Linice

A Linux Kernel Level Debugger

Version 2.6
www.linice.com

Author: Goran Devic
Contact email: author@linice.com

For the impatient, file README contains quick start instructions.

System Requirements

- Linux PC/x86 platform
- Minimum Pentium class CPU
- Linux kernels 2.4 or 2.6

Linice debugger has been developed on a number of Linux kernels, mainly on RedHat distributions. It was tested on RedHat 9.0 and SuSE 8.0. It should also work with no problems on some earlier versions of these distributions. Other distributions may work, but are not tested. You may have various successes with out-of-stock kernels or those earlier than 2.4. Support for kernels 2.6 is limited: This Linice version will compile and load under Debian 2.6 (as tested). Other distributions may or may not work; in particular FedoraCore2 will not work since its System.map file does not list some important symbols. You may be able to have it running if you recompile the kernel with those symbols exported. Linice should work with kernels 2.6.8 and below. The work is currently in progress to have integrated support for all 2.6 kernels, including the most recent ones after 2.6.8. If you think you can contribute, send me an email or join the discussion group at http://groups-beta.google.com/group/linice

Linice components

Linice consists of:

1) Linice kernel loadable module: kernel independent portion and kernel dependent portion. Build a generic object from within the "linice" directory. Kernel dependent loadable module is created by running a make within the "bin" directory. This directory is a symbolic link to one of the kernel-specific directories: bin-2.4 or bin-2.6
2) Linsym: User mode app that loads / unloads Linice kernel module, generates and manipulates symbol files.
3) Xice: Initiates a session on the X-Server with a loaded Linice.

Compiling Linice from the source

Initially, the complete package can be compiled by running “make_bin-2.4” or “make_bin-2.6” from the build directory. Later, individual components can be compiled by running “make” from within “linice”, “linsym” or “x” directories.

You will also need assembler NASM to successfully compile Linice. NASM is enclosed within Linice distribution. It can be found in the subdirectory tools/nasm.

When done compiling individual components, follow the instructions below to complete the kernel-dependent portion of the installation.

Compiling kernel-specific Linice code

1) After compiling kernel-independent code, it has to be linked with the iceface.c file which contains your particular kernel interfaces. This step actually creates a Linux loadable module for a particular kernel. You may want to adjust some configuration defines:

Modify ‘Makefile’ to suit your environment:

```
Default:           TARGET=
Configuration:     Single CPU, non-APIC
```

By default, Linice builds for a single CPU, non-APIC target machine.
If your machine has IO_APIC, you will need to add –DIO_APIC define.
If your machine is SMP, you will need to add –DSMP define.

```
Default:           #PCIHDR = PCIHDR
Configuration:     Include PCI header information file
```

Delete “#” in order to include the “pcihdr.h” file. This file contains a database of PCI devices. It is used by the PCI command to decode devices on a PCI bus. Inclusion of this file will enlarge the module by about 300K.

---

2) Run ‘make” (for 2.4 builds) or “compile_2.6” (for 2.6 builds) in order to generate the debugger module tailored to your running kernel.

3) Customize initialization file “linice.dat”

Specify the memory to be reserved for symbols, history buffer or display; change the init string or macros, or set the different keyboard layout to customize it to your particular locale. The default values are a good starting point and should work.

These keyboard layouts are supported:

```
us                         United States (default)
uk                         UK
finnish, finnish-latin1  Finland
de, de-latin1              Germany
```

```
fr, fr-latin1            France
dk, dk-latin1            Denmark
dvorak                   [Dvorak keyboard]
sg, sg-latin1            Switzerland (German)
sf, sf-latin1            Switzerland (French)
be                       Belgium
po                       Portugal
it                       Italy
sw                       Sweden
hu                       Hungary
jp106                    Japan
pl                       Poland
hr-cp852, hr-latin2      Croatia / Hrvatska
cz-qwerty, cz-qwertz     Czech Republic
```

If you are not using US keyboard, it is necessary to let Linice know of a proper keyboard layout since it does not use Linux code for keyboard handling.

