

**广东中兴新支点**

**CGEL API 参考手册**

**嵌入式操作平台**

# 声 明

本资料著作权属广东中兴新支点有限公司所有。未经著作权人书面许可，任何单位或个人不得以任何方式摘录、复制或翻译。

侵权必究。



是广东中兴新支点有限公司的注册商标。广东中兴新支点有限公司产品的名称和标志是广东中兴新支点有限公司的专有标志或注册商标。在本手册中提及的其他产品或公司的名称可能是其各自所有者的商标或商名。在未经广东中兴新支点有限公司或第三方商标或商名所有者事先书面同意的情况下，本手册不以任何方式授予阅读者任何使用本手册上出现的任何标记的许可或权利。

由于产品和技术的不断更新、完善，本资料中的内容可能与实际产品不完全相符，敬请谅解。如需查询产品的更新情况，请联系广东中兴新支点有限公司。

若需了解最新的资料信息，请访问网站 <http://www.gd-linux.com/>。

# 手册说明

《广东中兴新支点 CGEL API 参考手册》主要给用户介绍了广东中兴新支点 Carrier Grade Embedded Linux 3.x/4.x 所提供的特色功能的内核态和用户态编程接口函数。另外，用户还可以参考文档《广东中兴新支点 CGEL 用户手册》。

# 内容介绍

章 名	概 要
第 1 章 内存管理接口	介绍内存管理相关功能需要使用的接口函数。
第 2 章 IPC 信号增强接口	介绍 IPC 信号增强相关功能需要使用的接口函数。
第 3 章 文件系统接口	介绍 IPC 信号增强相关功能需要使用的接口函数。
第 4 章 多核差异化接口	介绍多核差异化功能需要使用的接口函数。
第 5 章 能耗接口	介绍能耗相关功能需要使用的接口函数。
第 6 章 调测接口	介绍调测相关功能需要使用的接口函数。
第 7 章 网络协议栈接口	介绍协议栈相关功能需要使用的接口函数。
第 8 章 虚拟化接口	介绍虚拟化相关功能需要使用的接口函数。
第 9 章 V2LIN 适配库接口	介绍 VxWorks 移植到 Linux 用户态应用的适配库接口函数。
第 10 章 CGEL4.x 与 CGEL3.x 差异	介绍 CGEL4.x 与 CGEL3.x 接口差异。
第 11 章 开源 LICENSE 说明	简单介绍 GPL

# 版本更新说明

文档版本	发布日期	备 注
CGEL_V4.01.20	2011-11-9	2011 年第一次版本发布。
CGEL_V4.02.20	2012-11-9	2012 年第一次版本发布。

# 本书约定

介绍符号的约定、键盘操作约定、鼠标操作约定以及四类标志。

## 1) 符号约定

- 样式 **按钮** 表示按钮名；带方括号 “**【属性栏】**” 表示人机界面、菜单条、数据表和字段名等；
- 多级菜单用 “->” 隔开，如 **【文件】->【新建】->【工程】** 表示 **【文件】** 菜单下的 **【新建】** 子菜单下的 **【工程】** 菜单项；
- 尖括号 **<路径>** 表示当前目录中 **include** 目录下的 **.h** 头文件，如 **<asm-m68k/io.h>** 表示 **/include/asm-m68k/io.h** 文件。

## 2) 标志

本书采用二个醒目标志来表示在操作过程中应该特别注意的地方：



提示：介绍提高效率的一些方法；



警告：提醒操作中的一些注意事项。

# 联系方式

电 话：020-87048510

电子信箱：[cgel@gd-linux.com](mailto:cgel@gd-linux.com)

公司地址：广东省天河区科技园高唐软件园基地高普路 1021 号 6 楼

邮 编：510663

# 目 录

<b>第 1 章 内存管理接口 .....</b>	<b>1</b>
<b>1.1 内存信息统计接口 .....</b>	<b>1</b>
1.1.1 内存总体信息统计接口 <code>baseinfo</code> .....	1
1.1.2 内核使用内存基本信息统计接口 <code>kernelinfo</code> .....	2
1.1.3 进程使用内存信息显示接口 <code>processinfo</code> .....	2
1.1.4 根据所在管理区布局显示页缓存占用内存信息显示接口 <code>cacheinfo</code> .....	3
1.1.5 以文件为单位显示页缓存占用内存信息显示接口 <code>fileinfo</code> .....	4
1.1.6 块设备占用内存信息统计接口 <code>bufferinfo</code> .....	5
<b>1.2 内存映射相关接口 .....</b>	<b>5</b>
1.2.1 <code>syscall</code> 分配物理内存接口 .....	5
1.2.2 <code>syscall</code> 释放物理内存接口 .....	5
1.2.3 内存即时映射查询接口 .....	6
1.2.4 剩余可使用物理内存查询接口 .....	6
<b>1.3 页缓存限制功能接口 .....</b>	<b>7</b>
1.3.1 页缓存占用占总物理内存比例配置接口 <code>pagecache_ratio</code> .....	7
<b>1.4 用户态虚拟地址与物理地址双向互转功能接口 .....</b>	<b>7</b>
1.4.1 虚拟地址向物理地址转换接口 <code>vir2phy</code> .....	7
1.4.2 物理地址向虚拟地址转换接口 <code>pageinfo</code> .....	8
<b>1.5 内存耗尽时的嵌入式处理流程接口 .....</b>	<b>8</b>
1.5.1 设置关键任务接口 <code>oom_adj</code> .....	8
1.5.2 设置非关键任务内存不足时最大重试次数接口 <code>vm/fail_retry_time</code> .....	8
1.5.3 设置非关键任务睡眠时间接口 <code>oom_wait_ms</code> .....	9
1.5.4 设置处理内存不足信号的进程 ID 接口 <code>oom_handleurg_process</code> .....	9
1.5.5 设置三条水准分配比率接口 <code>watermarks_alloc_ratio</code> .....	9
1.5.6 用户态数据接口 <code>siginfo_t</code> .....	10
<b>1.6 <code>tmpfs/ramfs</code> 文件系统占用内存统计功能接口 <code>meminfo</code> .....</b>	<b>11</b>
<b>1.7 更改系统巨页池的巨页数量接口 <code>nr_hugepages_mempolicy</code> .....</b>	<b>11</b>

1.8	虚拟内存分配策略配置查询接口 <code>overcommit_memory</code> .....	12
1.9	进程启动时预留用户态地址空间接口 .....	12
<b>第 2 章 IPC 信号增强接口 .....</b>		<b>13</b>
2.1	快速消息队列创建接口 .....	13
2.1.1	创建消息队列接口 <code>Uf_msgq_create</code> .....	13
2.2	快速消息队列删除接口 .....	13
2.2.1	删除消息队列接口 <code>uf_msgq_delete</code> .....	13
2.3	快速消息队列接收接口 .....	14
2.3.1	接收消息队列接口 <code>uf_msgq_recv</code> .....	14
2.4	快速消息队列发送接口 .....	14
2.4.1	发送消息队列接口 <code>uf_msgq_send</code> .....	14
2.5	快速信号量创建接口 .....	15
2.5.1	B 信号量初始化接口 <code>uf_semb_init</code> .....	15
2.5.2	M 信号量初始化接口 <code>uf_semm_init</code> .....	15
2.5.3	C 信号量初始化接口 <code>uf_semc_init</code> .....	16
2.6	快速信号量删除接口 .....	16
2.6.1	快速信号量删除接口 <code>uf_sem_del</code> .....	16
2.7	快速信号量获取接口 .....	17
2.7.1	B 信号量获取接口 <code>uf_semb_take</code> .....	17
2.7.2	M 信号量获取接口 <code>uf_semm_take</code> .....	17
2.7.3	C 信号量获取接口 <code>uf_semc_take</code> .....	17
2.8	快速信号量释放接口 .....	18
2.8.1	B 信号量释放接口 <code>uf_semb_give</code> .....	18
2.8.2	M 信号量释放接口 <code>uf_semm_give</code> .....	18
2.8.3	C 信号量释放接口 <code>uf_semc_give</code> .....	18
<b>第 3 章 文件系统接口 .....</b>		<b>20</b>
3.1	jffs2 文件系统信息查看功能接口 .....	20
3.1.1	jffs2 文件系统参数查看接口 <code>info_N</code> .....	20

3.1.2	jffs2 文件系统节点分布查看接口 inode_info.....	20
3.2	jffs2 文件系统主动垃圾回收功能接口 .....	21
3.2.1	主动垃圾回收功能接口 _sysctl .....	21
3.2.2	jffs2 主动垃圾回收接口 start_gc_N .....	22
3.2.3	jffs2 主动垃圾回收调试等级设置接口 debug_N .....	22
3.3	文件标志扩展接口 .....	22
3.3.1	CLOEXEC/CLOFORK 功能打开文件接口 open .....	22
3.4	访问磁盘失败次数统计功能接口 .....	23
3.4.1	块设备访问失败次数查询接口 diskerrorinfo.....	23
3.4.2	指定块设备访问失败次数查询接口 syscall.....	23
3.5	统一系统调用查询接口 .....	24
3.5.1	统一系统调用号查询接口 nr_unify_syscall .....	24

## 第 4 章 多核差异化接口 ..... 26

4.1	AMP 多核差异化主核启动参数接口 .....	26
4.2	AMP 多核差异化 Loader 加载器接口 .....	26
4.3	AMP 多核差异化共享内存统一系统调用接口 .....	27
4.3.1	P2020 单板创建共享内存接口 .....	28
4.3.2	P2020 单板写入共享内存接口 .....	28
4.3.3	P2020 单板读取共享内存接口 .....	29
4.3.4	P2020 单板共享内存控制接口 .....	29
4.3.5	P2020 单板删除共享内存接口 .....	29
4.4	查看共享内存信息接口 shminfo .....	30
4.5	设置或查询软中断占用率较低门限接口 sirqrate_low.....	30
4.6	设置或查询软中断占用率较高门限接口 sirqrate_high .....	30
4.7	算法开启与否的开关接口 sirq_balance .....	31
4.8	查询或设置软中断掩码接口 sirq_mask .....	31
4.9	查询或设置非实时任务集合运行比例接口 op_interval.....	32

4.10	查询或设置非实时任务集合每次运行的最大微秒数接口 <code>run_interval</code> .....	32
4.11	设置某核上不受分区调度影响的实时任务优先级接口 <code>guard_prio</code> .....	32
4.12	查询和设置排他性\独占\共享 CPU 集合接口 <code>exclusive_cpus</code> .....	33
4.13	查询或设置独占式\共享式 CPU 集合的迁移结果接口 <code>migrate_info</code> .....	34
4.14	排它核负载均衡配置接口 <code>exclusive_cpus_balance</code> .....	35
4.15	控制实时工作队列的优先级接口 <code>rt_events_high_priority</code> .....	35
4.16	查询或设置定时器绑定的 CPU 号接口 <code>bind_timer_cpu</code> .....	35
4.17	查询和设置 CPU 掩码集合接口 <code>softlockup_cpumask</code> .....	36
4.18	中断绑定接口 <code>smp_affinity</code> .....	36
<b>第 5 章</b>	<b>能耗接口.....</b>	<b>38</b>
5.1	系统能耗状态控制接口 <code>state</code> .....	38
5.2	STD 操作模式控制接口 <code>disk</code> .....	38
5.3	写磁盘内存映像大小控制接口 <code>image_size</code> .....	39
5.4	磁盘设备号指定接口 <code>resume</code> .....	39
5.5	wakeup 事件设备支持判定接口 <code>wakeup</code> .....	40
5.6	USB 设备支持判定接口 <code>level</code> .....	40
5.7	USB 设备最大空闲时间设置接口 <code>autosuspend</code> .....	40
5.8	设备恢复后运行时间显示接口 <code>active_duration</code> .....	40
5.9	设备连接后运行时间显示接口 <code>connected_duration</code> .....	41
5.10	CPU 当前运行频率显示接口 <code>cpuinfo_cur_freq</code> .....	41
5.11	CPU 最低运行频率显示接口 <code>cpuinfo_min_freq</code> .....	41
5.12	CPU 最高运行频率显示接口 <code>cpuinfo_max_freq</code> .....	42



5.13 CPU 频率切换时间延迟显示接口 <code>cpuinfo_transition_latency</code> .....	42
5.14 CPU 可支持频率列表显示接口 <code>scaling_available_frequencies</code> .....	42
5.15 CPU 可支持策略列表显示接口 <code>scaling_available_governors</code> .....	42
5.16 CPU 当前策略运行频率显示接口 <code>scaling_cur_freq</code> .....	43
5.17 CPU 当前硬件驱动显示接口 <code>scaling_driver</code> .....	43
5.18 CPU 当前频率策略显示接口 <code>scaling_governor</code> .....	43
5.19 CPU 当前策略频率最大显示接口 <code>scaling_max_freq</code> .....	43
5.20 CPU 当前策略频率最小显示接口 <code>scaling_min_freq</code> .....	44
5.21 CPU 当前速度设置接口 <code>scaling_setspeed</code> .....	44
5.22 进程 <code>nice</code> 值忽略控制接口 <code>ignore_nice_load</code> .....	44
5.23 设置 <code>ondemand</code> 降低 CPU 频率接口 <code>ondemand/powersave_bias</code> .....	45
5.24 性能采用时间间隔设置接口 <code>sampling_rate</code> .....	45
5.25 性能采用最小时间间隔显示接口 <code>sampling_rate_min</code> .....	45
5.26 性能采用最大时间间隔显示接口 <code>sampling_rate_max</code> .....	46
5.27 CPU 频率提高占用率设置接口 <code>up_threshold</code> .....	46
5.28 CPU 的 <code>idle</code> 状态调整使用驱动显示接口 <code>current_driver</code> .....	46
5.29 CPU 的 <code>idle</code> 状态调整策略显示接口 <code>current_governor_ro</code> .....	46
5.30 CPU 的 <code>idle</code> 状态简单描述显示接口 <code>desc</code> .....	47
5.31 CPU 的 <code>idle</code> 状态调整时间延迟显示接口 <code>latency</code> .....	47
5.32 CPU 的 <code>idle</code> 状态下名称显示接口 <code>name</code> .....	47
5.33 CPU 的 <code>idle</code> 状态下能量消耗显示接口 <code>power</code> .....	47
5.34 CPU 的 <code>idle</code> 状态下持续时间显示接口 <code>time</code> .....	48

5.35 CPU 的 idle 状态下进入次数显示接口 usage .....	48
---	----

## 第 6 章 调测接口..... 49

6.1 kprobe 动态插点接口.....	49
6.1.1 注册动态插点接口 register_kprobe .....	49
6.1.2 注销动态插点接口 unregister_kprobe .....	49
6.1.3 批量注册动态插点接口 register_kprobes.....	50
6.1.4 批量注销动态插点接口 unregister_kprobes .....	50
6.2 jprobe 动态插点接口.....	50
6.2.1 注册跳转动态插点接口 register_jprobe .....	50
6.2.2 注销跳转动态插点接口 unregister_jprobe .....	51
6.2.3 批量注册跳转动态插点接口 register_jprobes.....	51
6.2.4 批量注销跳转动态插点接口 unregister_jprobes.....	51
6.3 kretprobe 动态插点接口.....	52
6.3.1 函数返回点动态检查点注册接口 register_kretprobe .....	52
6.3.2 函数返回点动态检查点注销卸载接口 unregister_kretprobe .....	52
6.3.3 组动态检查点多函数返回点注册接口 register_kretprobes .....	52
6.3.4 组动态检查点函数返回点注销卸载接口 unregister_kretprobes .....	53
6.4 静态检查点调测动作接口.....	53
6.4.1 静态检查点创建接口 TRACE_EVENT .....	53
6.4.2 静态检查点调测钩子注册接口 register_static_trace .....	54
6.4.3 静态检查点调测钩子注销接口 unregister_static_trace .....	54
6.5 内核预设调测功能控制接口 .....	54
6.5.1 内核预设调测可用显示接口 available_tracers .....	54
6.5.2 内核预设调测功能当前选择接口 current_tracer .....	55
6.5.3 当前跟踪选项接口 trace_options.....	55
6.5.4 当前内核预设调测功能启动情况接口 tracing_enabled .....	55
6.6 无锁环形管道输出接口 .....	56
6.6.1 ring_buffer 分配接口 ring_buffer_alloc .....	56
6.6.2 ring_buffer 释放接口 ring_buffer_free .....	56
6.6.3 ring_buffer 数据块写入接口 ring_buffer_write .....	56
6.6.4 ring_buffer 无阻塞读出接口 trace .....	57
6.6.5 ring_buffer 阻塞读出接口 trace_pipe.....	57

<b>6.7</b>	<b>CPU 信息获取接口 .....</b>	<b>57</b>
6.7.1	CPU 寄存器信息获取接口 trace_get_regs .....	57
6.7.2	CPU 负载均衡统计信息查看接口 schedstat .....	58
6.7.3	CPU 调度域参数查看接口 domain<m> .....	58
<b>6.8</b>	<b>内核 SLAB 错误注入控制接口 .....</b>	<b>60</b>
6.8.1	错误概率参数设置接口 probability .....	60
6.8.2	两次错误之间成功次数设置接口 interval .....	60
6.8.3	允许错误注入最大次数设置接口 times .....	61
6.8.4	第一次错误植入前间隔次数设置接口 space .....	61
6.8.5	failslab 消息输出内容设置接口 verbose .....	61
6.8.6	failslab 任务过滤器开关设置接口 task-filter .....	62
6.8.7	failslab 堆栈回溯有效区域起始地址设置接口 require-start .....	62
6.8.8	failslab 堆栈回溯有效区域结束地址设置接口 require-end .....	62
6.8.9	failslab 堆栈回溯无效区域起始地址设置接口 reject-start .....	63
6.8.10	failslab 堆栈回溯无效区域结束地址设置接口 reject-end .....	63
6.8.11	failslab 堆栈回溯深度设置接口 stacktrace-depth .....	63
6.8.12	failslab 等待页面分配错误注入忽略设置接口 ignore-gfp-wait .....	64
<b>6.9</b>	<b>内核 PAGE 分配错误注入控制接口 .....</b>	<b>64</b>
6.9.1	错误概率参数设置接口 probability .....	64
6.9.2	两次错误之间成功次数设置接口 interval .....	65
6.9.3	允许错误注入最大次数设置接口 times .....	65
6.9.4	第一次错误植入前间隔次数设置接口 space .....	65
6.9.5	fail_page_alloc 消息输出内容设置接口 verbose .....	66
6.9.6	fail_page_alloc 任务过滤器开关设置接口 task-filter .....	66
6.9.7	fail_page_alloc 堆栈回溯有效区域起始地址设置接口 require-start .....	66
6.9.8	fail_page_alloc 堆栈回溯有效区域结束地址设置接口 require-end .....	67
6.9.9	fail_page_alloc 堆栈回溯无效区域起始地址设置接口 reject-start .....	67
6.9.10	fail_page_alloc 堆栈回溯无效区域结束地址设置接口 reject-end .....	67
6.9.11	fail_page_alloc 堆栈回溯深度设置接口 stacktrace-depth .....	68
6.9.12	fail_page_alloc 等待页面分配错误注入忽略设置接口 ignore-gfp-wait .....	68
6.9.13	fail_page_alloc 高端页面分配错误注入忽略接口 ignore-gfp-highmem .....	68
6.9.14	页面分配错误注入最小页面分配 order 接口 min-order .....	69
<b>6.10</b>	<b>内核 IO 请求错误注入控制接口 .....</b>	<b>69</b>
6.10.1	错误概率参数设置接口 probability .....	69
6.10.2	两次错误之间成功次数设置接口 interval .....	69

6.10.3	允许错误注入最大次数设置接口 times.....	70
6.10.4	第一次错误植入前间隔次数设置接口 space.....	70
6.10.5	fail_make_request 消息输出内容设置接口 verbose.....	70
6.10.6	fail_make_request 任务过滤器开关设置接口 task-filter.....	71
6.10.7	fail_make_request 堆栈回溯有效区域起始地址设置接口 require-start.....	71
6.10.8	fail_make_request 堆栈回溯有效区域结束地址设置接口 require-end.....	71
6.10.9	fail_make_request 堆栈回溯无效区域起始地址设置接口 reject-start .....	72
6.10.10	fail_make_request 堆栈回溯无效区域结束地址设置接口 reject-end .....	72
6.10.11	fail_make_request 堆栈回溯深度设置接口 stacktrace-depth.....	72
<b>6.11</b>	<b>异常可靠重启接口.....</b>	<b>73</b>
6.11.1	异常重启时的回调处理接口 .....	73
6.11.2	异常信息控制台输出设置接口 excep_not_output .....	73
<b>6.12</b>	<b>阻塞任务内核态堆栈回溯功能接口.....</b>	<b>73</b>
6.12.1	查看非运行状态进程的内存态堆栈回溯接口.....	73
6.12.2	查看非运行状态线程的内存态堆栈回溯接口.....	73
<b>6.13</b>	<b>系统黑匣子功能接口.....</b>	<b>74</b>
6.13.1	内存块设备创建接口 .....	74
6.13.2	获取系统黑匣子信息接口 .....	74
6.13.3	向系统黑匣子存储一条信息接口 .....	75
6.13.4	获取存储区域当前记录长度接口 .....	76
6.13.5	获取存储区域总长度接口 .....	76
6.13.6	黑匣子区域基地址、滚动区大小和非滚动区大小.....	76
6.13.7	黑匣子模块初始化接口 kbbbox_init.....	77
6.13.8	获取系统黑匣子信息接口 kbbbox_get_info .....	77
6.13.9	向系统黑匣子存储一条信息接口 kbbbox_store_info .....	78
6.13.10	获取存储区域当前记录长度接口 kbbbox_get_len.....	78
6.13.11	获取存储区域总长度接口 kbbbox_get_size .....	78
6.13.12	常规信息保存接口 kbbbox_store_context.....	79
6.13.13	黑匣子回调的注册接口 .....	79
6.13.14	黑匣子回调注销接口 kbbbox_unregister_callback .....	80
6.13.15	调用黑匣子回调的接口 .....	80
6.13.16	清空系统黑匣子的统一系统调用接口 .....	80
6.13.17	清空系统黑匣子的内核接口 .....	81
<b>6.14</b>	<b>softlockup 功能接口.....</b>	<b>81</b>

6.14.1	设置控制台输出时间间隔接口 oftlockup_print_interval.....	81
6.14.2	softlockup 回调接口 .....	81
<b>6.15</b>	<b>任务运行监控接口 .....</b>	<b>82</b>
6.15.1	对应线程非运行时间统计接口 outcpustatistics .....	82
6.15.2	任务长时间不运行时间间隔设置接口 moni_nonrunning/interval.....	82
6.15.3	监控任务长时间不运行启动接口 moni_nonrunning/pids .....	82
6.15.4	监控任务长时间运行启动接口 moni_running/interval.....	83
6.15.5	proc/sys/kernel/moni_running/cpus_allowed 接口 .....	83
<b>6.16</b>	<b>任务退出监测功能接口 .....</b>	<b>84</b>
6.16.1	监控功能任务退出激活查询和设置接口 enable .....	84
6.16.2	监控功能对象线程号查询和设置接口 thread_ids .....	84
6.16.3	清空当前所有带监控的任务 id 接口 reset.....	85
6.16.4	清除待监控任务列表中的某个 id 接口 delid .....	85
<b>6.17</b>	<b>数据断点接口 .....</b>	<b>85</b>
6.17.1	数据断点设置接口 sys_write_watchregs .....	85
6.17.2	设置或取消数据断点接口 hard_regs.....	86
6.17.3	数据断点区间设置接口 sys_write_watchregs .....	86
6.17.4	设置或取消数据区间断点接口 hard_range.....	87
6.17.5	用户态内存区域监控设置接口 sys_watchrange .....	87
<b>6.18</b>	<b>精确统计 IDLE 接口 .....</b>	<b>88</b>
6.18.1	IDLE 任务中断时间统计接口 stat.....	88
6.18.2	进程被中断打断时间统计接口 intr.....	88
6.18.3	线程被中断打断时间统计接口 intr.....	89
<b>6.19</b>	<b>死机死锁检测接口 .....</b>	<b>89</b>
6.19.1	开启 NMI 硬件看门狗接口 wdt_enable .....	89
6.19.2	关闭 NMI 硬件看门狗接口 wdt_disable .....	89
6.19.3	看门狗超时时间管理接口 wdt_period .....	90
6.19.4	开启 NMI 软件看门狗接口 soft_enable .....	90
6.19.5	proc/nmi/soft_disable 接口 .....	90
6.19.6	V2LIN ufipc 机制 taskLock/intLock 调度检测接口/proc/u2k_debug_enable .....	91
<b>6.20</b>	<b>深度睡眠任务长期不调度检测接口 .....</b>	<b>91</b>
6.20.1	设置监控范围 hung_task .....	91
6.20.2	设置是否产生 kernel panic 接口 hung_task_panic .....	91

