

Runtime debugging in embedded systems

available tools and usage

Nicholas Mc Guire
Opentech EDV-Research GmbH
Mistelbach, Austria
der.herr@hofr.at, <http://www.opentech.at>

Document Control

Version	Author	Date	Comment
1.0	Nicholas Mc Guire	2nd Feb 2004	init release

Abstract

The issue of runtime debugging in embedded systems arises with more or less any product simply because there is no such thing as bug-free code. For this reason there are two principal demands on a embedded runtime system.

- The ability to detect error-nous behavior
- The ability to locate the error in a way that allows fixing it in a short time.

The first issue is "solved" with watchdog, respawning processes when they exit, server structures where client processes can be re launched, etc.

The issue of localizing the fault is not so often treated, in fact in the consulting and support work I've been involved in to date, not a single embedded development environment was targeting this issue explicitly. Surprisingly enough most the necessary tools are there - they just seem to be unused, and most of them unknown. So if you always wondered what `libSegFault.so` is, what `/proc/PID/statm` contains, `bgcc...` or never noticed any of these, then this article is for you.

1 doc-ctrl

2 Introduction

Embedded system development generally has a higher emphasis on testing and system evaluation than a desk-top system. In an embedded system a failure or a error-nous behavior can stay unnoticed for quite some time, simply because nobody is going to notice it until a service his hurt or the device stops responding. But far before this situation occurs, problems may be noticeable and systems can be corrected before they fail. To achieve such behavior, or to at least improve the embedded systems behavior, a system needs to take system monitoring and post-mortem analysis into account from the very beginning. In this article we will scan some of the available resources for embedded GNU/Linux systems, that allow to target some of these long term debugging problems.

- memory leaks

- "random" segfaults
- resource access problems
- locating performance bottle-necks

3 Memory leaks

Memory leaks are a common reasons for embedded systems to slowly but shurly lock up, locating such problems should actually happen during system testing. But as many embedded systems utilize 'out-of-the-box' distributions an assessment of the core OS should be done.

A quick scan of /usr/sbin on my default system shows that there are a number of potential problems in this instalation - none of these might ever cause any real problems - but it clearly shows that checking for memory leaks is not being performed by some of the developers...

```
root@rtl14:~ # for APP in `ls /usr/sbin/*` ; do echo -n "$APP : " ; \
  mtrace $APP ; done 2>&1 | tee mtrace.sbin
root@rtl14:~ # cat mtrace.sbin
```

```
/usr/sbin/adsl-setup :- 0000000000 Realloc 300 was never alloc'd
/usr/sbin/lprng_certs :
Memory not freed:
```

```
-----
  Address      Size      Caller
0000000000      0 at
0000000000      0 at
/usr/sbin/nmbd :- 0000000000 Realloc 4481 was never alloc'd
```

```
Memory not freed:
-----
  Address      Size      Caller
0000000000      0 at
/usr/sbin/papd :- 0000000000 Realloc 58 was never alloc'd
/usr/sbin/sendmail :- 0000000000 Realloc 694 was never alloc'd
/usr/sbin/smbd :- 0000000000 Realloc 6762 was never alloc'd
...
```

These results would not be acceptable for a system that needs to give any reliability guarantee, unfortunately similar results can be reproduced on many systems. If an embedded system would be relying on any of these applications, a close investigation of the potential problems would need to be performed.

Note that reallocating memory that was never allocated will generally work, but the memory is not initialized which can cause hard to detect side effects. Not freeing memory, aside from resulting in memory "disapearing" can be a security problem.

Also it is quite typical for short lived applications to not bother with freeing up allocated memory, but just exiting and leaving it to the OS to clean it all up. This is also true for some system applications like /bin/lis .

3.1 top

To check if an application is actually causing a memory leak one can monitor the system via top.

All that one need to do is lauch top and use the "M" command to sort the list by memory usage, any memory leakage would show up after a while when the process slowly crawls to the top of the list.

```
rtl14:~ # top
```

```
...
  PID USER      PRI  NI  SIZE  RSS SHARE STAT  %CPU %MEM    TIME CPU  COMMAND
26343 root         15   0  1104 1104   816 R    0.5  0.4    0:00  0  top
26291 root         14   0  1672 1672  1508 S    0.1  0.7    0:01  0  sshd
   1 root         12   0    76   68    52 S    0.0  0.0    0:54  0  init
   2 root         12   0     0    0     0 SW   0.0  0.0    0:00  0  keventd
   3 root         19  19     0    0     0 SWN  0.0  0.0    0:00  0  ksoftirqd_CPU
   4 root         12   0     0    0     0 SW   0.0  0.0    9:54  0  kswapd
```

Hit M to sort by memory usage:

```
  PID USER      PRI  NI  SIZE  RSS SHARE STAT  %CPU %MEM    TIME CPU  COMMAND
26291 root         13   0  1672 1672  1508 S    0.4  0.7    0:01  0  sshd
26079 root         12   0  1420 1420  1312 S    0.0  0.6    0:01  0  sshd
26293 root         14   0  1344 1344  1052 S    0.0  0.6    0:00  0  bash
26367 root         16   0  1108 1108   820 R    1.2  0.4    0:00  0  top
25997 nobody       12   0  1056 1052   848 S    0.0  0.4    0:00  0  in.identd
```

Unfortunately top is quite inefficient as it performs a few hundred system call per output screen scanning the /proc filesystem. Generally top will not be available on an embedded system. So the solution is to use the same method as in top, but to scan the appropriate file manually which then only requires a command like "cat":

```
root@rtl14:~# chroot /opt/ecdk-0.1-i386/i386/dev_fs/
```

```
BusyBox v0.60.5 (2003.12.05-18:55+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.
```

```
# ps | grep sshd
 1086      root          3080    S  /usr/sbin/sshd
  8281      root          5732    S  sshd: root@pts/0
21332      root          5692    S  sshd: root@pts/1
23747      root          5696    S  sshd: root@pts/2
# cat /proc/1086/statm
355 355 328 36 0 319 55
#
```

doing this a few times manually or via a cron job can reveal memory leak problems. The meaning of the individual values listed are shown in the following table

Field	Description
size	total program size
resident	size of memory portions
shared	shared pages
trs	'code' pages
drs	data/stack
lrs	library
dt	dirty pages

Table : Contents of the statm files (note: one page = 4096bytes)

This might not seem very elegant - but on the other hand its a very simple way to locate long term memory leakage in the embedded environment. The ammount of data that is produced is fairly small so storing it locally over a longer period of

time should not be too big a problem. Testing on the embedded system is crucial as this might simply never happen on the development system, and a quick scan through the /proc filesystem in a running system can locate such problem far before the actually inflict any damage on the device.

3.2 bgcc

For applications being developed from scratch the search for memory leaks should start a bit earlier, gcc provides extensions for bounds checking, which reveals memory leaks at the compile stage. The bounds checking gcc is called bgcc and is available as a patch to mainstream gcc. bgcc [?] can provide more than just locating of memory leaks, but this is out side the scope of this article.

3.2.1 The good news on bgcc

BGCC can be reconfigured at runtime to pinpoint the exact problems via the GCC_BOUNDS_OPTS. To retrieve a list of available options just export GCC_BOUNDS_OPTS=--help and execute any program that was compile with bgcc.

```
rtl14:~ # export GCC_BOUNDS_OPTS=--help
rtl14:~ # ./hello
```

You may supply a list of the following arguments to a bounds-checked program by listing them in the environment variable 'GCC_BOUNDS_OPTS' before running the program. Separate the arguments by spaces.

General:

-no-message	Don't print introductory message.
-no-statistics	Don't print statistics.
-, -help	Print this table of usage.

Control runtime behaviour:

-array-index-check	*Check the index of all array references.
-no-array-index-check	Only check the pointer is within the array.
-never-fatal	Don't abort after a bounds error.
-reuse-heap	*Re-use the heap.
-reuse-age=<age>	Set the age limit before freeing (default: 0).
-no-reuse-heap	Never really free old heap blocks.
-warn-unchecked-statics	Warn if unchecked static objects are referenced.
-no-warn-unchecked-statics	*Switch off the above.
-warn-unchecked-stack	Warn if unchecked stack objects are referenced.
-no-warn-unchecked-stack	*Switch off the above.
-warn-free-null	*Warn if free (0) is used.
-no-warn-free-null	Switch off the above.
-warn-misc-strings	*Warn for miscellaneous strings usage.
-no-warn-misc-strings	Switch off the above.
-warn-illegal	Warn when ILLEGAL pointers are created.
-no-warn-illegal	*Switch off the above.
-warn-unaligned	*Warn when pointers are used unaligned.
-no-warn-unaligned	Switch off the above.
-warn-overlap	*Warn if memcpy arguments overlap.
-no-warn-overlap	Switch off the above.
-warn-compare-objects	*Warn if comparing pointers to different objects.
-no-warn-compare-objects	Switch off the above.
-warn-all	Turn on all warnings.
-print-heap	Print all heap data at exit.
-no-print-heap	*Don't print heap data at exit.

Debugging:

-print-calls	Print calls to the bounds-checking library.
--------------	---

```
-no-print-calls          *Don't print calls.  
Note: '*' means this is the default behaviour.
```

If one need to monitor a particular problem then simply setting up the bgcc environment will reduce the output to an absolute minimum.

```
}  
root@rtl14:/tmp # export GCC_BOUNDS_OPTS="-print-heap \  
-no-message -no-statistics"  
root@rtl14:/tmp # ./hello  
doing something  
dev_kit check  
Filename = <noname>, Line = 0, Function = calloc, Count = 1 Size = 96452
```

As the output of bgcc informations on exit go to stderr one can redirect them to prevent any irritations for users and still get all the informations needed to locate potential problems. Many little glitches might go unnoticed for quite some time, unfortunately bgccs output is limited in the sense that it does not report any commandline parameters that could help associate detected errors with a specific invocation. But it is not too hard to add this to the executables at startup. Simply dump anything of interest to stderr with `fprintf(stderr, "`

```
}  
root@rtl14:/tmp # ./hello  
./hello  
dev_kit check  
Bounds Checking GCC v gcc-2.95.3-2.20 Copyright (C) 1995 Richard W.M. Jones  
Bounds Checking comes with ABSOLUTELY NO WARRANTY. For details see file  
'COPYING' that should have come with the source to this program.  
Bounds Checking is free software, and you are welcome to redistribute it  
under certain conditions. See the file 'COPYING' for details.  
For more information, set GCC_BOUNDS_OPTS to '-help'  
Bounds library call frequency statistics:  
Calls to push, pop, param function:      1, 1, 2  
Calls to add, delete stack:              2, 2  
Calls to add, delete heap:                1, 0  
Calls to check pointer +/- integer:      0  
Calls to check array references:          0  
Calls to check pointer differences:       0  
Calls to check object references:         0  
Calls to check component references:      0  
Calls to check truth, falsity of pointers: 0, 0  
Calls to check <, >, <=, >= of pointers:  0  
Calls to check ==, != of pointers:       0  
Calls to check p++, ++p, p--, --p:       0, 0, 0, 0  
References to unchecked static, stack:   0, 0
```

Both the output and the configuration is sufficiently simple that this can be used by untrained personell to deliver better error information when contacting service points. A dump showing the executable invoked, the passed arguments and the bgcc output can help pinpoint problems a lot.