Installing and running Linice

4) Use `linsym` to load and unload the Linice module:

Load and install Linice kernel module: "`linsym –i`"
Unload Linice: "`linsym –x`"

You have to use linsym to load (and unload) linice since it needs some initial values that linsym provides which cannot be passed easily by manually using 'insmod'. Also, linsym reads the initialization file "linice.dat" and sends it to the module. It will search for this file first in the current directory, and then in the /etc directory. If the file could not be found at each of these locations, linsym will use default, built-in settings.

Linsym also searches for the map of kernel symbols in the /boot/System.map. That symbol map file has to match the running kernel! This is very important since linsym reads some kernel addresses that help it to locate the code into which to place various hooks (such is a keyboard hook or a pointer to the kernel module list.) If you have recompiled your kernel and the current System.map is named differently, Linice may refuse to load. In this case, use the option –m to specify the correct map file.

If you are calling linsym from anywhere other than the default linice/bin directory, you have to define environment variable LINICE and set it with the path to that directory, so the linsym knows where to find the debugger kernel module to load. As example, if you installed (and rebuilt) linice in /usr/src/linice directory, you would do:

```
# export LINICE=/usr/src/Linice/bin
```

Then, you will be able to call linsym from any directory.

You have to be “root” in order to load or unload linice. If you intend to become a root by using the “su” command, remember to use the “su –“format (the dash option will switch to the root environment variables).

5) Use `linsym` to translate symbols
Translate debug information from a program or a module:
“`linsym –t <binary>`”
This will create a symbol file <binary>.SYM that can be loaded.

6) Use `linsym` to load symbol file
Load symbol file into Linice: “`linsym –s <symbols.sym>`”

7) You may break into the Linice at any time by pressing the keyboard hotkey: “Ctrl+Q”

Running in the X-Window

To have Linice pop up on the X-Window, you have to have it loaded using the standard command “`linsym –i`”. If you are already in the X environment, it might look as the system froze since Linice actually popped up using the VGA frame buffer. Simply hit F5 (go) and you will get back the control on your terminal. Then simply run utility “`xice`”, and the Linice should appear on the top of your X window.

Note: Linice supports most common linear frame buffers in 8, 16 and 32 bits per pixel modes. If your current video mode is using a 24 bpp, switch it to one of the supported pixel modes.

Compiling the debugee

Notes on compiling your code that you want to debug:

Your program or module has to be compiled with the “`-gstabs+`” switch in order to generate symbolic information suitable for Linice translation.

To get the proper visibility into local symbols, you should **not** use the switch: “-fomit-frame-pointer” since with that switch the local variable information will not be accessible.

It is recommended to disable code optimization.

To get the visibility into non-static, un-initialized global symbols, you need to use the linker switch: “-dp” or “-d” or “-dc” to force the assignments of space for “common symbols”.

Preparing the Linux kernel for source debugging

This text describes modifications to the kernel 2.4 build process.

Although you can run Linice on top of the unmodified kernel, if you need to do source level debug on the kernel code, these steps should give you a general guidance on how to prepare the kernel code and symbols. You will need to rebuild the kernel in order to generate necessary stabs info, which will be used by the symbol translator. Edit the Linux kernel Makefile and insert the code in red/underline:

Linice needs stabs debugging information to be built with the Linux kernel.
Add "*-gstabs+*" to the CFLAGS and add "*-gstabs*" to the AFLAGS:

```
CFLAGS := -gstabs+ $(CPPFLAGS) -Wall -Wstrict-prototypes -Wno-trigraphs
-O2 -fno-strict-aliasing -fno-common -Wno-unused
AFLAGS := -gstabs -D__ASSEMBLY__ $(CPPFLAGS)
```

In order to be able to access local variables, remove the option -f*omit-frame-pointer*.