6.20.3	限制每次遍历的线程数目上限接口 <code>hung_task_check_count</code> .....	92
6.20.4	检测间隔时间接口 <code>hung_task_timeout_secs</code> .....	92
6.20.5	打印告警次数接口 <code>hung_task_warnings</code> .....	92
<b>6.21</b>	<b>其他信息获取接口 .....</b>	<b>93</b>
6.21.1	进程的内存与寄存器信息获取接口 <code>trace_get_dump</code> .....	93
6.21.2	当前进程的堆栈回溯信息获取接口 <code>bt_stacks</code> .....	93
6.21.3	用户态地址对应的内核态虚拟地址查询接口 <code>sys_user_to_kernel</code> .....	93
6.21.4	内核态保存记录结果获取接口 <code>sys_watchresult</code> .....	94
6.21.5	进程下子进程 <code>ra</code> 值查看接口 <code>extra_stat</code> .....	94
6.21.6	调度细节统计信息查看接口 <code>sched</code> .....	94
6.21.7	获取当前的时间标签信息接口 <code>get_timestamp</code> .....	95
6.21.8	查询核间中断统计计数接口 <code>inter_process_interrupt</code> .....	95
<b>6.22</b>	<b>SYSRQ 功能扩展魔法键接口 .....</b>	<b>96</b>
<b>第 7 章 网络协议栈接口 .....</b>		<b>97</b>
<b>7.1</b>	<b>RPS/RFS 功能接口 .....</b>	<b>97</b>
7.1.1	指定网卡包交给特定 CPU 处理接口 <code>rps_cpus</code> .....	97
7.1.2	<code>rps_sock_flow_table</code> 总数设置和获取接口 <code>rps_sock_flow_entries</code> .....	98
7.1.3	<code>rps_dev_flow</code> 总数设置和获取接口 <code>rps_flow_cnt</code> .....	98
<b>7.2</b>	<b>LOP 功能接口 .....</b>	<b>99</b>
7.2.1	获取 <code>PACKET_LOP</code> 值接口 <code>getsockopt</code> .....	99
7.2.2	设置 <code>PACKET_LOP</code> 值接口 <code>setsockopt</code> .....	99
<b>7.3</b>	<b>设置 <code>NUD_REACHABLE</code> 到 <code>NUD_STALE</code> 状态的切换间隔接口 .....</b>	<b>100</b>
<b>7.4</b>	<b>网卡数据包截获功能 .....</b>	<b>100</b>
7.4.1	<code>filter</code> 规则的设置查询接口 .....	101
7.4.2	回调注册接口 <code>netdev_register_intercept_func</code> .....	101
7.4.3	回调注销接口 <code>netdev_unregister_intercept_func</code> .....	102
<b>第 8 章 虚拟化接口 .....</b>		<b>103</b>
<b>8.1</b>	<b>虚拟机用户打开共享内存授权设备文件接口 .....</b>	<b>103</b>
<b>8.2</b>	<b>虚拟机用户关闭共享内存授权设备文件接口 .....</b>	<b>103</b>

8.3	虚拟机用户操作共享内存授权设备文件接口 .....	104
-----	---------------------------	-----

8.4	全虚拟化虚拟机 PV-on-HVM 驱动内核启动参数控制接口 .....	105
-----	--------------------------------------	-----

## 第 9 章 V2LIN 适配库接口 ..... 106

9.1	任务接口.....	106
-----	-----------	-----

9.1.1	taskSpawn 接口 .....	106
-------	--------------------	-----

9.1.2	taskInit 接口 .....	107
-------	-------------------	-----

9.1.3	taskActivate 接口 .....	108
-------	-----------------------	-----

9.1.4	taskDelete 接口 .....	108
-------	---------------------	-----

9.1.5	taskDeleteForce 接口 .....	109
-------	--------------------------	-----

9.1.6	taskSuspend 接口 .....	109
-------	----------------------	-----

9.1.7	taskResume 接口 .....	109
-------	---------------------	-----

9.1.8	taskRestart 接口 .....	110
-------	----------------------	-----

9.1.9	taskPrioritySet 接口 .....	110
-------	--------------------------	-----

9.1.10	taskPriorityGet 接口 .....	111
--------	--------------------------	-----

9.1.11	taskLock 接口 .....	111
--------	-------------------	-----

9.1.12	taskUnlock 接口 .....	111
--------	---------------------	-----

9.1.13	taskIdSelf 接口 .....	112
--------	---------------------	-----

9.1.14	taskIdVerify 接口 .....	112
--------	-----------------------	-----

9.1.15	taskTcb 接口.....	112
--------	-----------------	-----

9.1.16	taskInfoGet 接口 .....	112
--------	----------------------	-----

9.1.17	taskShow 接口.....	113
--------	------------------	-----

9.1.18	taskName 接口 .....	113
--------	-------------------	-----

9.1.19	taskNameToId 接口.....	113
--------	----------------------	-----

9.1.20	taskIsReady 接口.....	114
--------	---------------------	-----

9.1.21	taskIsSuspended 接口.....	114
--------	-------------------------	-----

9.1.22	taskIdListGet 接口 .....	114
--------	------------------------	-----

9.2	信号量接口.....	115
-----	------------	-----

9.2.1	semShow 接口.....	115
-------	-----------------	-----

9.2.2	semBCreate 接口.....	115
-------	--------------------	-----

9.2.3	semCCreate 接口.....	117
-------	--------------------	-----

9.2.4	semMCreate 接口 .....	118
-------	---------------------	-----

9.2.5	semTake 接口 .....	119
-------	------------------	-----

9.2.6	semGive 接口 .....	120
-------	------------------	-----

9.2.7	semFlush 接口 .....	121
-------	-------------------	-----

9.2.8	semDelete 接口 .....	121
<b>9.3</b>	<b>消息队列接口 .....</b>	<b>122</b>
9.3.1	msgQCreate 接口 .....	122
9.3.2	msgQDelete 接口 .....	122
9.3.3	msgQSend 接口 .....	123
9.3.4	msgQReceive 接口 .....	123
9.3.5	msgQNumMsgs 接口 .....	124
9.3.6	msgQShow 接口 .....	124
<b>9.4</b>	<b>TICK 值接口 .....</b>	<b>125</b>
9.4.1	tick64Get 接口 .....	125
9.4.2	tickGet 接口 .....	125
<b>9.5</b>	<b>TIMER 定时器接口 .....</b>	<b>125</b>
9.5.1	timer_create 接口 .....	126
9.5.2	timer_connect 接口 .....	126
9.5.3	timer_settime 接口 .....	126
9.5.4	timer_gettime 接口 .....	127
9.5.5	timer_cancel 接口 .....	127
9.5.6	timer_delete 接口 .....	128
<b>9.6</b>	<b>WD 定时器接口 .....</b>	<b>128</b>
9.6.1	wdCreate 接口 .....	128
9.6.2	wdDelete 接口 .....	128
9.6.3	wdStart 接口 .....	128
9.6.4	wdCancel 接口 .....	129
9.6.5	wdShow 接口 .....	129
<b>9.7</b>	<b>END 接口 .....</b>	<b>129</b>
9.7.1	mib2Init 接口 .....	130
9.7.2	mib2ErrorAdd 接口 .....	130
9.7.3	endObjInit 接口 .....	131
9.7.4	endObjFlagSet 接口 .....	131
9.7.5	endEtherAddressForm 接口 .....	132
9.7.6	endEtherPacketDataGet 接口 .....	132
9.7.7	endEtherPacketAddrGet 接口 .....	133
<b>9.8</b>	<b>网络缓存池接口 .....</b>	<b>134</b>



9.8.1	netBufLibInit 接口 .....	134
9.8.2	netPoolInit 接口 .....	134
9.8.3	netPoolDelete 接口 .....	135
9.8.4	netMblkFree 接口 .....	135
9.8.5	netCIBlkFree 接口 .....	135
9.8.6	netCIFree 接口 .....	136
9.8.7	netMblkCIFree 接口 .....	136
9.8.8	netMblkCIChainFree 接口 .....	137
9.8.9	netMblkGet 接口 .....	137
9.8.10	netCIBlkGet 接口 .....	138
9.8.11	netClusterGet 接口 .....	138
9.8.12	netMblkCIGet 接口 .....	139
9.8.13	netTupleGet 接口 .....	139
9.8.14	netCIBlkJoin 接口 .....	140
9.8.15	netMblkCIJoin 接口 .....	141
9.8.16	netCIPoolIdGet 接口 .....	141
9.8.17	netMblkToBufCopy 接口 .....	142
9.8.18	netMblkDup 接口 .....	143
9.8.19	netMblkChainDup 接口 .....	143
<b>9.9</b>	<b>LIST 接口 .....</b>	<b>144</b>
9.9.1	lstLibInit 接口 .....	144
9.9.2	lstInit 接口 .....	144
9.9.3	lstAdd 接口 .....	144
9.9.4	lstConcat 接口 .....	145
9.9.5	lstCount 接口 .....	145
9.9.6	lstDelete 接口 .....	146
9.9.7	lstExtract 接口 .....	146
9.9.8	lstFirst 接口 .....	147
9.9.9	lstGet 接口 .....	147
9.9.10	lstInsert 接口 .....	147
9.9.11	lstLast 接口 .....	148
9.9.12	lstNext 接口 .....	148
9.9.13	lstNext 接口 .....	149
9.9.14	lstPrevious 接口 .....	149
9.9.15	lstNStep 接口 .....	149
9.9.16	lstFind 接口 .....	150

9.9.17	lstFree 接口 .....	150
<b>9.10</b>	<b>事件接口 .....</b>	<b>151</b>
9.10.1	eventReceive 接口 .....	151
9.10.2	eventSend 接口 .....	152
9.10.3	eventClear 接口 .....	152
<b>9.11</b>	<b>HOST 接口 .....</b>	<b>152</b>
9.11.1	hostTblInit 接口 .....	152
9.11.2	hostAdd 接口 .....	153
9.11.3	hostDelete 接口 .....	153
9.11.4	hostGetByName 接口 .....	154
9.11.5	hostGetByAddr 接口 .....	154
<b>9.12</b>	<b>MUX 接口 .....</b>	<b>154</b>
9.12.1	muxLibInit 接口 .....	154
9.12.2	muxDevLoad 接口 .....	155
9.12.3	muxDevStart 接口 .....	155
9.12.4	muxBind 接口 .....	156
9.12.5	muxSend 接口 .....	157
9.12.6	muxReceive 接口 .....	157
9.12.7	muxIoctl 接口 .....	158
9.12.8	muxUnbind 接口 .....	158
9.12.9	muxDevUnload 接口 .....	159
9.12.10	endFindByName 接口 .....	159
9.12.11	muxDevExists 接口 .....	159
<b>9.13</b>	<b>中断接口 .....</b>	<b>160</b>
9.13.1	intLock 接口 .....	160
9.13.2	intUnlock 接口 .....	160
9.13.3	intEnable 接口 .....	161
9.13.4	intDisable 接口 .....	161
9.13.5	intConnect 接口 .....	161
9.13.6	intLocalLock 接口 .....	162
9.13.7	intLocalUnlock 接口 .....	162
<b>9.14</b>	<b>异常钩子函数接口 .....</b>	<b>162</b>
9.14.1	excHookAdd 接口 .....	163

<b>9.15 网络库接口 .....</b>	<b>163</b>
9.15.1 ifUnnumberedSet 接口 .....	163
9.15.2 ifAddrAdd 接口 .....	164
9.15.3 ifAddrSet 接口 .....	164
9.15.4 ifAddrDelete 接口 .....	165
9.15.5 ifAddrGet 接口 .....	165
9.15.6 ifBroadcastSet 接口 .....	165
9.15.7 ifBroadcastGet 接口 .....	166
9.15.8 ifDstAddrSet 接口 .....	166
9.15.9 ifDstAddrGet 接口 .....	167
9.15.10 ifMaskSet 接口 .....	167
9.15.11 ifMaskGet 接口 .....	168
9.15.12 ifFlagChange 接口 .....	168
9.15.13 ifFlagSet 接口 .....	169
9.15.14 ifFlagGet 接口 .....	169
9.15.15 ifunit 接口 .....	170
9.15.16 ifNameToIfIndex 接口 .....	170
9.15.17 ifIndexToIfName 接口 .....	170
9.15.18 ipAttach 接口 .....	171
9.15.19 ipDetach 接口 .....	171
<b>9.16 错误状态接口 .....</b>	<b>172</b>
9.16.1 errnoGet 接口 .....	172
9.16.2 errnoOfTaskGet 接口 .....	172
9.16.3 errnoSet 接口 .....	172
9.16.4 errnoOfTaskSet 接口 .....	173
<b>9.17 网络校验和接口 .....</b>	<b>173</b>
9.17.1 checksum 接口 .....	173

## **第 10 章 CGEL4.X 与 CGEL3.X 差异 ..... 175**

<b>10.1 PROC 及 SYS 目录差异 .....</b>	<b>175</b>
10.1.1 weight .....	175
10.1.2 stat .....	175
10.1.3 meminfo.....	175
10.1.4 stat .....	176
10.1.5 smaps .....	176

10.1.6	status .....	177
10.1.7	sched .....	177
<b>10.2</b>	<b>API 差异.....</b>	<b>178</b>
10.2.1	sched_setaffinity .....	178

## **第 11 章 开源 LICENSE 说明 ..... 179**

<b>11.1</b>	<b>GPL 简介.....</b>	<b>179</b>
<b>11.2</b>	<b>开发指导.....</b>	<b>179</b>
<b>11.3</b>	<b>源码获取方式.....</b>	<b>180</b>

# 第 1 章

## 内存管理接口

### 1.1 内存信息统计接口

#### 1.1.1 内存总体信息统计接口 **baseinfo**

Proc 接口名称：

/proc/mm/baseinfo

功能：查看内存总体信息统计。

用法：cat /proc/mm/baseinfo

输出参数：

- Memory 物理内存大小；
- Available 可用物理内存大小；
- Free 剩余物理内存；
- HighMem 高端内存大小；
- LowMem 低端内存大小；
- UserSpace 用户空间占用内存大小；
- KernelSpace 内核空间占用物理内存大小；
- PageCache 页缓存大小。

```
# cat /proc/mm/baseinfo
Memory:      2048000 kB
Available:    2027944 kB
Free:         1983664 kB
HighMem:      1130432 kB
LowMem:       897512 kB
UserSpace:    39739 kB
KernelSpace:  4541 kB
PageCache:    2216 kB
```

### 1.1.2 内核使用内存基本信息统计接口 kernelinfo

Proc 接口名称:

```
/proc/mm/kernelinfo
```

功能: 查看内核使用内存基本信息统计。

用法: cat /proc/mm/kernelinfo

输出参数:

- SlabTotal 占用内存信息;
- Vmalloc 使用内存情况;
- Buffers 占用内存大小;
- Cached 占用内存大小;
- SwapCached 占用内存大小;
- Reserved 保留内存大小。

```
# cat /proc/mm/kernelinfo
SlabTotal:    1968 kB
Vmalloc:      114680 kB
Buffers:      0 kB
Cached:       2216 kB
SwapCached:   0 kB
Reserved:     19604 kB
```

### 1.1.3 进程使用内存信息显示接口 processinfo

Proc 接口名称:

```
/proc/mm/processinfo
```

功能: 查看进程使用内存基本信息。

用法: cat /proc/mm/processinfo

说明：

输出参数：

- RSS 进程占用物理内存大小；
- CODE 进程代码段占用内存大小；
- DATA 进程数据段占用内存大小；
- LIB 进程使用库占用内存大小；
- ANON 进程匿名映射占用内存大小；
- STACK 进程堆栈占用内存大小；
- SHARE 进程共享内存大小；
- IO 进程 IO 映射占用物理内存大小。

#	cat	/proc/mm/processinfo						
PID	Name	RSS	CODE	DATA	LIB	ANON	STACK	
::	SHARE	IO						
1	init	1824 kB	1412 kB	8 kB	0 kB	412 kB	84 kB	
::	0 kB	0 kB						
696	telnetd	1824 kB	1412 kB	8 kB	0 kB	412 kB	84 kB	
::	0 kB	0 kB						
698	sw	34608 kB	1412 kB	8 kB	0 kB	33196 kB	84 kB	
::	0 kB	0 kB						
699	ash	1824 kB	1412 kB	8 kB	0 kB	412 kB	84 kB	
::	0 kB	0 kB						
704	cat	1824 kB	1412 kB	8 kB	0 kB	412 kB	84 kB	
::	0 kB	0 kB						

#### 1.1.4 根据所在管理区布局显示页缓存占用内存信息显示接口 **cacheinfo**

Proc 接口名称：

```
/proc/mm/cachelinfo
```

功能：查看页缓存占用内存信息。

用法：cat /proc/mm/cachelinfo

输出参数：

- PageCaches 指定管理区页缓存大小；
- SwapCaches 页缓存中用于交换缓存的大小；
- Locked 被锁定的页缓存大小；

- WriteBack 处于回写状态的页缓存大小；
- Dirty 脏页大小；
- Mapped 已与用户进程建立映射关系的页缓存大小；
- Active 处于活动状态的页缓存大小。

```
# cat /proc/mm/cacheinfo
Zone:      PageCaches  SwapCaches  Locked  WriteBack  Dirty  Mapped  Active
HighMem   :    2216 kB     0 kB     0 kB     0 kB     0 kB  1688 kB  1548 kB
```

### 1.1.5 以文件为单位显示页缓存占用内存信息显示接口 fileinfo

Proc 接口名称：

```
/proc/mm/fileinfo
```

功能：查看页缓存占用内存信息。

用法：cat /proc/mm/fileinfo

输出参数：

- Name 占有物理内存的文件名；
- SIZE 该文件占用内存实际大小；
- COUNT 文件 dentry 使用计数（dentry 已有现成记录）。

```
# cat /proc/mm/fileinfo
Name      SIZE      COUNT(3)
busybox   1200 kB   10
cat        4 kB     0
inittab    4 kB     0
rcS        4 kB     0
ash        4 kB     0
shm.out    416 kB    2
sw         4 kB     0
sh         4 kB     0
ifconfig   4 kB     0
mount      4 kB     0
telnetd    4 kB     0
fstab      4 kB     0
profile    4 kB     0
.ash_history 4 kB     0
init       4 kB     0
ps         4 kB     0
passwd     4 kB     0
```



### 1.1.6 块设备占用内存信息统计接口 **bufferinfo**

Proc 接口名称:


```
/proc/mm/bufferinfo
```

功能: 查看块设备占用内存信息统计。

用法: `cat /proc/mm/bufferinfo`

输出参数: 块设备名称与块设备占用物理内存大小。

---

 提示: 若系统无块设备, 则信息显示为空。

---

## 1.2 内存映射相关接口

### 1.2.1 **syscall** 分配物理内存接口

接口类型: 用户态接口。

函数调用名称:

```
long syscall (int NR_UNIFY_SYSCALL, unsigned int cmd, unsigned long addr, unsigned long len)
```

功能: 不影响虚拟地址空间的前提下分配物理内存。

返回值: 返回分配物理内存线性区的起始地址, 成功; -1, 失败。

参数说明:

- `cmd`, 代表具体的子功能号, 此处对应 `SYSCALL_EMMAP`。
- `addr`, 待分配物理内存的线性地址起点。
- `len`, 待分配物理内存的长度。

### 1.2.2 **syscall** 释放物理内存接口

接口类型: 用户态接口。

函数调用名称:

```
long syscall (int NR_UNIFY_SYSCALL, unsigned int cmd, unsigned long addr, unsigned long len)
```

功能：不影响虚拟地址空间的前提下释放物理内存。

返回值：返回线性区的起始地址：成功；-1：失败

参数说明：

- cmd，代表具体的子功能号，此处对应 SYSCALL\_EMUNMAP。
- addr，待释放物理内存的线性地址起点。
- len，待释放物理内存的长度。

### 1.2.3 内存即时映射查询接口

接口类型：用户态接口。

函数调用名称：

```
long syscall (int NR_UNIFY_SYSCALL, unsigned int cmd, unsigned long map_noalloc_ptr)
```

功能：返回内核中定义的 MAP\_NOALLOC 宏值。MAP\_NOALLOC 宏是在即时映射情况下一个特殊标志位。用户在调用 mmap 接口（将一个文件或者其它对象映射进内存）时通过参数 flag 字段设置 MAP\_NOALLOC 表示映射请求只分配虚拟地址空间不分配物理内存。

参数说明：

- cmd，代表具体的子功能号；
- map\_noalloc\_ptr，用户态程序变量的地址，用于接收内核返回的 MAP\_NOALLOC 宏值。

返回值：成功返回 0，失败返回-1。

### 1.2.4 剩余可使用物理内存查询接口

接口类型：用户态接口。

功能：查询当前系统剩余的可使用的物理内存大小，效率高于 Proc 接口。

函数调用名称：

```
int syscall(int nr, unsigned int id, unsigned long addr);
```

参数说明：

- nr：对应于架构下的统一系统调用号；

- **id**: 扩展子功能号, 获取剩余可使用物理内存的子功能号 `SYSCALL_GET_AVAILABLE_MEM`;
- **addr**: 出参, 用于返回当前系统的可使用剩余内存大小的指针 (用户输入应该为 `unsigned int` 的指针), 内存大小的单位为 `Kbytes`。

返回值: 0 表示调用成功, 小于 0 表示调用失败并返回错误码。错误码如下所示:

-ENOSYS 不支持该功能;

-EFAULT 传入的 `addr` 非法不可写。

## 1.3 页缓存限制功能接口

在多核环境下 (配置 `CONFIG_SMP`), 内核出于优化性能考虑, 当 `page cache` 增加时并不立即更新 `zone` 结构的页缓存计数值, 而是先更新每 `cpu` 变量的 `pcp` 结构页缓存计数值, 当其达到预先设置的阈值 `threshold` 时, 才将上次更新到本次更新的累加值更新到 `zone` 结构页缓存计数中。因此, `zone` 结构中页缓存计数值不是一个页面一个页面地累加, 而是一次累加 `threshold+1+threshold/2` 个页面, 用户通过 `/proc/zoneinfo` 实际看到的页缓存值可能会超出设置的限制值。这是内核性能优化带来的影响, 对页缓存的影响可以忽略。

### 1.3.1 页缓存占用占总物理内存比例配置接口 `pagecache_ratio`

Proc 接口名称:

```
/proc/sys/vm/pagecache_ratio
```

功能: 设置页缓存占用总物理内存的比例上限。如设置为 20, 表示页缓存最多占用 20% 总物理内存。

用法: `echo pagecache_ratio > /proc/sys/vm/pagecache_ratio`

说明: 或以其他文件写入方式设置页缓存占用总物理内存的比例, 取值为:  $5 \leq \text{pagecache\_ratio} \leq 100$ 。

## 1.4 用户态虚拟地址与物理地址双向互转功能接口

### 1.4.1 虚拟地址向物理地址转换接口 `vir2phy`

Proc 接口名称:

```
/proc/<pid>/vir2phy
```

功能: 作为输入和输出接口用于虚拟地址向物理地址的转换。

用法:

```
echo va > /proc/<pid>/vir2phy  
cat /proc/<pid>/vir2phy
```

说明：为输入时（或以其他文件写入的方式）将需要查询的线性地址写入所在进程的 vir2phy 文件；为输出时（或以其他文件读取的方式）便可读出该虚拟地址对应的物理地址。

### 1.4.2 物理地址向虚拟地址转换接口 pageinfo

Proc 接口名称：

```
/proc/pageinfo
```

功能：根据输入的物理地址，反查物理页面的使用情况，如果被进程使用打印出进程的 VMA 区间信息以及相应 tgid。

说明：cat 该文件，向串口打印输入的物理地址对应的页面使用信息(若无输入，默认为 0)，可通过 dmesg 命令查看；echo 该文件，可设置需要解析的输入物理地址。

## 1.5 内存耗尽时的嵌入式处理流程接口

### 1.5.1 设置关键任务接口 oom\_adj

Proc 接口名称：

```
/proc/pid/oom_adj
```

功能：设置关键任务。

用法：echo -17 > /proc/<pid>/oom\_adj

说明：关键任务可以是管理进程的关键线程、shell 代理线程等，这些线程不允许挂起或者被杀掉。在内存不足时允许该线程在水线下分配内存，以便最大程度的保证这些线程的运行，从而保证嵌入式系统的稳定性。（或以其他文件写入的方式）

### 1.5.2 设置非关键任务内存不足时最大重试次数接口 vm/fail\_retry\_time

Proc 接口名称：

```
/proc/sys/vm/fail_retry_time
```

功能：设置非关键任务内存不足时尝试分配内存的最大重试次数。

用法：echo N > /proc/sys/vm/fail\_retry\_time

说明：将需要重试的次数 N 写入 fail\_retry\_time。系统默认为 1 次。（或以其他文件写入的方式）

### 1.5.3 设置非关键任务睡眠时间接口 oom\_wait\_ms

Proc 接口名称：

```
/proc/sys/vm/oom_wait_ms
```

功能：设置非关键任务在尝试第二次内存分配之前等待的时间。由于在一次申请内存情况下会尝试多次，在第一次申请物理内存失败的情况下会尝试回收一些内存，然后再次尝试分配，因此这里等待的时间是一次申请过程的多次尝试时间。

用法：echo N > /proc/sys/vm/oom\_wait\_ms

说明：单位为毫秒，默认值为 10ms。

### 1.5.4 设置处理内存不足信号的进程 ID 接口 oom\_handleurg\_process

Proc 接口名称：

```
/proc/sys/vm/oom_handleurg_process
```

功能：设置内核向哪个进程发送内存不足的信号。

用法：echo ID > /proc/sys/vm/oom\_handleurg\_process

说明：如果 ID 为 0，则内核此时向所有进程发送此信号。如果非 0 就向此 ID 的进程发送此信号。默认值为 0。当内存不足时发送信号的结构见 1.5.6 节 [siginfo\\_t](#) 说明。

### 1.5.5 设置三条水线分配比率接口 watermarks\_alloc\_ratio

Proc 接口名称：

```
/proc/sys/vm/watermarks_alloc_ratio
```

功能：设置三级内存水线分配比率。

用法：echo a b c d > /proc/sys/vm/watermarks\_alloc\_ratio

说明：设置内核三级水线 ALLOC\_HARDER、ALLOC\_HARDEST 和 ALLOC\_HIGH 的分成比例。三级水线表示的分配类型如下：

ALLOC\_HARDER：实时非关键任务分配类型内存水线；

ALLOC\_HARDEST：内核态或用户态关键任务分配类型内存水线；

ALLOC\_HIGH: 中断或 ATOMIC 原子分配类型内存水线。

这三级水线将内核 MIN 水线下的内存分为四部分，分别是 ALLOC\_HARDER、ALLOC\_HARDEST、ALLOC\_HIGH 和内核保留部分，[a, b, c, d]代表这四部分的分成比例，默认是 1: 1: 1: 1。

### 1.5.6 用户态数据接口 siginfo\_t

接口说明：内存不足时通过信号反馈内核当前内存分配的信息。

结构：

```
struct siginfo {
    int si_signo;          /* 信号值，内存不足的信号为 SIGURG */
    int si_errno;          /* errno 值，为 0 */
    int si_code;           /* 内存不足时复用为内存分配类型和分配标志 gfp_mask */
    union{                 /* 联合数据结构，内存不足时使用 _rt 成员 */
        ...
        /* POSIX.1b signals */
        struct {
            __kernel_pid_t _pid; /* 发送信号进程的 pid */
            __ARCH_SI_UID_T _uid;
            sigval_t _sigval;     /* 返回内核控制路径 */
        } _rt;
        ...
    }
} siginfo_t;
```

