Enforcer

COLLABORATORS			
	TITLE : Enforcer		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 18, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Enforcer **1**

1.1 main . . . 1
1.2 credits . . . 1
1.3 credits_testers . . . 2
1.4 enforcer . . . 3
1.5 findhit . . . 4
1.6 lawbreaker . . . 6
1.7 move4k . . . 7
1.8 segtracker . . . 8
1.9 rebootoff . . . 11
1.10 debuggers1 . . . 11
1.11 debuggers2 . . . 13
1.12 notes1 . . . 15
1.13 notes2 . . . 16
1.14 notes3 . . . 16
1.15 notes4 . . . 16
1.16 option_quiet . . . 18
1.17 option_tiny . . . 18
1.18 option_small . . . 18
1.19 option_showpc . . . 19
1.20 option_stacklines . . . 19
1.21 option_stackcheck . . . 19
1.22 option_aregcheck . . . 19
1.23 option_dregcheck . . . 20
1.24 option_datestamp . . . 20
1.25 option_deadly . . . 20
1.26 option_fspace . . . 20
1.27 option_verbose . . . 21
1.28 option_led . . . 21
1.29 option_parallel . . . 21

1.30 option_rawio . . . 21
1.31 option_file . . . 22
1.32 option_stdio . . . 22
1.33 option_buffersize . . . 22
1.34 option_intro . . . 23
1.35 option_priority . . . 23
1.36 option_noalertpatch . . . 23
1.37 option_on . . . 23
1.38 option_quit . . . 23
1.39 output . . . 24
1.40 findseg . . . 25
1.41 quotes . . . 28
1.42 copyright . . . 29
1.43 detailexample . . . 29
1.44 output_datestamp . . . 33
1.45 output_write . . . 33
1.46 output_address . . . 33
1.47 output_writedata . . . 33
1.48 output_pc . . . 33
1.49 output_buserror . . . 33
1.50 output_sr . . . 34
1.51 output_sw . . . 34
1.52 output_decode . . . 34
1.53 output_tcb . . . 34
1.54 output_dataregs . . . 34
1.55 output_d0 . . . 35
1.56 output_d1 . . . 35
1.57 output_d2 . . . 35
1.58 output_d3 . . . 35
1.59 output_d4 . . . 35
1.60 output_d5 . . . 35
1.61 output_d6 . . . 35
1.62 output_d7 . . . 36
1.63 output_addrregs . . . 36
1.64 output_a0 . . . 36
1.65 output_a1 . . . 36
1.66 output_a2 . . . 36
1.67 output_a3 . . . 36
1.68 output_a4 . . . 36

1.69 output_a5 . . . 37
1.70 output_a6 . . . 37
1.71 output_a7 . . . 37
1.72 output_stack . . . 37
1.73 output_stackword . . . 37
1.74 output_segtracker . . . 37
1.75 output_segtrackeraddress . . . 38
1.76 output_segtrackername . . . 38
1.77 output_segtrackerhunk . . . 38
1.78 output_segtrackeroffset . . . 38
1.79 output_name . . . 38
1.80 output_taskname . . . 38
1.81 output_cliname . . . 38
1.82 output_alert . . . 39
1.83 output_alertnum . . . 39
1.84 output_showpc . . . 39
1.85 output_showpc_m8 . . . 39
1.86 output_showpc_m7 . . . 39
1.87 output_showpc_m6 . . . 39
1.88 output_showpc_m5 . . . 39
1.89 output_showpc_m4 . . . 40
1.90 output_showpc_m3 . . . 40
1.91 output_showpc_m2 . . . 40
1.92 output_showpc_m1 . . . 40
1.93 output_showpc_p0 . . . 40
1.94 output_showpc_p1 . . . 40
1.95 output_showpc_p2 . . . 40
1.96 output_showpc_p3 . . . 40
1.97 output_showpc_p4 . . . 41
1.98 output_showpc_p5 . . . 41
1.99 output_showpc_p6 . . . 41
1.100 output_showpc_p7 . . . 41
1.101 index . . . 41

Chapter 1

Enforcer

1.1 main

Table of contents:

Enforcer
SegTracker
FindHit
LawBreaker
RebootOff
Move4K

Copyright
Credits

+---+
| Michael Sinz |
| UUNET: michael.sinz@scala.com |
| 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 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	jim2@bix.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
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@bix.com
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
      BridgeBoard