3.2.2 The bad news on bgcc

The bad news is simple - size.

```
}
```

```
root@rtl14:/tmp # ls -l hello
-rwxr-xr-x  1 root  root  95964 Jan 31 16:10 hello*
root@rtl14:/tmp # file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), stripped
```

94k for a hello world is not bad.

3.3 ltrace

ltrace, discussed later, can also help in locating memory leaks simply by checking the calls to malloc and frinds during execution. But memory-leak detection is not the prime functionality of ltrace.

3.4 njamd

njamd is a malloc debugger - well it calls it selfe "Not Just Another Malloc Debugger" but that is its prime usage - for POSIX systems.

njamd is an example of using library prelaod techniques, discussed later on, is libnjamd, which is a malloc debugger implemented as preloadable library which "wraps" the normal libc memory allocation library:

```
void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

The basic interface consists of a preloaded library and some environment variables to control its behavior, in addition issuing signal SIGUSR1 will dump memory leak information when running under control of gdb. These memory usage diagnostics dumped to standard error are fairly human readable and allow quick pinpointing of problems.

njamd also commes with a nice utility called njamdpm that allows to do postmortem heap analysis. For more on njamd read up at [?].

3.4.1 using njamd

To use njamd you need to build and install it, which ran out of the box for me with the cvs download version. We need an application with some memory leaks and a segfault - so here is the modified hello.

```
#include <stdlib.h> /* getenv */
#include <stdio.h>
main(){
  char *junk;
  char *junk2;
  junk = (char *)malloc(6);
  junk2 = (char *)malloc(32);
  sprintf(junk, "123456");
  free(junk);
  return 0;
}
```

As the string "123456" is seven chars long the malloc'ed junk is too small, this will segfault. Calling it with libnjamd.so preloaded not only catches the segfault, but displays memory leak information

```
root@rtl14:~# LD_PRELOAD=libnjamd.so ./hello
```

```
Segmentation fault (caught by NJAMD)
```

```
NJAMD: Cause of fault: Access to protected region
```

```
Offending Pointer: 0x402c4000
```

```
called from ./hello(malloc+0x184) [0x8048478]
```

```
called from /lib/libc.so.6(__libc_start_main+0xc6) [0x40054d06]
```

```
called from ./hello(free+0x4d) [0x8048371]
```

```
0x402c3ffa-0x402c3fff: Aligned length 6
```

```
Allocation callstack:
```

```
called from ./hello(malloc+0x15e) [0x8048452]
```

```
called from /lib/libc.so.6(__libc_start_main+0xc6) [0x40054d06]
```

```
called from ./hello(free+0x4d) [0x8048371]
```

```
Not Freed
```

```
NJAMD totals:
```

```
Allocation totals:      2 total, 2 leaked
```

```
Leaked User Memory:    38 bytes
```

```
Peak User Memory:      38 bytes
```

```
NJAMD Overhead at user peak:  7.963 kB
```

```
Peak NJAMD Overhead:      7.963 kB
```

```
Average NJAMD Overhead:                3.981 kB per alloc
```

```
Address space used:      16.000 kB
```

```
NJAMD Overhead at exit:                7.963 kB
```

```
0x402c3ffa-0x402c3fff: Aligned length 6
```

```
Allocation callstack:
```

```
called from ./hello(malloc+0x15e) [0x8048452]
```

```
called from /lib/libc.so.6(__libc_start_main+0xc6) [0x40054d06]
```

```
called from ./hello(free+0x4d) [0x8048371]
```

```
Not Freed
```

```
0x402c5fe0-0x402c5fff: Aligned length 32
```

```
Allocation callstack:
```

```
called from ./hello(malloc+0x16e) [0x8048462]
```

```
called from /lib/libc.so.6(__libc_start_main+0xc6) [0x40054d06]
```

```
called from ./hello(free+0x4d) [0x8048371]
```

```
Not Freed
```

```
Aborted
```

The output is dumped on exit in a fairly human readable form, atleast for the above case diagnostics of the problem are trivial. Note that this works for stripped binaries and stripped libnjamd.so in which case it is reduced to 85k. There is a certain runtime overhead though as the above report shows.

4 Sporadic Segfaults

We start out with a deterministic segfault:

```

rtl14:~ # cat segfault.c

#include <stdio.h>
main(){
    char *junk=NULL;
    sprintf(junk,"my segfault");
    printf("%s\n",junk);
    return 0;
}

rtl14:~ # cc segfault.c -o segfault
rtl14:~ # ./segfault
Segmentation fault

```

So this little program works just as expected - but if this were in the middle of your application server and it only happens once every few days on the target board and is not reproducible - what do you do ?

4.1 catchsegv

catchsegv is a shell script - nothing really fancy - what this does is use the libSegFault.so that is available in glibc by default. libSegFault is used by preloading it - more on that below.

```

rtl14:~ # catchsegv ./segfault
*** Segmentation fault
Register dump:

EAX: 00000073   EBX: 4015ed24   ECX: 03fffffff   EDX: 00000000
ESI: 00000000   EDI: bffff820   EBP: bffff168   ESP: bffff130

EIP: 4009f4a2   EFLAGS: 00010246

CS: 0023   DS: 002b   ES: 002b   FS: 0000   GS: 0000   SS: 002b

Trap: 0000000e   Error: 00000006   OldMask: 00000000
ESP/signal: bffff130   CR2: 00000000

Backtrace:
/lib/libc.so.6(_IO_str_overflow+0x122) [0x4009f4a2]
/lib/libc.so.6(_IO_default_xsputn+0x98) [0x4009e3d8]
/lib/libc.so.6(_IO_vfprintf+0x1c6) [0x40076326]
/lib/libc.so.6(vsprintf+0x8c) [0x400936bc]
/lib/libc.so.6(sprintf+0x2d) [0x4008076d]
?:0(main) [0x8048393]
/lib/libc.so.6(__libc_start_main+0xc6) [0x40041d06]
./segfault(__libc_start_main+0x59) [0x80482d1]

Memory map:

08048000-08049000 r-xp 00000000 03:02 3965059 /root/segfault
08049000-0804a000 rw-p 00000000 03:02 3965059 /root/segfault
0804a000-0804e000 rwxp 00000000 00:00 0
40000000-40015000 r-xp 00000000 03:02 655389 /lib/ld-2.3.2.so
40015000-40016000 rw-p 00014000 03:02 655389 /lib/ld-2.3.2.so
40016000-40019000 r-xp 00000000 03:02 655388 /lib/libSegFault.so

```

```
40019000-4001a000 rw-p 00002000 03:02 655388 /lib/libSegFault.so
4002b000-4002c000 rw-p 00000000 00:00 0
4002c000-4015b000 r-xp 00000000 03:02 655392 /lib/libc-2.3.2.so
4015b000-40160000 rw-p 0012f000 03:02 655392 /lib/libc-2.3.2.so
40160000-40162000 rw-p 00000000 00:00 0
bffffe000-c0000000 rwxp fffff000 00:00 0
```

Lets verify the results by using gdb on the same segfault.

```
rtl14:~ # gdb ./segfault
GNU gdb 5.3
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-slackware-linux"...
(gdb) run
Starting program: /root/segfault

Program received signal SIGSEGV, Segmentation fault.
0x4009a4a2 in _IO_str_overflow_internal () from /lib/libc.so.6
(gdb) bt
#0 0x4009a4a2 in _IO_str_overflow_internal () from /lib/libc.so.6
#1 0x400993d8 in _IO_default_xsputn_internal () from /lib/libc.so.6
#2 0x40071326 in vfprintf () from /lib/libc.so.6
#3 0x4008e6bc in vsprintf () from /lib/libc.so.6
#4 0x4007b76d in sprintf () from /lib/libc.so.6
#5 0x08048425 in main () at segfault.c:4
#6 0x4003cd06 in __libc_start_main () from /lib/libc.so.6
(gdb) quit
The program is running. Exit anyway? (y or n)
```

So we can get the same infomation from the backtrace in gdb or from the dump that catchsegfault produced - just that libSegFault.so is only a few kB.

```
-rwxr-xr-x 1 root root 14616 May 19 2003 /lib/libSegFault.so*
```

Building for the chroot environment:

The makefile is not very elegant as all the cross build related entries are hard coded - but for segfaulting this should do:

```
ALL: segfault

CROSS:=i386-linux
CC:=$(CROSS)-gcc
STRIP:=$(CROSS)-strip
C_FLAGS:= -Wall -Os -march=i386
INCLUDE:=-I/opt/ecdk-0.1-i386/i386/include/
LD_FLAGS:=-Wl,-rpath,/lib
STRIP_FLAGS:=-s --remove-section=.note --remove-section=.comment hello hello.o

segfault: segfault.c
    $(CC) $(C_FLAGS) $(INCLUDE) -c segfault.c -o segfault.o
    $(CC) $(LD_FLAGS) -o segfault segfault.o
```

```
$(STRIP) $(STRIP_FLAGS) segfault
```

```
clean:
```

```
rm -f *.o segfault
```

Note that we are going to trace a stripped file - so this is what you typically want to be able to do on an embedded system - trace some stripped executable without the need to actually modify it or provide any large tool-chain.

what catchsegv ./segfault actually is doing is loading libsegfault.so so you can achieve the same results on an embedded system by doing

```
# export LD_PRELOAD=${LD_PRELOAD:+${LD_PRELOAD}:}/lib/libSegFault.so
# export SEGFAULT_USE_ALTSTACK=1
# export SEGFAULT_OUTPUT_NAME=/tmp/segfault.trace
```

in case you don't have any other preloaded libs the following is sufficient

```
# export LD_PRELOAD=/lib/libSegFault.so
```

back to the embedded fs again (don't have bash on it so we need to set the shell in our current session before change rooting again).

```
root@rtl14:~# export SHELL=/bin/ash
root@rtl14:~# chroot /opt/ecdk-0.1-i386/i386/dev_fs/
```

```
BusyBox v0.60.5 (2003.12.05-18:55+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.
```

```
# export LD_PRELOAD=/lib/libSegFault.so
# export SEGFAULT_USE_ALTSTACK=1
# export SEGFAULT_OUTPUT_NAME=/tmp/segfault.trace
# cd /tmp
# ls
hello                segfault
# ./segfault
Segmentation fault
# ls
hello                segfault            segfault.trace
# cat segfault.trace
*** Segmentation fault
Register dump:
```

```
EAX: 00000000  EBX: 4012eb90  ECX: 03ffffff  EDX: 00000073
ESI: bffff80c  EDI: 00000000  EBP: bffff17c  ESP: bffff154
```

```
EIP: 40083d64  EFLAGS: 00010246
```

```
CS: 0023  DS: 002b  ES: 002b  FS: 0000  GS: 0000  SS: 002b
```

```
Trap: 0000000e  Error: 00000006  OldMask: 00000000
ESP/signal: bffff154  CR2: 00000000
```

```
Backtrace:
```

```
/lib/libc.so.6(_IO_str_overflow+0x13c) [0x40083d64]
/lib/libc.so.6(_IO_default_xsputn+0xab) [0x40082ec7]
```

```

/lib/libc.so.6(_IO_vfprintf+0x1f3) [0x40062d5f]
/lib/libc.so.6(vsprintf+0x63) [0x4007ab6b]
/lib/libc.so.6(sprintf+0x25) [0x4006b91d]
./segfault [0x8048429]
/lib/libc.so.6(__libc_start_main+0xbb) [0x4003216f]
./segfault(__libc_start_main+0x51) [0x8048365]
# exit

```

note that there is no memory map dump on the embedded fs which is due to a problem in libSegFault when cross compiling if you find a solution to this little glitch let me know. The essential information though is that we can pinpoint the cause for the segfault.

libSegFault.so can not only be preloaded for an individual application (actually the LD_PRELOAD is active for a given shell session) but can be put into the file /etc/ld.so.preload to be active on a system scope - if this is done - don't forget to define an appropriate output file.

Non-exhaustive tests showed no performance penalty for using libSegFault - so this might well be a generally recommendable way of ensuring that sporadic segfaults can be traced.

4.2 core dumps

For quite some time generating core-dumps was a common practice, but it seems to have gotten out of "style"... One of the problems for embedded applications is that the core files generated can be quite large, clearly they give you the best picture of what happened if they can be retrieved and passed on to the responsible programmer.

Some daemons will provide core files in a configurable manner - like squid offers a coredump tag in its config file. Generally this is a good idea to put any generated core files in a well defined place or the usefulness of these will diminish simply because nobody knows where to look and if they stumble across one what to do with it.