字段说明：

- si\_signo: 信号值，内存不足的信号为 SIGURG;
- isi\_errno: errno 值，未使用，为 0;
- si\_code: 内存不足时复用为内存分配类型和分配标志 gfp\_mask;
- si\_code 的高 4 位用于根据内核的内存分配类型，提供给用户三种处理建议：

```
//中断或 ATOMIC 原子分配类型用户不做处理。
#define PROPOSAL_IGNORE    0x10000000
//用户态关键任务和非关键任务由用户自行处理。
#define PROPOSAL_DEFAULT   0x20000000
//其他分配类型建议重启。
#define PROPOSAL_RESTART   0x40000000
```

- `si_code` 的低 28 位用于返回内核当前的分配标志 `gfp_mask`;
- `pid`: 用于返回发送信号时进程的 `pid`;
- `sigval`: 传递参数用于返回内核控制路径。

内核控制路径包括硬中断上下文、软中断上下文、内核线程上下文和用户态系统调用，对应的宏为：

<code>#define CTL_PATH_HARDIRQ</code>	<code>0x01</code>
<code>#define CTL_PATH_SOFTIRQ</code>	<code>0x02</code>
<code>#define CTL_PATH_KERNEL_THREAD</code>	<code>0x04</code>
<code>#define CTL_PATH_USER_SYSCALL</code>	<code>0x08</code>

在内存不足发送信号后，用户态接收 `SIGURG` 后使用此结构接收内核信息。

## 1.6 tmpfs/ramfs 文件系统占用内存统计功能接口 `meminfo`

Proc 接口名称：

```
/proc/meminfo
```

功能：查看 `tmpfs`、`ramfs` 文件系统占用内存的尺寸。

用法：`cat /proc/meminfo`

说明：或以其他文件读取方式查看 `RamfsCached` 和 `TmpfsCached` 的值。

## 1.7 更改系统巨页池的巨页数量接口 `nr_hugepages_mempolicy`

功能：根据当前进程的 `NUMA` 内存策略来更改系统中巨页池的巨页数量。

Proc 接口名称：

```
/proc/sys/vm/nr_hugepages_mempolicy
```

用法：

```
//修改巨页池数量，策略为当前调用这个命令的进程的内存策略
echo nr > /proc/sys/vm/nr_hugepages_mempolicy
//查询巨页池数量
cat /proc/sys/ vm/nr_hugepages_mempolicy
```

## 1.8 虚拟内存分配策略配置查询接口 `overcommit_memory`

功能：虚拟内存分配策略配置查询。

Proc 接口名称：

```
/proc/sys/vm/overcommit_memory
```

用法：

```
echo N > /proc/sys/vm/ overcommit_memory      //修改虚拟内存分配策略
cat /proc/sys/ vm/ overcommit_memory           //查询巨页池数量
```

其中，N 值各值表示 0 启发式过量使用；1 一直过量使用；2 不过量使用。

## 1.9 进程启动时预留用户态地址空间接口

功能：用户进程启动时预留一段用户态虚拟地址出来，供应用特殊使用。

接口：进程启动时加上启动参数 `-resvV:[size]@[base]`

参数说明：内核检测参数无效时通过 `printk` 打印错误信息。

- `size`：预留的用户态虚拟地址空间大小；
- `base`：预留的用户态虚拟地址空间基址。

返回值：无。



# 第 2 章

## IPC 信号增强接口

### 2.1 快速消息队列创建接口

#### 2.1.1 创建消息队列接口 `Uf_msgq_create`

接口类型：用户态接口。

函数调用名称：

```
Uf_msgq_t * Uf_msgq_create(int max_msgs, int max_msglen, int opt)
```

功能：创建消息队列。

输入参数：

- `max_msgs` 最大消息数；
- `max_msglen` 消息最大长度；
- `opt` 消息选项（同 Vxworks 接口的 `msgQCreate` 的 `opt` 参数）。

返回值：非空，成功；NULL，失败

### 2.2 快速消息队列删除接口

#### 2.2.1 删除消息队列接口 `uf_msgq_delete`

接口类型：用户态接口。

函数调用名称：

```
int uf_msgq_delete(uf_msgq_t * pMsgQ)
```

功能：删除消息队列。

输入参数：

➤ pMsgQ 要删除的消息队列指针。

返回值：0，成功；非 0，失败。

## 2.3 快速消息队列接收接口

### 2.3.1 接收消息队列接口 uf\_msgq\_recv

接口类型：用户态接口。

函数调用名称：

```
int uf_msgq_recv(uf_msgq_t * pMsgQ, char *buffer, unsigned int buf_len, int timeout)
```

功能：接收消息队列。

输入参数：

- pMsgQ：消息队列结构指针；
- buffer：接收缓冲区；
- buf\_len：接收长度，单位为字节；
- timeout：超时时间，单位为毫秒。

返回值：0，成功；-1，失败。

## 2.4 快速消息队列发送接口

### 2.4.1 发送消息队列接口 uf\_msgq\_send

接口类型：用户态接口。

函数调用名称：

```
int uf_msgq_send(uf_msgq_t * pMsgQ, char *buffer, unsigned int nBytes, int timeout, int pri)
```

功能：发送消息队列。

输入参数:

- pMsgQ: 消息队列结构指针;
- buffer: 发送缓冲区;
- nBytes: 发送长度, 单位为字节;
- timeout: 超时时间, 单位为毫秒;
- pri: 发送消息优先级, 设置 0 为普通消息; 1 为紧急消息。

返回值: 0, 成功; 非 0, 失败。

## 2.5 快速信号量创建接口

### 2.5.1 B 信号量初始化接口 uf\_semb\_init

接口类型: 用户态接口。

函数调用名称:

```
int uf_semb_init(uf_sem_t *p_sem,int opts,uf_semb_state_t ini_state)
```

功能: B 信号量初始化

输入参数:

- p\_sem: 传入的信号量;
- opts: 创建信号量的选项 (同 Vxworks 接口的 semBCreate 的 opt 参数);
- ini\_state: 信号量初始化时的状态。

返回值: 0, 成功; 非 0, 失败。

### 2.5.2 M 信号量初始化接口 uf\_semm\_init

接口类型: 用户态接口。

函数调用名称:

```
int uf_semm_init(uf_sem_t *p_sem,int opts)
```

功能: M 信号量初始化

输入参数:

- p\_sem: 传入的信号量;
- opts: 创建信号量的选项 (同 Vxworks 接口的 semMCreate 的 opt 参数)。

返回值：0，成功；非 0，失败。

### 2.5.3 C 信号量初始化接口 `uf_semc_init`

接口类型：用户态接口。

函数调用名称：

```
int uf_semc_init(uf_sem_t *p_sem,int opts,int ini_cnt)
```

功能：C 信号量初始化。

输入参数：

- 输入参数 `p_sem`：传入的信号量；
- 输入参数 `opts`：创建信号量的选项（同 Vxworks 接口的 `semCCreate` 的 `opt` 参数）；
- 输入参数 `ini_cnt`：信号量初始化时的状态。

返回值：0，成功；非 0，失败。

## 2.6 快速信号量删除接口

### 2.6.1 快速信号量删除接口 `uf_sem_del`

接口类型：用户态接口。

函数调用名称：

```
int uf_semc_del(uf_sem_t *p_sem)
```

功能：信号量删除。

输入参数：

- 输入参数 `p_sem`：删除信号量的结构体地址；
- 输入参数 `opts`：创建信号量的选项；
- 输入参数 `ini_cnt`：信号量初始化时的状态。

返回值：0，成功；非 0，失败。

## 2.7 快速信号量获取接口

### 2.7.1 B 信号量获取接口 `uf_semb_take`

接口类型：用户态接口。

函数调用名称：

```
int uf_semb_take(uf_sem_t* p_sem, int timeout)
```

功能：获取 B 信号量。

输入参数：

- `p_sem`：传入的信号量；
- `timeout`：获取信号量时的超时时间，单位为毫秒。

返回值：0，成功；非 0，失败。

### 2.7.2 M 信号量获取接口 `uf_semm_take`

接口类型：用户态接口。

函数调用名称：

```
int uf_semm_take(uf_sem_t* p_sem, int timeout)
```

输入参数：

- `p_sem`：传入的信号量；
- `timeout`：获取信号量时的超时时间，单位为毫秒。

功能：获取 M 信号量。

返回值：0，成功；非 0，失败。

### 2.7.3 C 信号量获取接口 `uf_semc_take`

接口类型：用户态接口。

函数调用名称：

```
int uf_semc_take(uf_sem_t* p_sem, int timeout)
```

功能：获取 C 信号量。

输入参数：

- 输入参数 `p_sem`：传入的信号量；
- 输入参数 `timeout`：获取信号量时的超时时间，单位为毫秒。

返回值：0，成功；非 0，失败。

## 2.8 快速信号量释放接口

### 2.8.1 B 信号量释放接口 `uf_semb_give`

接口类型：用户态接口。

函数调用名称：

```
int uf_semb_give(uf_sem_t* p_sem)
```

功能：释放 B 信号量。

输入参数：`p_sem` 为传入的信号量。

返回值：0，成功；非 0，失败

### 2.8.2 M 信号量释放接口 `uf_semm_give`

接口类型：用户态接口。

函数调用名称：

```
int uf_semm_give(uf_sem_t* p_sem)
```

功能：释放 M 信号量。

输入参数：`p_sem` 为传入的信号量。

返回值：0，成功；非 0，失败。

### 2.8.3 C 信号量释放接口 `uf_semc_give`

接口类型：用户态接口。

函数调用名称：

```
int uf_semc_give(uf_sem_t* p_sem)
```

功能：释放 C 信号量。

输入参数：p\_sem 为传入的信号量。

返回值：0，成功；非 0，失败。

# 第 3 章

## 文件系统接口

### 3.1 jffs2 文件系统信息查看功能接口

#### 3.1.1 jffs2 文件系统参数查看接口 info\_N

Proc 接口名称:

```
/proc/sys/fs/jffs2/分区名/info_N
```

功能: 查看 jffs2 文件系统参数。

用法: `cat /proc/sys/fs/jffs2/分区名/info_N`

说明: 或其他文件读取方式查看该分区下 jffs2 文件系统参数。

#### 3.1.2 jffs2 文件系统节点分布查看接口 inode\_info

Proc 接口名称:

```
/proc/fs/jffs2/分区名/inode_info
```

功能: 查看 jffs2 文件系统节点分布信息。

用法: `cat /proc/fs/jffs2/分区名/inode_info`

说明: 或其他文件写入方式查看 jffs2 文件系统节点分布信息, 各类节点约等于四个链表同类节点的和, 主要反映一个节点分布的趋势。



## 3.2 jffs2 文件系统主动垃圾回收功能接口

### 3.2.1 主动垃圾回收功能接口\_sysctl

接口类型：用户态接口。

函数调用名称：

```
int _sysctl(struct __sysctl_args *args)
```

功能：jffs2 主动垃圾回收接口。

输入参数：参数\*args 为 jffs2 文件系统相关信息，说明如下。

```
struct __sysctl_args
{
    int    *name;    /* integer vector describing variable */
    int     nlen;    /* length of this vector */
    void   *oldval;  /* 0 or address where to store old value */
    size_t *oldlenp; /* available room for old value, overwritten by actual size of old value */
    void   *newval;  /* 0 or address of new value */
    size_t  newlen;  /* size of new value */
};
```

此结构中各参数含义：

- **name**：调用路径，比如要获取分区 0 的文件系统信息，需定义如下结构，并赋值给 **name**：

```
int part0_info_ctl[] = {CTL_FS,FS_JFFS2, JFFS2_PART0,PART_INFO};
```

其中 CTL\_FS 已包含在头文件 sysctl.h 中，无须定义；FS\_JFFS2 在内核定义为 22；JFFS2\_PART0 在内核定义为 0+1=1；PART\_INFO 在内核定义为 1，用户需要自定义。

- **nlen**：name 所指的路径的长度，可以使用宏 SIZE(x)来获取长度；比如：

```
#define SIZE(x) sizeof(x)/sizeof(x[0])
nlen = SIZE(part0_info_ctl);
```

- **oldval**：如需获取文件系统信息，该参数指定用户定义的 struct jffs2\_flash\_info 变量，同时返回信息保存在该结构中；如果是触发垃圾回收该参数设置为 0；
- **oldlenp**：参数 oldval 长度；

- newval: 如需获取文件系统信息, 该参数设置为 0; 如果触发垃圾回收该参数设置为 1 即可;
- newlen: 参数 Newval 长度;

返回值: 0, 成功; -1, 失败, 返回错误码 `errno` (`EFAULT` 错误地址; `ENODEV` 设备不存在; `ENOMEM` 空间不足; `EINVAL` 入参不正确; `EIO` 为 I/O 错误)。

### 3.2.2 jffs2 主动垃圾回收接口 `start_gc_N`

Proc 接口名称:

```
/proc/sys/fs/jffs2/分区名/start_gc_N
```

功能: 启动 jffs2 主动垃圾回收功能。

用法: `echo N > /proc/sys/fs/jffs2/分区名/start_gc_N`

说明: 或其他文件写入方式制定合适的垃圾回收策略, 在用户态主动触发垃圾回收, 当 `start_gc_N` 不为 0 的时候触发垃圾回收。

### 3.2.3 jffs2 主动垃圾回收调试等级设置接口 `debug_N`

Proc 接口名称:

```
/proc/sys/fs/jffs2/分区名/debug_N
```

功能: 设置或查看 jffs2 主动垃圾回收调试等级。

用法:

```
cat /proc/sys/fs/jffs2/分区名/debug_N      /*查看该分区下 jffs2 文件系统调试级别。*/  
echo N > /proc/sys/fs/jffs2/分区名/debug_N  /*设置当前分区 jffs2 调试级别。*/
```

说明: 或其他文件写入方式设置调试等级, `N` 的取值范围为 0、1、2、4。其中, 0 为关闭调试; 1 为异常流程; 2 为正常流程有打印输出; 4 为正常流程无打印输出, 有信息收集代码运行。

## 3.3 文件标志扩展接口

### 3.3.1 CLOEXEC/CLOFORK 功能打开文件接口 `open`

接口类型: 用户态接口。

函数调用名称:

```
long sys_open(const char __user *filename, int flags, int mode)
```

功能：打开文件是否支持 CLOEXEC 或 CLOFORK 功能。

输入参数：

- filename：文件名；
- flags：文件打开标志；
  - ✧ O\_CLOFORK：打开文件时支持 O\_CLOFORK 标志，使得在执行 fork 时，新进程不继承该文件句柄；
  - ✧ O\_CLOEXEC：打开文件时支持 O\_CLOEXEC 标志，使得在执行 exec 后，新进程不继承该文件句柄。
- mode：文件访问权限。

返回值：大于等于 0，成功；小于 0，失败

## 3.4 访问磁盘失败次数统计功能接口

### 3.4.1 块设备访问失败次数查询接口 diskerrorinfo

Proc 接口名称：

```
/proc/diskerrorinfo
```

功能：查询每个块设备的访问失败次数。

用法：cat /proc/diskerrorinfo

说明：输出环境中所有块设备的失败次数，设备名和失败次数一一对应。

### 3.4.2 指定块设备访问失败次数查询接口 syscall

接口类型：用户态接口。

函数调用名称：

```
int syscall(int syscall_nr, SYSCALL_DEV_DISK_FAIL_COUNT, char* disk_name, unsigned int  
name_len, unsigned int* fail_count);
```

功能：查询指定块设备访问的失败次数。

输入参数：

- `syscall_nr`: 当前内核版本的统一系统调用的调用号（不同版本可能不一样），在运行的环境输入下面的命令查询：`/ # cat /proc/nr_unify_syscall`
- `SYSCALL_DEV_DISK_FAIL_COUNT`: 用户程序必须与内核中的系统调用方式一样，必须采用如下定义，也在运行的环境输入命令查询：`/ # cat /proc/nr_unify_syscall`

```
#define SYS_DEV      5
#define SYS_DISK_FAIL_COUNT    2
#define SYSCALL_DEV_DISK_FAIL_COUNT \
SYS_DEV<<24 | SYS_DISK_FAIL_COUNT<<16
```

- `disk_name`: 要查询的磁盘名称被存放的地址；
- `name_len`: 要查询的磁盘名称的长度，不能超过 32；
- `fail_count`: 记录输出的失败次数存放地址。

返回值：0，成功，将失败次数记录在 `fail_count`；-1，失败，根据 `errno` 来查询实际错误。

说明：

- `ENOSYS`: 没有这个系统调用；
- `EINVAL`: 输入参数错误，磁盘名称长度不能超过 32；
- `EFAULT`: 用户地址空间错误，发生在内核与用户空间拷贝数据时；
- `ENXIO`: 查询的磁盘设备不存在。

## 3.5 统一系统调用查询接口

### 3.5.1 统一系统调用号查询接口 `nr_unify_syscall`

函数调用名称：

```
/proc/nr_unify_syscall
```

功能：供用户态查询当前系统的统一系统调用的系统调用号。

用法：`cat /proc/nr_unify_syscall`

```
# cat /proc/nr_unify_syscall
299
```

说明：输出统一系统调用的调用号，不同的内核版本统一系统调用号可能不同（或者其他读取文件的方法）。当用户查询到统一系统调用号后，使用 `syscall` 进入统一系统调用的子功能函数：

```
int syscall(nr_unify_syscall, SUBSYSCALL_CMD, arg, ...)
```

其中：

SUBSYSCALL\_CMD：统一系统调用子功能号；

arg：子功能需要的参数列表。

# 第 4 章

## 多核差异化接口

### 4.1 AMP 多核差异化主核启动参数接口

接口名称：无。

功能：控制多操作系统管理器的资源划分。

说明：

- mosm\_support: 为 1 表示支持配置了 mosm 模块，支持多核差异化运行；
- linux\_cpu\_mask, 检查每个 bit 是否为 1 确定由 Linux 管理的 cpu 核；
- linuxmemsz: 单位可为 G/g/M/m/K/k, Linux 操作系统可使用的物理内存总数；
- appshmemsz: 单位可为 G/g/M/m/K/k, 多个操作系统之间的应用共享内存区大小；
- mosm\_highest\_phyaddr: 共享内存的最高地址，在规划地址空间的时候，要计算共享内存的大小，为多核虚拟化各模块的总和，注意不要发生空间覆盖，包括启动相关大小（powerpc 下为 0x1000）加共享内存机制控制区 1k 字节加 NR\_CPUS\*CONFIG\_SHMDATA\_SIZE。

### 4.2 AMP 多核差异化 Loader 加载器接口

命令格式：

```
%s load -f file -m <cpu_mask> [options] [args...]
```

功能：通过用户态 Loader 程序加载从核。

参数:

- -f: 目标操作系统的 ELF 文件;
- -m: 目标操作系统被加载的 CPU id;
- options: 参数选项, 包括以下;
  - ✧ -z: 目标操作系统被加载的内存起始物理地址和占据的物理内存大小;
  - ✧ -T: 选择 console (virt\_uart/ttyS1);
  - ✧ --dual-core: ARM 架构使用 DualCore 模式启动;
  - ✧ -v: 输出详细的内存映射信息;
  - ✧ -h: 显示帮助说明。

### 4.3 AMP 多核差异化共享内存统一系统调用接口

接口名称:

```
enum {
SYS_KERN=1,
    SYS_VM,
    SYS_NET,
    SYS_FS,
    SYS_DEV,
    SYS_MOSM,
    /* adds anything new here */
    MAX_NR_SYSCALL = SYS_MOSM,
};
enum {
    /* adds anything new here */
    SYS_MOSM_SHMGET=1,
    SYS_MOSM_SHMCTL,
    SYS_MOSM_SHMDELETE,
    SYS_MOSM_SHMREAD,
    SYS_MOSM_SHMWRITE,
    MAX_NR_VM = SYS_MOSM_SHMWRITE,
};
#define SYS_ID(first, second) ((first << 24) | (second << 16))
```

```
#define SYSCALL_MOSM_SHMGET SYS_ID(SYS_MOSM, SYS_MOSM_SHMGET)
#define SYSCALL_MOSM_SHMCTL SYS_ID(SYS_MOSM, SYS_MOSM_SHMCTL)
#define SYSCALL_MOSM_SHMDELETE SYS_ID(SYS_MOSM, SYS_MOSM_SHMDELETE)
#define SYSCALL_MOSM_SHMREAD SYS_ID(SYS_MOSM, SYS_MOSM_SHMREAD)
#define SYSCALL_MOSM_SHMWRITE SYS_ID(SYS_MOSM, SYS_MOSM_SHMWRITE)
```

参数：number：统一系统调用的系统调用号。

### 4.3.1 P2020 单板创建共享内存接口

接口名称：

```
syscall(int number, SYSCALL_MOSM_SHMGET, int key, size_t size, int shmflg)
```

功能：通过统一系统调用接口创建共享内存，也可通过此接口查看创建是否成功。

参数：

- number：统一系统调用的系统调用号；
- Key：共享内存的唯一标识，必须为正整数；
- Size：申请共享内存的大小；
- Shmflg：代表查找还是创建，0 代表创建，1 代表查找。

返回值：成功返回传入的 key；失败返回错误码。

### 4.3.2 P2020 单板写入共享内存接口

接口名称：

```
syscall(int number, SYSCALL_MOSM_SHMWRITE, int key, const void __user* buf, size_t size, int flag)
```

功能：通过统一系统调用接口往共享内存中写数据。

参数：

- number：统一系统调用的系统调用号；
- Key：共享内存的唯一标识；
- Buf：用户缓冲区；
- Size：需要写入的数据字节数；
- Flag：暂时不用，传入 0。



返回值：成功返回写入的字节数，否则返回错误码。

### 4.3.3 P2020 单板读取共享内存接口

接口名称：

```
syscall(int number, SYSCALL_MOSM_SHMREAD, int key, const void __user* buf, size_t size, int flag)
```

功能：通过统一系统调用接口从共享内存中读数据。

参数：

- number：统一系统调用的系统调用号；
- Key：共享内存的唯一标识；
- Buf：用户缓冲区；
- Size：需要读出的数据字节数；
- Flag：阻塞时间，0 代表不阻塞。

返回值：成功返回读出的字节数，否则返回错误码。

### 4.3.4 P2020 单板共享内存控制接口

接口名称：

```
syscall(int number, SYSCALL_MOSM_SHMCTL, int key, int cpu, int flag)
```

功能：通过统一系统调用接口写完成后通知唤醒对方读进程。

参数：

- number：统一系统调用的系统调用号；
- Key：共享内存的唯一标识；
- cpu：对方 cpu 的 id，从 0 开始；
- Flag：暂时不用，传入 0。

返回值：成功返回 0，否则返回错误码。

### 4.3.5 P2020 单板删除共享内存接口

接口名称：

```
syscall(int number, SYSCALL_MOSM_SHMDELETE, int key)
```

功能：通过统一系统调用接口删除共享内存。删除之前必须保证读写进程已经退出，并且必须由创建者删除。

参数：

- **number**：统一系统调用的系统调用号；
- **Key**：共享内存的唯一标识；
- **Flag**：暂时不用，传入 0。

返回值：成功返回 0，否则返回错误码。

## 4.4 查看共享内存信息接口 shminfo

接口名称：

```
/proc/mosm/shminfo
```

功能：查看共享内存的物理，虚拟地址，以及共享内存的控制信息和数据。

用法：cat /proc/mosm/shminfo

## 4.5 设置或查询软中断占用率较低门限接口 sirqrate\_low

接口名称：

```
/proc/sys/kernel/sirqrate_low
```

功能：控制软中断负载均衡算法接口，用于设置或查询软中断占用率较低的门限，默认为 15，允许范围[5, 100]。

用法：

```
cat /proc/sys/kernel/sirqrate_low           //查询软中断占用率较低的门限。
echo 10 > /proc/sys/kernel/sirqrate_low      //设置软中断占用率较低的门限。
```

## 4.6 设置或查询软中断占用率较高门限接口 sirqrate\_high

接口名称：

```
/proc/sys/kernel/sirqrate_high
```

功能：控制软中断负载均衡算法接口，用于设置或查询软中断占用率较高的门限，默认为 75，允许范围[5, 100]，设置时请保证 `sirqrate_high > sirqrate_low`。

用法：

```
cat /proc/sys/kernel/sirqrate_high      //查询软中断占用率较高的门限。
echo 90 > /proc/sys/kernel/sirqrate_high //设置软中断占用率较高的门限。
```

## 4.7 算法开启与否的开关接口 `sirq_balance`

接口名称：

```
/proc/sys/kernel/sirq_balance
```

功能：控制软中断负载均衡算法开启与否的开关，非 0 表示开启，0 表示关闭，默认为开启。

用法：

```
cat /proc/sys/kernel/sirq_balance
echo 0 > /proc/sys/kernel/sirq_balance //关闭算法。
echo 1 > /proc/sys/kernel/sirq_balance //开启算法。
```

## 4.8 查询或设置软中断掩码接口 `sirq_mask`

接口名称：

```
/proc/sys/kernel/sirq_mask
```

功能：控制软中断负载均衡算法接口，用于查询或设置软中断掩码，默认为 12，即  $(1 \ll \text{NET\_TX\_SOFTIRQ}) \mid (1 \ll \text{NET\_XX\_SOFTIRQ})$ 。

用法：

```
cat /proc/sys/kernel/sirq_mask      //查询设置的软中断掩码。
echo 12 > /proc/sys/kernel/sirq_mask //12 即 0xC 表示激活 NET RX 和 TX 软中断均衡。
```

## 4.9 查询或设置非实时任务集合运行比例接口 `op_interval`

接口名称：

```
/proc/sys/kernel/part_sched/cpuN/op_interval
```

功能：设置某个核上，非实时任务集合每秒可运行的最大微秒数，即运行比例。

用法：

```
//可查看非实时任务集合每秒可运行的最大微秒数的设置情况。  
cat /proc/sys/kernel/part_sched/cpuN/op_interval  
  
//可设置非实时任务集合每秒可运行的最大微秒数。  
echo /proc/sys/kernel/part_sched/cpuN/op_interval
```

## 4.10 查询或设置非实时任务集合每次运行的最大微秒数接口 `run_interval`

接口名称：

```
/proc/sys/kernel/part_sched/cpuN/run_interval
```

功能：设置某个核上，非实时任务集合每次运行的最大微秒数（小于等于 `op_interval`），以保证系统的实时性。

用法：

```
//可查看非实时任务集合每次运行的最大微秒数的设置情况。  
cat /proc/sys/kernel/part_sched/cpuN/run_interval  
  
//可设置非实时任务集合每次运行的最大微秒数。  
echo /proc/sys/kernel/part_sched/cpuN/run_interval
```

