D if the task is DISABLE state
      ||  +----- This will have a F if the task is FORBID state
      |+-------- This is the processor IPL level (0 is normal code)
      +--------- This is the processor state: U=user, S=supervisor
```

1.58 output_tcb

This is the address of the Task Control Block, also known
as the task structure. (See exec/tasks.h) This is used by

Enforcer
to tell you who caused the hit.

1.59 output_dataregs

This line contains a dump of the data registers at the time
of the
Enforcer
hit.

1.60 output_d0

The D0 register of the 680x0 CPU.

See
Data:

1.61 output_d1

The D1 register of the 680x0 CPU.

See
Data:

1.62 output_d2

The D2 register of the 680x0 CPU.

See

Data:

1.63 output_d3

The D3 register of the 680x0 CPU.

See

Data:

1.64 output_d4

The D4 register of the 680x0 CPU.

See

Data:

1.65 output_d5

The D5 register of the 680x0 CPU.

See

Data:

1.66 output_d6

The D6 register of the 680x0 CPU.

See

Data:

1.67 output_d7

The D7 register of the 680x0 CPU.

See

Data:

1.68 output_addrregs

```
                This line contains a dump of the address register at the
time of the
                Enforcer
                 hit.
```

1.69 output_a0

```
                The A0 register of the 680x0 CPU.

See
                Addr:
```

1.70 output_a1

```
                The A1 register of the 680x0 CPU.

See
                Addr:
```

1.71 output_a2

```
                The A2 register of the 680x0 CPU.

See
                Addr:
```

1.72 output_a3

```
                The A3 register of the 680x0 CPU.

See
                Addr:
```

1.73 output_a4

```
                The A4 register of the 680x0 CPU.

See
                Addr:
```

1.74 output_a5

```
                The A5 register of the 680x0 CPU.

See
                Addr:
```

1.75 output_a6

```
                The A6 register of the 680x0 CPU.

See
                Addr:
```

1.76 output_a7

```
                The A7 register of the 680x0 CPU is also known as the
Stack Pointer or SP.  In the
                Enforcer
                 hit, the USER SP
(the stack of the task that caused the hit) is displayed
in the USP: field.

See
                Addr:
```

1.77 output_stack

```
                These lines contain stack dumps from the task that caused
the
                Enforcer
                 hit.  It can be used to figure out what the
program was doing and what routines called the current
routine by looking at the values on the stack.
```

1.78 output_stackword

```
                This is a longword on the stack of the task that caused the hit
See
                Stck:
                 for more details.
```

1.79 output_segtracker

```
                This symbol "---->" identifies a line produced via the

                SegTracker
                 utility.

See
                FindHit
                 for details as to how to use this information.
```

1.80 output_segtrackeraddress

```
                This is the address that the hunk/offset describes.  This is here ↩
                   such
that you can cross-reference it with a value on the stack, in a register,
or the program counter.  The hunk/offset on the same line are produced
when this address is processed via
                SegTracker
                .
See
                FindHit
                 for details as to how to use this information.
```

1.81 output_segtrackername

```
                This is the name of the file, as passed to LoadSeg, which was ↩
                   found to
be loaded around the address given.  See
                FindHit
                 for details as to how
to use this information.
```

1.82 output_segtrackerhunk

```
                This is the hunk in the load file that was loaded around the
given address.  See
                FindHit
                 for details as to how
to use this information.
```

1.83 output_segtrackeroffset

```
                This is the offset from the start of the hunk that this address is
at within the given load file.  See
                FindHit
                 for details as to how
to use this information.
```

1.84 output_name

This line contains the decoding of the TCB into the TASK name,
the CLI command (if a CLI), and if the hit happened in the SegList
attached to the process, the hunk and offset for the hit. Note
that this hunk/offset is not produced by
SegTracker
.

1.85 output_taskname

This field contains the task name as stored in the TCB of the task
that caused the
Enforcer
hit. If the TCB is invalid, it will say so.

1.86 output_cliname

This field will contain the name of the CLI command that caused the
hit if the TCB is a CLI process and there was a command loaded.
If the task is not a CLI process or no command is loaded, this
field will not be displayed.

1.87 output_alert

This output happens when a program or the OS calls the EXEC Alert
function.
Enforcer
catches these calls and will display the alert
information as seen above. (With the data and time as needed)

1.88 output_alertnum

This field contains the alert number that was generated. Check
the include file exec/alerts.h or exec/alerts.i for details as
to how to decode this number.

1.89 output_showpc

If the SHOWPC option is turned on,
Enforcer
will dump the 8 longwords
before the program counter and the 8 longwords starting at the PC.

This can be used to help debug programs by being able to look at the
code around the hit by disassembling it.

1.90 output_showpc_m8