Next, we will make the default build generate an intermediate kernel version with all the stabs information, then have them stripped for the version that we will be loading:

```
      $(LD_VMLINUX) $(LD_VMLINUX_KALLSYMS) -o vmlinux.debug
      $(STRIP) -S -o vmlinux vmlinux.debug
      $(NM) vmlinux | grep -v '\(compiled\)\|\(\.o$$\)\|\( [aUw]
\)\|\(\.\.ng$$\)\|\(LASH[RL]DI\)' | sort > System.map
```

This modified Linux Makefile will now generate the file vmlinux.debug, which will be used to generate the Linice symbol file. It will also generate a regular kernel code that will be installed. Follow the standard kernel build procedure:

```
# make mrproper
# make xconfig or make menuconfig or make config
# make dep
# make clean
# make bzImage
```

`vmlinux.debug` and `vmlinux` images are now built. `vmlinux` is a stripped version of the `vmlinux.debug`.

```
# make modules
# make modules_install
# make install
```

Translate `vmlinux.debug` kernel symbols into Linice symbol file:

```
# linsym –t vmlinux.debug
```

Add new kernel to the boot loader and reboot using it, so the new symbols will match the running kernel when you load them into Linice.

A note about the serial connection

When you use a serial port and a VT100 terminal is connected to the other side of a serial link (on another machine), *the local keyboard may still be used as an active input device,* in addition to the input from a remote serial terminal. Although that behavior appears odd, it helps to keep control when the connection parameters are not quite right.

Some Toshiba notebooks have a hidden serial port connector under the keyboard. It uses a non-standard IO port of 0x1E0. Linice supports it and the port is enumerated as COM5.

Serial VT100 terminal output driver is expanded to support 24, 25, 48 or 50 lines ("lines" command) with 80 or 132 columns ("width" command).

Bugs

Please send all bug reports to bugs@linice.com.

Include Linice version, the version of the kernel that you are running it on (or trying to run it on), and any other information that would help reproduce the problem.

Debugging and developing Linice

If you are inclined to help out and work on Linice, please send me a note and I will try to help you get going as best as I can.

Since all the sources are freely published, you are free to tinker with it. However, please send me any modification that you feel would be useful to others, so I can add them to the "official" package for everyone's benefit. Of course, you will be credited as well ☺.I prefer getting the complete modified source files, so I can diff them (instead of getting the usual diff files).

Features and wish list

If you would like to see a feature implemented or have other suggestion for improvement, please email to features@linice.com.

USB Keyboard not supported

This limitation stems from the fact that Linice is handling keyboard at the low-level and currently only knows how to handle a legacy PS/2 keyboard interface, and it does not have a USB stack.

If you simply cannot plug in the PS/2 keyboard (which would solve this problem), and you really have to use a USB keyboard, you will need to disable Linux support for all USB devices. This will allow the system BIOS to handle the USB keyboard so it will appear to the software as if you have a legacy PS/2 keyboard. If you enable any USB support in the Linux kernel, it will turn off SBIOS handling of the USB keyboard, and Linice will not be able to handle it.

New commands and modified commands

Some of these commands are added and don't exist on SoftIce, and some are slightly modified to enhance the functionality or to make them more appropriate for the Linux environment.

ASCII

(new) Prints an ASCII character table

XWIN

(new) Redirect console to a DGA frame buffer (X-Window). This command will have effect only after you have already initialized Linice's X-Window driver by running xice program and have Linice popped up once onto the display.

VAR

(new) Defines a user variable to be a value or an expression. User variable can later be used within any expression, at which time it will be evaluated. Expressions and user variables may be nested. Example:

```
var u1 = eax + 1
var u2 = u1 + ebx
? u2
```

CALL <function-address> (arguments)

(new) Invoke an arbitrary function. If the symbol table is loaded, and you are in the context of a program or a module whose function you want to call, you can issue this call. Optionally, you may pass a number of arguments to the function. (The syntax is strict in that it requires you to type the enclosing "()" even if you don't specify any function parameters to prevent you from accidentally calling a function.)
Example: CALL funct (eax, 0)
Upon return, Linice displays the value of EAX register as the result of a function call. Calling arbitrary functions sometimes may have undesired side-effects. If you need to preserve CPU registers, you may want to use the CPU [s | r] commands.