## 4.11 设置某核上不受分区调度影响的实时任务优先级接口 `guard_prio`

接口名称：

```
/proc/sys/kernel/part_sched/cpuN/guard_prio
```

功能：设置某个核上，不受分区调度影响的实时任务优先级。高于此优先级的任务将不受分区调度影响。

用法：

```
//可查看不受分区调度影响的实时任务优先级。
cat /proc/sys/kernel/part_sched/cpuN/guard_prio

//可设置不受分区调度影响的实时任务优先级。
echo /proc/sys/kernel/part_sched/cpuN/guard_prio
```

## 4.12 查询和设置排他性\独占\共享 CPU 集合接口 exclusive\_cpus

接口名称：

```
/proc/exclusive_cpus
```

功能：proc 接口，供用户查询和设置排他性 CPU 集合，以及独占 CPU 集合、共享 CPU 集合。

用法：

- 不配置 MIGRATE\_KTHREAD（智能迁移功能）选项

以下方式可输出当前排他性 CPU 集合，或者其他读取文件的方法。

```
cat /proc/exclusive_cpus
```

以下方式可输入待设置的排它性 CPU 集合，X 代表输入值，或者其他写文件的方法。

```
echo X > /proc/exclusive_cpus
```

X 以 16 进制表示，无需在前面加 0x 前缀，例如：

设置排它性集合为 cpu2、cpu3，那么输入 `echo c > /proc/exclusive_cpus`；

设置排它性集合为 cpu2、cpu3、cpu5，那么输入 `echo 2c > /proc/exclusive_cpus`；

用户对线程的绑定优先于排它性 cpu 集合设置的反向绑定。

用户对线程的绑定会清除反向绑定标志 reverse binding flag。

要取消排它性 cpu 集合，执行 `echo 0 > /proc/exclusive_cpus`。

- 配置 MIGRATE\_KTHREAD（智能迁移功能）选项

以下方式可输出当前排他性 CPU 集合、独占 CPU 集合、共享 CPU 集合。（或者其他读取文件的方法）

```
cat /proc/exclusive_cpus
```

以下方式可输入待设置的排它性 CPU 集合、独占 CPU 集合、共享 CPU 集合。X Y Z 分别代表输入的 CPU 掩码，以 16 进制表示，中间使用空格隔开，无需在前面加 0x 前缀。掩码之间的约束规则为：独占式 CPU 集合不能超过排他性 CPU 集合的大小；共享式 CPU 集合不能超过非排他性 CPU 集合的大小。（或者其他写文件的方法）

```
echo X Y Z> /proc/exclusive_cpus
```

以下是合法的输入格式：

```
//输入一个参数默认为排他性 CPU 集合，而独占 CPU 集合默认等于排他 CPU 集合，共享 CPU  
集合默认等于非排他 CPU 集合。
```

```
echo X > /proc/exclusive_cpus
```

```
//三种 CPU 集合同时指定。
```

```
echo X Y Z> /proc/exclusive_cpus
```

以下是非法的输入格式：

```
echo X Y > /proc/exclusive_cpus //只指定两种 CPU 集合非法输入
```

```
echo X Y Z A> /proc/exclusive_cpus //指定超过三种 CPU 集合输入非法输入
```

## 4.13 查询或设置独占式\共享式 CPU 集合的迁移结果接口 migrate\_info

接口名称：

```
/proc/migrate_info
```

功能：proc 接口，供用户查询或设置独占式 CPU 集合和共享式 CPU 集合的迁移结果。

用法：查看上一次设置独占式 CPU 集合和共享式 CPU 集合的迁移结果。（或者其他读取文件的方法）

```
cat /proc/migrate_info
```

#### 4.14 排它核负载均衡配置接口 `exclusive_cpus_balance`

接口名称：

```
/proc/exclusive_cpus_balance
```

功能：通过配置将排它核上负载均衡开启或关闭。

用法：

输出当前排它核上负载均衡配置情况（或者其他读取文件的方法）：

```
cat /proc/exclusive_cpus_balance
```

输入待设置的值（或者其他写文件的方法）：

```
echo X > /proc/exclusive_cpus_balance
```

X 为 0 时，表示关闭排它核上负载均衡功能；X 为 1 时，表示开启排它核上负载均衡功能；X 为其他值时，为非法值，无效。

#### 4.15 控制实时工作队列的优先级接口 `rt_events_high_priority`

接口名称：

```
proc/sys/kernel/rt_events_high_priority
```

功能：用于控制实时工作队列的优先级，或查看、设置优先级。

用法：可读、可写。

```
cat /proc/sys/kernel/rt_events_high_priority //查看优先级。  
echo num > /proc/sys/kernel/rt_events_high_priority //设置优先级。
```

#### 4.16 查询或设置定时器绑定的 CPU 号接口 `bind_timer_cpu`

接口名称：

```
/proc/sys/kernel/bind_timer_cpu
```

功能：设置定时器绑定的 CPU 号以及查询当前绑定的 CPU 号

用法:

```
echo CPU_ID > /proc/sys/kernel/bind_timer_cpu    //设置希望将定时器绑定的 cpu 号。  
cat /proc/sys/kernel/bind_timer_cpu              //读取当前定时器绑定的 cpu 号。
```

## 4.17 查询和设置 CPU 掩码集合接口 softlockup\_cpumask

接口名称:

```
/proc/sys/kernel/softlockup_cpumask
```

功能: proc 接口, 供用户查询和设置 CPU 掩码集合, 可用于开启或关闭指定 CPU 上 softlockup 功能, 允许用户将某些 CPU 上的 softlockup 功能关闭, 以减小对该 CPU 的性能影响。集合中为 1 的项, 表示开启该 CPU 上的 softlockup 功能; 集合中为 0 的项, 表示关闭该 CPU 上的 softlockup 功能。(CGEL 内核选项已默认配置支持该功能)

用法:

查询当前 CPU 掩码集合:

```
cat /proc/sys/kernel/softlockup_cpumask
```

设置新的 CPU 掩码集合, 将会覆盖旧的 CPU 掩码集合:

```
echo X > /proc/sys/kernel/softlockup_cpumask
```

X 以 16 进制表示, 无需在前面加 0x 前缀, 下面以 8 个 CPU 的单板为例:

```
cat /proc/sys/kernel/softlockup_cpumask
```

若显示的值为 “ff”, 表示在所有 CPU 上都开启了 softlockup 功能。

```
echo fc > /proc/sys/kernel/softlockup_cpumask
```

上面命令表示关闭 cpu0 和 cpu1 上的 softlockup 功能。

## 4.18 中断绑定接口 smp\_affinity

接口名称: 无。

功能: 通过配置将中断绑定到指定的 CPU 上。



用法:

```
/proc/irq/IRQ#/smp_affinity
```

其中 `smp_affinity` 默认值为 `0xffffffff`, 为 CPU 绑定的位图, 例如 `0x00000003` 为将中断绑定到 CPU0 和 CPU1 上。

# 第 5 章

## 能耗接口

### 5.1 系统能耗状态控制接口 state

系统调用接口名称：

```
/sys/power/state
```

功能：控制系统能耗的状态。

用法：

```
cat /sys/power/state  
echo xxx > /sys/power/state
```

说明：可以读出三种支持的状态：standby、mem、disk。或以其他文件写入方式设置，写入这三种状态中的一种到 state 文件中，系统能耗的状态就会改变。

### 5.2 STD 操作模式控制接口 disk

系统调用接口名称：

```
/sys/power/disk
```

功能：控制系统能耗的状态。

用法：

```
cat /sys/power/disk
```

```
echo xxx > /sys/power/disk
```

说明：可以读出支持的操作模式：platform（如果系统支持）、shutdown、reboot、testproc、test。或以其他文件写入方式设置，在/sys/power/state 中写入字符串 disk 以后再根据 disk 文件的模式去操作。

- platform：使用平台相关的驱动去挂起系统，如 ACPI；
- shutdown：关闭系统；
- reboot：重启系统；
- testproc：关闭所有从 CPU，然后挂起所有任务，等待 5 秒以后再重启从 CPU，然后恢复任务；
- test：关闭所有从 CPU，挂起所有任务，然后挂起外设，等待 5 秒以后再重启从 CPU，恢复设备，然后恢复任务。

### 5.3 写磁盘内存映像大小控制接口 image\_size

系统调用接口名称：

```
/sys/power/image_size
```

功能：控制写到磁盘上的内存映像的大小。

用法：echo num > /sys/power/image\_size

说明：或以其他文件写入方式设置，写入非负数，如果写入 0 则由系统自动分配映像的大小。

### 5.4 磁盘设备号指定接口 resume

系统调用接口名称：

```
/sys/power/resume
```

功能：指定写入的磁盘设备号。

用法：echo xxx > /sys/power/resume

说明：或以其他文件写入方式设置，以 major: minor 的格式保存，指定设备的主设备号和次设备号。

## 5.5 wakeup 事件设备支持判定接口 wakeup

系统调用接口名称：

```
/sys/bus/XXX/devices/YYY/power/wakeup
```

功能：表示设备是否支持 wakeup 事件。

用法：cat /sys/bus/XXX/devices/YYY/power/wakeup

说明：XXX 是总线的名称，YYY 是设备号。表示设备是否支持 wakeup 事件，如果设备不支持则为换行符"\n"；如果事件被打开则为 "enabled"，如果事件被关闭则为 "disabled"。

## 5.6 USB 设备支持判定接口 level

系统调用接口名称：

```
/sys/bus/usb/devices/XXX/power/level
```

功能：表示 USB 设备支持的模式。

用法：cat /sys/bus/usb/devices/XXX/power/level

说明：支持三种模式 "on"，"auto" 和 "suspend"。on 是一直打开，auto 是动态管理，suspend 是强制挂起。

## 5.7 USB 设备最大空闲时间设置接口 autosuspend

系统调用接口名称：

```
/sys/bus/usb/devices/XXX/power/autosuspend
```

功能：设置设备的最大空闲时间。

用法：echo xxx > /sys/bus/usb/devices/XXX/power/autosuspend

说明：如果在这段时间内设备空闲则将设备挂起。单位是秒，如果写入 -1 就是强制挂起设备。

## 5.8 设备恢复后运行时间显示接口 active\_duration

系统调用接口名称：

```
/sys/bus/usb/devices/XXX/power/active_duration
```

功能：显示设备恢复以后的正常运行时间。

用法：cat /sys/bus/usb/devices/XXX/power/active\_duration

## 5.9 设备连接后运行时间显示接口 connected\_duration

系统调用接口名称：

```
/sys/bus/usb/devices/XXX/power/connected_duration
```

功能：显示设备连接上以后的时间。

用法：cat /sys/bus/usb/devices/XXX/power/connected\_duration

## 5.10 CPU 当前运行频率显示接口 cpuinfo\_cur\_freq

系统调用接口名称：

```
/sys/devices/system/cpu/cpuX/cpufreq/cpuinfo_cur_freq
```

功能：显示 cpu 运行的当前频率。

用法：cat /sys/devices/system/cpu/cpuX/cpufreq/cpuinfo\_cur\_freq

说明：显示 cpu 运行的当前频率（kHz）。

## 5.11 CPU 最低运行频率显示接口 cpuinfo\_min\_freq

系统调用接口名称：

```
/sys/devices/system/cpu/cpuX/cpufreq/cpuinfo_min_freq
```

功能：显示 cpu 运行的最低频率。

用法：cat /sys/devices/system/cpu/cpuX/cpufreq/cpuinfo\_min\_freq

说明：显示 cpu 运行的最低频率（kHz）。

## 5.12 CPU 最高运行频率显示接口 `cpuinfo_max_freq`

系统调用接口名称：

```
/sys/devices/system/cpu/cpuX/cpufreq/cpuinfo_max_freq
```

功能：显示 cpu 运行的最高频率。

用法：cat /sys/devices/system/cpu/cpuX/cpufreq/cpuinfo\_max\_freq

说明：显示 cpu 运行的最高频率（kHz）。

## 5.13 CPU 频率切换时间延迟显示接口 `cpuinfo_transition_latency`

系统调用接口名称：

```
/sys/devices/system/cpu/cpuX/cpufreq/cpuinfo_transition_latency
```

功能：显示 cpu 频率切换的时间延迟。

用法：cat /sys/devices/system/cpu/cpuX/cpufreq/cpuinfo\_transition\_latency

说明：显示 cpu 频率切换的时间延迟（纳秒）。

## 5.14 CPU 可支持频率列表显示接口 `scaling_available_frequencies`

系统调用接口名称：

```
/sys/devices/system/cpu/cpuX/cpufreq/scaling_available_frequencies
```

功能：显示 cpu 可支持的频率列表。

用法：cat /sys/devices/system/cpu/cpuX/cpufreq/scaling\_available\_frequencies

说明：显示 cpu 可支持的频率列表（kHz）。

## 5.15 CPU 可支持策略列表显示接口 `scaling_available_governors`

系统调用接口名称：

```
/sys/devices/system/cpu/cpuX/cpufreq/scaling_available_governors
```

功能：显示 cpu 可支持的策略列表。

用法：cat /sys/devices/system/cpu/cpuX/cpufreq/scaling\_available\_governors

说明：显示 cpu 可支持的频率列表。

## 5.16 CPU 当前策略运行频率显示接口 scaling\_cur\_freq

系统调用接口名称：

```
/sys/devices/system/cpu/cpuX/cpufreq/scaling_cur_freq
```

功能：显示 cpu 当前策略所决定的运行频率。

用法：cat /sys/devices/system/cpu/cpuX/cpufreq/scaling\_cur\_freq

说明：显示 cpu 当前策略所决定的运行频率（kHz）。

## 5.17 CPU 当前硬件驱动显示接口 scaling\_driver

系统调用接口名称：

```
/sys/devices/system/cpu/cpuX/cpufreq/scaling_driver
```

功能：显示 cpu 当前硬件驱动。

用法：cat /sys/devices/system/cpu/cpuX/cpufreq/scaling\_driver

## 5.18 CPU 当前频率策略显示接口 scaling\_governor

系统调用接口名称：

```
sys/devices/system/cpu/cpuX/cpufreq/scaling_governor
```

功能：显示 cpu 当前频率策略。

用法：cat /sys/devices/system/cpu/cpuX/cpufreq/scaling\_governor

## 5.19 CPU 当前策略频率最大显示接口 scaling\_max\_freq

系统调用接口名称：

```
/sys/devices/system/cpu/cpuX/cpufreq/scaling_max_freq
```

功能：显示 cpu 当前策略所支持的最大频率。

用法：cat /sys/devices/system/cpu/cpuX/cpufreq/scaling\_max\_freq

说明：显示 cpu 当前策略所支持的最大频率(kHz)，可以重新写入来改变。

## 5.20 CPU 当前策略频率最小显示接口 scaling\_min\_freq

系统调用接口名称：

```
/sys/devices/system/cpu/cpuX/cpufreq/scaling_min_freq
```

功能：显示 cpu 当前策略所支持的最小频率。

用法：cat /sys/devices/system/cpu/cpuX/cpufreq/scaling\_max\_freq

说明：显示 cpu 当前策略所支持的最小频率(kHz)，可以重新写入来改变。

## 5.21 CPU 当前速度设置接口 scaling\_setspeed

系统调用接口名称：

```
/sys/devices/system/cpu/cpuX/cpufreq/scaling_setspeed
```

功能：设置 cpu 当前速度。

用法：echo xxx > /sys/devices/system/cpu/cpuX/cpufreq/scaling\_setspeed

说明：或以其他方式写入这个文件来改变 cpu 的运行速度，但必须在 scaling\_min\_freq 和 scaling\_max\_freq 之间。

## 5.22 进程 nice 值忽略控制接口 ignore\_nice\_load

系统调用接口名称：

```
/sys/devices/system/cpu/cpuX/cpufreq/ondemand/ignore_nice_load
```

功能：控制进程 nice 值的忽略。

用法：echo xxx > /sys/devices/system/cpu/cpuX/cpufreq/ondemand/ignore\_nice\_load



说明：或以其他方式写入，可以被设置为 0 或 1，如果被设置为 1 则不计算进程的 nice 值，如果被设置为 0 则计算进程的 nice 值。进程的 nice 值在计算当前 cpu 的性能时用到。

### 5.23 设置 **ondemand** 降低 CPU 频率接口 **ondemand/powersave\_bias**

系统调用接口名称：

```
/sys/devices/system/cpu/cpuX/cpufreq/ondemand/powersave_bias
```

功能：设置 **ondemand** 的 CPU 占用率的阈值从而降低 CPU 频率。

用法：echo xxx > /sys/devices/system/cpu/cpuX/cpufreq/ondemand/powersave\_bias

说明：或以其他方式写入，设置 **ondemand** 策略中 CPU 占用率的阈值，当系统侧 CPU 占用率小于设置的阈值时内核就自动降低 CPU 频率，阈值可以设为 1 到 1000 的值。

### 5.24 性能采用时间间隔设置接口 **sampling\_rate**

系统调用接口名称：

```
/sys/devices/system/cpu/cpuX/cpufreq/ondemand/sampling_rate
```

功能：设置性能采用的时间间隔。

用法：echo xxx > /sys/devices/system/cpu/cpuX/cpufreq/ondemand/sampling\_rate

说明：或以其他方式写入，设置性能采用的时间间隔，范围在 **sampling\_rate\_min** 和 **sampling\_rate\_max** 之间。

### 5.25 性能采用最小时间间隔显示接口 **sampling\_rate\_min**

系统调用接口名称：

```
/sys/devices/system/cpu/cpuX/cpufreq/ondemand/sampling_rate_min
```

功能：显示性能采用的最小时间间隔。

用法：cat /sys/devices/system/cpu/cpuX/cpufreq/ondemand/sampling\_rate\_min

## 5.26 性能采用最大时间间隔显示接口 `sampling_rate_max`

系统调用接口名称：

```
/sys/devices/system/cpu/cpuX/cpufreq/ondemand/sampling_rate_max
```

功能：显示性能采用的最大时间间隔。

用法：cat /sys/devices/system/cpu/cpuX/cpufreq/ondemand/sampling\_rate\_min

## 5.27 CPU 频率提高占用率设置接口 `up_threshold`

系统调用接口名称：

```
/sys/devices/system/cpu/cpuX/cpufreq/ondemand/up_threshold
```

功能：设置 cpu 频率提高的占用率。

用法：echo xxx > /sys/devices/system/cpu/cpuX/cpufreq/ondemand/up\_threshold

说明：设置 ondemand 进行频率调整的 cpu 占用率，当大于该占用率是提高 cpu 的频率，缺省为 80%。

## 5.28 CPU 的 idle 状态调整使用驱动显示接口 `current_driver`

系统调用接口名称：

```
/sys/devices/system/cpu/cpuidle/current_driver
```

功能：显示 cpu 的 idle 状态调整使用的驱动。

用法：cat /sys/devices/system/cpu/cpuidle/current\_driver

## 5.29 CPU 的 idle 状态调整策略显示接口 `current_governor_ro`

系统调用接口名称：

```
/sys/devices/system/cpu/cpuidle/current_governor_ro
```

功能：显示 cpu 的 idle 状态调整使用的策略。

用法: `cat /sys/devices/system/cpu/cpuidle/current_governor_ro`

### 5.30 CPU 的 idle 状态简单描述显示接口 desc

系统调用接口名称:

```
/sys/devices/system/cpu/cpuX/cpuidle/stateY/desc
```

功能: 显示 cpu 的 idle 状态的简单描述。

用法: `cat /sys/devices/system/cpu/cpuX/cpuidle/stateY/desc`

### 5.31 CPU 的 idle 状态调整时间延迟显示接口 latency

系统调用接口名称:

```
/sys/devices/system/cpu/cpuX/cpuidle/stateY/latency
```

功能: 显示 cpu 的 idle 状态调整的时间延迟。

用法: `cat /sys/devices/system/cpu/cpuX/cpuidle/stateY/latency`

说明: 显示 cpu 的 idle 状态调整的时间延迟（微秒）。

### 5.32 CPU 的 idle 状态下名称显示接口 name

系统调用接口名称:

```
/sys/devices/system/cpu/cpuX/cpuidle/stateY/name
```

功能: 显示 cpu 在这种 idle 状态下的名称。

用法: `cat /sys/devices/system/cpu/cpuX/cpuidle/stateY/name`

### 5.33 CPU 的 idle 状态下能量消耗显示接口 power

系统调用接口名称:

```
/sys/devices/system/cpu/cpuX/cpuidle/stateY/power
```

功能：显示 cpu 在这种 idle 状态下的能量消耗。

用法：cat /sys/devices/system/cpu/cpuX/cpuidle/stateY/power

### 5.34 CPU 的 idle 状态下持续时间显示接口 time

系统调用接口名称：

```
/sys/devices/system/cpu/cpuX/cpuidle/stateY/time
```

功能：显示 cpu 在这种 idle 状态下的持续时间。

用法：cat /sys/devices/system/cpu/cpuX/cpuidle/stateY/time

说明：显示 cpu 在这种 idle 状态下的持续时间(微秒)。

### 5.35 CPU 的 idle 状态下进入次数显示接口 usage

系统调用接口名称：

```
/sys/devices/system/cpu/cpuX/cpuidle/stateY/usage
```

功能：显示 cpu 在这种 idle 状态下的进入次数。

用法：cat /sys/devices/system/cpu/cpuX/cpuidle/stateY/usage

# 第 6 章

## 调测接口

### 6.1 kprobe 动态插点接口

#### 6.1.1 注册动态插点接口 `register_kprobe`

接口类型：内核态接口。

函数调用名称：

```
int __kprobes register_kprobe(struct kprobe *p)
```

功能：注册动态插点接口。

输入参数：

➤ `p`: `kprobe` 结构体指针。

返回值：对应内核错误码。

#### 6.1.2 注销动态插点接口 `unregister_kprobe`

接口类型：内核态接口。

函数调用名称：

```
void __kprobes unregister_kprobe(struct kprobe *p)
```

功能：注销动态插点接口。

输入参数：

- p: kprobe 结构体指针。

返回值：无。

### 6.1.3 批量注册动态插点接口 register\_kprobes

接口类型：内核态接口。

函数调用名称：

```
int __kprobes register_kprobes(struct kprobe **kps, int num)
```

功能：批量注册动态插点接口。

输入参数：

- kps: kprobe 结构体指针数组指针。
- num: kprobe 指针的个数。

返回值：对应内核错误码。

### 6.1.4 批量注销动态插点接口 unregister\_kprobes

接口类型：内核态接口。

函数调用名称：

```
int __kprobes register_kprobes(struct kprobe **kps, int num)
```

功能：批量注销动态插点接口。

输入参数：

- kps: kprobe 结构体指针数组指针；
- num: kprobe 指针的个数。

返回值：无。

## 6.2 jprobe 动态插点接口

### 6.2.1 注册跳转动态插点接口 register\_jprobe

接口类型：内核态接口。

函数调用名称：

```
int __kprobes register_jprobe(struct jprobe *jp)
```

功能：注册跳转动态插点接口。

输入参数：

➤ p: jprobe 结构体指针。

返回值：对应内核错误码。

### 6.2.2 注销跳转动态插点接口 unregister\_jprobe

接口类型：内核态接口。

函数调用名称：

```
void __kprobes unregister_jprobe(struct jprobe *p)
```

功能：注销跳转动态插点接口。

输入参数：

➤ p: jprobe 结构体指针。

返回值：无。

### 6.2.3 批量注册跳转动态插点接口 register\_jprobes

接口类型：内核态接口。

函数调用名称：

```
int __kprobes register_jprobes(struct jprobe **jps, int num)
```

功能：批量注册跳转动态插点接口。

输入参数：

➤ jps: jprobe 结构体指针数组指针；

➤ num: jprobe 指针的个数。

返回值：对应内核错误码。

### 6.2.4 批量注销跳转动态插点接口 unregister\_jprobes

接口类型：内核态接口。

函数调用名称：

```
void __kprobes unregister_jprobes(struct jprobe **jps, int num)
```

功能：批量注销跳转动态插点接口。

输入参数：

➤ jps: jprobe 结构体指针数组指针。

返回值：无。

## 6.3 kretprobe 动态插点接口

### 6.3.1 函数返回点动态检查点注册接口 register\_kretprobe

接口类型：内核态接口。

函数调用名称：

```
int register_kretprobe(struct kretprobe *p)
```

功能：用来向任一函数返回点注册插入一个动态检查点。

输入参数：该参数包含了函数返回点地址和调测动作的所有信息。

返回值：0，成功；负数，错误原因。

### 6.3.2 函数返回点动态检查点注销卸载接口 unregister\_kretprobe

接口类型：内核态接口。

函数调用名称：

```
int unregister_kretprobe(struct kretprobe *p)
```

功能：用来注销卸载的一个函数返回点的动态检查点。

输入参数：

➤ p: 该参数指向了待注销卸载的函数返回点动态检查点。

返回值：0，成功；负数，错误原因。

### 6.3.3 组动态检查点多函数返回点注册接口 register\_kretprobes

接口类型：内核态接口。



函数调用名称：

```
int register_kretprobes(struct kretprobe **rps, int num)
```

功能：用来注册插入多个函数返回点的一组动态检查点。

输入参数：

- **jps**：该参数指向一组待插入函数返回点动态检查点；
- **num**：动态检查点的个数。

返回值：0，成功；负数，错误原因。

### 6.3.4 组动态检查点函数返回点注销卸载接口 `unregister_kretprobes`

接口类型：内核态接口。

函数调用名称：

```
int unregister_kretprobes(struct kretprobe **rps, int num)
```

功能：用来注销卸载的一组函数返回点的动态检查点。

输入参数：

- **jps**：该参数指向一组待注销卸载的函数返回点的动态检查点；
- **num**：动态检查点的个数。

返回值：0，成功；负数，错误原因。

## 6.4 静态检查点调测动作接口

### 6.4.1 静态检查点创建接口 `TRACE_EVENT`

接口类型：内核态接口。

函数调用名称：

```
TRACE_EVENT (name, proto, args)
```

功能：用来定义一个静态检查点的宏模板，通过该模板，可以扩展出静态检查点调测动作钩子挂接函数，调测动作钩子卸载函数和调测动作钩子调用管理函数。

输入参数：

- **name**：静态检查点名字；

- proto: 输入钩子的参数类型声明;
- args: 输入钩子的参数声明。

返回值: 无。

#### 6.4.2 静态检查点调测钩子注册接口 `register_static_trace`

接口类型: 内核态接口。

函数调用名称:

```
int register_static_trace (name, void (*probe)(void))
```

功能: 用来向指定名字的静态检查点注册一个调测动作钩子。

输入参数:

- name: 静态检查点名字;
- probe: 调测动作函数钩子地址;

返回值: 0, 成功; 负数, 错误原因。

#### 6.4.3 静态检查点调测钩子注销接口 `unregister_static_trace`

接口类型: 内核态接口。

函数调用名称:

```
int unregister_static_trace (name, void (*probe)(void))
```

功能: 用来从指定名字的静态检查点注销一个调测动作钩子。

输入参数:

- name: 静态检查点名字;
- probe: 调测动作函数钩子地址。

返回值: 0, 成功; 负数, 错误原因。

### 6.5 内核预设调测功能控制接口

#### 6.5.1 内核预设调测可用显示接口 `available_tracers`

系统调用接口名称:

```
/sys/kernel/debug/tracing/available_tracers
```

功能：可用的内核预设调测功能显示。

用法：cat /sys/kernel/debug/tracing/available\_tracers

说明：输出可用的内核预设调测功能。

### 6.5.2 内核预设调测功能当前选择接口 current\_tracer

系统调用接口名称：

```
/sys/kernel/debug/tracing/current_tracer
```

功能：当前的内核预设调测功能选择情况。

用法：

```
cat /sys/kernel/debug/tracing/current_tracer  
echo xxx > /sys/kernel/debug/tracing/current_tracer
```

说明：查看和设置当前选择的内核调测功能。

### 6.5.3 当前跟踪选项接口 trace\_options

系统调用接口名称：

```
/sys/kernel/debug/tracing/trace_options
```

功能：当前的跟踪选项显示和设置。

用法：

```
cat /sys/kernel/debug/tracing/trace_options  
echo xxx > /sys/kernel/debug/tracing/trace_options
```

说明：查看和设置当前的跟踪选项。

### 6.5.4 当前内核预设调测功能启动情况接口 tracing\_enabled

系统调用接口名称：

```
/sys/kernel/debug/tracing/tracing_enabled
```

功能：当前选择的内核预设调测功能的启动情况。

用法:

```
cat /sys/kernel/debug/tracing/trace_options  
echo xxx > /sys/kernel/debug/tracing/trace_options
```

说明: 查看当前选择的内核预设调测功能的启动情况; 设置内核预设调测功能的启动和关闭。

## 6.6 无锁环形管道输出接口

### 6.6.1 ring\_buffer 分配接口 ring\_buffer\_alloc

接口类型: 用户态接口。

函数调用名称:

```
struct ring_buffer * ring_buffer_alloc(size, flags)
```

功能: 分配 ring\_buffer。

输入参数:

- size: 环形缓存的大小;
- Flags: 环形缓存的标志。

返回值: NULL, 失败; 指针, 环形缓存指针

### 6.6.2 ring\_buffer 释放接口 ring\_buffer\_free

接口类型: 用户态接口。

函数调用名称:

```
void ring_buffer_free(struct ring_buffer *buffer)
```

功能: 释放 ring\_buffer。

输入参数:

- buffer: 环形缓存指针

返回值: 无

### 6.6.3 ring\_buffer 数据块写入接口 ring\_buffer\_write

接口类型: 用户态接口。

函数调用名称:

```
int ring_buffer_write(struct ring_buffer *buffer, unsigned long length, void *data)
```

功能: 写入数据块到 ring\_buffer。

输入参数:

- buffer: 环形缓存指针;
- Length: 写入数据长度。

返回值: 0, 成功; 负数, 错误原因。

#### 6.6.4 ring\_buffer 无阻塞读出接口 trace

系统调用接口名称:

```
/sys/kernel/debug/tracing/trace
```

功能: 无阻塞读出 ring\_buffer 的内容。

用法: cat /sys/kernel/debug/tracing/trace

说明: 查看内核预设调测功能的跟踪结果; 有多少, 读多少, 不等待。

#### 6.6.5 ring\_buffer 阻塞读出接口 trace\_pipe

系统调用接口名称:

```
sys/kernel/debug/tracing/trace_pipe
```

功能: 阻塞式读出 ring\_buffer 的内容。

用法: cat /sys/kernel/debug/tracing/ trace\_pipe

说明: 查看内核预设调测功能的跟踪结果; 读完当前以后, 就阻塞式等待下一次的调测结果输出。

### 6.7 CPU 信息获取接口

#### 6.7.1 CPU 寄存器信息获取接口 trace\_get\_regs

接口类型: 内核态接口。

函数调用名称:

```
struct pt_regs* trace_get_regs(struct pt_regs *regs, int cpu)
```

功能：用于常规调测的库函数，获取某个 CPU 寄存信息。

输入参数：

- cpu : cpu 编号；
- buffer: 接收寄存器的结构缓存指针；

返回值：寄存器缓存指针（同入参 regs 地址），成功；NULL，失败。

### 6.7.2 CPU 负载均衡统计信息查看接口 schedstat

Proc 接口名称：

```
/proc/schedstat
```

功能：查看各 cpu 负载均衡统计信息，即为所有相关的 CPU 显示特定于运行队列的统计信息以及 SMP 系统中特定于域的统计信息。

用法：cat /proc/schedstat

说明：需要内核的配置宏 CONFIG\_SCHEDSTATS。

### 6.7.3 CPU 调度域参数查看接口 domain<m>

Proc 接口名称：

```
/proc/sys/kernel/sched_domain/cpu<n>/domain<m>/
```

功能：查看各 cpu 调度域参数。

用法：

```
cat /proc/sys/kernel/sched_domain/cpu0/domain0/flags
echo xxx > /proc/sys/kernel/sched_domain/cpu0/domain0/flags
```

说明：

1) cpu<n>表示第几个逻辑 cpu，如 cpu0；domain<m>表示 cpu<n>的第几个调度域，如 domain0；

2) /proc/sys/kernel/sched\_domain/cpu<n>/domain<m>/是目录，其中包含多个文件，每一个文件对应于一个调度域参数；

- span 字段表示调度域包含的 CPU 集合，在调度组在内核初始化后就固定了，不允许修改；

- `last_balance` 记录了上次做定时均衡的时间点，单位为 `tick`，只在定时均衡中更新，和 `balance_interval` 一起用于计算下次做定时均衡的时间；
- `balance_interval` 也只在定时均衡中更新，单位为毫秒，用于计算下次做定时均衡的时间：  
`last_balance+msecs_to_jiffies(balance_interval)`，最终根据情况将结果更新记录于运行队列 `rq->next_balance` 中；定时均衡成功迁移任务后，意味着此时系统的负载倾向于不均衡，将 `balance_interval` 置为 `min_interval`；定时均衡没有发生任务迁移，意味着此时系统的负载倾向于均衡，将 `balance_interval` 加倍，但不超过 `max_interval`（如果当前 CPU 的任务都已绑定，则不超过 `MAX_PINNED_INTERVAL`，定义为 512ms）；
- `busy_factor` 只用于定时均衡，如果当前 CPU 不是空闲的（定时忙平衡），就将变量 `interval` 间隔乘上此字段，这样在当前 CPU 忙时不会太频繁进入负载均衡算法，避免负载均衡算法本身带来较大的开销；
- `imbalance_pct` 用作均衡判别，见 1.5 节；此值增大可放宽负载失衡的判别条件，从而降低负载均衡的频率；
- `cache_nice_tries`, `nr_balance_failed` 用于定时均衡中连续迁移任务失败后，发起推任务操作，判断条件为 `nr_balance_failed > (cache_nice_tries+2)`；
- `busy_idx`, `idle_idx`, `newidle_idx`, `wake_idx`, `forkexec_idx` 用作下标选择运行队列中的历史权重 `cpu_load[]` 数组中的元素；
- `flags` 用于配置不同的均衡算法以及一些特殊的标志，其值含义如下：

```
#define SD_LOAD_BALANCE      1 /* Do load balancing on this domain. */
#define SD_BALANCE_NEWIDLE  2 /* Balance when about to become idle */
#define SD_BALANCE_EXEC      4 /* Balance on exec */
#define SD_BALANCE_FORK      8 /* Balance on fork, clone */
#define SD_WAKE_IDLE        16 /* Wake to idle CPU on task wakeup */
#define SD_WAKE_AFFINE       32 /* Wake task to waking CPU */
#define SD_WAKE_BALANCE      64 /* Perform balancing at task wakeup */
#define SD_SHARE_CPUPOWER    128 /* Domain members share cpu power */
#define SD_POWERSAVINGS_BALANCE 256 /* Balance for power savings */
#define SD_SHARE_PKG_RESOURCES 512 /* Domain members share cpu pkg resources */
#define SD_SERIALIZE         1024 /* Only a single load balancing instance */

#define BALANCE_FOR_MC_POWER \
    (sched_smt_power_savings ? SD_POWERSAVINGS_BALANCE : 0)

#define BALANCE_FOR_PKG_POWER \
    (((sched_mc_power_savings || sched_smt_power_savings) ? \
    SD_POWERSAVINGS_BALANCE : 0)
```

- ✧ `SD_LOAD_BALANCE` 表示做负载均衡；做负载均衡就会做定时均衡，定时均衡不能单独关闭；
- ✧ `SD_BALANCE_NEWIDLE` 表示做闲时均衡；
- ✧ `SD_BALANCE_EXEC` 表示做创建时均衡中的 `execve` 均衡；
- ✧ `SD_BALANCE_FORK` 表示做创建时均衡中的 `fork` 均衡；

- ✧ SD\_WAKE\_IDLE 表示开启超线程后在 try\_to\_wake\_up()中将被唤醒任务迁移至一个空闲 CPU 的上运行(如果有的话);
- ✧ SD\_WAKE\_AFFINE 表示在 try\_to\_wake\_up()中考虑任务的 cache 亲和, 如果存在 cache 亲和就不将此任务拉至当前 CPU;
- ✧ SD\_WAKE\_BALANCE 功能类似于 SD\_WAKE\_AFFINE, 只是算法判别准则不同;
- ✧ SD\_SHARE\_CPUPOWER 表示共享 CPU 的处理能力, 主要用于同一个核的超线程 CPU 间; 从而让操作系统感知到超线程硬件特性予以特殊处理;
- ✧ SD\_SHARE\_PKG\_RESOURCES 用于初始化调度组的 CPU 能力;
- ✧ SD\_SERIALIZE 主要用于不同 CPU 间的负载均衡算法的串行化(增加了一把自旋锁), 一般 NUMA 域可设置此选项;
- ✧ BALANCE\_FOR\_MC\_POWER 和 BALANCE\_FOR\_PKG\_POWER 用于节能, 负载均衡可以和 CPU 的变频技术以及 S5 节能技术结合起来, 用以将空闲的 CPU 降频或休眠, 具体可参考 find\_busiest\_group()函数的实现, 基本思想就是开启节能选项后(选项开关可通过 sched\_mc\_power\_savings 和 sched\_smt\_power\_savings 控制选中 SD\_POWERSAVINGS\_BALANCE)将负载轻的 CPU 的任务迁移至负载重的 CPU, 直至系统中产生空闲的 CPU。

## 6.8 内核 SLAB 错误注入控制接口

### 6.8.1 错误概率参数设置接口 probability

系统调用接口名称:

```
/sys/kernel/debug/failslab/probability
```

功能: 错误概率参数设置。

用法:

```
cat /sys/kernel/debug/failslab/probability  
echo xxx > /sys/kernel/debug/failslab/probability
```

说明: 查看或设置 failslab 的错误概率参数。

### 6.8.2 两次错误之间成功次数设置接口 interval

系统调用接口名称:



```
/sys/kernel/debug/failslab/interval
```

功能：用于设置两次错误之间的成功次数。

用法：

```
cat /sys/kernel/debug/failslab/interval  
echo xxx > /sys/kernel/debug/failslab/interval
```

说明：查看或设置 failslab 的两次错误之间的成功次数。

### 6.8.3 允许错误注入最大次数设置接口 times

系统调用接口名称：

```
/sys/kernel/debug/failslab/times
```

功能：用于设置 failslab 的允许错误注入的最大次数。

用法：

```
cat /sys/kernel/debug/failslab/times  
echo xxx > /sys/kernel/debug/failslab/times
```

说明：查看或设置 failslab 的允许错误注入的最大次数。

### 6.8.4 第一次错误植入前间隔次数设置接口 space

系统调用接口名称：

```
/sys/kernel/debug/failslab/space
```

功能：用于设置 failslab 的第一次错误植入之前的间隔次数。

用法：

```
cat /sys/kernel/debug/failslab/space  
echo xxx > /sys/kernel/debug/failslab/space
```

说明：查看或设置 failslab 的第一次错误植入之前的间隔次数。

### 6.8.5 failslab 消息输出内容设置接口 verbose

系统调用接口名称：

```
/sys/kernel/debug/failslab/verbose
```

功能：用于设置 failslab 的消息输出内容设置。

用法：

```
cat /sys/kernel/debug/failslab/verbose  
echo xxx > /sys/kernel/debug/failslab/verbose
```

说明：设置值 0：无消息输出；1：每注入一次错误，输出一行消息；2：带堆栈回溯消息。

### 6.8.6 failslab 任务过滤器开关设置接口 task-filter

系统调用接口名称：

```
/sys/kernel/debug/failslab/task-filter
```

功能：用于设置 failslab 的是否打开任务过滤器，该接口和/proc/pid/make-it-fail 配合使用。

用法：

```
cat /sys/kernel/debug/failslab/task-filter  
echo xxx > /sys/kernel/debug/failslab/task-filter
```

说明：failslab 的任务过滤器开关：Y-打开，N-关闭。

### 6.8.7 failslab 堆栈回溯有效区域起始地址设置接口 require-start

系统调用接口名称：

```
/sys/kernel/debug/failslab/require-start
```

功能：failslab 中堆栈回溯的有效区域设定，有效区域的起始虚拟地址设定。

用法：

```
cat /sys/kernel/debug/failslab/require-start  
echo xxx > /sys/kernel/debug/failslab/require-start
```

说明：查看或设置 failslab 的有效区域的起始虚拟地址。

### 6.8.8 failslab 堆栈回溯有效区域结束地址设置接口 require-end

系统调用接口名称：

```
/sys/kernel/debug/failslab/require-end
```

功能：failslab 中堆栈回溯的有效区域设定，有效区域的结束虚拟地址设定。

用法：

```
cat /sys/kernel/debug/failslab/require-end  
echo xxx > /sys/kernel/debug/failslab/require-end
```

说明：查看或设置 failslab 的有效区域的结束虚拟地址。

### 6.8.9 failslab 堆栈回溯无效区域起始地址设置接口 reject-start

系统调用接口名称：

```
/sys/kernel/debug/failslab/reject-start
```

功能：failslab 中堆栈回溯的有效区域设定，有效区域的起始虚拟地址设定。

用法：

```
cat /sys/kernel/debug/failslab/reject-start  
echo xxx > /sys/kernel/debug/failslab/reject-start
```

说明：查看或设置 failslab 的无效区域的起始虚拟地址。

### 6.8.10 failslab 堆栈回溯无效区域结束地址设置接口 reject-end

系统调用接口名称：

```
/sys/kernel/debug/failslab/reject-end
```

功能：failslab 中堆栈回溯的有效区域设定，有效区域的结束虚拟地址设定。

用法：

```
cat /sys/kernel/debug/failslab/reject-start  
echo xxx > /sys/kernel/debug/failslab/reject-start
```

说明：查看或设置 failslab 的无效区域的结束虚拟地址。

### 6.8.11 failslab 堆栈回溯深度设置接口 stacktrace-depth

系统调用接口名称：

```
/sys/kernel/debug/failslab/stacktrace-depth
```

功能：failslab 中堆栈回溯的深度设置。

用法：

```
cat /sys/kernel/debug/failslab/stacktrace-depth  
echo xxx > /sys/kernel/debug/failslab/stacktrace-depth
```

说明：查看或设置 failslab 的堆栈回溯深度。

## 6.8.12 failslab 等待页面分配错误注入忽略设置接口 ignore-gfp-wait

系统调用接口名称：

```
/sys/kernel/debug/failslab/ignore-gfp-wait
```

功能：failslab 中对有等待的页面分配的错误注入是否需要忽略。

用法：

```
cat /sys/kernel/debug/failslab/ignore-gfp-wait  
echo xxx > /sys/kernel/debug/failslab/ignore-gfp-wait
```

说明：查看或设置对有等待的页面分配的错误注入是否需要忽略，Y 忽略，N 不忽略。

## 6.9 内核 PAGE 分配错误注入控制接口

### 6.9.1 错误概率参数设置接口 probability

系统调用接口名称：

```
/sys/kernel/debug/fail_page_alloc/probability
```

功能：错误概率参数设置。

用法：

```
cat /sys/kernel/debug/fail_page_alloc/probability  
echo xxx > /sys/kernel/debug/fail_page_alloc/probability
```

说明：查看或设置 fail\_page\_alloc 的错误概率参数。

### 6.9.2 两次错误之间成功次数设置接口 interval

系统调用接口名称：

```
/sys/kernel/debug/fail_page_alloc/interval
```

功能：用于设置两次错误之间的成功次数。

用法：

```
cat /sys/kernel/debug/fail_page_alloc/interval  
echo xxx > /sys/kernel/debug/fail_page_alloc/interval
```

说明：查看或设置 fail\_page\_alloc 的两次错误之间的成功次数。

### 6.9.3 允许错误注入最大次数设置接口 times

系统调用接口名称：

```
/sys/kernel/debug/fail_page_alloc/times
```

功能：用于设置 fail\_page\_alloc 的允许错误注入的最大次数。

用法：

```
cat /sys/kernel/debug/fail_page_alloc/times  
echo xxx > /sys/kernel/debug/fail_page_alloc/times
```

说明：查看或设置 fail\_page\_alloc 的允许错误注入的最大次数。

### 6.9.4 第一次错误植入前间隔次数设置接口 space

系统调用接口名称：

```
/sys/kernel/debug/fail_page_alloc/space
```

功能：用于设置 fail\_page\_alloc 的第一次错误植入之前的间隔次数。

用法：

```
cat /sys/kernel/debug/fail_page_alloc/space  
echo xxx > /sys/kernel/debug/fail_page_alloc/space
```

说明：查看或设置 fail\_page\_alloc 的第一次错误植入之前的间隔次数。

### 6.9.5 fail\_page\_alloc 消息输出内容设置接口 verbose

系统调用接口名称：

```
/sys/kernel/debug/fail_page_alloc/verbose
```

功能：用于设置 fail\_page\_alloc 的消息输出内容设置。

用法：

```
cat /sys/kernel/debug/fail_page_alloc/verbose  
echo xxx > /sys/kernel/debug/fail_page_alloc/verbose
```

说明：设置值 0：无消息输出；1：每注入一次错误，输出一行消息；2：带堆栈回溯消息。

### 6.9.6 fail\_page\_alloc 任务过滤器开关设置接口 task-filter

系统调用接口名称：

```
/sys/kernel/debug/fail_page_alloc/task-filter
```

功能：用于设置 fail\_page\_alloc 的是否打开任务过滤器，该接口和/proc/pid/make-it-fail 配合使用。

用法：

```
cat /sys/kernel/debug/fail_page_alloc/task-filter  
echo xxx > /sys/kernel/debug/fail_page_alloc/task-filter
```

说明：fail\_page\_alloc 的任务过滤器开关：Y-打开，N-关闭。

### 6.9.7 fail\_page\_alloc 堆栈回溯有效区域起始地址设置接口 require-start

系统调用接口名称：

```
/sys/kernel/debug/fail_page_alloc/require-start
```

功能：fail\_page\_alloc 中堆栈回溯的有效区域设定，有效区域的起始虚拟地址设定。

用法：

```
cat /sys/kernel/debug/fail_page_alloc/require-start  
echo xxx > /sys/kernel/debug/fail_page_alloc/require-start
```

说明：查看或设置 fail\_page\_alloc 的有效区域的起始虚拟地址。

### 6.9.8 fail\_page\_alloc 堆栈回溯有效区域结束地址设置接口 require-end

系统调用接口名称：

```
/sys/kernel/debug/fail_page_alloc/require-end
```

功能：fail\_page\_alloc 中堆栈回溯的有效区域设定，有效区域的结束虚拟地址设定。

用法：

```
cat /sys/kernel/debug/fail_page_alloc/require-end  
echo xxx > /sys/kernel/debug/fail_page_alloc/require-end
```

说明：查看或设置 fail\_page\_alloc 的有效区域的结束虚拟地址。

### 6.9.9 fail\_page\_alloc 堆栈回溯无效区域起始地址设置接口 reject-start

系统调用接口名称：

```
/sys/kernel/debug/fail_page_alloc/reject-start
```

功能：fail\_page\_alloc 中堆栈回溯的有效区域设定，有效区域的起始虚拟地址设定。

用法：

```
cat /sys/kernel/debug/fail_page_alloc/reject-start  
echo xxx > /sys/kernel/debug/fail_page_alloc/reject-start
```

说明：查看或设置 fail\_page\_alloc 的无效区域的起始虚拟地址。

### 6.9.10 fail\_page\_alloc 堆栈回溯无效区域结束地址设置接口 reject-end

系统调用接口名称：

```
/sys/kernel/debug/fail_page_alloc/reject-end
```

功能：fail\_page\_alloc 中堆栈回溯的有效区域设定，有效区域的结束虚拟地址设定。

用法：

```
cat /sys/kernel/debug/fail_page_alloc/reject-start  
echo xxx > /sys/kernel/debug/fail_page_alloc/reject-start
```

说明：查看或设置 fail\_page\_alloc 的无效区域的结束虚拟地址。

### 6.9.11 fail\_page\_alloc 堆栈回溯深度设置接口 stacktrace-depth

系统调用接口名称：

```
/sys/kernel/debug/fail_page_alloc/stacktrace-depth
```

功能：fail\_page\_alloc 中堆栈回溯的深度设置。

用法：

```
cat /sys/kernel/debug/fail_page_alloc/stacktrace-depth  
echo xxx > /sys/kernel/debug/fail_page_alloc/stacktrace-depth
```

说明：查看或设置 fail\_page\_alloc 的堆栈回溯深度。

### 6.9.12 fail\_page\_alloc 等待页面分配错误注入忽略设置接口 ignore-gfp-wait

系统调用接口名称：

```
/sys/kernel/debug/fail_page_alloc/ignore-gfp-wait
```

功能：fail\_page\_alloc 中对有等待的页面分配的错误注入是否需要忽略。

用法：

```
cat /sys/kernel/debug/fail_page_alloc/ignore-gfp-wait  
echo xxx > /sys/kernel/debug/fail_page_alloc/ignore-gfp-wait
```

说明：查看或设置对有等待的页面分配的错误注入是否需要忽略,Y 忽略，N 不忽略。

### 6.9.13 fail\_page\_alloc 高端页面分配错误注入忽略接口 ignore-gfp-highmem

系统调用接口名称：

```
/sys/kernel/debug/fail_page_alloc/ignore-gfp-highmem
```

功能：fail\_page\_alloc 中对有等待的页面分配的错误注入是否需要忽略。

用法：

```
cat /sys/kernel/debug/fail_page_alloc/ignore-gfp-highmem  
echo xxx > /sys/kernel/debug/fail_page_alloc/ignore-gfp-highmem
```

说明：查看或设置对对有等待的页面分配的错误注入是否需要忽略，Y 忽略，N 不忽略。



### 6.9.14 页面分配错误注入最小页面分配 order 接口 min-order

系统调用接口名称：

```
/sys/kernel/debug/fail_page_alloc/min-order
```

功能：在页面分配时，可以进行错误注入的最小页面分配 order。

用法：

```
cat /sys/kernel/debug/fail_page_alloc/min-order  
echo xxx > /sys/kernel/debug/fail_page_alloc/min-order
```

说明：查看或设置可以进行错误注入的最小页面分配 order。

## 6.10 内核 IO 请求错误注入控制接口

### 6.10.1 错误概率参数设置接口 probability

系统调用接口名称：

```
/sys/kernel/debug/fail_make_request/probability
```

功能：错误概率参数设置。

用法：

```
cat /sys/kernel/debug/fail_make_request/probability  
echo xxx > /sys/kernel/debug/fail_make_request/probability
```

说明：查看或设置 fail\_make\_request 的错误概率参数。

### 6.10.2 两次错误之间成功次数设置接口 interval

系统调用接口名称：

```
/sys/kernel/debug/fail_make_request/interval
```

功能：用于设置两次错误之间的成功次数。

用法：

```
cat /sys/kernel/debug/fail_make_request/interval  
echo xxx > /sys/kernel/debug/fail_make_request/interval
```

说明：查看或设置 fail\_make\_request 的两次错误之间的成功次数。

### 6.10.3 允许错误注入最大次数设置接口 times

系统调用接口名称：

```
/sys/kernel/debug/fail_make_request/times
```

功能：用于设置 fail\_make\_request 的允许错误注入的最大次数。

用法：

```
cat /sys/kernel/debug/fail_make_request/times  
echo xxx > /sys/kernel/debug/fail_make_request/times
```

说明：查看或设置 fail\_make\_request 的允许错误注入的最大次数。

### 6.10.4 第一次错误植入前间隔次数设置接口 space

系统调用接口名称：

```
/sys/kernel/debug/fail_make_request/space
```

功能：用于设置 fail\_make\_request 的第一次错误植入之前的间隔次数。

用法：

```
cat /sys/kernel/debug/fail_make_request/space  
echo xxx > /sys/kernel/debug/fail_make_request/space
```

说明：查看或设置 fail\_make\_request 的第一次错误植入之前的间隔次数。

### 6.10.5 fail\_make\_request 消息输出内容设置接口 verbose

系统调用接口名称：

```
/sys/kernel/debug/fail_make_request/verbose
```

功能：用于设置 fail\_make\_request 的消息输出内容设置。

用法：

```
cat /sys/kernel/debug/fail_make_request/verbose  
echo xxx > /sys/kernel/debug/fail_make_request/verbose
```