```
                This is the longword at the memory address (PC - $20)
where PC is the
                Program Counter
                 of the
                Enforcer
                 hit.
```

1.91 output_showpc_m7

```
                This is the longword at the memory address (PC - $1C)
where PC is the
                Program Counter
                 of the
                Enforcer
                 hit.
```

1.92 output_showpc_m6

```
                This is the longword at the memory address (PC - $18)
where PC is the
                Program Counter
                 of the
                Enforcer
                 hit.
```

1.93 output_showpc_m5

```
                This is the longword at the memory address (PC - $14)
where PC is the
                Program Counter
                 of the
                Enforcer
                 hit.
```

1.94 output_showpc_m4

```
                This is the longword at the memory address (PC - $10)
where PC is the
                Program Counter
                 of the
                Enforcer
                 hit.
```

1.95 output_showpc_m3

```
                This is the longword at the memory address (PC - $0C)
where PC is the
                Program Counter
                 of the
                Enforcer
                 hit.
```

1.96 output_showpc_m2

```
                This is the longword at the memory address (PC - $08)
where PC is the
                Program Counter
                 of the
                Enforcer
                 hit.
```

1.97 output_showpc_m1

```
                This is the longword at the memory address (PC - $04)
where PC is the
                Program Counter
                 of the
                Enforcer
                 hit.
```

1.98 output_showpc_p0

```
                This is the longword at the memory address (PC)
where PC is the
                Program Counter
                 of the
                Enforcer
                 hit.
```

1.99 output_showpc_p1

```
                This is the longword at the memory address (PC + $04)
where PC is the
                Program Counter
                 of the
                Enforcer
                 hit.
```

1.100 output_showpc_p2

```
                This is the longword at the memory address (PC + $08)
where PC is the
                Program Counter
                 of the
                Enforcer
                 hit.
```

1.101 output_showpc_p3

```
                This is the longword at the memory address (PC + $0C)
where PC is the
                Program Counter
                 of the
                Enforcer
                 hit.
```

1.102 output_showpc_p4

```
                This is the longword at the memory address (PC + $10)
where PC is the
                Program Counter
                 of the
                Enforcer
                 hit.
```

1.103 output_showpc_p5

```
                This is the longword at the memory address (PC + $14)
where PC is the
                Program Counter
                 of the
                Enforcer
                 hit.
```

1.104 output_showpc_p6

```
                This is the longword at the memory address (PC + $18)
where PC is the
                Program Counter
                 of the
                Enforcer
                 hit.
```

1.105 output_showpc_p7

```
                This is the longword at the memory address (PC + $1C)
where PC is the
                Program Counter
                 of the
                Enforcer
                 hit.
```

1.106 sourcecode

```
                Advanced Enforcer V37 - Source Code

I have never charged money for the use of Enforcer as a tool to
help make better Amiga programs.  Enforcer has helped make Amiga
software some of the most reliable software on any computer
platform.

I have not, however, made the source available.  The main reason
being that I want to keep the quality of Enforcer high and reduce
the chances that there would be "derivative" versions of Enforcer.

People still ask about how to get the source...

Well, you asked for it; now you can get it...

You can now get the source code to Enforcer and all of Enforcer's
tools.  For $60 (US) I will send you a disk or EMail you an
archive that contains the complete build of Enforcer.  Including
source and object files, makefiles, icons, etc.

The price is not even as high as people had suggested to sell
Enforcer for, but I do wish to make some money so that I can
keep doing Amiga work.

The source is fully commented and should be of interest to anyone who
likes looking how things work on the inside.

However, there are some restrictions:

  a)  The source code is not to be distributed.  If you did not
      get it directly from me, please let me know.

  b)  Any programs made with the source code are not to be
      distributed without first letting see the new code.

  c)  For commercial use of the source or programs developed from
      the source, please contact MKSoft Development.

If you have special needs or requirements, please contact me.
I would be more than willing to address any concerns/wishes
you may have.