```

# TAG: coredump_dir
# By default Squid leaves core files in the first cache_dir
# directory. If you set 'coredump_dir' to a directory
# that exists, Squid will chdir() to that directory at startup
# and coredump files will be left there.

```

As the comment here shows, all your app needs to do is to chdir before segfaulting, so our segfault program from above would simply be:

```

#include <stdio.h>
#include <unistd.h>
main(){
    char *junk=NULL;

    chdir("/tmp/");
    sprintf(junk,"some string to segfault it");
    printf("%s\n",junk);
    return 0;
}

```

And the core files are located in /tmp/ now. Note that you can use the file command to check what application caused the core dump

```

root@rtl14:/tmp # file /tmp/core
/tmp/core: ELF 32-bit LSB core file Intel 80386, version 1 (SYSV), SVR4-style, from 'segfault'

```

All you need to do to enable core-dumps is to set the appropriate user limit (default being 0 - no core files generated). so

```

rtl14:~ # export SHELL=/bin/ash
rtl14:~ # chroot /opt/ecdk-0.1-i386/i386/dev_fs

BusyBox v0.60.5 (2003.12.05-18:55+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

```

```

# ulimit -c 200
# cd /tmp
# ./segfault
Segmentation fault - core dumped
# ls
core                segfault            segfault.trace
hello               segfault.segv.5193
# ls -l
-rw-----   1 root    root          69632 Dec  7 12:58 core
-rwxr-xr-x   1 root    root           2488 Dec  5 19:31 hello
-rwxr-xr-x   1 root    root           2568 Dec  5 19:38 segfault
-rw-r--r--   1 root    root            723 Dec  6 09:34 segfault.trace
# exit

```

Note that the ulimit unit is 1k by default, so the core file was limited to 200k, as can be seen the core file of the segfault event for this trivial case is about two orders larger than what libSegFault produced. So for some systems core files may be unacceptable. Not also that the limit set with ulimit is a hard limit so in case the core file exceeds this size you simply get no usefull core file resulting most likely in no usefull diagnostics.

```

root@rtl14:~ # ulimit -c 20
root@rtl14:~ # ./segfault
Segmentation fault (core dumped)
root@rtl14:~ # ls -l core
-rw-----   1 root    root          20480 Dec  7 14:20 core
root@rtl14:~ # gdb --core core
GNU gdb 5.3
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-slackware-linux".
Core was generated by './segfault'.
Program terminated with signal 11, Segmentation fault.
#0  0x4009a4a2 in ?? ()
(gdb) bt
#0  0x4009a4a2 in ?? ()
Cannot access memory at address 0xbffff1a8
(gdb)

```

Here we could extract no more from the core file than the fact that the termination was due to a segmentation fault.

4.3 Core file names

The default in GNU/Linux is to name core file core, other UNIX flavors allow to configure the name (i.e. via sysctl in BSD), in Linux the core file base name can be set via sysctl and in addition the pid can be added

```

root@rtl14:/proc/sys/kernel# cat core_uses_pid

```

```
0
root@rtl14:/proc/sys/kernel# cat core_pattern
core
```

The default being to not append the pid.

```
root@rtl14:~# echo 1 > /proc/sys/kernel/core_uses_pid
root@rtl14:~# ./segfault
Segmentation fault (core dumped)
root@rtl14:~# ls -l core.*
-rw-----  1 root    root      53248 Dec  7 15:46 core.17193
root@rtl14:~#
```

This does not really help much unless you record all PIDs when launching applications... A more elaborate patch is available though and for distributed embedded systems and clusters this is quite interesting.

The Patch by Michael Sinz provides the following format options :

Format	Description
%P	process ID, pid
%U	user ID, uid
%N	commandline name of the process
%H	hostname
%%	A "%"

Table : Format options available in the coredump-file-control patch

```
sysctl -w "kernel.core_name_format=/coredumps/
```

```
echo "/coredumps/%U/%N-%P.core" > /proc/sys/kernel/core_name_format
```

putting the core files in user specific directories like

```
/coredumps/root/segfault-8023.core
```

4.4 debugging core files

After generating a core file in the embedded filesystem located at /opt/ecdk-0.1-i386/i386/dev_fs/ in my case, this core file is analyzed with gdb.

```
rtl14:~ # cd /opt/ecdk-0.1-i386/i386/dev_fs/
rtl14:~ # gdb
GNU gdb 5.3
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-slackware-linux".
(gdb) set solib-absolute-prefix /opt/ecdk-0.1-i386/i386/dev_fs/
(gdb) file /usr/src/dev_kit/test/segfault
Reading symbols from /usr/src/dev_kit/test/segfault...
(no debugging symbols found)...done.
```

```

(gdb) target core tmp/core
Core was generated by './segfault'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /opt/ecdk-0.1-i386/i386/dev_fs/lib/libc.so.6...done.
Loaded symbols for /opt/ecdk-0.1-i386/i386/dev_fs/lib/libc.so.6
Reading symbols from /opt/ecdk-0.1-i386/i386/dev_fs/lib/ld-linux.so.2...done.
Loaded symbols for /opt/ecdk-0.1-i386/i386/dev_fs/lib/ld-linux.so.2
#0 0x4007fd64 in _IO_str_overflow (fp=0xbffff87c, c=115) at strops.c:178
178      *fp->_IO_write_ptr++ = (unsigned char) c;
(gdb) bt
#0 0x4007fd64 in _IO_str_overflow (fp=0xbffff87c, c=115) at strops.c:178
#1 0x4007eec7 in _IO_default_xsputn (f=0xbffff87c, data=0x8048490, n=26)
    at genops.c:466
#2 0x4005ed5f in _IO_vfprintf (s=0xbffff87c,
    format=0x8048490 "some string to segfault it", ap=0xbffff95c)
    at vfprintf.c:1307
#3 0x40076b6b in _IO_vsprintf (string=0x0,
    format=0x8048490 "some string to segfault it", args=0xbffff95c)
    at iovsprintf.c:46
#4 0x4006791d in sprintf (s=0x0,
    format=0x8048490 "some string to segfault it") at sprintf.c:38
#5 0x08048429 in sprintf () at sprintf.c:37
#6 0x4002e16f in __libc_start_main (main=0x8048414 <sprintf+224>, argc=1,
    ubp_av=0xbffff9d4, init=0x80482bc, fini=0x804846c <sprintf+312>,
    rtdl_fini=0x40009cf0 <_dl_fini>, stack_end=0xbffff9cc)
    at ../sysdeps/generic/libc-start.c:129
(gdb)

```

the procedure is no different if we would have a "real" cross-debug session, in this case its i386 on a i686. If you copy the core files from the target system to your local host system then you must make sure you have a identical filesystem structure and the same library versions etc. or your results may be garbage. Unfortunately it is not so that you get NO results just because you use wrong libs ! if you run the above session without the set solib-absolute-prefix command gdb will silently use the host libs and may point you in a totally wrong direition.

5 Performance issues

Linux user-land is designed for desk-top setups, there are a number of problems for embedded systems when transferring des-top apps to embedded environments. Some of the key issues are:

- too many options
- too much processing spent for pritty output
- testing for many different setups and default config files
- rich feture set not needed on embedded systems

These issues arise basically because most embedded console interfaces are not designed around the assumption that someone is going to sit there 8 hours a day, but they are for system maintenance and inspection on errors only. UNIX itselfe has a large set of redundant functionality, just thinkg of dropping a file to screen, you can use cat, more, tail, sort, grep and even dd..., so the first step is to eliminate redundancy on the application layer. The next step is to locate poisble perfomance bottle necks that can be induced by rarely run applications. Just think of putting top on a system for monitoring, this well can produce a system load that is problematic, even on a PI 200 MHz top can consume 5-10% of the CPU !

Locating performance problems requires:

- resolve details of library dependencies
- check system call usage
- profile application internal calls

Once this has been done one has a set of potential hot-spots in the code and then can go on to explicitly tracing these events.

5.1 ltrace

ltrace can help you locate application bottlenecks - basically what you would want to do is build your application using standard libraries - but after that you analyze it to see where most of the time is spent.

```
root@rtl14:~/junk# ltrace -c -ttt ls -l
total 0
-rw-r--r--  1 root    root          0 Dec  9 16:55 a
-rw-r--r--  1 root    root          0 Dec  9 16:55 b
-rw-r--r--  1 root    root          0 Dec  9 16:55 c
% time      seconds  usecs/call   calls      function
-----
 22.16     0.000787      787         1  getpuid
 11.71     0.000416         9        43  strlen
  6.14     0.000218        43         5  __overflow
...
  0.34     0.000012         12         1  __cxa_atexit
  0.31     0.000011         11         1  strncmp
-----
100.00     0.003552                173 total
root@rtl14:~/junk#
```

This trace could suggest that we should pay attention to the optimization of strlen and getpuid, but one needs to be careful with such analysis as it may be very dependant on the specific data set an application is to process. Looking at a second trace output would lead to possibly totally different results.

```
root@rtl14:/usr/src/dev_kit/src/ltrace-0.3.31# ./ltrace -c -ttt ls
BUGS          configure      elf.h          options.o
COPYING       configure.in   elf.o          output.c
...<snip>
config.status display_args.o options.c      wait_for_something.c
config.sub    elf.c          options.h     wait_for_something.o
% time      seconds  usecs/call   calls      function
-----
 35.85     0.011869    11869         1  qsort
 16.03     0.005307         10        496  __errno_location
 11.52     0.003814         9        384  __ctype_get_mb_cur_max
  9.30     0.003079         10        300  strcoll
...<snip>
  0.04     0.000014         14         1  __fpending
  0.04     0.000013         13         1  _setjmp
-----
100.00     0.033106                2009 total
root@rtl14:/usr/src/dev_kit/src/ltrace-0.3.31#
```

This trace would show that the greatest optimisation could be achieved with qsort improvement (...which will not be easy to achieve though). The real consequence for an embedded system would be to review if sorting the output is really necessary - in case of ls this might not be the case so we could alias ls to ls -f, instructing ls to produce output in the order they are stored on disk.