说明：设置值 0：无消息输出；1：每注入一次错误，输出一行消息；2：带堆栈回溯消息。

### 6.10.6 fail\_make\_request 任务过滤器开关设置接口 task-filter

系统调用接口名称：

```
/sys/kernel/debug/fail_make_request/task-filter
```

功能：用于设置 fail\_make\_request 的是否打开任务过滤器，该接口和 /proc/pid/make-it-fail 配合使用。

用法：

```
cat /sys/kernel/debug/fail_make_request/task-filter  
echo xxx > /sys/kernel/debug/fail_make_request/task-filter
```

说明：fail\_make\_request 的任务过滤器开关：Y 打开，N 关闭。

### 6.10.7 fail\_make\_request 堆栈回溯有效区域起始地址设置接口 require-start

系统调用接口名称：

```
/sys/kernel/debug/fail_make_request/require-start
```

功能：fail\_make\_request 中堆栈回溯的有效区域设定，有效区域的起始虚拟地址设定。

用法：

```
cat /sys/kernel/debug/fail_make_request/require-start  
echo xxx > /sys/kernel/debug/fail_make_request/require-start
```

说明：查看或设置 fail\_make\_request 的有效区域的起始虚拟地址。

### 6.10.8 fail\_make\_request 堆栈回溯有效区域结束地址设置接口 require-end

系统调用接口名称：

```
/sys/kernel/debug/fail_make_request/require-end
```

功能：fail\_make\_request 中堆栈回溯的有效区域设定，有效区域的结束虚拟地址设定。

用法：

```
cat /sys/kernel/debug/fail_make_request/require-end  
echo xxx > /sys/kernel/debug/fail_make_request/require-end
```

说明：查看或设置 fail\_make\_request 的有效区域的结束虚拟地址。

### 6.10.9 fail\_make\_request 堆栈回溯无效区域起始地址设置接口 reject-start

系统调用接口名称：

```
/sys/kernel/debug/fail_make_request/reject-start
```

功能：fail\_make\_request 中堆栈回溯的有效区域设定，有效区域的起始虚拟地址设定。

用法：

```
cat /sys/kernel/debug/fail_make_request/reject-start  
echo xxx > /sys/kernel/debug/fail_make_request/reject-start
```

说明：查看或设置 fail\_make\_request 的无效区域的起始虚拟地址。

### 6.10.10 fail\_make\_request 堆栈回溯无效区域结束地址设置接口 reject-end

系统调用接口名称：

```
/sys/kernel/debug/fail_make_request/reject-end
```

功能：fail\_make\_request 中堆栈回溯的有效区域设定，有效区域的结束虚拟地址设定。

用法：

```
cat /sys/kernel/debug/fail_make_request/reject-start  
echo xxx > /sys/kernel/debug/fail_make_request/reject-start
```

说明：查看或设置 fail\_make\_request 的无效区域的结束虚拟地址。

### 6.10.11 fail\_make\_request 堆栈回溯深度设置接口 stacktrace-depth

系统调用接口名称：

```
/sys/kernel/debug/fail_make_request/stacktrace-depth
```

功能：fail\_make\_request 中堆栈回溯的深度设置。

用法：

```
cat /sys/kernel/debug/fail_make_request/stacktrace-depth  
echo xxx > /sys/kernel/debug/fail_make_request/stacktrace-depth
```

说明：查看或设置 fail\_make\_request 的堆栈回溯深度。

## 6.11 异常可靠重启接口

### 6.11.1 异常重启时的回调处理接口

函数调用名称：

```
void (*callback_system_reset)(void)
```

功能：异常重启时的回调处理。

说明：异常重启时，可以通过该函数指针挂接相关的处理函数。

### 6.11.2 异常信息控制台输出设置接口 `excep_not_output`

Proc 接口名称：

```
/proc/sys/kernel/excep_not_output
```

功能：控制异常信息是否输出到控制台，0 表示关闭，非 0 为打开。

使用：echo 0 > /proc/sys/kernel/excep\_not\_output

## 6.12 阻塞任务内核态堆栈回溯功能接口

### 6.12.1 查看非运行状态进程的内核态堆栈回溯接口

功能：查看处于非运行状态进程的内核态堆栈回溯。

Proc 接口名称：

```
/proc/[pid]/stack
```

参数：pid 为待查看的进程号。

用法：看进程号为 pid 的进程的内核态堆栈回溯。（或以其他方式打开）

```
cat /proc/[pid]/stack
```

### 6.12.2 查看非运行状态线程的内核态堆栈回溯接口

功能：查看处于非运行状态线程的内核态堆栈回溯。

Proc 接口名称:

```
/proc/[pid]/task/[tid]/stack
```

参数: pid 为待查看的进程号; tid 为待查看的线程号。

用法: 查看进程号为 pid 的进程下 tid 号线程的内核态堆栈回溯。(或以其他方式打开)

```
cat /proc/[pid]/task/[tid]/stack
```

## 6.13 系统黑匣子功能接口

若使用黑匣子内核态相关的全局变量和函数须包含黑匣子的头文件:

```
#include <linux/bbox.h>
```

### 6.13.1 内存块设备创建接口

接口:

```
syscall(unify_syscall, SYSCALL_DEV_MEMBLK, unsigned long addr, unsigned long size, char  
*path);  
#define SYSCALL_DEV_MEMBLK ((5<<24) | (1<<16))
```

功能: 应用程序通过统一系统调用创建内存块设备。支持将保留内存作为块设备供用户使用; 同时也可将 ext 文件系统 mount 到该设备上, 供用户以文件方式访问该块设备。

输入参数:

- SYSCALL\_DEV\_MEMBLK: 内存块设备功能号, 为固定值;
- Addr: 内存块设备基址;
- size: 内存块设备容量;
- path: 创建的设备节点路径, 可以对这个设备节点格式化和挂载文件系统。

返回值: 大于 0, 成功; 其它, 失败。

### 6.13.2 获取系统黑匣子信息接口

接口:

```
int syscall(int nr, int id, char __user * buf, unsigned int offset,
```

```
unsigned int size, int region);
```

功能：该接口用于用户态获取指定的区域偏移处开始一定大小的信息段。

输入参数：

- **nr**：对应于架构下的统一系统调用号；
- **id**：扩展子功能号，读取子功能号为 **SYSCALL\_KBBOX\_GETINFO**；
- **buf**：指明要保存获取信息的 **buf** 起始地址；
- **offset**：指明要获取的信息的起始偏移，该偏移依赖于黑匣子的区域；
- **size**：要获取信息的长度，为 0 时代表获取整个区域的信息；
- **region**：指明具体存储的区域是非滚动区还是滚动区。

返回值：

- 大于 0 表示调用成功，并返回读取的字节数，若需要读取的字节数大于实际的返回值，则表示已经读取到信息的末尾；
- 等于 0 表示已经读取到区域中信息的末尾，无新数据可以读；
- 小于 0 表示调用失败，并返回错误码（-EINVAL 参数错误；-EFAULT 其他错误）。

注意：参数中提供的 **buf** 缓冲的长度必须大于等于参数 **size** 的大小，否则将可能引起访问越界。

### 6.13.3 向系统黑匣子存储一条信息接口

接口：

```
int syscall(int nr, int id, char __user * buf, unsigned int size, int region);
```

功能：该接口用于用户态向黑匣子中存储一条信息。

输入参数：

- **nr**：对应于架构下的统一系统调用号；
- **id**：扩展子功能号，存储信息子功能号为 **SYSCALL\_KBBOX\_STOREINFO**；
- **buf**：指明要保存的信息的起始地址；
- **size**：要保存信息的长度；
- **region**：指明具体存储的区域是非滚动区还是滚动区。

返回值：

- 大于 0 表示调用成功，并返回读取的字节数，当要保存的信息长度大于返回值时，表示

区域（非滚动区）已经写满，剩余信息已经不能保存；

- 等于 0 表示区域（非滚动区）已经填满，无空间可以保存信息；
- 小于 0 表示调用失败，并返回错误码（-EINVAL 参数错误；-EFAULT 其他错误）。

注意：参数中提供的 buf 缓冲的长度必须大于等于参数 size 的大小，否则将可能引起访问越界。

#### 6.13.4 获取存储区域当前记录长度接口

接口：

```
int syscall(int nr, int id, int region);
```

功能：该接口用于用户态获取指定区域已有记录信息的总长度。

输入参数：

- nr：对应于架构下的统一系统调用号；
- id：扩展子功能号，获取信息长度子功能号为 SYSCALL\_KBBOX\_GETLEN；
- region：指明具体存储的区域是非滚动区还是滚动区。

返回值：返回当前记录长度，如果是滚动区已经产生回环则返回整个滚动区大小。

#### 6.13.5 获取存储区域总长度接口

功能：该接口用于用户态获取指定区域的总长度。

接口：

```
int syscall(int nr, int id, int region);
```

输入参数：

- nr：对应于架构下的统一系统调用号；
- id：扩展子功能号，获取信息长度子功能号为 SYSCALL\_KBBOX\_GETSIZE；
- region：指明具体存储的区域是非滚动区还是滚动区。

返回值：返回当前存储区域的总大小。

#### 6.13.6 黑匣子区域基地址、滚动区大小和非滚动区大小

接口类型：内核态接口。

接口：



```
unsigned long bbox_base_phyaddr;    /* 黑匣子物理基地址 */
unsigned int bbox_const_size;       /* 黑匣子非滚动区大小 */
unsigned int bbox_loop_size;        /* 黑匣子非滚动区大小 */
```

功能：黑匣子内存布局的划分。

说明：BSP 须提供以上三个变量的值，并保证 4 字节对齐。

### 6.13.7 黑匣子模块初始化接口 kbbox\_init

接口类型：内核态接口。

接口名称：

```
int kbbox_init(void);
```

功能：该接口用于黑匣子模块的系统级初始化。

返回值：

- 等于 0 表示初始化成功；
- 小于 0 表示调用失败，并返回错误码（-EINVA 参数错误；-ENOMEM 临时缓冲申请失败）。

### 6.13.8 获取系统黑匣子信息接口 kbbox\_get\_info

接口：

```
int kbbox_get_info(void * buf, unsigned int offset, unsigned int size, int region);
```

功能：用于内核态获取区域信息的接口。

输入参数：

- buf：指明要保存获取信息的缓冲的起始地址；
- offset：指明要获取的信息的起始偏移，该偏移依赖于黑匣子的区域；
- size：要获取信息的长度，为 0 时代表获取整个区域的信息；
- region：指明具体存储的区域是非滚动区还是滚动区。

返回值：

- 大于 0 表示调用成功，并返回读取的字节数，若需要读取的字节数大于实际的返回值，则表示已经读取到信息的末尾；

- 等于 0 表示已经读取到区域中信息的末尾，无新数据可以读；
- 小于 0 表示调用失败，并返回错误码（-EINVAL 参数错误；-EFAULT 其他错误）。

注意：参数中提供的 buf 缓冲的长度必须大于等于参数 size 的大小，否则将可能引起越界。

### 6.13.9 向系统黑匣子存储一条信息接口 `kbbox_store_info`

接口：

```
int kbbbox_store_info(void * buf, unsigned int size, int region);
```

功能：该接口用于内核态向黑匣子中存储一条信息。

输入参数：

- buf：指明要保存信息的缓冲区起始地址；
- size：要保存的信息的长度；
- region：指明具体存储的区域是非滚动区还是滚动区。

返回值：

- 大于 0 表示调用成功，并返回读取的字节数，当要保存的信息长度大于返回值时，表示区域（非滚动区）已经写满，剩余信息已经不能保存；
- 等于 0 表示区域（非滚动区）已经填满，无空间可以保存信息；
- 小于 0 表示调用失败，并返回错误码（-EINVAL 参数错误；-EFAULT 其他错误）。

注意：参数中提供的 buf 缓冲的长度必须大于等于参数 size 的大小，否则将可能引起越界。

### 6.13.10 获取存储区域当前记录长度接口 `kbbox_get_len`

接口：

```
int kbbbox_get_len(region);
```

功能：该接口用于内核态获取指定区域已有记录信息的总长度。

输入参数：region 指明具体存储的区域是非滚动区还是滚动区。

返回值：返回指定区域的当前偏移值，如果是滚动区已经产生回环则返回整个滚动区大小。

### 6.13.11 获取存储区域总长度接口 `kbbox_get_size`

接口：

```
int kbbbox _get_size(int region);
```

功能：该接口用于内核态获取指定区域已有记录信息的总长度。

输入参数：region 指明具体存储的区域是非滚动区还是滚动区。

返回值：返回当前存储区域的总大小。

### 6.13.12 常规信息保存接口 kbbbox\_store\_context

接口：

```
int kbbbox_store_context(struct pt_regs *regs, int region);
```

功能：该接口用于向黑匣子中记录异常的寄存器现场、堆栈、内存和运行队列等信息。

输入参数：

- regs：该指针指向异常时保存的寄存器现场；
- region：指明具体存储的区域是非滚动区还是滚动区。

返回值：

- 等于 0 表示调用成功；
- 小于 0 表示调用失败，并返回错误码（-EINVAL 参数错误；-EFAULT 其他错误）。

### 6.13.13 黑匣子回调的注册接口

接口：

```
typedef int (*kbbbox_callback)(void *);  
int kbbbox_register_callback(kbbbox_callback callback);
```

功能：该接口用于注册黑匣子回调函数。

输入参数：callback 指向要注册的回调函数的指针。

返回值：

等于 0 表示调用成功；

小于 0 表示调用失败，并返回错误码（-EINVAL 参数错误；-EFAULT 其他错误）。

#### 6.13.14 黑匣子回调注销接口 `kbbox_unregister_callback`

接口：

```
int kbbox_unregister_callback(void);
```

功能：该接口用于注销黑匣子回调函数。

返回值：

- 等于 0 表示调用成功；
- 小于 0 表示调用失败，并返回错误码（-EINVAL 参数错误；-EFAULT 其他错误）。

#### 6.13.15 调用黑匣子回调的接口

接口：

```
int kbbox_callback(void * ptr);
```

功能：调用已注册的黑匣子回调函数，若未注册任何黑匣子回调函数，则返回。

输入参数：ptr 指向回调使用的参数，无需传参数时直接填写 NULL。

返回值：

- 等于 0 表示调用成功；
- 小于 0 表示调用失败，并返回错误码（-EINVAL 参数错误；-EFAULT 其他错误）。

#### 6.13.16 清空系统黑匣子的统一系统调用接口

接口：

```
int syscall(int nr, int id);
```

功能：黑匣子信息保存到非易失性物理介质后，就可用该接口来清除黑匣子区域了，主要是清除黑匣子状态相关的数据，清除黑匣子区域的数据。

输入参数：

- nr：对应于架构下的统一系统调用号；
- id：扩展子功能号，获取区域长度子功能号为 SYSCALL\_KBBOX\_CLEAN。

返回值：

- 0 表示调用成功；

- 小于 0 表示调用失败并返回错误码（-EINVAL 参数错误；-EFAULT 其他错误）。

### 6.13.17 清空系统黑匣子的内核接口

接口：

```
int kbbox_clean(void);
```

功能：该函数黑匣子信息保存到非易失性物理介质后，就可清除黑匣子区域了。

返回值：

0 表示调用成功；

小于 0 表示调用失败并返回错误码（-EINVAL 参数错误；-EFAULT 其他错误）。

## 6.14 softlockup 功能接口

### 6.14.1 设置控制台输出时间间隔接口 oftlockup\_print\_interval

功能：由于 softlockup 输出信息可能非常频繁，为了避免输出信息过多，允许用户设置每两次输出到控制台时间间隔，缺省间隔 30s 输出一次。目前仅 CGEL V3 版本支持。

Proc 接口名称：

```
/proc/sys/kernel/softlockup_print_interval
```

用法：可读可写。

```
cat /proc/sys/kernel/softlockup_print_interval    输出间隔时间。  
echo 40 >/proc/sys/kernel/softlockup_print_interval    设置间隔时间。
```

该值为整数，单位为 ms，最小 1，最大为 0x7FFFFFFF。

### 6.14.2 softlockup 回调接口

接口：

```
void (*callback_softlockup)(int cpuid) = NULL
```

功能：全局变量，为了使上层程序可以感知 softlockup 事件发生，从而采取合适动作，在每次产生 softlockup 事件时，调用该回调接口通知。目前仅 CGEL V3 版本支持。

用法：可赋值，在项目初始化时对该变量赋值。

## 6.15 任务运行监控接口

### 6.15.1 对应线程非运行时间统计接口 outcpustatistics

Proc 接口名称：

```
/proc/<pid>/outcpustatistics
```

功能：Outcpustatistics 即为统计所对应 pid 线程非运行时间显示接口。

用法：cat /proc/<pid>/outcpustatistics

说明：一段时间对进程所作的统计，单位纳秒（ns）。

### 6.15.2 任务长时间不运行时间间隔设置接口 moni\_nonrunning/interval

Proc 接口名称：

```
/proc/sys/kernel/moni_nonrunning/interval
```

功能：设置监控任务长时间不运行时间间隔，目前仅 CGEL V3 版本支持，需配置内核选项【Kernel hacking】-->【[\*]Monitor tasks which run too long or too little time】支持。

用法：可读可写。（或者其他读取文件的方法）

```
cat /proc/sys/kernel/moni_nonrunning/interval //输出监控间隔时间。
echo 40 >/proc/sys/kernel/softlockup_print_interval //设置监控时间。
```

该接口单位为 ms，默认值为 1000，精度为 1 个 jiffies，四舍五入。

### 6.15.3 监控任务长时间不运行启动接口 moni\_nonrunning/pids

Proc 接口名称：

```
/proc/sys/kernel/moni_nonrunning/pids
```

功能：监控任务长时间不运行功能，默认不启动，在设置监控任务后启动。目前仅 CGEL V3 版本支持，需配置内核选项【Kernel hacking】-->【[\*]Monitor tasks which run too long or too little time】支持。

用法：可读可写。（或者其他读取文件的方法）

```
cat /proc/sys/kernel/moni_nonrunning/pids          //输出监控任务。
echo xx[,xx,xx,xx,xx] >/proc/sys/kernel/moni_nonrunning/ pids    //设置监控任务。
```

默认值为-1，表示关闭状态，最多设置 5 个，后续设置覆盖前面设置，输入有效监控任务，启动该功能，系统只监控用户数据的有效任务 id，忽略无效任务 id。

说明：当检测到指定的任务不运行超过用户设定的时间间隔时，如果该任务为阻塞状态，打印出该任务调用链；如果为 R 状态，打印出该任务所在核上获得 CPU 资源的任务 PID 号，任务名字，抢占计数，优先级。

#### 6.15.4 监控任务长时间运行启动接口 `moni_running/interval`

Proc 接口名称：

```
/proc/sys/kernel/moni_running/interval
```

功能：任务长时间运行，是指一个任务在指定 CPU 上连续运行时间超过一个用户指定的时间间隔。此接口用于设置监控长时间运行任务，默认不启动，在设置监控时间后启动。目前仅 CGEL V3 版本支持，需配置内核选项【Kernel hacking】-->【(\*)Monitor tasks which run too long or too little time】支持。

用法：可读可写。（或者其他读取文件的方法）

```
cat /proc/sys/kernel/moni_running/interval          //输出监控时间。
echo xx >/proc/sys/kernel/moni_running/interval      //设置监控任务。
```

说明：当检测到任务连续运行超过用户设定的时间间隔时，打印出该任务所在 CPU，以及任务名字，PID 号，优先级，抢占计数信息。设置监控时间为有效值（大于 0）时，开启对所有任务监控，设置为小于等于 0 时，关闭该功能（默认值为-1）。该接口单位为 ms，基于 timer 来检测，精度最小为 1 个 jiffies，四舍五入。

#### 6.15.5 `/proc/sys/kernel/moni_running/cpus_allowed` 接口

Proc 接口名称：

```
/proc/sys/kernel/moni_running/cpus_allowed
```

功能：proc 接口，设置监控核，默认全部监控。目前仅 CGEL V3 版本支持，需配置内核选项【Kernel hacking】-->【(\*)Monitor tasks which run too long or too little time】支持。

用法：可读可写。（或者其他读取文件的方法）

```
cat /proc/sys/kernel/mon_i_running/cpus_allowed    //输出 cpu 掩码。  
echo xx >/proc/sys/kernel/mon_i_running/cpus_allowed  //设置 cpu 掩码。
```

## 6.16 任务退出监测功能接口

### 6.16.1 监控功能任务退出激活查询和设置接口 enable

Proc 接口名称：

```
/proc/capture/enable
```

功能：作为输入和输出接口用于激活任务退出监控功能和查询当前的任务退出功能是否激活。

用法：

```
cat /proc/capture/enable  
echo x > /proc/capture/enable
```

说明： $x$  为 1，激活任务退出监控功能； $x$  为 0，关闭任务退出监控功能。

### 6.16.2 监控功能对象线程号查询和设置接口 thread\_ids

Proc 接口名称：

```
/proc/capture/thread_ids
```

功能：作为输入和输出接口用于查询和设置监控对象的线程号。

用法：

```
cat /proc/capture/thread_ids  
echo x > /proc/capture/thread_ids
```

说明：

- 输入参数如果  $x$  为一个进程的 pid，则所有该进程的线程都将被监控；
- 输入参数如果  $x$  为一个线程的 tid，则仅仅监控该线程；

该功能可以同时监控 TRHEAD\_MAX 个 id，该宏的缺省值为 100：

- 如果任务是正常退出，内核不会自动清除 thread\_ids 中对应任务的 pid；



- 如果任务是异常退出，则内核自动清除 `thread_ids` 中对应任务的 `pid`；
- 任务退出监控模块输出的打印信息非常多，如果用户使用串口调试，请用户在使用时将串口终端的当前打印级别调至 4（`dmesg -n4`），防止串口终端过于频繁，导致单板复位。

### 6.16.3 清空当前所有带监控的任务 id 接口 `reset`

Proc 接口名称：

```
/proc/capture/reset
```

功能：清空所有 `/proc/capture/thread_ids` 中的所有 `id`。

用法：`echo 1 > /proc/capture/reset`

说明：或以其他文件写入的方式清空当前所有带监控的任务 `id`。

### 6.16.4 清除待监控任务列表中的某个 id 接口 `delid`

Proc 接口名称：

```
/proc/capture/delid
```

功能：清除待监控任务 `id` 列表中的某个 `id`。

用法：`echo x > /proc/capture/delid`

说明：或以其他文件写入的方式清除待监控任务 `id` 列表中的某个 `id`。

## 6.17 数据断点接口

### 6.17.1 数据断点设置接口 `sys_write_watchregs`

接口类型：用户态接口。

函数调用名称：

```
syscall(nr, SYSCALL_KERN_WATCHREGS, tid, idx, addr, flag)
```

功能：设置数据断点的系统调用。

输入参数：

- `nr`：对应于架构下的统一系统调用号；
- `tid`：用户态时指线程号；

- **idx**: 设置数据断点时使用的硬件断点资源号;
- **addr**: 数据断点监控的地址。

说明:

- **idx** 不再架构相关的硬件断点最大支持数范围内, 则返回失败;
- 如果要设置的 **idx** 号断点已经之前已经设置, 则返回失败;
- **addr** 根据内核态还是用户态判断地址是否合理, 不合理则返回失败;
- **addr** 为 0 时清空相应 **idx** 号断点的监控。

### 6.17.2 设置或取消数据断点接口 **hard\_regs**

功能: 设置或取消数据断点 (仅 POWERPC BOOKE、E300 架构)。

Proc 接口名称:

```
/proc/watchpoint/hard_regs
```

说明: Proc 文件系统接口与统一系统调用的接口本质上是一致的, 不同的是在解析参数的时候不同, Proc 通过对输入的命令来解析参数, 而统一系统调用采用 **syscall** 调用把参数传入。需要注意的是, 当命令行需要传入地址参数时, 必须采用 16 进制, 参数前面加 0x 前缀, 其他数值参数均为 10 进制。

用法:

```
echo 625 1 0x1009ab60 0 > /proc/watchpoint/hard_regs
```

注意上述 **echo** 命令中 625 代表 **tid** 号, 1 代表硬件断点的索引, 0x1009ab60 表示要设置的断点地址 (注意此处为地址要加 0x 前缀), 0 代表用户态断点。

### 6.17.3 数据断点区间设置接口 **sys\_write\_watchregs**

接口类型: 用户态接口。

函数调用名称:

```
syscall(nr, SYSCALL_KERN_WATCHRANGE, tid, start, len, flag)
```

功能: 设置数据断点区间的系统调用 (仅 POWERPC BOOKE 架构)。

输入参数:

- **nr**: 对应于架构下的统一系统调用号;
- **tid**: 用户态时指线程号;

- start: 数据区间断点监控的起始地址;
- len: 数据区间的长度;
- flag: 用于标志设置内核态还是用户态数据断点, 0 代表用户态, 1 代表内核态。

说明:

- 如果之前已经设置数据断点或数据区间断点, 则返回失败;
- start 根据内核态还是用户态判断地址是否合理, 不合理则返回失败;
- start 为 0 时清空数据区间断点的监控。

#### 6.17.4 设置或取消数据区间断点接口 `hard_range`

功能: 设置或取消数据区间断点。

Proc 接口名称:

```
/proc/watchpoint/hard_range
```

说明: Proc 文件系统接口与统一系统调用的接口本质上是一致的, 不同的是在解析参数的时候不同, Proc 通过对输入的命令来解析参数, 而统一系统调用采用 `syscall` 调用把参数传入。需要注意的是, 当命令行需要传入地址参数时, 必须采用 16 进制, 参数前面加 0x 前缀, 其他数值参数均为 10 进制。

用法:

```
echo 622 0x1009ab60 32 0 > /proc/watchpoint/hard_range
```

注意上述 `echo` 命令中 622 代表 tid 号, 0x1009ab60 表示要设置的断点地址 (注意此处为地址要加 0x 前缀), 32 代表断点区间长度, 0 代表用户态断点。

#### 6.17.5 用户态内存区域监控设置接口 `sys_watchrange`

接口类型: 用户态接口。

函数调用名称:

```
syscall(nr, SYSCALL_KERN_SOFTWARE, tid, start, len, ignores)
```

功能: 软件区间监控功能中设置和清除监控用户态内存区域的系统调用。设置监控后, 在调度的时候监控用户态虚拟地址的内容有没有被修改。

输入参数:

- nr: 对应于架构下的统一系统调用号;

- tid: 用户态时指线程号;
- start: 监控区间的起始地址, 为 0 时情况所有监控;
- len: 监控区间的长度;
- ignores: 扩展参数, 目前暂定为忽略线程的作用, 即指定了哪些线程不监控。