CPU [s | r]

(enhanced) CPU command normally dumps all CPU registers. The enhanced command adds parameters "s" and "r" which will **s**ave and **r**estore general purpose CPU registers from an internal store.
For example, you may want to use this command to save registers before calling an arbitrary function (via command CALL), so they can be restored after being modified by a called function.

Breakpoints

(enhanced) A new flag has been added to all types of breakpoints – a "one-time" breakpoint flag, option "O" (a letter). When a breakpoint with this flag hits, that breakpoint will be cleared (deleted).
Example: BPX o module!getkey

Expressions

Linice expression evaluator provides full C-language operator precedence with a number of operators and functions. The following are supported operators:

<table>
<tr><td>||</td><td>logical-OR</td><td>( )</td><td>parenthesis support</td></tr>
<tr><td>&&</td><td>logical-AND</td><td>[ ]</td><td>array index</td></tr>
<tr><td>==</td><td>compare equality</td><td>|</td><td>bitwise-OR</td></tr>
<tr><td>!=</td><td>compare inequality</td><td>^</td><td>bitwise-XOR</td></tr>
<tr><td><<</td><td>shift left</td><td>&</td><td>bitwise-AND</td></tr>
<tr><td>>></td><td>shift right</td><td><</td><td>compare less</td></tr>
<tr><td><=</td><td>compare less or equal</td><td>></td><td>compare greater</td></tr>
<tr><td>>=</td><td>compare greater or equal</td><td>%</td><td>modulo</td></tr>
<tr><td>-></td><td>pointer operator</td><td>!</td><td>logical-NOT</td></tr>
<tr><td>+ - * /</td><td>math</td><td>~</td><td>bitwise-NOT</td></tr>
<tr><td>:</td><td>selector:offset</td><td></td><td></td></tr>
</table>

The following are the functions that can be called within any expression:

byte(..)	truncates the argument into a byte size (8 bits)
word(..)	truncates the argument into a word size (16 bits)
dword(..)	returns dword (32 bits)
hibyte(..)	returns high-order byte
hiword(..)	returns the upper word of the argument (bits [31:16])
sword(..)	convert byte into a signed word
ptr(..)	dereferences the argument assuming it is an address

The following functions do not take any arguments:

bpcount()	returns the breakpoint instance count
bpmiss()	breakpoint miss count
bptotal()	breakpoint total count
bpindex()	current breakpoint index number
bplog()	breakpoint silent log
DataAddr()	returns the address of the first data item displayed in the data window
CodeAddr()	returns the address of the first byte in the code window
EAddr()	returns the effective address of the current instruction
EValue()	returns the current value associated with EAddr()

The following tokens return the state of the corresponding EFLAGS register bits:

CFL	carry flag
PFL	Parity flag
AFL	Auxiliary flag
ZFL	Zero flag
SFL	Sign flag
OFL	Overflow flag
RFL	Resume flag
TFL	Trap flag
DFL	Direction flag
IFL	Interrupt flag
NTFL	Nested Task Flag
IOPL	IOPL level
VMFL	Virtual Machine flag

Numbers are evaluated in the following order:

Hexadecimal is the default radix for all numeric input and output except for selected commands such is window size. Hex number can optionally be prefixed by “0x”.

Decimal number is specified by explicit prefix “+” or “-“ (unary operator).

Binary number is specified with the prefix “0b” as in “0b1010”.

Octal number is specified with the prefix “0o” as in “0o3777”.

The *line number* operator “.” also changes the default radix to decimal.

Character literal values ‘123’ evaluate as the ascii string of up to 4 characters.

Character constants such as ‘\123’ the default radix is decimal unless the constant starts with ‘\x’ as in ‘\xABC’.

CPU registers and variations: AL, AH, AX, EAX, etc.

Breakpoint address such is ‘bp0’

Symbol specified using the name or module!name formats.

User variable that is going to be evaluated (see command VAR)

DOT-extension token implemented within the extension interface.