```

root@rtl14:/usr/src/dev_kit/src/ltrace-0.3.31# ./ltrace -c -ttt ls -f
.                config.sub          options.c        options.o
..               debian             options.h        elf.o
etc              defs.h                process_event.c output.o
...<snip>
install-sh      ltrace.h           ltrace.o
wait_for_something.c  proc.c           Makefile
% time          seconds  usecs/call      calls      function
-----
25.15    0.003836         9        396  __ctype_get_mb_cur_max
19.21    0.002930        11        256  __overflow
14.16    0.002160        10        202  __errno_location
...<snip>
 0.13    0.000020        20         1  closedir
 0.10    0.000015        15         1  __fpending
 0.09    0.000013        13         1  __cxa_atexit
-----
100.00    0.015255                1438 total
root@rtl14:/usr/src/dev_kit/src/ltrace-0.3.31#

```

This is a principal problem of bottle-neck detection, it generally is not possible to optimize for all cases of possible data sets. In embedded systems one often has a fairly well defined set of processing tasks to do allowing to optimize for a specific data set, so optimization beyond what is possible for the general case is possible and ltrace can guide you to the point where to start optimizing. Furthermore, as this simple examples with ls shows, not all tools designed for convenience on desk-top systems, are really that suited for embedded systems, so in many cases ltrace can lead to simple to implement improvements.

It is important to note though that one may not simply read the numbers in ltrace and believe that the ls -l really takes that long. The output of ltrace must be read as relative values, so we can see what library call took the most time, but not really how long it took. To put this relative values into context, run the commands with time prepended.

```

root@rtl14: # time /usr/src/dev_kit/src/strace/strace -c -ttt ls -l
...<snip>
real    0m0.010s
user    0m0.004s
sys     0m0.005s
root@rtl14: # time ls -l
t
...<snip>
real    0m0.006s
user    0m0.003s
sys     0m0.001s

```

This shows the distortion due to tracing with ltrace, naturally as this is a very non-deterministic system you would have to run this a few times to get any usable results and generally for very short execution times results always will be very inaccurate. What one *can* extract from ltrace precisely is in what library functions you want to possibly optimize, or maybe reimplement in your application code for a less general case.

5.2 strace

Applications are spending their time in libraries, system calls, and in application code, what we saw above was how to locate library bottle necks. Next we want to see details of the time spent in kernel mode:

```
root@rtl14:~# strace -c -tt ./hello
12:03:37.368036 execve("./hello", ["./hello"], [/* 35 vars */]) = 0
% time      seconds  usecs/call   calls   errors syscall
-----
 94.27     0.009598      1600         6         2 open
  3.72     0.000379         42         9         writev
...snip...
  0.01     0.000001          1         1      sigaltstack
  0.00     0.000000          0         1      getpid
-----
100.00     0.010181                58         1 total
root@rtl14:~#
```

The output of strace not only shows the number of system calls but also the time spent per system call, so bottle necks can be located and especially system calls that are returning with error codes, which can result in error processing overhead and, if these errors don't alter the intended behavior, indicate obviously useless code (in the above case the two opens failed on /etc/ld.so.cache and /etc/ld.so.preload). This can be combined with time to achieve an overview

```
root@rtl14:~ # time strace -tt -c ./hello
root@rtl14:~# time strace -tt -c ./hello
12:28:59.320956 execve("./hello", ["./hello"], [/* 31 vars */]) = 0
% time      seconds  usecs/call   calls   errors syscall
-----
 32.69     0.000017          6         3         1 open
 25.00     0.000013          3         5      old_mmap
...snip...
  5.77     0.000003          2         2      fstat64
  3.85     0.000002          1         2      close
-----
100.00     0.000052                16         1 total

real    0m0.037s
user    0m0.000s
sys     0m0.004s
root@rtl14:~#
```

Note though that for applications running only very short time, like this hello world, the actual numbers are not very precise - never the less, one can put the system call related times into relation to the user-space time and the real, wall-clock, time, which gives a sufficient overview to locate potential performance bottle necks.

If a daemon is launched in this mode you need to specify the -f flags to include all child processes.

```
root@rtl14:~# time strace -c -tt -f crond
12:35:15.117242 execve("/usr/sbin/crond", ["crond"], [/* 31 vars */]) = 0
Process 23746 attached
% time      seconds  usecs/call   calls   errors syscall
```

```

-----
20.76    0.000115        115        1        send
17.69    0.000098         7         15       4 open
...snip
  0.18    0.000001         1          1       getuid32
  0.00    0.000000         0          3       2 nanosleep
-----
100.00   0.000554                123       8 total

real    0m41.721s
user    0m0.003s
sys     0m0.003s
root@rtl14:~#

```

In all traces shown here one can find errors on open - using strace to locate such problems showed that the file /etc/ld.so.preload did not exist on the system and was causing this error. Generally for embedded systems with a high reliability requirements it is advisable to ensure that the expected system resources are really available or the application does not request them even if in the case of /etc/ld.so.preload this error is without consequences. Cleaning up the applicatoins will simplify and speed up error analysis in case something really goes wrong.

A clear disadvantage of both ltrace and strace is that one has to run an application in a special setup and there is a considerable performance overhaed when tracing. Both are thus limited to tracing specific applications, one can not trace an entire system with these tools, atleast not at a tolerable expense.

5.3 Profiling with gprof

In this section we will run through a quick profileing session with gprof, we assume that the libraries are not built with profiling on the embedded system (typically this is not posible for size and perform reasons), but the application is built with profiling. Thus this is limited to profiling the applicati function calls but will not reveal any weeknesses in the libraries used.

```
ALL: hello
```

```
CROSS:=i386-linux
CC:=$(CROSS)-gcc
C_FLAGS:= -Wall -g -O -march=i386
INCLUDE:=-I/opt/profile/i386/include/
LD_FLAGS:=-Wl,-rpath,/lib
```

```
hello: hello.c
    $(CC) $(C_FLAGS) $(INCLUDE) -c hello.c -o hello.o -pg
    $(CC) $(LD_FLAGS) -o hello hello.o -pg
clean:
    rm -f *.o hello
```

The application it selfe has nothing special in it - just a call to "something" in our case.

```
#include <stdio.h>

int something (void){
    printf("doing something\n");
    return 0;
}
```

```

}

int main(int argc, char **argv){
    something();
    printf("dev_kit check\n");
    return 0;
}

```

5.3.1 usage of gprof

compile with -pg (limited to -static if libs were not compiled with -pg)

```
gcc -static -v -g -pg app.c -o app
```

If one has application libraries one can compile these specific libs with profiling, but compiling all system libraries with profiling for a embedded target is somewhat problematic. For development this might be ok, but as we are interested in runtime debugging we assume here that shared libs are not compiled with profiling. Note that in some cases it may be tolerable to compile the application with static libs once one indentified the application that is causing problems.

As an example of how bad this is - the above hello.c compiled with -static...

```
-rwxr-xr-x  1 root    root      1560675 Jan 14 09:14 hello*
```

were as profiling of only the application code leads to a moderately increased filesize which is generally tolerable.

```
-rwxr-xr-x  1 root    root       36145 Jan 14 09:16 hello*
```

running hello, described in the previous section, will log the output in gmon.out (default file name, which *can't* be changed, and an exiting file will be overwritten if it is in the working directory of the process).

reading with gprof which is part of the binutils package will generate the flat profile and a call graph (default settings).

Flat profile:

Each sample counts as 0.000999001 seconds.

no time accumulated

%	cumulative	self		self	total	
time	seconds	seconds	calls	Ts/call	Ts/call	name
0.00	0.00	0.00	1	0.00	0.00	something

The call graph reveals cal depth and also allows better location of circumstances under which excessive execution time had been reached.

Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) no time propagated

index	% time	self	children	called	name
		0.00	0.00	1/1	main [14]
[1]	0.0	0.00	0.00	1	something [1]

5.3.2 drawback

- Mention the function call overhead introduced by -pg
- libs must be compiled with -pg or -static.
- libs and code are substantially larger
- better suited for development than for runtime testing

5.4 Code coverage with gcov

Using the above example hello.c file again, we compile it with:

```
gcc -fprofile-arcs -ftest-coverage hello.c -o hello
```

The fairly small data files compared to gprof and also the low impact on the executable size make this very useful for locating hot-spots in embedded applications.

```
-rwxr-xr-x  1 root    root      15405 Jan 14 09:12 hello*
-rw-r--r--  1 root    root        108 Jan 14 09:12 hello.bb
-rw-r--r--  1 root    root        156 Jan 14 09:12 hello.bbg
-rw-r--r--  1 root    root        174 Jan 14 07:44 hello.c
-rw-r--r--  1 root    root         48 Jan 14 09:12 hello.da
-rw-r--r--  1 root    root     11192 Jan 14 09:02 hello.o
```

5.4.1 generating reports

reports can be generated from data and source - no access to binary necessary.

```
root@rtl14:/usr/src/dev_kit/test# gcov -b hello.c
100.00% of 7 source lines executed in file hello.c
No branches in file hello.c
100.00% of 3 calls executed in file hello.c
Creating hello.c.gcov.
```

These reports allow very quick localization of dead code and of code that is executed very frequently, thus a good place to consider for optimization and profiling.

```
    1  int something (void){
    1          printf("doing something\n");
call 0 returns = 100%
    1          return 0;
    }

    1  int main(int argc, char **argv){
    1          something();
call 0 returns = 100%
    1          printf("dev_kit check\n");
call 0 returns = 100%
    1          return 0;
    }
```

The main issues that gcov can provide are

- code coverage
- execution path check
- execution hot spots localization

naturally gcov can only work properly if the applications are tested under circumstances that will allow gcov to reveal problems. So codecoverage requires well designed test-suits - untested branches will not be evaluated.

5.5 applogger

On embedded systems one often has the problem that one does not have the sources for a given application available or that one can't easily modify them, so methods that allow tracing the installed binaries are convenient. One such method is to use the preload capability of GNU glibc and plug in a little logger library that can dump information - in our case via the heavy weight syslog call.

The following example is based on a post by Jeffrey Streifling on the web - I unfortunately did not save the URL so I can't give full reference here. What this is doing here is quite cruel and can't be recommended for any real system, but to keep the example short...

This file is compiled into a library and then preloaded via the `export LD_PRELOAD=` shell command. From then on all applications launched from this shell will cause a syslog entry when they make a libc call to `open`.