说明:

- 长度 len 不超过一个页面;
- 用户态最多监控 6 个区间地址;
- ignores 中指定忽略的 tid 数小于 10。

## 6.18 精确统计 IDLE 接口

### 6.18.1 IDLE 任务中断时间统计接口 stat

Proc 接口名称:

```
/proc/stat
```

功能: 输出每个核上面的中断、软中断、IDLE 任务执行时间。

用法: cat /proc/stat

输出结果: 在内核已有的接口上多输出了 hard\_irq\_run\_time\_cpu、irq\_run\_time\_cpu0 两个字段, 同时修改了 idle\_exec\_runtime\_cpu 字段的实现。

- idle\_exec\_runtime\_cpu: 表示 IDLE 任务在所有核运行的总时间;
- idle\_exec\_runtime\_cpu0: 表示 IDLE 任务在 0 核运行的总时间;
- hard\_irq\_run\_time\_cpu: 表示硬中断在所有核运行的总时间;
- hard\_irq\_run\_time\_cpu0: 表示硬中断在 0 核运行的总时间;
- irq\_run\_time\_cpu: 表示中断在所有核运行的总时间;
- irq\_run\_time\_cpu0: 表示中断在 0 核运行的总时间。

### 6.18.2 进程被中断打断时间统计接口 intr

Proc 接口名称:

```
/proc/<pid>/intr
```

功能: 输出进程 (含该进程的所有线程) 被中断打断的时间。

用法: `cat /proc/<pid>/intr`

说明: 输出的四个值分别表示对象进程(含所有线程)被硬中断打断了多少时间(微妙)、被中断打断了多少时间、被硬中断打断了多少个 `clock_t`, 被中断打断了多少个 `clock_t`。

### 6.18.3 线程被中断打断时间统计接口 `intr`

Proc 接口名称:

```
cat /proc/<pid>/task/<tid>/intr
```

功能: 输出线程被中断打断的时间。

用法: `cat /proc/<pid>/task/<tid>/intr`

说明: 输出的四个值分别表示对象进程的对象线程, 被硬中断打断了多少时间(微妙)、被中断打断了多少时间、被硬中断打断了多少个 `clock_t`, 被中断打断了多少个 `clock_t`。

## 6.19 死机死锁检测接口

### 6.19.1 开启 NMI 硬件看门狗接口 `wdt_enable`

Proc 接口名称:

```
/proc/nmi/wdt_enable
```

功能: 开启硬件 NMI 看门狗。

用法: `echo 1 > /proc/nmi/wdt_enable`

说明: X86 架构无需此接口, 配置 NMI 硬件看门狗后默认看门狗打开。而 PowerPC 8XX 架构, 需在 boot 阶段修改代码设置寄存器, 配合 Kuboot 版本一起使用打开硬件看门狗。

### 6.19.2 关闭 NMI 硬件看门狗接口 `wdt_disable`

Proc 接口名称:

```
/proc/nmi/wdt_disable
```

功能: 关闭硬件 NMI 看门狗。

用法: `echo 1 > /proc/nmi/wdt_disable`

说明: 支持 PowerPC BOOKE 架构和 MIPS 架构。而 X86、PowerPC 8XX、PowerPC E300 架

构均不支持该接口，硬件看门狗开启后无法关闭。

### 6.19.3 看门狗超时时间管理接口 wdt\_period

Proc 接口名称：

```
/proc/nmi/wdt_period
```

功能：调整看门狗超时的时间。

用法：echo N > /proc/nmi/wdt\_period

说明：超时时间 N 单位和具体单板平率相关。

注意：wdt\_period 指定设置 TB 寄存器（64 位，0-63）的哪一位，该位连续两次从 0 变 1 时才可能触发 NMI 看门狗，由于 TB 值是不停增长的，无法确定开始死机时 TB 寄存器的值，也就无法确定还有多长时间指定的 TB 寄存器位会从 0 变为 1。如果设置的是第 3 位（从 0 位开始），一次转变的时间区间是 1~16 个寄存器值。如果设置 NMI 设置寄存器的第 3 位，则死机后触发 NMI 所需时间区间为：

最短：1 + 16 = 17（TB 寄存器单位）；

最长：16 + 16 = 32（TB 寄存器单位）。

### 6.19.4 开启 NMI 软件看门狗接口 soft\_enable

Proc 接口名称：

```
/proc/nmi/soft_enable
```

功能：开启 NMI 软件看门狗。

用法：echo 1 > /proc/nmi/soft\_enable

说明：支持 MIPS XLS 与 MIPS XLP 架构，而其他架构均不支持 NMI 软件看门狗。

### 6.19.5 /proc/nmi/soft\_disable 接口

Proc 接口名称：

```
/proc/nmi/soft_disable
```

功能：关闭 NMI 软件看门狗

说明：echo 1 > /proc/nmi/soft\_disable

说明：支持 MIPS XLS 与 MIPS XLP 架构，而其他架构均不支持 NMI 软件看门狗。

### 6.19.6 V2LIN ufipc 机制 taskLock/intLock 调度检测接口/proc/u2k\_debug\_enable

Proc 接口名称：

```
/proc/u2k_debug_enable
```

功能：使能或关闭内核监控 V2LIN 模块在 ufipc 机制的 taskLock/intLock 接口中发生调度功能。通过对/proc/u2k\_debug\_enable 文件进行写操作（echo 操作）来使能或关闭内核监控 V2LIN 模块在 ufipc 机制的 taskLock/intLock 接口中发生调度功能。

用法：为 1 时表示使能，为 0 时表示关闭。

```
echo 1> /proc/u2k_debug_enable      //打开监控功能
echo 0> /proc/u2k_debug_enable      //关闭监控功能
```

## 6.20 深度睡眠任务长期不调度检测接口

### 6.20.1 设置监控范围 hung\_task

Proc 接口名称：

```
/proc/hung_task
```

功能：设置监控范围。

用法：

- all：监控所有线程（默认为 all），比如：echo all > /proc/hung\_task
- pidN...pidM：监控指定 pidN 到 pidM 范围的线程，比如：echo 1020...1029 > /proc/hung\_task

### 6.20.2 设置是否产生 kernel panic 接口 hung\_task\_panic

Proc 接口名称：

```
/proc/sys/kernel/hung_task_panic
```

功能：当检测出有睡眠任务不调度时，设置是否产生 kernel panic。

说明：读写接口，可以通过内核配置宏选项 `CONFIG_DETECT_HUNG_TASK` 指定，默认值为 `CONFIG_BOOTPARAM_HUNG_TASK_PANIC_VALUE`，也可以通过启动参数“`hung_task_panic=`”在运行时决定是否开启，设置成 1 生效。

### 6.20.3 限制每次遍历的线程数目上限接口 `hung_task_check_count`

Proc 接口名称：

```
/proc/sys/kernel/hung_task_check_count
```

功能：限制每次遍历的线程数目上限，避免 hungtask 后台任务执行时间过长。

说明：读写接口，默认值为 pid 的上限，完成一次遍历之后，下一次将重新开始遍历。

### 6.20.4 检测间隔时间接口 `hung_task_timeout_secs`

Proc 接口名称：

```
/proc/sys/kernel/hung_task_timeout_secs
```

功能：深度睡眠长时间不调度监控功能，通过睡眠时间 `hung_task_timeout_secs` 进行设置。当系统中存在任务，该任务的睡眠时间超过这个值都没有被调度的时候，就会打印该任务的信息。

说明：读写接口，可以通过内核配置宏选项 `CONFIG_DETECT_HUNG_TASK` 指定，默认值是 `CONFIG_DEFAULT_HUNG_TASK_TIMEOUT`，通常值是 120 秒。

注意：当需要监控睡眠时间为 A 秒的任务时，只有将 `hung_task_timeout_secs` 的监控时间设置为 A/2 秒时，才能确保监控到所有睡眠时间超过 A 秒的任务，当然这样设置的话，睡眠时间超过 A/2 秒的任务也有可能被打印。即这个监控任务睡眠时间的功能的精度只能保证睡眠时间超过 `hung_task_timeout_secs` 的 2 倍的任务都会被有效监控。

### 6.20.5 打印告警次数接口 `hung_task_warnings`

Proc 接口名称：

```
/proc/sys/kernel/hung_task_warnings
```

功能：当检测出有睡眠任务不调度时，总共打印告警的次数。

说明：读写接口，默认值为 10。



## 6.21 其他信息获取接口

### 6.21.1 进程的内存与寄存器信息获取接口 `trace_get_dump`

接口类型：内核态接口。

函数调用名称：

```
int trace_get_dump(struct task_struct * task)
```

功能：用于常规调测的库函数，获取指定进程的内存数据与寄存器信息，按照 ELF 格式保存到根目录下 `core.pid` 文件。

输入参数：`task` 为需要 dump 的任务控制结构。

返回值：0，成功；负数，失败原因。

### 6.21.2 当前进程的堆栈回溯信息获取接口 `bt_stacks`

接口类型：内核态接口。

函数调用名称：

```
int bt_stacks(int depth, char *buf, int buf_len)
```

功能：用于常规调测的库函数：获取当前进程的堆栈回溯信息；

输入参数：

- `depth`：回溯层数，0 表示不限；
- `buf`：回溯结果保存缓存；
- `buf_len`：缓存长度。

返回值：0，成功；负数，失败原因。

### 6.21.3 用户态地址对应的内核态虚拟地址查询接口 `sys_user_to_kernel`

接口类型：用户态接口。

函数调用名称：

```
syscall(nr, SYSCALL_KERN_KERNADDR, tid, addr, buf, size)
```

说明：

- 输入参数 **nr**: 对应于架构下的统一系统调用号;
- 输入参数: **tid** 用户态时指线程号;
- 输入参数 **addr**: 用户态虚拟地址;
- 输入参数 **buf**: 数组起始地址, 用于存放相应的内核虚拟地址;
- 输入参数 **size**: 传入的是指向 **buf** 大小的整形指针, 当调用完成时返回 **buf** 中保存的内核虚拟地址的个数。

#### 6.21.4 内核态保存记录结果获取接口 **sys\_watchresult**

接口类型: 用户态接口。

函数调用名称:

```
syscall(nr, SYSCALL_KERN_SOFRANGE, buf, len, index)
```

功能: 从内核态获取保存记录结果的数据区, 并获取当前记录的游标。获取记录结果的数据区并放入用户态 **buf** 中, 根据监控发生顺序显示监控结果。

输入参数:

- **nr**: 对应于架构下的统一系统调用号;
- **buf**: 用户态数据区地址;
- **len**: 用户态数据区的长度, 至少为一个页面大小;
- **index**: 当前记录的游标。

#### 6.21.5 进程下子进程 **ra** 值查看接口 **extra\_stat**

Proc 接口名称:

```
/proc/<pid>/task/<tid>/extra_stat
```

功能: 显示进程下某子进程的 **ra** 值。

用法: `cat /proc/<pid>/task/<tid>/extra_stat`

说明: 该功能只指针对 **RMI\_PHOENIX** 架构。

#### 6.21.6 调度细节统计信息查看接口 **sched**

Proc 接口名称:

```
/proc/pid/task/tid/sched
```

功能：显示调度细节统计信息。

用法：cat /proc/pid/task/tid/sched

说明：包含如下统计信息：

- 任务的总调度次数（被动和主动调度次数）；
- 任务被抢占时间统计；
- 任务阻塞时间统计；
- 任务阻塞情况下：信号量、消息队列、主动睡眠的时间统计；
- 任务所阻塞的消息队列 id 以及阻塞的信号量显示；
- 任务内核栈堆栈回溯。

### 6.21.7 获取当前的时间标签信息接口 get\_timestamp

接口类型：内核态接口。

函数调用名称：

```
unsigned long long get_timestamp (void)
```

功能：获取当前的时间标签信息。

返回值：当前时间标签，单位为纳秒。

### 6.21.8 查询核间中断统计计数接口 inter\_process\_interrupt

Proc 接口名称：

```
/proc/inter_process_interrupt
```

功能：查询核间中断统计计数，目前仅 CGEL4.X 内核提供对 MIPS64、PowerPC、X86\_64 架构的支持。

用法：通过以下读操作命令来查询各种核间中断（请求函数调用中断、请求重新调度中断、TLB 无效中断）的统计累计发生次数，方便分析核间中断对系统的影响。显示结果组织形式类似 /proc/interrupt 接口。

```
cat /proc/inter_process_interrupt
```

注意：如果需要统计动态插入的模块中触发的核间中断，应该保证在统计过程中和查询结果时，该模块不被移除。因为模块是动态链接到内核中，模块中的函数地址在模块移除并重新插入后将发生变化，目前统计功能将以函数地址为准，模块移除后将不会显示模块中触发核间中断的

函数名字。

## 6.22 SYSRQ 功能扩展魔法键接口

SYSRQ 功能是利用串口通信键盘控制可扩展一键式调测功能。通过按下 Ctrl+^组合键或者 Ctrl+break 组合键以后，在 5 秒内按下 command 字符，会得到对应的打印信息。其中 command 是一个字符，可以是 1 - 9、a-z、A-Z（字母不区分大小写）。

相关 command 对应的显示信息如下所示：

- 0-9: 设置 printk 的打印级别。和 /proc/sys/kernel/printk 的第一个参数意思相同；
- b: 立即重启单板，不进行磁盘同步等操作；调用内核的函数为 emergency\_restart；
- c: 进行 kexec reboot，需要 KEXC 支持。调用内核的函数为 crash\_kexec；
- d: 显示此有的锁，需要配置 CONFIG\_LOCKDEP，调用函数为 debug\_show\_all\_locks；
- e: 向所有进程（init 除外）发送 SIGTERM 信号，发送信号调用 force\_sig 函数；
- f: 进入 out-of-memory 流程，杀死一个进程，调用 out\_of\_memory 函数；
- i: 向所有进程（init 除外）发送 SIGKILL 信号；
- m: 显示当前内存信息，调用内核 show\_mem 函数；
- P: 显示当前寄存器的值；
- q: 显示当前的 timer，调用 timer\_list\_show；
- s: 同步当前所有的文件系统，调用 emergency\_sync 函数；
- t: 显示当前系统上所有的进程信息，调用函数为 show\_state；
- u: 重新 mount 所有文件系统，调用函数为 emergency\_remount；
- w: 显示系统中所有处于 uninterruptable 状态的进程，调用函数为 show\_state。

注意：powerpc 单板，在按下 Ctrl+break 组合键后会产生中断风暴，为了解决此问题打过两次补丁，导致在 powerpc 单板在按下 Ctrl+break 后需要按两次 command 才能生效，Ctrl+^不受影响。

# 第 7 章

## 网络协议栈接口

### 7.1 RPS/RFS 功能接口

目前单个 CPU 的处理能力已经难以适应高性能网卡接收/发送报文的速度，因此提高多核处理器数据报文并行处理能力则成为 SMP 系统下提高网络吞吐量的必要手段。

**RPS**：全称是 **Receive Packet Steering**，这是 Google 工程师 Tom Herbert 提交的内核补丁。这个 patch 采用软件模拟的方式，实现了多队列网卡所提供的功能，把不同流的数据包分发给不同的 CPU 核来处理，分散了在多 CPU 系统上数据接收时的负载，把软中断分到各个 CPU 处理，而不需要硬件支持，大大提高了网络性能。

**RFS**：全称是 **Receive Flow Steering**，这也是 Tom 提交的内核补丁，它是用来配合 RPS 补丁使用的，是 RPS 补丁的改进扩展补丁。它不仅能够把同一流的数据包分发给同一个 CPU 核来处理，还能够分发给其“被期望”的 CPU 核来处理，提高 cache 的命中率。

#### 7.1.1 指定网卡包交给特定 CPU 处理接口 `rps_cpus`

系统调用接口名称：

```
/sys/class/net/<device>/queues/rx-<n>/rps_cpus
```

功能：通过给此接口设置一个 mask 指定特定网卡 device 或者网卡上的某个接收队列 rx-<n>（如果支持多接受队列的话）可以包交给特定的几个 CPU 处理。

用法：`echo xxx > /sys/class/net/<device>/queues/rx-<n>/rps_cpus`

说明：或以其他文件写入方式设置，默认此接口的当值是 0，RPS 是无效的。包的 hash 值和 `rps_cpus` 的值共同决定了该数据包会放入哪个 CPU 的 backlog 队列。

注意：

- 一个数据包到达后，中断所到达的 CPU 负责分配，这时会通过 SKB 内容计算 hash，由于是第一次读这个 SKB，所以发生 cache miss 正常。但是之后这个数据包可能会被分配到其他 CPU 上处理，对于另一个 CPU 来说，SKB 仍然是一个未知内容，所以 cache miss 又会发生，这次的 cache miss 就是 RPS 带来的新开销。不过幸好，很多网卡都天生支持计算这样的 hash，驱动只需要把该值存入 SKB，就可以提高 cache hit，以及计算机整体性能。
- rps\_cpus 可以在运行时被动态改变，但是这样的改动可能会导致包的乱序。因为改动前某个 transaction 的数据包都是由某个 CPU 处理，而 rps\_cpus 改变，可能导致该 transaction 的包被发往另一个 CPU 上处理，如果后到的包在第二个 CPU 上被处理的速度快于先到的包在第一个 CPU 上被处理的速度，就会发生乱序。所以尽可能不要频繁改变 rps\_cpus 的值。

### 7.1.2 rps\_sock\_flow\_table 总数设置和获取接口 rps\_sock\_flow\_entries

系统调用接口名称：

```
/proc/sys/net/core/rps_sock_flow_entries
```

功能：设置和获取全局数据流表 rps\_sock\_flow\_table 的总数。

用法：

```
cat /proc/sys/net/core/rps_sock_flow_entries  
echo xxx > /proc/sys/net/core/rps_sock_flow_entries
```

说明：或以其他文件写入方式设置。

### 7.1.3 rps\_dev\_flow 总数设置和获取接口 rps\_flow\_cnt

系统调用接口名称：

```
/sys/class/net/<device>/queues/rx-<n>/rps_flow_cnt
```

功能：设置和获取每个队列 rxqueue 的数据流表 rps\_dev\_flow\_table 的 rps\_dev\_flow 总数。

用法：

```
cat /sys/class/net/<device>/queues/rx-<n>/rps_flow_cnt  
echo xxx > /sys/class/net/<device>/queues/rx-<n>/rps_flow_cnt
```

说明：或以其他文件写入方式设置。

每个接收队列若要启用 RFS，需要同时设置 `/proc/sys/net/core/rps_sock_flow_entries` 与 `/sys/class/net/<dev>/queues/rx-<n>/rps_flow_cnt` 两个参数，参数的值会被进位到最近的 2 的幂次方值。（例如，参数的值是 7，则实际有效值是 8；参数是值 32，则实际值就是 32。）在设置时注意，流计数依赖于期待的有效的连接数，这个值明显小于连接总数。经测试，`rps_sock_flow_entries` 设置成 32768，在中等负载的服务器上工作效率明显提升。对于单队列设备，单队列的 `rps_flow_cnt` 值建议被配置成与 `rps_sock_flow_entries` 相同。对于一个多队列设备，每个队列的 `rps_flow_cnt` 建议被配置成 `rps_sock_flow_entries/N`，N 是队列总数。例如，如果 `rps_sock_flow_entries` 设置成 32768，并且有 16 个接收队列，每个队列的 `rps_flow_cnt` 最好被配置成 2048。

## 7.2 LOP 功能接口

### 7.2.1 获取 PACKET\_LOP 值接口 `getsockopt`

接口类型：用户态接口。

函数调用名称：

```
int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen)
```

功能：添加了获取 `PACKET_LOP` 选项相关的值。有时，一个程序需要确定为当前为一个套接口进行哪些选项设置。这对于一个子程序库函数尤其如此，因为这个库函数并不知道为这个套接口进行哪些设置，而这个套接口需要作为一个参数进行传递。程序也许需要知道类似于流默认使用的缓冲区的大小。

输入参数：

- `s`：要进行选项检验的套接口；
- `level`：选项检验所在的协议层；
- `optname`：要检验的选项；
- `optval`：指向接收选项值的缓冲区的指针；
- 指针 `optlen`：同时指向输入缓冲区的长度和返回的选项长度值。

返回值：0，成功；-1，失败，错误原因会存放在外部变量 `errno` 中。

用法：以下方式用于获取 `lop` 标记位。

```
getsockopt(sockfd_packet, SOL_PACKET, PACKET_LOP, (void *)&socklop, &len)
```

### 7.2.2 设置 `PACKET_LOP` 值接口 `setsockopt`

接口类型：用户态接口。

函数调用名称:

```
int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen)
```

功能: 添加了设置 PACKET\_LOP 选项相关的值。

输入参数:

- s: 选项改变所要影响的套接口;
- level: 选项的套接口层次;
- optname: 要设计的选项名;
- optval: 指向要为新选项所设置的值的指针;
- 指针 optlen: 选项长度值。

返回值: 0, 成功; -1, 失败, 错误原因会存放在外部变量 errno 中。

用法: 以下方式用于获取 lop 标记位。

```
setsockopt(sockfd_packet, SOL_PACKET, PACKET_LOP, (void *)&socklop, sizeof(socklop))
```

## 7.3 设置 NUD\_REACHABLE 到 NUD\_STALE 状态的切换间隔接口

接口:

```
/proc/sys/net/ipv4/neigh/<接口名>/base_reachable_time
```

功能: ARP 条目在收到 ARP 确认报文后进入 NUD\_REACHABLE 状态, 在一段时间内未使用此条目后, 内核切换为 NUD\_STALE 状态 (可疑状态)。在 NUD\_STALE 状态下, 可能主动发送 arp 请求确认对端状态。完善开源内核, 在如上接口中设置 NUD\_REACHABLE 状态切换到 NUD\_STALE 状态的时间间隔, 并立即生效。

用法:

- cat 该文件: 可查看 NUD\_REACHABLE 状态切换到 NUD\_STALE 状态的时间间隔;
- echo 该文件: 可设置 NUD\_REACHABLE 状态切换到 NUD\_STALE 状态的时间间隔, 并立即生效。

## 7.4 网卡数据包截获功能

使用此功能需在 menuconfig 中配置选中 **【Networking support】** ---> **【Net Packet Intercept】**



选项以提供支持。

### 7.4.1 filter 规则的设置查询接口

接口：

```
syscall(int __NR_unify_syscall, unsigned int cmd, int ifindex, unsigned int opt, void *buf,
unsigned int size)
```

功能：网卡数据包截获功能 filter 规则的设置/取消/查询接口（用户态）。

输入参数：

- `__NR_unify_syscall`：为当前架构的统一系统调用号；
- `cmd`：代表具体的子功能号，此处对应 `SYSCALL_PACKET_INTERCEPT`；
- `ifindex`，为要设置的物理网卡索引，不支持网卡别名、bond、网桥等，由上层保证有效性。
- `opt`：选项如下；
  - ✧ `--SET_INTERCEPT_RECVFILTER`，设置网卡接收报文时的截获 filter；
  - ✧ `--SET_INTERCEPT_SENDFILTER`，设置网卡发送报文时的截获 filter；
  - ✧ `--GET_INTERCEPT_RECVFILTER`，获取网卡接收报文时的截获 filter；
  - ✧ `--GET_INTERCEPT_SENDFILTER`，获取网卡发送报文时的截获 filter。
- `buf`：如果是设置 filter，则为用户态传入的 filter 规则的地址，filter 定义为 `struct sock_fprog`，使用同 `setsockopt` 的 `SO_ATTACH_FILTER`；为 `NULL` 则表示取消 filter 设置。如果是获取 filter，则用来保存内核返回的 filter；
- `size`，为 `buf` 的大小。

返回值：成功，0；失败，-1，并显示错误码（`ENOSYS` 无此系统调用；`ENODEV` 无此设备；`EINVAL` 非法参数；`EPERM` 获取 filter，但对应的 filter 并未设置；`EFAULT` buf 指针不可访问）。

说明：若重复设置，将覆盖前一次设置。

### 7.4.2 回调注册接口 `netdev_register_intercept_func`

接口：

```
int netdev_register_intercept_func(char *ifname, intercept_func recv_func, intercept_func
send_func)
```

功能：注册网卡数据包截获功能的回调函数。

intercept\_func 结构：回调函数类型定义在 linux/netdevice.h，引用时需包含该头文件。

```
typedef int (*intercept_func)(struct sk_buff *);
```

其中，参数 sk\_buff 为网络报文 buffer。返回值 PACKET\_UNINTERCEPTED 为报文未被截获，内核继续处理；PACKET\_INTERCEPTED 为报文被截获，内核将其丢弃。

输入参数：

- ifname：为要设置的物理网卡名，不支持网卡别名、bond、网桥等，由上层保证有效性；
- recv\_func：接收报文的截获处理函数，为 NULL 即不注册这个处理；
- send\_func：发送报文的截获处理函数，为 NULL 即不注册这个处理。

返回值：成功，0；失败，-ENODEV，无此设备。

说明：重复注册，将覆盖前一次注册的处理函数。

### 7.4.3 回调注销接口 netdev\_unregister\_intercept\_func

接口：

```
int netdev_unregister_intercept_func(char *ifname, int which)
```

功能：注销网卡数据包截获功能的回调函数。

输入参数：

- ifname：为要设置的物理网卡名，不支持网卡别名、bond、网桥等，由上层保证有效性；
- which：如果设置了 RECV\_INTERCEPT\_FUNC 位，就注销接收报文的截获处理函数；如果设置了 SEND\_INTERCEPT\_FUNC 位，就注销发送报文的截获处理函数。

返回值：成功，0；失败，-ENODEV，无此设备。

# 第 8 章

## 虚拟化接口

### 8.1 虚拟机用户打开共享内存授权设备文件接口

功能：虚拟机用户打开共享内存授权设备接口。

接口：

```
int  
open(  
    Const char* pathname,  
    int flags  
)
```

输入参数：

- **pathname**：授权设备文件路径，一般为/dev/xen/gntdev 文件，不建议用户移动此文件位置；
- **flags**：授权设备文件打开方式，一般为 O\_RDWR。