Not implemented commands

These commands are not (yet) ported from the SoftIce©:
A (Assemble code)
ADDR (Display/change address contexts)
BH (Breakpoint history)
BPINT (Breakpoint on interrupt)
DEVICE (Display info about a device)
GENINT (Generate an interrupt)
PAGEIN (Load a page)
PRN (Set printer output)
QUERY (Display a process virtual address space map)
SHOW (display from backtrace buffer)
SS (Search source module for string)
THREAD (Show thread information)
TRACE (Enter back trace simulation mode)
XG (Trace simulation)
XP (Step in trace simulation)
XRSET (Reset trace history buffer)
XT (Step in trace simulation)
XFRAME (Display active exception frames)

Implemented commands

```
 SETTING BREAK POINTS
BPM    - Breakpoint on memory access
BPMB   - Breakpoint on memory access, byte size
BPMW   - Breakpoint on memory access, word size
BPMD   - Breakpoint on memory access, double word size
BPIO   - Breakpoint on I/O port access
BPX    - Breakpoint on execution
```

```
BSTAT  - Breakpoint Statistics
```
MANIPULATING BREAK POINTS
```
BPE    - Edit breakpoint
BPT    - Use breakpoint as a template
BL     - List current breakpoints
BC     - Clear breakpoint
BD     - Disable breakpoint
BE     - Enable breakpoint
```
DISPLAY/CHANGE MEMORY
```
R      - Display/change register contents
U      - Un-assembles instructions
D      - Display memory
DB     - Display memory, byte size
DW     - Display memory, word size
DD     - Display memory, double word size
E      - Edit memory
EB     - Edit memory, byte size
EW     - Edit memory, word size
ED     - Edit memory, double word size
PEEK   - Read from physical address
PEEKB  - Read from physical address a byte
PEEKW  - Read from physical address a word
PEEKD  - Read from physical address a dword
POKE   - Write to physical address
POKEB  - Write to physical address a byte
POKEW  - Write to physical address a word
POKED  - Write to physical address a dword
H      - Help on the specified function
HELP   - Help on the specified function
?      - Evaluate expression
VER    - Linice version
WATCH  - Add watch variable
FORMAT - Change format of data window
DATA   - Change data window
```
DISPLAY SYSTEM INFORMATION
```
GDT    - Display global descriptor table
LDT    - Display local descriptor table
IDT    - Display interrupt descriptor Table
TSS    - Display task state segment
CPU    - Display cpu register information
PCI    - Display PCI device information
MODULE - Display kernel module list
PAGE   - Display page table information
PHYS   - Display all virtual addresses for physical address
STACK  - Display call stack
PROC   - Display process information
WHAT   - Identify the type of an expression
```
I/O PORT COMMANDS
```
I      - Input data from I/O port
IB     - Input data from I/O port, byte size
IW     - Input data from I/O port, word size
ID     - Input data from I/O port, double word size
O      - Output data to I/O port
OB     - Output data to I/O port, byte size
OW     - Output data to I/O port, word size
OD     - Output data to I/O port, double word size
```
FLOW CONTROL COMMANDS

```
X      - Return to host debugger or program
G      - Go to address
T      - Single step one instruction
P      - Step skipping calls, Int, etc.
HERE   - Go to current cursor line
HALT   - System APM Off
HBOOT  - System boot (total reset)
```

MODE CONTROL

```
I1HERE - Direct INT1 to LinICE, globally or kernel only
I3HERE - Direct INT3 to LinICE, globally or kernel only
ZAP    - Zap embedded INT1 or INT3
FAULTS - Enable/disable LinICE fault trapping
SET    - Change an internal system variable
VAR    - Change a user variable
```

CUSTOMIZATION COMMANDS

```
PAUSE  - Controls display scroll mode
ALTKEY - Set key sequence to invoke window
FKEY   - Display/set function keys
DEX    - Display/assign window data expressions
CODE   - Display instruction bytes in code window
COLOR  - Display/set screen colors
TABS   - Set/display tab settings
LINES  - Set/display number of lines on screen
WIDTH  - Set/display number of columns on screen
MACRO  - Define a named macro command
```

UTILITY COMMANDS

```
S      - Search for data
F      - Fill memory with data
M      - Move data
C      - Compare two data blocks
ASCII  - Prints an ASCII character table
```