```

/*
 * Written by Der Herr Hofrat, <der.herr@hofr.at>
 * Copyright (C) 2003 OpenTech EDV Research GmbH
 * License: GPL Version 2
 */
/*
 * Compile as shared library with:
 * gcc -fPIC -Wall -g -O2 -shared -o applogger.so.0 applogger.c -lc
 *
 * Log all calls to glibc's open function with all its parameters passed
 * open logfile on load of applogger lib and close on removal.
 *
 * The libraries init function is used to extract the logfile name in the
 * environment and open the file, the exit method closes the file.
 */

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>

#define _FCNTL_H
#include <bits/fcntl.h>

extern int __open(const char *pathname, int flags, mode_t mode);

/* initialize and cleanup logfile(s) on load/unload of lib */
void __applogger_init(void) __attribute__((constructor));
void __applogger_exit(void) __attribute__((destructor));

FILE *logfile;
char default_fname []="/tmp/applogger";
char *logfile_name;

```

```

void __applogger_init(void)
{
    if ((logfile_name = getenv("APPROG_FILE")) != 0) {
        printf("using %s\n",logfile_name);
    } else {
logfile_name=default_fname;
        printf("using %s (not APPROG_FILE set in environment)\n",logfile_name);
    }

if((logfile=fopen(logfile_name,"a+")) == NULL )
{
perror("Cannot open logfile\n");
exit(-1);
}
}

void __applogger_exit(void)
{
fclose(logfile);
}

/* any open called will cause a log entry */
int open(const char *pathname, int flags, mode_t mode) {

fprintf(logfile,"%d: opened %s (%d,%d)\n", getpid(), pathname, flags, mode);

/* now do the real libc open */
return __open(pathname, flags, mode);
}

```

Compile this with the above noted gcc command. And either preload the libs as shown below to make them take effect for all applications or just load them for one particular application, as shown later with njamd usage, by preceding the command with the LD_PRELOAD=/PATH/lib

```

hofrat@rtl114:~/C/libs > export LD_PRELOAD=/home/hofrat/C/libs/applogger.so.0
hofrat@rtl114:~/C/libs > export APPROG_FILE=/tmp/log
hofrat@rtl114:~/C/libs > vi junk
<just close the file again>
hofrat@kanga:~/C/libs > cat /tmp/log
17551: opened /usr/share/terminfo/x/xterm (0,-1073751172)
17551: opened junk (0,0)
17551: opened .junk.swp (0,0)
17551: opened .junk.swp (194,384)
17551: opened .junk.swpx (0,0)
17551: opened .junk.swpx (194,384)
17551: opened .junk.swp (578,384)
17551: opened junk (0,0)
hofrat@kanga:~/C/libs > unset LD_PRELOAD

```

This shows that open is a bad trace point unless you are a bit more selective, and it shows that this version of vi might be an overkill version from an embedded system (8 files opened/reopened for a single editor session). Also using this brute force approach together with a call to syslog(...) can easily result in a syslog party on your embedded system... The clear advantage of this method is though that no modification to the existing application is required one can trace stripped application binaries basically at any library call.

5.6 tracing a specific library function

As noted above gprof is somewhat limited as it requires shared libs to be compiled with -pg, making the quite large, or one needst to compile apps as static executables. Aside from the size problems on some embedded systems, both with the gprof output files and with the profiling overhaed of the libs, this method requires source access to the libs or atleast a vendor willing to deliver profiling versions of there libs.

A limited work around is to "profile" a library call by a preloaded lib that does nothing else but timestamp the call and return and log the elapsed time along with some parameters to a specified log file. This method can be applied quite selectively and will produce fairly small log files aswell as introducing a minimum overhead, both size wise and time wise, but it requires that one has already pinpointed the potential trouble makers first. So this is an addition to grprof/gcov, by no means a replacement.

```
/*
 * Written by Der Herr Hofrat, <der.herr@hofr.at>
 * Copyright (C) 2004 OpenTech EDV Research GmbH
 * License: GPL Version 2
 */
/*
 * Compile as shared library with:
 * gcc -fPIC -Wall -g -O2 -shared -o timestamp.so.0 timestamp.c -lc
 *
 * timestamp all calls and returns to open, log elapsed time.
 */

#include <stdio.h>
#include <unistd.h> /* exit, gettimeofday */
#include <sys/types.h>
#include <stdlib.h> /* getenv */
#include <sys/time.h> /* gettimeofday */
#include <time.h>

#define _FCNTL_H
#include <bits/fcntl.h>

extern int __open(const char *pathname, int flags, mode_t mode);

/* initialize and cleanup logfile(s) on load/unload of lib */
void __timestamp_init(void) __attribute__((constructor));
void __timestamp_exit(void) __attribute__((destructor));

FILE *logfile;
char default_fname[]="/tmp/timelog";
char *logfile_name;

void __timestamp_init(void)
{
    int verbos=0;
    if (getenv("TIMELOG_VERB") != 0) {
verbos=1;
    }
    if ((logfile_name = getenv("TIMELOG_FILE")) != 0) {
        if(verbos)
sprintf("using %s\n",logfile_name);
    } else {
```

```

logfile_name=default_fname;
    if(verbos)
printf("using %s (no TIMELOG_FILE set in env)\n",logfile_name);
    }

if((logfile=fopen(logfile_name,"a+")) == NULL )
{
perror("Cannot open logfile\n");
exit(-1);
}
}

void __timestamp_exit(void)
{
fclose(logfile);
}

int open(const char *pathname, int flags, mode_t mode) {

    int fd;
    struct timeval fun_call;
    struct timeval fun_ret;
    struct timeval fun_runtime;

    /* timestamp call and return */
    gettimeofday(&fun_call,NULL);
    fd = __open(pathname, flags, mode);
    gettimeofday(&fun_ret,NULL);

    /* fix timeval usec overruns */
    fun_runtime.tv_sec = fun_ret.tv_sec - fun_call.tv_sec;
    fun_runtime.tv_usec = fun_ret.tv_usec - fun_call.tv_usec;
    if (fun_runtime.tv_usec < 0) {
--fun_runtime.tv_sec;
fun_runtime.tv_usec += 1000000;
}

    /* log it */
    fprintf(logfile,"%d: open %s (0x%x,%o,%d) in %ld us\n",
    getpid(),
    pathname,
    flags,
    mode,
    fd,
    fun_runtime.tv_usec + ( fun_runtime.tv_sec * 1000 ));

    /* don't forget to return the original functions value ;) */
    return fd;
}

```

The read_write application called below is simply a open of a input file with O_RDONLY and open of the output file with O_WRONLY—O_TRUNC—O_CREATE, and then copy the data from input to output.

```
hofrat@kanga:~ > unset LD_PRELOAD
```

```

hofrat@kanga:~ > export TIMELOG_VERB=1
hofrat@kanga:~ > export LD_PRELOAD=/home/hofrat/C/libs/timestamp.so.0
hofrat@kanga:~ > ./read_write
  <just quite the editor again>
using /tmp/timelog (no TIMELOG_FILE set in env)
hofrat@kanga:~ > unset TIMELOG_VERB
hofrat@kanga:~ > ./read_write
hofrat@kanga:~ > tail /tmp/timelog
19588: open /tmp/input (0x0,1001101560,4) in 10 us
19588: open /tmp/output (0x241,644,5) in 107 us
19602: open /tmp/input (0x0,1001101560,4) in 8 us
19602: open /tmp/output (0x241,644,5) in 4160 us

```

Looking at this output - one might be surprised that the last call to open took 4 milli seconds (and this is on a PIII/800) - looking at the flags one can see open ran with O_TRUNC and the output file was quite large before so that explains the large time. This is exactly the type of problem that is hard to locate during testing because one typically does not have the accumulated files and long time environment like embedded applications that are running for months or many years and then under some specific circumstances open the file with O_TRUNC (for instance if there is too little space left on the system).

The nice thing about this solution though is that this type of preload "profiling" can be added to a running system without the need to modify applications or libraries that are on the system.

5.7 ltp

When writing up this summary I was unsure where to put this section, its not strictly related to runtime debugging, but it is more of a prerequisite to start with any serious runtime debugging and system profiling. The Linux Test Plan [?] provides a "regression test" for the kernel proper. The core test components are:

- filesystem stress tests
- disk I/O tests
- memory management stress tests
- ipc stress
- scheduler tests
- commands functional verification tests
- system call functional verification tests

Especially the stress tests are of interest not only for "normal" embedded linux but also for the different real-time variants. Some of the features of interest here related to hot-spot detection are:

- Kernel Code Coverage:
The Linux kernel code coverage is a kernel patch that allows to identify the areas in the kernel that are influenced most by the LTP test-suite.
- Gcov Extension(lcov):
This is basically a gcov variant that will allow correlating data collected with the test-suits run, thus simplifying the interpretation of collected data.
- Gcov-kernel Extension:
To allow judgment of the test-suite effects in the kernel proper, this code coverage extension to the linux kernel (reporting via /proc interface), allows identifying kernel areas that were most impacted by a given LTP test.

It's kind of out of the scope of this article to go into details - so give <http://ltp.sourceforge.net/> a close look !

5.8 oprofile

For runtime profiling at a system scope the above tools will not due, locating bottle necks that might not be application specific but are due to the overall system configuration requires a tool that monitors the entirety of a system. For Linux based embedded systems oprofile [?] allows exactly that.

For 2.4.X kernels, you must have the kernel source available for the kernel you want to run oprofile under (passed at build time with `./configure --with-linux=/path/to/kernel/source, make, make install`). In 2.6.X oprofile is already available in the kernel, it just needs to be enabled (builtin or as module), and then one can build oprofile (`./configure --with-kernel-support`). Support is somewhat X86 centric (again) - but other architectures are emerging in 2.6.X. The overhead of oprofile has been reported as low as 1-8%, mainly dependant on the interrupt load of the system. This is why it is mentioned here as this low overhead makes it well suited for runtime debugging on deployed systems.

Oprofile allows some kernel level profiling with 2.4.X kernels, as of Linux-2.6.X it is integrated in the kernel and one should expect this to become one of the main runtime monitoring tools in the future Linux kernel development.

One quote from the oprofile web-page that maby summs it all up very well:

Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is. - Rob Pike

The oprofile package comes with the necessary utilities to configure and start/stop profiling at runtime:

- `opcontrol` - used for starting and stopping
- `opreport` - generate symbol-based profile summarie
- `opannotate` - to output annotated, mixed source or assembly output
- `opgprof` - generate a gprof-format profile

When using oprofile it is advisable to read the entire man-page, to avoid large loads due to profiling one must carefully select set-up parameters !

6 Kernel-space runtime debugging

6.1 kernel debugging

In the late 2.4.X and in the 2.6.X series of kernels a number of builtin kernel level debug featurrs have been added, these should be used for kernel based application development before trying to build new facilities !

Many of the bugs that can be detected by kernel debugging are typical bugs that can stay undetected for a long time. Thus one should run kernel space apps with debugging turned on before releasing them.

```
Kernel hacking --->
[*] Kernel debugging
[ ] Check for stack overflows (NEW)
[ ] Debug high memory support (NEW)
[ ] Debug memory allocations (NEW)
[ ] Memory mapped I/O debugging (NEW)
[ ] Magic SysRq key (NEW)
[ ] Spinlock debugging (NEW)
[ ] Compile the kernel with frame pointers (NEW)
```

- Check for stack overflows:
Sets: `CONFIG_DEBUG_STACKOVERFLOW`
This will make `do_IRQ()` check for enough stack space being left, which typically is not the case if interrupts start stacking up endlessly (for instance if a level triggered irq is never reset). The

error will be reported if the stack has less than 1kB of free space left on it, if this is the case a stack dump is written via `show_stack`

- **Debug high memory support:**

Sets: `CONFIG_DEBUG_HIGHMEM`

Add error checking for high memory access - this will slow down high memory considerably so it is only suitable during development, for production systems this needs to be turned off. This will trigger an `out_of_line_bug()` (2.4.x only) if `kmap_atomic` (the IRQ safe variants) find that there is no page table entry for the requested area. `kunmap_atomic` will force the page table entry of the this page to be cleared so that unclean reuse, that is without remapping it, will oops.

- **Debug memory allocations:**

Sets: `CONFIG_DEBUG_SLAB`

The two things does is

- red zoning which allows detection of

- * write before start
- * write past end
- * double free

note that these things need not immediately lead to a system crash, and thus they can stay undetected for a long time.

- cache poisoning, which means that the memory area is filled up with a known pattern and terminated with a defined sequence, thus the history of an object can be checked.

For more detail see the section on debugging memory allocation below.

- **Memory mapped I/O debugging:**

Sets: `CONFIG_DEBUG_IOVIRT`

I/O address ranges can't be checked for validity in general, but in some cases the kernel can detect obviously invalid addresses, like those that were `ioremap`'ed. This debug check will be dropped as it is primarily there to catch drivers ported from Linux-2.0.X/2.2.X which used unmapped ISA addresses.

- **Magic SysRq key:**

Sets: `CONFIG_DEBUG_SYSRQ`

Allows a system that is stuck in kernel mode to be inspected by directly intercepting the keyboard input and triggering debug functions on pressing of `<CNTRL>-<PrintScrn>-<CHAR>` - with the following being predefined by the kernel:

- 0-9 raise/lower the log level of the system
- b reboot the system
- e send a `SIG_TERM` to all processes via `send_sig_all(SIGTERM)`
- i send a `SIG_KILL` to all processes via `send_sig_all(SIGKILL)`
- k secure attention key - kills all programs currently using this tty and allow kernel messages to be displayed.
- m display memory information
- p register dump
- s sync
- t status
- u remount the system read-only

For more details and examples see the section `sysrq` below. i

- **Spinlock debugging:**
Sets: `CONFIG_DEBUG_SPINLOCK`
Catches missing spinlock initialization by adding a spinlock magic number `SPINLOCK_MAGIC 0xdead4ead`, which ensures that one can't simply use some random memory location as spinlock when locking/unlocking. It also ensures that attempts to unlock spin locks that are not locked will be reported via `BUG()`.
- **Compile the kernel with frame pointers:**
Sets: `CONFIG_FRAME_POINTER`
The kernel will be compiled without `-fomit-frame-pointer`, which set by default, prevents keeping the frame pointer in a register for functions that don't need one and avoids saving and restoring it. This makes debugging of function call chain possible in `gdb`. Generally this should be disabled for production releases as it slows down the system noticeably.

Additionally to the above debugging capabilities the 2.6.X series of kernels (well actually 2.6.0 for now) offer:

- **Page alloc debugging:**
Sets: `CONFIG_FRAME_PAGEALLOC`
This performs extra checks when freeing pages with `kmem_cache_free` or `kfree` via `__cache_free`:

```
__cache_free
-> cache_free_debugcheck:
    -> kfree_debugcheck: verify that pointers are valid
    -> store_stackinfo: store
```

note that the comment in `mm/slab.c` for `kfree_debugcheck` states:

```
* - detect bad pointers.
* - POISON/RED_ZONE checking
* - destructor calls, for caches with POISON+dtor
```

but the code seems to only check pointers, atleast I was not able to see how `POISONING/RED_ZONE` checks are done here... This should not be active on production systems as it slows down the kernel in the very sensitiv memory subsystem.

- **Compile the kernel with debug info:**
Sets: `CONFIG_DEBUG_INFO`
If one needs to debug the kernel using `gdb` one should turn this option on, it will result in a very large kernel but will deliver usefull information in the debugger. If this option is enabled one also should turn on the above noted `CONFIG_FRAME_POINTER`.
- **:**
Sets: `CONFIG_DEBUG_SPINLOCK_SLEEP`
As sleeping with held spinlocks is generally not a good idea, this option causes many kernel functions to become noisy if called within held spinlocks.

6.1.1 Debug memory allocations

TODO

- mark each slab in the cache with `RED_MAGIC1` on call of `kmem_cache_init_objs`, a `BUG()` will be issued if the objects red zone is not found after poisoning the cache object.
- slab memory object initialization check, this printk's any functions that requests status of none initialized slab cache objects.

- issue a BUG() in `__kmem_cache_shrink_locked` (called via `kmem_cache_shrink`) if a cache is shrunk while the slab is marked inuse.
- issue a BUG() if adding of a red zoned object can't find the previous objects red zone in `kmem_cache_alloc_one_tail`
- check slab on freeing objects, an overrun of a cache object must not necessarily lead to a crash of the system, and can stay undetected for some time. These extra checks verify that the object being freed is in the bounds of the slab, that the freed object is at the end of the slab and that it is not on the free list yet.
- memory poisoning, which fills the released cache memory area up with 0x5a terminated by POISON_END which is 0xa5, this poison and poison end is then checked for in `kmem_check_poison_obj`. Checks for cache memory poisoning are done in the following functions with the error responses indicated by ->:
 - `kmem_slab_destroy` -i BUG()
 - `kmem_cache_create` -i printk warning if poisoning is requested with objects that use a ctor method
 - `is_chained_kmem_cache` -i return 1 if cache is in chain, 0 otherwise. This allows sanity checking of caches in code by a call to `is_chained_kmem_cache`
 - `kmem_cache_alloc_one_tail` -i BUG()

6.1.2 using sysrq

To use Sys-Req one need to configure it at kernel compile time by enabling 'Magic SysRq key (CONFIG_MAGIC_SYSRQ)' when configuring the kernel. At runtime it can be switched off and on, on a kernel with SysRq compiled in, by the following command:

```
rt114:~ # echo "0" > /proc/sys/kernel/sysrq
rt114:~ # echo "1" > /proc/sys/kernel/sysrq
```

respectively.

Note that older versions disabled sysrq by default, requiring explicit enabling at run-time, recent versions have changed this and make sysrq default on. Sysrq is also adjustable via sysctl calls (CTL_KERN, KERN_SYSRQ=38), thus it is easy to integrate this feature in some centralized management level like SNMP.

The sysrq operations include the handler to execute, a help field that is displayed if an invalid sysrq is keyed in, which then shows a list of available system requests. The convention being to capitalize the letter used to trigger the respective system request. As there is no sysrq registered for "y" by default on 2.6.0 systems, you should get the following output

```
<CNTRL>-<PrintScr>-<y>
SysRq: HELP: loglevel0-8 reBoot tErm kIll saK showMem showPc
unRaw Sync showTasks Unmount
```

This allows you to monitor the system at a very low level, and is an essential tool for post mortem analysis during development, especially of kernel components like drivers.

6.1.3 adding your sysrq handler

System requests are handled in `drivers/char/sysrq.c`, they are not dependant on any user-space application layer but directly "intercept" the keyboard driver, thus a sysrq key-sequence never reaches any normal terminal or application but is caught by the kernel proper. This ensures that they will be honored even if the kernel has been put in a state where user-space is no longer serviced or parts of the kernel are messed up, naturally if the sysrq code itself was damaged, i.e. by writing to the code area via some stray pointers, then sysrq will not help you in any way. Thus this is an improvement over user-space application control but it can't guarantee that a system will be responsive.

That said - on to setting up a sysrq handler for a kernel space application. Define your handler and the sysrq operations by adding them to `drivers/char/sysrq.c`:

```

unsigned long my_app_status = 0;
static void sysrq_app_status(int key, struct pt_regs *pt_regs,
                             struct tty_struct *tty)
{
    console_loglevel = 8;
    Gprintk("application status = %ld\n",my_app_status);
}

```

A more usefull thing to do is to declare a function pointer and conditionally call the application status function if non-NULL. But this is up to the programmer, basically there is no real limit here, except that this is statically compiled into the kernel and thus generally the application module is not yet loaded at boot time.

As noted above the conventinn is to mark the respective key in the `.help_msg` string by capitalization, if you can't fit it in a meaningfull word and you get stuck with a free letter like "z", then just add it in braces, trying to forfully stick to this convention with strings like `statuZ` will cause confusion....

The `.action_msg` is printed previous to invoking the handler, so in our case `<CNTRL>-<PrintScr>-<z>` would output the folowing on the consoleon the console:

```

<CNTRL>-<PrintScr>-<z>
SysRq: Schow applicatoin status
application status = 0

```

The core structure for the sysrq system is the `sysrq_key_op`, consiting of three elemens, the actual handler, a short textual help message and a message to explain what the handler is doing.

```

struct sysrq_key_op {
    void (*handler)(int, struct pt_regs *,
                   struct kbd_struct *, struct tty_struct *);
    char *help_msg;
    char *action_msg;
};

```

In oure example we would use the folowing,

```

static struct sysrq_key_op sysrq_app_status_op = {
    .handler      = sysrq_app_status,
    .help_msg     = "status(z)",
    .action_msg   = "Schow applicatoin status",
};

```

which is mapping this handler in the sysrq table to a specific key - z in our case. The above `sysrq_app_status` is not doing anything usefull, it is just being executed in kernel space and printing to the console, but as it is executed in kernel context one can display any data item from kernel space. In this example it will display a unsigned long `my_app_status` which is made globally available in kernel context by adding the line

```
EXPORT_SYMBOL(my_app_status);
```

at the end of `sysrq.c` and thus can be set by the kernel space application.

```

#define SYSRQ_KEY_TABLE_LENGTH 36
static struct sysrq_key_op *sysrq_key_table[SYSRQ_KEY_TABLE_LENGTH] = {
/* 0 */ &sysrq_loglevel_op,
/* 1 */ &sysrq_loglevel_op,
/* 2 */ &sysrq_loglevel_op,

```

```

...
/* w */ NULL,
/* x */ NULL,
/* y */ NULL,
/* z */ &sysrq_app_status_op,
};

```

Now recompile the kernel, and after booting hold down <ALT>-<Print>-<Z> (on X86 sysrq maps to <ALT>-<PrintScr> for other architectures see Documentation/sysrq.txt)

SysRq key strokes are also logged via the kernel's log facility so `dmesg` will also display the log messages:

```

SysRq : Show application status
application status = 0

```

Note that the loglevel must be set to 8 if a message should go to the console, so

```

console_loglevel = 8;
printk("application status = %ld\n",my_app_status);

```

will print to the console and also to the kernel ring buffer which then is typically logged via `syslogd` to the appropriate log files in `/var/log`, the invocation of the SysRq handler is done via the `__handle_sysrq_nolock` function in `drivers/char/sysrq.c` which ensures that the log level changes in any of the handler functions is temporary, so no need to save and restore log levels in handlers.

6.1.4 dynamic sysrq

The System request interface in Linux is not built for dynamic allocation of system request handlers. But never the less one can allocate them dynamically, it must be noted though that this is very linux kernel version specific and the code shown here will work for linux-2.6.0 and possibly for no other version. The concept though is quite generic.

Basically all that needs to be done is grab the sysrq table and run through the list of sysrqs until one finds one that is a NULL pointer. then map your handler with the appropriate format to that key - that's it.

The kernel exports some sysrq related management functions that can be used to dynamically register sysrq handlers. Obviously this does not require patching the kernel, and thus is the preferred way for testing.

The registration is done by adding a key operations in the key op lookup table, which is again in `drivers/char/sysrq.c`. The key table is limited to `SYSRQ_KEY_TABLE_LENGTH`, which corresponds to 0-9,a-z. A set of predefined sysrq keys are compiled in as shown above, so the free sysrq table entries can be used for our application specific sysrq.

The sysrq system export six functions as a minimum registration interface