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 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.
```

1.8 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
```

1.9 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
```

1.10 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 68020 and 68030 version...
;
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
NewVector:      cmp.l   #4,$10(sp)              ; 68020 and 68030
                beq.s   TraceSkip               ; If AbsExecBase, OK
                ;
                ; 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:
                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 68040 version...
;
NewVector040:   cmp.l   #4,$14(sp)              ; 68040
                beq.s   TraceSkip040            ; If AbsExecBase, OK
                ;
                ; 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:
                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...
                ;
TraceSkip040:   move.l  OldVector(pc),-(sp)     ; Ready to return
                rts
```

1.11 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);
}
```

1.12 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
      BridgeBoard

1.13 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.

1.14 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.

1.15 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 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.

1.16 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.

1.17 option_tiny

TINY/S

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

1.18 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)

1.19 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.
```

1.20 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.
```

1.21 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.
```

1.22 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.
```

1.23 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.
```

1.24 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.
```

1.25 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.
```

1.26 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.
```

1.27 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.
```

1.28 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.)
```

1.29 option_parallel

```
PARALLEL/S

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

1.30 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.
```

1.31 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.
```

1.32 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.
```

1.33 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.
```

1.34 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.
```

1.35 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.
```

1.36 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.
```

1.37 option_on

```
ON/S

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

1.38 option_quit

```
QUIT=OFF/S

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

1.39 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 data and time as needed)

See also the Detail Example for information.
```

1.40 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);
}
```

1.41 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.
```

1.42 copyright

1.43 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
```

1.44 output_datestamp

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

1.45 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
```

1.46 output_address

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

1.47 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.

1.48 output_pc

This field displays the program counter at the time of the MMU
trap of the invalid axxess. 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
```

1.49 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.

1.50 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.

1.51 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.

1.52 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
```

1.53 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.

1.54 output_dataregs

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

1.55 output_d0

The D0 register of the 680x0 CPU.

See Data:

1.56 output_d1

The D1 register of the 680x0 CPU.

See Data:

1.57 output_d2

The D2 register of the 680x0 CPU.

See Data:

1.58 output_d3

The D3 register of the 680x0 CPU.

See Data:

1.59 output_d4

The D4 register of the 680x0 CPU.

See Data:

1.60 output_d5

The D5 register of the 680x0 CPU.

See Data:

1.61 output_d6

The D6 register of the 680x0 CPU.

See Data:

1.62 output_d7

```
The D7 register of the 680x0 CPU.

See Data:
```

1.63 output_addrregs

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

1.64 output_a0

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

See Addr:
```

1.65 output_a1

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

See Addr:
```

1.66 output_a2

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

See Addr:
```

1.67 output_a3

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

See Addr:
```

1.68 output_a4

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

See Addr:
```

1.69 output_a5

The A5 register of the 680x0 CPU.

See Addr:

1.70 output_a6

The A6 register of the 680x0 CPU.

See Addr:

1.71 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:

1.72 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.

1.73 output_stackword

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

1.74 output_segtracker

This symbol "---->" identifies a line produced via the
SegTracker utility.

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

1.75 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.

1.76 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.

1.77 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.

1.78 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.

1.79 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.

1.80 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.

1.81 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.

1.82 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)

1.83 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.

1.84 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.

1.85 output_showpc_m8

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

1.86 output_showpc_m7

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

1.87 output_showpc_m6

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

1.88 output_showpc_m5

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

1.89 output_showpc_m4

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

1.90 output_showpc_m3

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

1.91 output_showpc_m2

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

1.92 output_showpc_m1

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

1.93 output_showpc_p0

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

1.94 output_showpc_p1

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

1.95 output_showpc_p2

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

1.96 output_showpc_p3

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

1.97 output_showpc_p4

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

1.98 output_showpc_p5

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

1.99 output_showpc_p6

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

1.100 output_showpc_p7

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

1.101 index

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

  68020 Notes
  68030 Notes
  68040 Notes
  Debuggers: Not causing a hit
  Debuggers: Trapping a hit
  Detail Example Hit
  Enforcer
  Enforcer - Copyright © 1992-1996
  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
Example Enforcer output
Famous MKSoft Quotes
FindHit
FindSeg: A SegTracker example
General Notes
LawBreaker
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
SegTracker
```