LINE EDITOR KEY USAGE

```
up     - Recall previous command line
down   - Recall next command line
right  - Move cursor right
left   - Move cursor left
BKSP   - Back over last character
HOME   - Start of line
END    - End of line
INS    - Toggle insert mode
DEL    - Delete character
ESC    - Cancel current command
```

WINDOW COMMANDS

```
WC     - Toggle code window
WD     - Toggle data window
WL     - Toggle locals window
WR     - Toggle register window
WS     - Toggle call stack window
WW     - Toggle  watch window
EC     - Enter/exit code window
.      - Locate current instruction
```

WINDOW CONTROL

```
VGA    - Switch to a VGA text display
MDA    - Switch to a MDA (Monochrome) text display
XWIN   - Redirect console to a DGA frame buffer
SERIAL - Redirect console to a serial terminal
CLS    - Clear window
```

```
RS     - Restore program screen
ALTSCR - Change to alternate display
FLASH  - Restore screen during P and T
```

SYMBOL/SOURCE COMMANDS

```
SYM    - Display symbols
EXP    - Display exported symbols from a kernel or a module
SRC    - Toggle between source, mixed & code
TABLE  - Select/remove symbol table
FILE   - Change/display current source file
TYPES  - List all types, or display type definition
LOCALS - Display locals currently in scope
 SPECIAL OPERATORS
.      - Preceding a decimal number specifies a line number
@      - Preceding an address specifies indirection
```

Linsym – Symbol loader and translator

The following is a list of arguments supported by the `linsym` utility:

Option: --install

Short option: -i
Installs Linice debugger module and breaks. Optionally uses the environment variable LINICE to get to the Linice 'bin' directory where the kernel module is.
Example: `# linsym –i`

Option: --map <System.map>

Short option: -m <System.map>
In order to successfully load Linice, Linsym needs a current System.map file, which it will try to find at certain default locations (/boot/System.map, /boot/System.map-<kernel name>). If you have a custom-compiled kernel, the current system map may be at a different location, or may be even named differently. Use this option when loading Linice to specify the correct path and name of that file.
Example: `# linsym –m /boot/System.map-test -i`

Option: --uninstall

Short option: -x
This option uninstalls Linice debugger module.
Example: `# linsym –x`

Option: --translate <program>

Short option: -t <program>
Translates debug symbols from your module, kernel or executable program and creates a separate symbol file. This symbol file contains all the available *stabs* debug information compiled and linked in with the target program: global and local symbols, source code, type definition and other pertinent debugging information.
Example: `# linsym –t module.o`

Option: --output <alt_name.sym>

Short option: -o <alt_name.sym>
Specifies alternate file name for the symbol file generated by the translation (option "-t".)
Example: `# linsym –t module.o –o symbol.sym`

Option: --path <orig-path>:<new-path>

Short option: -p <orig-path>:<new-path>
Specifies path substitution for the source code. This option is useful when you are building a symbol file from another computer, and the source code that you would include resides on a different directory path. This option lets you substitute a path prefix. Note that the two paths are separated by a colon: the first path is the path that will be

matched against all original absolute paths in the stabs debugging section (specifying the path to the source), and the second path is the path to be used instead.
Example: `# linsym –t module –p /usr/src/mod:/mnt/usr/src/mod`

Option: --sym <symbol.sym>

Short option: -s <symbol.sym>
Loads one or more symbol files into the running Linice. You can load multiple symbol files by separating them with a colon ":". The names of symbol files usually end with the ".sym", which has to be specified as part of the file name. This is in contrast to the "unload" command, which specifies the base name only.
Example: `# linsym –s module.sym`

Option: --unload <symbol.sym>

Short option: -u <symbol>
Unloads one or more symbol files from the running Linice. You can unload multiple symbol files by separating them with a colon ":". You can also unload symbol files from within the Linice using the command "table". The specified name is the name of the symbol table as listed in the Linice using the "table" command. (It is not the file name which should be used when loading a symbol file.)
Example: `# linsym –u module`

Option: --logfile [<filename>][,append]