```

__sysrq_lock_table
__sysrq_unlock_table
__sysrq_get_key_op
__sysrq_put_key_op
__sysrq_swap_key_ops
__sysrq_swap_key_ops_nolock

```

When manipulating the sysrq always table one must always lock the table before reading or writing, naturally followed by an unlock operation. To deregister a handler it simply must be set to a NULL pointer, the sysrq functions `handle_sysrq` which does the locking and then calls `__handle_sysrq_nolock`, checks for NULL pointers, but if an invalid pointer is left in place strange things will happen.

To set up a sysrq handler dynamically one should use the function `__sysrq_swap_key_op`, which does the necessary locking before calling its `_nolock` counterpart to actually add the new handler.

6.2 msgdump

The kernel is using `printk` to produce debug and status output, which normally goes to `syslogd` via `klogd`, and then into log files. But in case of a system crash this user-space daemons will hardly work properly and valuable debug information may be lost, making post-mortem analysis quite painful if one is forced to copy down messages of the console.

The `kmsgdump` kernel patch tries to fix this problem by using a `sysrq` handler to dump this log information to a floppy or the parallel port. Unfortunately this is very X86 specific and not available for other platforms

7 Kernel debug "API"

As a general note on kernel debugging API, one can notice that the "API" is not that consistent when it comes to low level stuff. So one needs to relax the expressions when searching in other than x86 architectures for these functions. As an example, on ppc architectures the register dump is called `show_regs` and not `show_registers`, but the path of events is quite similar on all archs.

The worst case, the oops, is generated via an invalid instruction that then goes via the appropriate trap handler and finally will call `die` which will produce the, hopefully not that well known, oops screen.

7.1 printk()

`printk` is a `printf` variation for the kernel, with the addition of a prepended log level and the limit of not printing any of the floating point formats (but floats in kernel space are generally a no-no any way). `printf` is a very robust function, it can be called in more or less any context, including interrupt context and while holding locks. There are also variants for the hard-realtime extensions to Linux.

- `rtl_printf` is the RTLinux variant that can be called safely from rt-context.
- `printk` in ADEOS is patched to be safe from any domain context
- `rt_printk` is the RTAI variant that also is rt-context safe.

7.1.1 log level and log buffer

`printk` has a log level prepended to the format - these log levels are used by the kernel to decide where the message needs to go based on the current log level setting in the kernel, and are used by `klogd`/`syslogd` when sorting out log messages in `/var/log/*`. Especially when putting `printk` messages into a error branch of code that could cause serious problems one should lower the log level so that the message is dumped to the console, as the `syslogd` path might not work in case of a kernel bug.

To ensure that a message reaches the console do:

```
...
console_loglevel = DEFAULT_CONSOLE_LOGLEVEL;
printk("This always will go to the console\n");
...
```

Note that you can also set this via the magic system request (for details on `sysrq` see above) by using `<CNTRL>-<PrintScrn>-<0>` - `<CNTRL>-<PrintScrn>-<9>` (not sure what 9 would do, the kernel seems to only define 0-7, but it can be set via `sysrq...`).

When using `printk` in your error path don't forget that if the error is non-transient that you can cause log floods to the system. To protect against such cases one can limit the logging message either by intervals or by absolute counts. For details see the chapter on debugging in Rober Loves "Linux Kernel Development" [?].

7.1.2 before printk is available

Before printk is actually available, that is before the kernel has a console device, debug messages are a problem. On some architectures a `early_printk()` is provided which is available after `setup_early_printk` called very early in the boot process on x86_64 ppc64 and alpha (don't think its available on other archs - corrections appreciated).

For the first few lines of output that one sees on the screen "Uncompressing Linux..." the kernel can't access any real kernel functions yet, for debugging in this setup part of `vmlinux`, one must use the architecture specific functions.

- `puts`:
put a string on the boot console
- `puthex`:
put a variable as hexadecimal number on the boot console

`puts` is supported on some m68k, ppc, mips and i386. `puthex` is a bit more selective and appears in architecture specific naming (i.e. `udbg_puthex` on ppc64).

7.1.3 preventing log floods

7.2 BUG() and variants:

BUG is intentionally for in-lined functions, as it prints the file and line that caused the problem, so if used in functions that are called and not in-lined the information available via BUG is more or less useless. The mechanism used by BUG is to trigger an invalid instruction by a call to one of the undefined instructions `ud0`, `ud1`, `ud2` which are explicitly for this purpose. As the use of `ud0` (or was it `ud1` ?) is AMD specific, `ud2` is used for x86 platforms.

The invalid instruction `ud*` will cause a trap execution via `do_trap` which then subsequently will cause `die` (from `arch/i386/kernel/traps.c` which then causes ensures that the console log level is set to dump directly to the console (via `console_verbose()`) and then dumps the debug information to the screen by calling `show_registers`. Further more, within `show_registers` the stack is dumped to produce the oops message.

```
from include/asm/bug.h:
/*
 * Tell the user there is some problem.
 * The offending file and line are encoded after the "officially
 * undefined" opcode for parsing in the trap handler.
 */

#define BUG() \
__asm__ __volatile__( "ud2\n" \
    "\t.word %c0\n" \
    "\t.long %c1\n" \
    : : "i" (__LINE__), "i" (__FILE__))
```

`BUG_ON` is just a wrapper to `BUG` and generally will be better readable in the code.

```
#define BUG_ON(condition) do { if (unlikely((condition)!=0)) BUG(); } while(0)
```

The memory subsystem defines its own `BUG` variant to catch `NULL` pages. The `dowhile(0)` construction is to allow `gcc` to optimize it away in case that `BUG` is undefed.

```
#define PAGE_BUG(page) do { \
    BUG(); \
} while (0)
```

A more verbatim variant is the WARN_ON, the above BUG reports will only tell you a bug occurred but don't tell much about the conditions and the state in which this happened - so unless this is sufficiently clear (i.e for a driver specific function that will be called only in known code paths) BUG is fine - for code that could be called from any where in the kernel WARN_ON is mor useful as it allows tracking back the cause for the problem.

```
#define WARN_ON(condition) do { \
    if (unlikely((condition)!=0)) { \
        printk("Badness in %s at %s:%d\n", __FUNCTION__, \
            __FILE__, __LINE__); \
        dump_stack(); \
    } \
} while (0)
```

when using BUG and friends one needs to ensure that a error does not lead to a printk flood resulting in a syslog party and consequent overload of the system, especially on low resource systems this can easily happen, see not in the above printk section.

7.2.1 preempt_count()

For debugging relaxed lock protected objects (basically all per cpu variables), checking the kernels preemption status is necessary. Also when holding locks, preemption should be off. For these (and other) cases the kernel provides some functions to inspect the content of the preempt

In the 2.6 series of kernels these functions are based on the global preempt_count variable which is comprised of three counters. One for preemption one for soft interrupts (bottom halves) and one for hard interrupts.

include/asm/hardirq.h:

bits 0-7 are the preemption count (max preemption depth: 256)

bits 8-15 are the softirq count (max # of softirqs: 256)

bits 16-23 are the hardirq count (max # of hardirqs: 256)

bit 26 is the PREEMPT_ACTIVE flag.

- preempt_count()
preempt_count, if 0 the kernel can be preempted, if greater 0 preemption has been disabled and if it drops below 0 -> BUG.
- in_interrupt()
returns non 0 if in interrupt context (hardware interrupt or bottom halves).
- in_irq()
returns non 0 if in interrupt context due to a hardware interrupt
- hardirq_trylock() which is the negation of in_interrupt()
- in_softirq()
returns non 0 if in interrupt context due to a active bottom half

Note that the 2.4 series of kernels provides in_interrupt and in_irq() as well, but these are based on separate counters and not on the preempt_count (which is not available in the default 2.4 kernel series - patches are available though).

7.2.2 GCCs __builtin_return_address(x)

As noted above BUG only is use full for in lined functions as __FILE__ etc. filed in by the preprocessor - so for functions that are called one needs a means to identify the caller. Unless the function is only called by a very small set of known locations, which allows manual inspection, GCCs __builtin_return_address is the way to go.

```
unsigned long caller = (unsigned long) __builtin_return_address(0);
```

`__builtin_return_address` returns the return address of the current function, which is the caller. the argument is the number of frames to go up the call stack, but this does not always return reasonable results (at least not on x86). A frame number of 0, returning the direct caller, does work reliably though (provided the stack was not corrupted). Unfortunately the return value does not allow too much interpretation - info gcc says:

```
On some machines it may be impossible to determine the return
address of any function other than the current one; in such cases,
or when the top of the stack has been reached, this function will
return '0' or a random value. In addition,
'__builtin_frame_address' may be used to determine if the top of
the stack has been reached.
```

The problem is related to the availability of a frame pointer register on some architectures, for those the stack does not contain the necessary structure to use `__builtin_return_address` or `__builtin_frame_address`, as the frame pointer is omitted for optimization. This can be changed at compile time in the top level Makefile of the kernel though - recommended only for debugging and not for production systems, so its not recommended for runtime debugging.

example usage:

```
kernel/time.c:
signed long schedule_timeout(signed long timeout)
{
    ...
    switch (timeout)
    {
    case MAX_SCHEDULE_TIMEOUT:
        schedule();
        goto out;
    default:
        /*
         * Another bit of PARANOID. Note that the retval
         * will be 0 since no piece of kernel is supposed
         * to do a check for a negative retval of
         * schedule_timeout() (since it * should never
         * happens anyway). You just have the printk()
         * that will tell you if something is gone wrong
         * and where. */
        if (timeout < 0)
        {
            printk(KERN_ERR "schedule_timeout: wrong timeout "
                "value %lx from %p\n", timeout,
                __builtin_return_address(0));
            current->state = TASK_RUNNING;
            goto out;
        }
    }
}
```

For functions that can be called from any where in the kernel this is a very convenient way to find the actual source of the problem. The address can then be resolved via the kernel symbol table (`/proc/kallsyms` for 2.6 and `/proc/ksyms` for 2.4 as well as in the `System.map` file generated during kernel compilation).

7.3 show_registers()/show_regs()

This simply dumps the content of the registers and also for the stack if not in a system call. In the 2.6 series of kernels the symbol names are available in the kernel so one can produce directly usable messages, prior to 2.6 one has to "decode" the output with ksymoops.

```
print_symbol("EIP is at %s\n", regs->eip);
```

The rest is not too exciting - just a dump of the architecture specific registers.

```
printk("eax: %08lx  ebx: %08lx  ecx: %08lx  edx: %08lx\n",
      regs->eax, regs->ebx, regs->ecx, regs->edx);
printk("esi: %08lx  edi: %08lx  ebp: %08lx  esp: %08lx\n",
      regs->esi, regs->edi, regs->ebp, esp);
printk("ds: %04x  es: %04x  ss: %04x\n",
      regs->xds & 0xffff, regs->xes & 0xffff, ss);
printk("Process %s (pid: %d, threadinfo=%p task=%p)",
      current->comm, current->pid, current_thread_info(), current);
```

7.4 dump_stack()/show_trace/show_stack

dump_stack is just a wrapper to show_trace in 2.4, in 2.6 kernels dump_stack will call show_trace passing it the current tasks thread_info structure. show_trace will walk up the stack until it reaches the thread_info structure that is stored in the stack. The 2.4 kernel simply dumps the hexadecimal values it finds on the stack, in 2.6 print_symbol() is called on all values to produce directly usable output.

The show_stack also invokes show_trace() but it dumps the raw stack content before doing so.

7.5 print_symbol()

To print a human readable message of where things are going wrong one can use the instruction pointer available via regs->eip and pass it to print_symbol. This will print the symbol name that belongs to this address, this is the kernel function name or garbage in case the stack is corrupted.

8 Simple tools

In this section we will briefly summarize the available tools as an overview with some notes on the underlying mechanisms.

8.1 strace

In the simplest case strace runs the specified command until it exits. It intercepts and records the system calls which are called by a process and the signals which are received by a process. The name of each system call, its arguments and its return value are printed on standard error or to a specified file. Within the linux kernel a process is marked as traced by the flag:

```
#define PF_PTRACED 0x00000010 /* 2.2.X kernels */
#define PT_PTRACED 0x00000001 /* 2.4.x/2.6.x kernels */
```

whereby the ptrace flag gives a parent process access to the actual code pages of a process, allowing to modify it (i.e. insert breakpoints) and is used by debuggers. strace also makes heavy use of the ptrace function to control the executable

running as a child process of the `strace` command. The core functions for attaching/detaching and accessing the child process memory space is coded in `/kernel/ptrace.c`.

`strace` is a diagnostic, instructional, and debugging tool to not only locate failures but also to locate application bottlenecks and performance problems. Especially when it comes to diagnosing problems for applications where source code is not available tools like `strace` are a means to look deep into the internal structure of an application.

A perl script for post-processing of `strace` data, written by Richard Braakman, is also in the current `strace` release which allows to display a graph of invoked subprocesses including execution times (via `-t/-tt/-ttt` flags to `strace`)

For details and options to connect to running processes see the man (1) `strace`, the latest versions of `strace` are available via `cvs`. `Strace` is supported on a number of platforms and is easily cross-compiled (builds with `autoconf`).

8.2 ltrace

`ltrace` is a debugging program which runs a specified command until it exits, intercepting and records all dynamic library calls. It also records any signals received by the traced process. `ltrace` can intercept and trace system calls executed by the program, but is less elaborate in its feature set for this purpose than `strace`. Again, as in `strace` the program to be traced need not be recompiled, it is thus usable for binary only code debugging as well.

The operation of `ltrace` is quite similar to that of `strace`, especially the output with the `-t/tt/ttt` options allow quick location of hot-spots in the library calls. Unfortunately `ltrace` does impact quite heavily on the applications performance so its use in actually deployed systems needs to be considered carefully. Also a clear disadvantage of `ltrace` as of version 0.31 is that it does not cleanly cross-compile (`./configure --target=...` will not work), but versions for different architectures are available as debian packages.

TODO: the rest of them :)

9 Checkpoint Restart

A critical issue for embedded systems is post-mortem analysis and the ability to continue at the place where an application left of, i.e. at power failure, when the system comes up again. Checkpoint restart technologies provide this capability, and it is better under all circumstances to use available implementations than to reinvent the wheel, which is commonly done when it comes to checkpoint/restart capabilities. The BLCR, Berkeley Lab Checkpoint Restart library is used in clusters, and works not only for single nodes but also with distributed computations based on message passing (i.e. LAM, Local Area Multicomputer).

9.1 BLCR installation and configuration

BLCR is a checkpoint/restart system that can be used standalone or in conjunction with LAM/MPI in order to checkpoint parallel jobs. Check-pointing is very useful to troubleshoot applications because it makes errors reproducible and it save a lot of time because you can start debugging from the last checkpoint and not from the beginning of the application. So let's download the BPROC tarball and unpack it to the `/usr/src` directory:

```
rtl29: # cd /usr/src
rtl29: # wget http://ftg.lbl.gov/twiki/pub/Whiteboard/CheckpointDownloads/blcr-0.3.1.tar.gz
rtl29: # tar -xzf blcr-0.3.1.tar.gz
rtl29: # cd blcr-0.3.1
rtl29: # ./configure --with-linux=/usr/src/linux
rtl29: # make
rtl29: # make install (as root)
```

After downloading the next step is to configure and compile the BLCR modules and utilities. (Compilation does not work with a Slackware 9.1 default installation). Make sure that the `/usr/src/linux` link is set to the correct linux kernel

and that Loadable module support -> Set version information on all module symbols is not activated, also the kernel tree must contain the compiled kernel (actually the System.map file). This is due to the fact that the BLCR modules are compiled without modversion and the busybox insmod executable is not able to cut off versions during the insert of the module.

Note that you may get a number of warnings regarding redefinitions - but we found no fatal problem involved with these warnings - it seems that the include files distributed with bcr are not yet quite clean (afterall its version 0.3).

9.2 A simple example

A somewhat artificial example but perfectly fine to show the concepts. This simple application will count an integer and checkpoint the application every ten iterations. If you kill the job with a fatal signal or power off the system, the application can restart at the last checkpoint. Also if it does not die but behaves strange, you can restart it at an earlier point in its history and rerun it. This is especially useful to distinguish transient failures from real bugs, which is otherwise very hard to do!

```
/* counting.c
 *
 * GPL V2,
 * (C) 2004, Florian Bruckner <florian.bruckner@aon.at>
 * (C) 2004, Der Herr Hofrat <der.herr@hofr.at>
 * (C) 2004, Michael Poeltl <michael.poeltl@univie.ac.at>
 * OpenTech EDV Research GmbH <http://www.opentech.at>
 *
 * BLCR example program
 * increments an integer in a loop and creates a checkpoint every
 * 10 iterations
 *
 * compile:
 * gcc -lcr -o counting counting.c
 */

#include <stdio.h>
#include <unistd.h>
#include <libcr.h>

#define ITERATIONS 10

int main(void)
{
    char filename[100];
    int i;
    cr_client_id_t id;

    printf("Counting demo: PID %d\n", (int)getpid());
    printf("Checkpoints every %d iterations\n", (int)(ITERATIONS));

    sprintf(filename, "context.%d", (int)getpid());
    printf("checkpoint-file: %s\n", filename);

    /* initialize checkpoint system */
    id = cr_init();
```

```

    for (i=1; i<100; ++i) {
if (!(i%(ITERATIONS))) {
cr_request_file(filename);
printf("checkpoint!\n");
}
printf("Count = %d\n", i);
fflush(stdout);
sleep(1);
    }

    return 0;
}

```

9.3 Usage

9.3.1 Preparing Linux for BLCR

After the make install step you must of course run ldconfig so the new library is found. The modules are installed but not loadable yet as you must update the module dependencies - this is done by calling depmode with the -a flag to rebuild the dependencies. But first we test it before copying the modules to the system wide module directory.

```

rtl29: # ldconfig
rtl29: # cd /tmp/blcr-0.3.0/vmadump/
rtl29: # insmod vmadump.o
rtl29: # cd ../cr_module/
rtl29: # insmod blcr.o

```

If this works out OK then copy them to /lib/modules/misc and update the dependencies.

```

rtl29: # cp cr_module/blcr.o /lib/modules/misc
rtl29: # cp vmadump/vmadump.o /lib/modules/misc
rtl29: # depmode -a

```

9.3.2 Compiling apps with BLCR

Then its simply compiling your application with a call to cr_init, which will set up the necessary data structures for check-pointing and calls to cr_request.file in the code at points where you want to preserve execution status.

```

rtl29: # gcc -lcr -o counting counting.c
rtl29: # ./counting
Counting demo: PID 280
Checkpoints every 10 iterations
checkpoint-file: context.280
Count = 1
Count = 2
Count = 3
....
checkpoint!
Count = 10
Count = 11
Count = 12
...
checkpoint!

```

```
Count = 20
Count = 21
<CNTRL>-C
```

In this case we would have successfully completed execution, but to see check-pointing we kill it with a fatal <CNTRL>-C then to restart at the last checkpoint, you simply restart it with:

```
rtl29: # cr_restart context.280
checkpoint!
Count = 20
Count = 21
Count = 22
...
```

We can't diagnose the <CNTRL>-C, but we can say that this application failure was a transient error as it continues now. Furthermore if it were a non transient error we could monitor events that lead up to the fatal event. You would restart the application and connect a debugger to it and could watch it fail. Especially if this failure happens only after running for weeks or maybe months this is very powerful .

9.3.3 Common problems

```
rtl29: # insmod vmadump/vmadump.o
vmadump.o: init_module: Invalid argument
vmadump.o: Hint: insmod errors can be caused by incorrect module parameters, including invalid IO or IR
    You may find more information in syslog or the output from dmesg
```

If you see this you most likely are not running exactly the same kernel that you compile blcr against, blcr is accessing low level data structures in the kernel and only can operate properly if it fits the kernel exactly, that is not only the version but also the config options must be the same ! Checking with dmesg after the unsuccessful insmod would give you something like:

```
vmadump: from bproc-3.1.10 Erik Hendriks <erik@hendriks.cx>
vmadump: Modified for blcr 0.3.0 <http://ftg.lbl.gov/checkpoint>
Running kernel does not match the System.map used to build vmadump.o
```

If you forget to load the blcr and/or vmadump module then you will get the error when running the checkpointed application - so always make sure that it is loaded otherwise you will crash the application due to check-pointing - which is not quite the intended use !

```
cr_core.c:388 cr_request_file: called w/o first calling cr_init()
```

Other than that there is very little one can do wrong while installing blcr.

10 Function Instrumentation

GCC allows to instrument function calls by calling it with the `-finstrument-function` flag and providing two functions either in a library or in the source of the code. GCC will place the two functions:

```
void __cyg_profile_func_enter ( void *this_fun,
                               void *call_site);
void __cyg_profile_func_exit ( void *this_fun,
                              void *call_site);
```

just before a call and just after the exit from any function call done in the code. The arguments are the address of the first instruction of the function being entered, and the second argument is the address at which the function was called, this is provided because the `__builtin_return_address(0)` does not work on all platforms supported by gcc.

To utilize this function instrumentation for profiling and temporal debugging one simply needs to gather the time stamp in the enter/exit function and log it to a file. Symbol resolution can be performed based on the output of `nm`. For details see the `kfi` ?? document.

11 Conclusion

An embedded OS becomes valuable for industrial applications when it is reliable and this reliability can be verified. This verification process requires some tools, and as the above sections hopefully show, embedded Linux has many tools in this area. The tools described in the article are not going to cover every situation, but they should help with many of the common debug problems in embedded systems.

As the embedded Linux community is very active, new tools are evolving all the time, in the recent 2.6.X kernel releases provisions for system wide profiling and built in ksymoops are integrated, simplifying runtime debugging of Linux based embedded systems. It is to be expected that more work in this area will happen in the near future, increasing the value of embedded Linux, and its reliability.

References

[GNU] GNU not UNIX,

<http://www.gnu.org/>, <ftp://ftp.gnu.org/>

[proc-howto] Terrehon Bowden terrehon@pacbell.net, Bodo Bauer bb@ricochet.net, Jorge Nerin jcomandante@zaralinux.com, Documentations/`proc.txt`,
`/linux/Documentation/filesystems/proc.txt`.

[BGCC] Richard W.M. Jones rjones@orchestream.com, Herman A.J. ten Brugge Haj.Ten.Brugge@net.HCC.nl (Maintainer), `bgcc` <http://www.inter.NL.net/hcc/Haj.Ten.Brugge>

[STRACE] Paul Kranenburg, Branko Lankester, Rick Sladkey, `strace` <http://www.liacs.nl/wichert/strace/>, `cvs -d :pserver:anonymous@cvs.strace.sourceforge.net:/cvsroot/strace`

[LTRACE] Juan Cespedes, `ltrace`, <http://freshmeat.net/projects/ltrace/>

[NJAMD] Mike Perry mikeperry@fscked.org, `njamd` <http://sourceforge.net/projects/njamd>

[KMSGDUMP] Randy Dunlap rddunlap@osdl.org, `kmsgdump kernel patches` <http://w.ods.org/tools/kmsgdump/>
<http://developer.osdl.org/rddunlap/kmsgdump/>

[OPROFILE] <http://oprofile.sourceforge.net/>

[LTP] <http://ltp.sourceforge.net/>

[RML] Robert M. Love rml@tech9.net, `Linux Kernel Development` http://tech9.net/rml/kernel_book/

[KFI] Nicholas Mc Guire, Kernel Function Instrumentation - tool analysis, <http://www.opentech.at/documents.html>, OpenTech, 2005