To order on-line, go to http://www.iam.com/amiga/enforcer.html
```

```
or fill out the
                order form
                 and send to:

      Michael Sinz
      MKSoft Development
      105 Keim Road
      Elverson, PA 19520

      Enforcer@sinz.org
```

1.107 orderform

```
  ------------------------------------------------------------
  |                                                          |
  |           Single user, non-commercial license            |
  |                                                          |
  |                                                          |
  |  Name: ____________________________________________      |
  |                                                          |
  |                                                          |
  |  Address: _________________________________________      |
  |                                                          |
  |                                                          |
  |           _________________________________________      |
  |                                                          |
  |                                                          |
  |  E-Mail:  _________________________________________      |
  |                                                          |
  |  ------------------------------------------------       |
  |                                                          |
  |   Single user, non-commercial license:        $60.00     |
  |                                                          |
  |   Shipping:  ($2 USA/$6 elsewhere)            ______     |
  |                                                          |
  |   Internet E-Mail delivery:  (no cost)          0.00     |
  |                                                          |
  |  ------------------------------------------------       |
  |                                                          |
  |   Total:                             Total:   ______     |
  |                                                          |
  ------------------------------------------------------------

      All checks in US funds.
      Make checks payable to:  MKSoft Development
```

1.108 index

```
                Index of all nodes in the Enforcer.guide document:
```

68020 Notes

68030 Notes

68040 Notes

68060 Notes

68040.library & 68060.library notes

68040.library patches

Copyback and DMA

Debuggers: Not causing a hit

Debuggers: Trapping a hit

Detail Example Hit

Enforcer

Enforcer - Copyright © 1992-1998

Enforcer Beta Testers

Enforcer Credits

Enforcer Documentation

Enforcer Output: A0 Register

Enforcer Output: A1 Register

Enforcer Output: A2 Register

Enforcer Output: A3 Register

Enforcer Output: A4 Register

Enforcer Output: A5 Register

Enforcer Output: A6 Register

Enforcer Output: A7 Register

Enforcer Output: Address hit

Enforcer Output: Address Register Dump

Enforcer Output: Alert Number

Enforcer Output: Alerts

Enforcer Output: Bus Error

Enforcer Output: CLI Command Name

```
Enforcer Output: CPU Status Register

Enforcer Output: D0 Register

Enforcer Output: D1 Register

Enforcer Output: D2 Register

Enforcer Output: D3 Register

Enforcer Output: D4 Register

Enforcer Output: D5 Register

Enforcer Output: D6 Register

Enforcer Output: D7 Register

Enforcer Output: Data Register Dump

Enforcer Output: Data Write

Enforcer Output: Date Stamp

Enforcer Output: Hunk

Enforcer Output: Offset

Enforcer Output: Program Counter

Enforcer Output: SegTracker

Enforcer Output: SegTracker Address

Enforcer Output: SegTracker Name

Enforcer Output: Show PC

Enforcer Output: Show PC+$00

Enforcer Output: Show PC+$04

Enforcer Output: Show PC+$08

Enforcer Output: Show PC+$0C

Enforcer Output: Show PC+$10

Enforcer Output: Show PC+$14

Enforcer Output: Show PC+$18

Enforcer Output: Show PC+$1C

Enforcer Output: Show PC-$04
```

Enforcer Output: Show PC-$08

Enforcer Output: Show PC-$0C

Enforcer Output: Show PC-$10

Enforcer Output: Show PC-$14

Enforcer Output: Show PC-$18

Enforcer Output: Show PC-$1C

Enforcer Output: Show PC-$20

Enforcer Output: Special information

Enforcer Output: Special Status Word

Enforcer Output: Stack Dump

Enforcer Output: Stack Word

Enforcer Output: Task Control Block

Enforcer Output: Task Name

Enforcer Output: Task/Process Name

Enforcer Output: Write Hit

Enforcer Source Code

Example Enforcer output

Famous MKSoft Quotes

FindHit

FindSeg: A SegTracker example

General Notes

LawBreaker

MMU tool

Move4K

Option: AREGCHECK

Option: BUFFERSIZE

Option: DATESTAMP

Option: DEADLY

Option: DREGCHECK

```
Option: FILE

Option: FSPACE

Option: INTRO

Option: LED

Option: NOALERTPATCH

Option: ON

Option: PARALLEL

Option: PRIORITY

Option: QUIET

Option: QUIT

Option: RAWIO

Option: SHOWPC

Option: SMALL

Option: STACKCHECK

Option: STACKLINES

Option: STDIO

Option: TINY

Option: VERBOSE

Order Form

SegTracker
```