Short option: -l [<filename>][,append]
Saves the content of the Linice history buffer (the command line window) into a file. You can optionally provide a file name; if you don't, the default file name "linice.log" will be used. If the file already exists, it will be truncated, unless you specified ",append" also, which will preserve the original content and append a new one.
Example: `# linsym –l out.log,append`

Option: --verbose {0-3}

Short option: -v {0-3}
This option specifies the verbose level. The default is 0, which is silent.
If Linice refuses to load, or if you encounter some other error, repeat the command with the verbose level set to 3 and see if the message dump helps. If not, send me an email with the dump attached. The most common problem is incorrect system.map file (if you have compiled a custom kernel), or /dev/ice not deleted.

Linice configuration file: linice.dat

Configuration file “linice.dat”, which has to reside either in the current directory, or in the /etc directory, contains initial parameters and switches used to initialize Linice debugger. For most users, it should work just fine without any modifications.

This file is a text file, and it is fully compatible with the SoftIce version. This section describes the parameter values that are used by Linice (the rest of the keywords are ignored; they may be used in the future versions of Linice):

lowercase = [on | off]

Specifies if the disassembly will be shown in uppercase or lowercase letters.

sym = <buffer in Kb>

Linice will reserve this much memory for all the symbol files that are to be loaded. Be sure it is large enough for your current set of symbol tables since it cannot be modified without reloading the debugger.

hst = <buffer in Kb>

Specify the size of the history buffer (which is also known as command line buffer.) If you intend to save this buffer to a file, you may want to give it a larger size.

macros = <number>

Number of keyboard macros that are going to be allocated.

drawsize = <buffer in Kb>

If you intend to use Linice on top of the X-Window, specify the value of this buffer to be large enough to store the background frame buffer. The size of the buffer must be sufficient to store the rectangular frame that is obscured by the Linice window. The exact size depends on the current bits per pixel value as well as how many lines and what width do you wish to display in Linice.

init = <init commands;>

This line specifies a set of Linice commands that will be executed immediately upon Linice load.

F1…F12 Function keys assignment

SF1…SF12 Shift + Function keys assignment

AF1…AF12 Alt + Function keys assignment

CF1…CF12 Ctrl + Function keys assignment

Each key in these combinations may be assigned a command or set of commands.

layout = [country-code]

Specify the keyboard layout as a country code, which is described earlier. The default layout is US.

Advanced Topics – Debugger Extensions

Linice debugger supports custom plug-ins, so-called "dot-commands" (since they are typed after a dot/period on the command line).

Command: ?	List all standard Linice commands
Command: .?	List all registered custom dot commands
Command: .<cmd>	Execute a registered dot command

You can write a debugger extension that implements one or more dot-commands as a kernel loadable module. There are 2 functions provided by Linice which are used to register and unregister extension interface.

Please refer to the header file LiniceExt.h for the details of the interface. There is also a sample module to illustrate the function and capability of the interface. You can compile it and run it, or use it as a starting point for your own extensions.

*int LiniceRegisterExtension(TLINICEEXT *pExt);*

Use this function to register a debugger extension interface. It returns one of the error codes specified in the LiniceExt.h header file. TLINICEEXT is a structure that describes the interface and contains all the callback function pointers.

The caller is responsible to initialize these portions of the structure TLINICEEXT:

version
Set it to the macro LINICEEXTVERSION. This field is mandatory. Linice is using it to verify the correct and supported interface version.

size
Set it to the macro LINICEEXTSIZE. This field is also mandatory.

pDotName
Set it to the ASCIIZ name of the dot command that you implement using this interface structure. This field is mandatory. Your code may support a number of extensions, each being represented by one TLINICEEXT structure.

pDotDescription
This field is optional. If not used, set it to NULL; otherwise, set it to the ASCIIZ string with the description of the command. This string will be used when a user lists all dot commands using the command ".?". The description helps by telling what the command does, and serves no other purpose.

```
int (*Command)(char *pCommand);
```

Specify the address of your function which will be called when a dot command is invoked. This can be NULL, which would make sense only if you don't intent to provide a command handler, but only a function token handler (described below). Note that you still need to provide pDotName string even if you don't have a handler.
When this function gets called, argument *pCommand* points to the arguments of the command, or the rest of the line.
Example: If you registered a command "dump", and the user typed ".dump 1 2 3" on the command line, *pCommand* would point to " 1 2 3" (also note the space before "1".)

```
void (*Notify)(int Notification);
```

Specify the address of your function which will be called on various debugger system events. This can optionally be NULL if you don't care about the events.
The debugger events ("*Notification*") are:

```
PEXT_NOTIFY_ENTER – Linice got control. Break into debugee.
PEXT_NOTIFY_LEAVE – Linice released control. Debugee continues to run.
```

```
int (*QueryToken)(int *pResult, char *pToken, int len);
```

Specify the address of your function which will be called when an unknown token is encountered within an expression. Your extension may be able to help parsing expressions and provide values to add-on functions or tokens.
These are the parameters sent by Linice: *pToken* points to the start of the expression token, *len* specifies the suggested length of the token, and *pResult* points to a variable of the type 'int' where you should store the result.
After you examine a token (probably using a function 'strncmp()' or similar), if you detect that you don't handle it, simply return 0.
If you do handle the token, store the final value in the **pResult* and return the number of characters to advance past the token size. This may or may not be the same value as parameter *len*.
If the token is a function, it is possible to recursively call the expression evaluator for the function arguments in parenthesis. The sample extension module shows how to do it.

After the extension is successfully registered, Linice fills in the pointers to some of its utility functions that your module can call:

```
TLINICEREGS *(*GetRegs)(void);
```

Returns the address of the internal Linice structure that holds the CPU registers of the program being debugged. This address will not change for the duration of Linice session. You can read and write CPU registers when the debugger is active.

```
int (*Eval)(int *pValue, char *pExpr, char **ppNext);
```

This function evaluates a string expression into a number. One possible use is to resolve an expression that a user might have typed as arguments to your dot command.
Set *pValue* to where the result should be stored. Set *pExpr* to the expression string, and set *ppNext* to the char* variable to receive the end of the evaluated string. (*ppNext* is optional: if you don't care where the expression ended, set it to NULL. If you need to parse multiple expressions that are given one after the other, then you may want to know where the previous expression ended.)

On success, the function returns a non-zero value. On failure, the function returns 0, *pValue* is not modified, and *ppNext* points to the character which caused the error.

```
int (*Disasm)(char *pBuffer, int sel, int offset);
```
Disassemble a line of x86-code into your buffer from the given address. *pBuffer* needs to be at least 80 characters long to store disassembled instruction. *sel* and *offset* define the target address. If you specify 0 for *sel*, kernel CS will be used instead.

```
int (*dprint)(char *format,...);
```
This function provides a way to print out any message with variable number of arguments into the Linice command buffer. It works similar to the standard "C" function printf(), except that you should not use special character for new line "\n".

```
int (*Execute)(char *pCommand);
```
This function executes any command that you might also be able to type on the command line.

```
int (*Getch)(int fPolled);
```
Reads and returns a character from the input stream. Use this function for the interactive option menus within the extension handler. Set *fPolled* to TRUE to have the function wait until a key becomes available.

```
int (*MemVerify)(int sel, int offset, int size);
```
This function verifies that a range of memory addresses is present and accessible. When it returns a nonzero value, you can access the memory range using your pointers. *sel* and *offset* specify the start address, and *len* is the size of the memory range in bytes. If you specify 0 for *sel*, kernel DS will be used instead.

*void LiniceUnregisterExtension(TLINICEEXT *pExt);*

Use this function to unregister previously registered interface. Be sure to call this function before unloading your extension module, so the Linice stops calling it via the registered structure. Failure to do so will result in a crash.

You have to unload all of your extension modules before you unload Linice, since the modules are linked to Linice by the means of those two exported functions. Run the Linux command "lsmod" to see the module dependency. Alternatively, list the registered extensions with the Linice command ".?".

Be sure not to call any of the callbacks from the extension interface structure if the registration failed.