返回值：授权设备句柄，成功；-1，失败。

### 8.2 虚拟机用户关闭共享内存授权设备文件接口

功能：虚拟机用户关闭共享内存授权设备文件接口。

接口：

```
int
close(
int fd
)
```

输入参数：fd 授权设备文件句柄。

返回值：0 成功；-1 失败。

### 8.3 虚拟机用户操作共享内存授权设备文件接口

功能：虚拟机用户操作共享内存授权设备文件接口。

接口：

```
int
ioctl(
int fd,
int cmd,
...
)
```

输入参数：

- fd：授权设备文件句柄；
- cmd：授权设备操作命令码，见下表 1；
- ...：授权设备不同命令码对应的传入操作参数类型，见下表 1。

返回值：返回值参见下表对应命令码的返回值定义。

表 1 授权设备操作命令参数对应关系

命令码	传入参数类型	功能	返回值
IOCTL_GNTDEV_ALLOC_GRANT_REFS	ioctl_gntdev_alloc_grant_refs	虚拟机授权表分配授权映射	0 表示成功 非 0 表示失败
IOCTL_GNTDEV_FREE_GRANT_REFS	ioctl_gntdev_free_grant_refs	虚拟机授权表释放授权映射	0-成功 非 0-失败
IOCTL_GNTDEV_CLAIM_GRANT_REF	ioctl_gntdev_claim_grant_ref	从申请授权映射组中声明一个授权映射	授权表索引值 非法值表示执行

			失败
<b>IOCTL_GNTDEV_RELEASE_GRANT_REF</b>	<code>ioctl_gntdev_release_grant_ref</code>	释放一个授权映射到授权映射组	0 释放成功 非 0 释放失败
<b>IOCTL_GNTDEV_GRANT_FOREIGN_ACCESS_REF</b>	<code>ioctl_gntdev_grant_foreign_access_ref</code>	设置授权映射外部虚拟机访问	0 授权成功 非 0 授权失败
<b>IOCTL_GNTDEV_END_FOREIGN_ACCESS_REF</b>	<code>ioctl_gntdev_end_foreign_access_ref</code>	取消授权映射被外部虚拟机访问	0 取消授权成功 非 0 取消授权失败
<b>IOCTL_GNTDEV_MAP_GRANT_REF</b>	<code>ioctl_gntdev_map_grant_ref</code>	添加一个授权映射管理结构	0-成功 非 0-失败
<b>IOCTL_GNTDEV_UNMAP_GRANT_REF</b>	<code>gntdev_ioctl_unmap_grant_ref</code>	删除一个授权映射管理结构	0-成功 非 0-失败
<b>IOCTL_GNTDEV_GET_OFFSET_FOR_VADDR</b>	<code>ioctl_gntdev_get_offset_for_vaddr</code>	根据输入的虚拟地址获取相应的授权映射管理结构的索引号	0-成功 非 0-失败
<b>IOCTL_GNTDEV_SET_MAX_GRANTS</b>	<code>ioctl_gntdev_set_max_grants</code>	设置一次可以操作的最大授权索引数	0-成功 非 0-失败

## 8.4 全虚拟化虚拟机 PV-on-HVM 驱动内核启动参数控制接口

功能：控制全虚拟化虚拟机 PV-on-HVM 驱动的启用及 Qemu 设备卸载。

接口：无

说明：

**xen\_pv\_hvm**：为 `disable` 表示不启用 PV-on-HVM 驱动，其他情况下默认启用。

**xen\_emul\_unplug**：指定卸载哪类 Qemu 设备，默认为 `all`。

- `ide-disks`：卸载主磁盘设备；
- `aux-ide-disks`：卸载非主磁盘设备；
- `nics`：卸载掉网卡设备；
- `all`：卸载掉所有磁盘和网卡设备。

# 第 9 章

## V2LIN 适配库接口

V2LIN 模块处于用户态程序中，提供了与 VxWorks 相同的 API 接口。用户移植 VxWorks 应用程序时，只需在 VxWorks 的应用代码中加入 V2LIN 模块，然后再作很少的修改工作即可得到底层操作系统接口的支持，减少移植的工作量，提高工作效率。本章主要对 V2LIN 的适配接口进行列举介绍，更多使用方法的详细介绍请参考 [《V2LIN 使用指南》](#)。

### 9.1 任务接口

头文件：taskLib.h

#### 9.1.1 taskSpawn 接口

功能：V2LIN 适配库创建任务。

结构：

```
int taskSpawn
(
char * name,
int priority,          /* priority of new task */
int options,          /* task option word */
int stackSize,
FUNCPTR entryPt,      /* entry point of new task */
int arg1,
int arg2,
int arg3,
```

```

int    arg4,
int    arg5,
int    arg6,
int    arg7,
int    arg8,
int    arg9,
int    arg10
)

```

输入参数：

- name: 创建任务名字；
- priority: 创建任务优先级；
- options: 创建任务选项；
- stackSize: 兼容 Vxworks 堆栈参数（目前保留，未使用）；
- entryPt: 创建任务的入口函数；
- arg1 - arg10: 创建任务的参数。

返回值：0 成功；-1 失败。

### 9.1.2 taskInit 接口

功能：V2LIN 适配库用于初始化任务相关的数据结构。

结构：

```

STATUS taskInit
(
    WIND_TCB * pTcb,          /* address of new task's TCB */
    char *     name,
    int        priority,      /* priority of new task */
    int        options,       /* task option word */
    char *     pStackBase,    /* base of new task's stack */
    int        stackSize,     /* size (bytes) of stack needed */
    FUNCPTR    entryPt,       /* entry point of new task */
    int        arg1,
    int        arg2,
    int        arg3,
    int        arg4,

```

```
int    arg5,  
int    arg6,  
int    arg7,  
int    arg8,  
int    arg9,  
int    arg10  
)
```

输入参数：

- pTcb: 需要初始化的任务控制块结构；
- name: 创建任务名字；
- priority: 创建任务优先级；
- options: 创建任务选项；
- stackSize: 兼容 Vxworks 堆栈参数（目前保留，未使用）；
- entryPt: 创建任务的入口函数；
- arg1 - arg10: 创建任务的参数。

返回值：0 成功；-1 失败。

### 9.1.3 taskActivate 接口

功能：V2LIN 适配库激活任务（使一个新任务运行）接口。

结构：

```
STATUS taskActivate  
(  
    int tid                /* task ID of task to activate */  
)
```

输入参数：tid 需要激活的任务 ID。

返回值：0 成功；-1 失败。

### 9.1.4 taskDelete 接口

功能：V2LIN 适配库任务删除接口。

结构：



```
STATUS taskDelete
(
    int tid                /* task ID of task to delete */
)
```

输入参数：tid 需要删除的任务 ID。

返回值：0 成功；-1 失败

### 9.1.5 taskDeleteForce 接口

功能：V2LIN 适配库强制删除接口（即使处于安全状态的任务也要被删除）。

结构：

```
STATUS taskDeleteForce
(
    int tid                /* task ID of task to delete */
)
```

输入参数：tid 需要强制删除的任务 ID。

返回值：0 成功；-1 失败

### 9.1.6 taskSuspend 接口

功能：V2LIN 适配库 taskSuspend 接口。

结构：

```
STATUS taskSuspend
(
    int tid                /* task ID of task to suspend */
)
```

输入参数：tid 需要挂起的任务 ID。

返回值：0 成功；-1 失败。

### 9.1.7 taskResume 接口

功能：V2LIN 适配库任务恢复接口。

结构:

```
STATUS taskResume
(
    int tid                /* task ID of task to resume */
)
```

输入参数: tid 需要输入的任务 ID。

返回值: 0 成功; -1 失败。

### 9.1.8 taskRestart 接口

功能: V2LIN 适配库任务重启接口。

结构:

```
STATUS taskRestart
(
    int tid                /* task ID of task to restart */
)
```

输入参数: tid 需要重启的任务 ID。

返回值: 0 成功; -1 失败。

### 9.1.9 taskPrioritySet 接口

功能: V2LIN 适配库优先级设置。

结构:

```
STATUS taskPrioritySet
(
    int tid,                /* task ID */
    int newPriority          /* new priority */
)
```

返回值: 0 成功; -1 失败。

输入参数:

- tid: 设置优先级的任务 ID;

- newPriority: 设置的新优先级数字。

### 9.1.10 taskPriorityGet 接口

功能：V2LIN 适配库获取任务优先级接口。

结构：

```
STATUS taskPriorityGet
(
    int    tid,           /* task ID */
    int * pPriority        /* return priority here */
)
```

返回值：0 成功；-1 失败。

输入参数：

- tid: 获取优先级的任务 ID；
- pPriority: 用于存储任务的优先级。

### 9.1.11 taskLock 接口

功能：V2LIN 适配库关任务调度接口。

结构：

```
STATUS taskLock (void)
```

返回值：0 成功；-1 失败。

### 9.1.12 taskUnlock 接口

功能：V2LIN 适配库打开任务调度。

结构：

```
STATUS taskUnlock (void)
```

返回值：0 成功；-1 失败。

### 9.1.13 taskIdSelf 接口

功能：获取 V2LIN 任务的 taskId。

结构：

```
int taskIdSelf(void)
```

返回值：task id 成功；-1 失败。

### 9.1.14 taskIdVerify 接口

功能：验证 V2LIN 任务的 taskId。

结构：

```
STATUS taskIdVerify (void)
```

返回值：0 成功；-1 失败。

### 9.1.15 taskTcb 接口

功能：通过 taskId 获取相应 V2LIN 任务的 TCB 结构地址。

结构：

```
WIND_TCB *taskTcb(int taskid)
```

输入参数：taskid 为任务 ID。

返回值：成功，TCB 结构地址；NULL，失败。

### 9.1.16 taskInfoGet 接口

功能：根据 tid 获取 V2LIN 任务信息。

结构：

```
STATUS taskInfoGet
(
    int          tid,          /* ID of task for which to get info */
    TASK_DESC    *pTaskDesc    /* task descriptor to be filled in */
)
```

输入参数：

- tid：任务 ID，即任务 TCB 结构指针；
- pTaskDesc：将获取到的任务信息，存放到该指针指向的 TASK\_DESC 结构内。

返回：成功，OK；失败，ERROR。

### 9.1.17 taskShow 接口

功能：根据 tid 打印 V2LIN 任务信息。

结构：

```
STATUS taskShow (int tid, int level)
```

输入参数：

- tid：任务 ID，即任务 TCB 结构指针；
- level：打印信息详细级别。目前不支持该参数。

返回：成功，OK；失败，ERROR。

### 9.1.18 taskName 接口

头文件：taskInfo.h

功能：通过 taskId 获取 V2LIN 任务名。

结构：

```
char *taskName(int tid)
```

输入参数：tid 为任务 ID，即任务 TCB 结构指针。

返回：成功，非 NULL，任务名字符串指针；失败，NULL。

### 9.1.19 taskNameToId 接口

头文件：taskInfo.h

功能：通过 V2LIN 任务名获取 taskId。

结构：

```
int taskNameToId(char *name)
```

输入参数：name 为任务名。

返回：成功，其他，对应任务的 ID，即 TCB 结构指针；失败，ERROR。

### 9.1.20 taskIsReady 接口

头文件：taskInfo.h

功能：判断任务是否已准备运行。

结构：

```
BOOL taskIsReady(int taskid)
```

输入参数：taskid 为任务 ID。

返回：TRUE，任务状态为 READY；FALSE，任务状态非 READY。

### 9.1.21 taskIsSuspended 接口

头文件：taskInfo.h

功能：判断任务是否已挂起。

结构：

```
BOOL taskIsSuspended(int taskid)
```

输入参数：taskid 为任务 ID，即任务 TCB 结构指针。

返回：TRUE，任务状态为 Suspend；FALSE，任务状态非 Suspend。

### 9.1.22 taskIdListGet 接口

头文件：taskInfo.h

功能：将当前所有 V2LIN 任务 taskId 放入 list 指定的数组中。

结构：

```
int taskIdListGet(int list[], int maxIds)
```

输入参数：

- list：存放 taskId 的数组地址；

- maxIds: list 数组能容纳的最大任务 taskId 的数量。

返回: 获取得到的 taskId 的数量。

## 9.2 信号量接口

头文件: semLib.h

### 9.2.1 semShow 接口

头文件: semShow.h

功能: 打印 sem 信号量的具体状态信息。

结构:

```
STATUS semShow (SEM_ID semId, int level)
```

输入参数:

- semId: 信号量 ID;
- level: 打印级别, level 为 1, 将打印挂在该信号量上的所有任务信息。

返回值: 成功, OK; 失败, ERROR。

与 Vxworks 相应接口的差异: V2LIN 信号量有三种实现方式(见下面备注说明), 如果使用 ufipc 方式的信号量, semShow()接口目前没有实现, 直接返回 ERROR; 如果配置\_USE\_SEM\_FUTEX\_, semShow()接口不支持 level 参数。

Vxworks 错误码: S\_smObjLib\_NOT\_INITIALIZED 为 semShow 回调函数没有被初始化。

备注: V2LIN 信号量有三种实现方式, 分别为: 不做任何配置情况下, 默认使用 c 库接口实现信号量; 配置\_USE\_SEM\_FUTEX\_, 表示使用 futex 系统调用实现信号量; 配置\_USE\_UF\_IPC\_LIB\_, 表示使用 ufipc 方式实现信号量, 需要与 ufipc 库配合使用。

### 9.2.2 semBCreate 接口

功能: 创建并初始化 B 型信号量。

结构:

```
V2LINsem_t *semBCreate(int opt, SEM_B_STATE initial_state)
```

参数说明:

- **opt**: 信号量创建选项。
  - ✧ **SEM\_Q\_FIFO**: 等待任务按照 FIFO 方式被唤醒。
  - ✧ **SEM\_Q\_PRIORITY**: 等待任务按优先级顺序被唤醒。
  - ✧ **SEM\_EVENTSEND\_ERR\_NOTIFY**: 不支持。
- **initial\_state**: 信号量初始化状态。
  - ✧ **SEM\_EMPTY**: B 型信号量, 初始值为 EMPTY。
  - ✧ **SEM\_FULL**: B 型信号量, 初始值为 FULL。

返回: 成功, 创建的 B 型信号量 ID; 失败, ERROR。

错误码:

- **S\_semLib\_INVALID\_OPTION**: 输入的 **opt** 参数错误。
- **S\_semLib\_INVALID\_STATE**: 输入的 **initial\_state** 参数错误。
- **ERR\_UF\_SEM\_NO\_MEM**: **ufipc** 方式下, 为 **ufipc** 保留的内存池空间不足。
- **ERR\_UF\_SEM\_INVALID\_ID**: **ufipc** 方式下, 信号量 ID 错误。
- **ERR\_UF\_SEM\_INVALID\_OPTION**: **ufipc** 方式下, 输入的 **opt** 参数错误。
- **ERR\_UF\_SEM\_INVALID\_STATE**: **ufipc** 方式下, 输入的 **initial\_state** 参数错误。

与 VxWorks 相应接口的差异:

1) V2LIN 中, 该接口 **opt** 参数不支持 **SEM\_EVENTSEND\_ERR\_NOTIFY**。

2) V2LIN 中 B 型信号量有三种实现方式。如果使用 **FUTEX** 方式的信号量, 或者 c 库接口实现信号量, 暂时不支持 **SEM\_Q\_FIFO** 参数; 当使用 **SEM\_Q\_PRIORITY** 参数, 等待在该信号量上的任务, 将按照进入等待队列时的任务优先级先后顺序依次被唤醒, 而非按照唤醒时刻等待任务的实时优先级被唤醒。如果使用 **ufipc** 方式, 暂时不支持 **SEM\_Q\_PRIORITY** 参数。

3) VxWorks 错误码:

- **S\_intLib\_NOT\_ISR\_CALLABLE**: 该函数在中断上下文被调用。
- **S\_memLib\_NOT\_ENOUGH\_MEMORY**: 内存池剩余空间不足, 无法创建信号量。
- **S\_semLib\_INVALID\_OPTION**: **opt** 参数输入错误。
- **S\_semLib\_INVALID\_STATE**: 输入的 **initial\_state** 参数错误。

备注: V2LIN 中 B 型信号量有三种实现方式。如果使用 **FUTEX** 方式的信号量, 或者 c 库接口实现信号量, 暂时不支持 **SEM\_Q\_FIFO** 参数; 如果使用 **ufipc** 方式, 暂时不支持 **SEM\_Q\_PRIORITY** 参数。



### 9.2.3 semCCreate 接口

功能：创建并初始化 C 型信号量。

结构：

```
V2LINsem_t * semCCreate (int opt, int initial_count)
```

参数说明：

- opt: 信号量创建选项。
  - ✧ SEM\_Q\_FIFO: 等待任务按照 FIFO 方式被唤醒。
  - ✧ SEM\_Q\_PRIORITY: 等待任务按优先级顺序被唤醒。
  - ✧ SEM\_EVENTSEND\_ERR\_NOTIFY: 不支持。
- initial\_count: C 信号量初始化 count 值。

返回：成功，创建的 C 型信号量 ID；失败，ERROR。

错误码：

- S\_semLib\_INVALID\_OPTION: 输入的 opt 参数错误。
- ERR\_UF\_SEM\_NO\_MEM: ufipc 方式下为 ufipc 保留的内存池空间不足。
- ERR\_UF\_SEM\_IVALID\_ID: ufipc 方式下，信号量 ID 错误。
- ERR\_UF\_SEM\_INVALID\_OPTION: ufipc 方式下，输入的 opt 参数错误。
- ERR\_UF\_SEM\_INVALID\_STATE: ufipc 方式下，输入的 initial\_state 参数错误。

与 VxWorks 相应接口的差异：

1) V2LIN 中，该接口 opt 参数不支持 SEM\_EVENTSEND\_ERR\_NOTIFY。

2) V2LIN 中 C 型信号量有三种实现方式。如果使用 FUTEX 方式的信号量，或者 c 库接口实现信号量，暂时不支持 SEM\_Q\_FIFO 参数；当使用 SEM\_Q\_PRIORITY 参数，等待在该信号量上的任务，将按照进入等待队列时的任务优先级先后顺序依次被唤醒，而非按照唤醒时刻等待任务的实时优先级被唤醒。如果使用 ufipc 方式，暂时不支持 SEM\_Q\_PRIORITY 参数。

3) VxWorks 错误码：

- S\_intLib\_NOT\_ISR\_CALLABLE: 该函数在中断上下文中被调用。
- S\_memLib\_NOT\_ENOUGH\_MEMORY: 内存池剩余空间不足，无法创建信号量。
- S\_semLib\_INVALID\_OPTION: opt 参数输入错误。

备注：V2LIN 中 C 型信号量有三种实现方式。如果使用 FUTEX 方式的信号量，或者 c 库接口

实现信号量，暂时不支持 SEM\_Q\_FIFO 参数；如果使用 ufipc 方式，暂时不支持 SEM\_Q\_PRIORITY 参数。

#### 9.2.4 semMCreate 接口

功能：创建并初始化 M 型信号量。

结构：

```
V2LINsem_t * semMCreate (int opt)
```

参数说明：opt 为信号量创建选项，选项如下。

- ✧ SEM\_Q\_FIFO：等待任务按照 FIFO 方式被唤醒。
- ✧ SEM\_Q\_PRIORITY：等待任务按优先级顺序被唤醒。
- ✧ SEM\_DELETE\_SAFE：保护任务在获取到 M 信号量时不会被删除。不支持。
- ✧ SEM\_INVERSION\_SAFE：防止 M 信号量的优先级翻转。
- ✧ SEM\_EVENTSEND\_ERR\_NOTIFY：不支持。

返回：成功，创建的 M 型信号量 ID；失败，ERROR。

错误码：

- S\_semLib\_INVALID\_OPTION：输入的 opt 参数错误。
- ERR\_UF\_SEM\_NO\_MEM：ufipc 方式下为 ufipc 保留的内存池空间不足。
- ERR\_UF\_SEM\_INVALID\_ID：ufipc 方式下，信号量 ID 错误。
- ERR\_UF\_SEM\_INVALID\_OPTION：ufipc 方式下，输入的 opt 参数错误。
- ERR\_UF\_SEM\_INVALID\_STATE：ufipc 方式下，输入的 initial\_state 参数错误。

与 VxWorks 相应接口的差异：

1) V2LIN 中，该接口 opt 参数不支持 SEM\_EVENTSEND\_ERR\_NOTIFY。

2) V2LIN 中 M 型信号量有三种实现方式。如果使用 FUTEX 方式的信号量，或者 c 库接口实现信号量，暂时不支持 SEM\_Q\_FIFO 参数；当使用 SEM\_Q\_PRIORITY 参数，等待在该信号量上的任务，将按照进入等待队列时的任务优先级先后顺序依次被唤醒，而非按照唤醒时刻等待任务的实时优先级被唤醒。

3) 如果使用 ufipc 方式，当未使用 SEM\_INVERSION\_SAFE 参数时，等待任务的唤醒方式为 SEM\_Q\_FIFO，不支持 SEM\_Q\_PRIORITY；当使用 SEM\_INVERSION\_SAFE 参数时，等待任务的唤醒方式为 SEM\_Q\_PRIORITY，不支持 SEM\_Q\_FIFO。

另外，如果使用 c 库接口实现信号量，将不支持 SEM\_INVERSION\_SAFE 参数；另外两种信号量实现方式，均支持 SEM\_INVERSION\_SAFE 参数。

4) VxWorks 错误码：

- S\_intLib\_NOT\_ISR\_CALLABLE：该函数在中断上下文中被调用。
- S\_memLib\_NOT\_ENOUGH\_MEMORY：内存池剩余空间不足，无法创建信号量。
- S\_semLib\_INVALID\_OPTION：opt 参数输入错误。

备注：V2LIN 中 M 型信号量有三种实现方式。如果使用 FUTEX 方式的信号量，或者 c 库接口实现信号量，暂时不支持 SEM\_Q\_FIFO 参数；如果使用 ufipc 方式，暂时不支持 SEM\_Q\_PRIORITY 参数。另外，如果使用 c 库接口实现信号量，将不支持 SEM\_INVERSION\_SAFE 参数。

### 9.2.5 semTake 接口

功能：获取信号量，如果信号量无法被获取到，将会使任务等待，直到获取得到信号量，或者超时。

结构：

```
STATUS semTake(V2LINsem_t * semaphore, int max_wait)
```

参数说明：

- semaphore：需要获取的信号量 ID。
- max\_wait：任务等待的超时时间，单位：tick。
  - ✧ NO\_WAIT：任务不等待直接返回。
  - ✧ WAIT\_FOREVER：除非获取到信号量，否则任务将永远等待。

返回：成功，OK；失败，ERROR。

错误码：

- S\_objLib\_OBJ\_ID\_ERROR：信号量 ID 错误。
- S\_objLib\_OBJ\_UNAVAILABLE：信号量无法被获取得到。当参数为 NO\_WAIT，同时信号量无法被获取时返回。
- S\_objLib\_OBJ\_TIMEOUT：信号量获取超时。
- ERR\_UF\_SEM\_INVALID\_ID：ufipc 方式下，信号量 ID 错误。
- ERR\_UF\_SEM\_UNAVAILABLE：ufipc 方式下，信号量无法被获取得到。
- ERR\_UF\_SEM\_INVALID\_TIMEOUT：ufipc 方式下，获取信号量超时。
- ERR\_UF\_SEM\_DELETED：ufipc 方式下，表示操作的信号量已经被删除。

- **ERR\_UF\_SEM\_EFAULT**: ufipc 方式下, 信号量等待错误。
- 与 VxWorks 相应接口的差异: VxWorks 错误码如下。
- **S\_intLib\_NOT\_ISR\_CALLABLE**: 该函数在中断上下文中被调用。
- **S\_objLib\_OBJ\_ID\_ERROR**: semaphore 参数输入错误。
- **S\_objLib\_OBJ\_UNAVAILABLE**: 信号量无法被获取得到。当参数为 NO\_WAIT, 同时信号量无法被获取时返回。
- **S\_objLib\_OBJ\_TIMEOUT**: 信号量等待超时。

备注: 非 V2LIN 创建的任务不能使用 V2LIN 的信号量。

### 9.2.6 semGive 接口

功能: 释放信号量。如果有任务正因为等待该信号量而被挂起, 该任务将会被唤醒。

结构:

```
STATUS semGive(V2LINsem_t * pSem)
```

参数说明: pSem 为需要释放的信号量 ID。

返回: 成功, OK, 失败, ERROR。

错误码:

- **S\_objLib\_OBJ\_ID\_ERROR**: 信号量 ID 错误。
- **S\_semLib\_INVALID\_OPERATION**: 信号量操作错误。
- **ERR\_UF\_SEM\_IVALID\_ID**: ufipc 方式下, 信号量 ID 错误。
- **ERR\_UF\_SEM\_UNAVAILABLE**: ufipc 方式下, 信号量无法被释放。

与 VxWorks 相应接口的差异: VxWorks 错误码如下。

- **S\_intLib\_NOT\_ISR\_CALLABLE**: 该函数在中断上下文中被调用。
- **S\_objLib\_OBJ\_ID\_ERROR**: pSem 参数输入错误。
- **S\_semLib\_INVALID\_OPERATION**: 当前任务不是 M 信号量的拥有者。
- **S\_eventLib\_EVENTSEND\_FAILED**: 信号量向注册的任务发送事件失败。这个错误码只有在创建时带有 SEM\_EVENTSEND\_ERR\_NOTIFY 参数的信号量才有可能得到。

备注: 非 V2LIN 创建的任务不能使用 V2LIN 的信号量。

### 9.2.7 semFlush 接口

功能：刷新信号量。该接口只能用于 B 或者 C 型信号量，将会把所有在该信号量上等待的任务都唤醒。

结构：

```
STATUS semFlush(V2LINsem_t * pSem)
```

参数说明：pSem 为需要刷新的信号量 ID。

返回：成功，OK；失败，ERROR。

错误码：

- S\_objLib\_OBJ\_ID\_ERROR：信号量 ID 错误。
- S\_semLib\_INVALID\_OPERATION：信号量操作错误。
- ERR\_UF\_SEM\_DELETED：ufipc 方式下，表示操作的信号量已经被删除。
- ERR\_UF\_SEM\_INVALID\_OPERATION：ufipc 方式下，信号量操作错误
- ERR\_UF\_SEM\_INVALID\_ID：ufipc 方式下，信号量 ID 错误。
- ERR\_UF\_SEM\_UNAVAILABLE：ufipc 方式下，信号量无法被使用。

与 VxWorks 相应接口的差异：VxWorks 错误码如下。

- S\_intLib\_NOT\_ISR\_CALLABLE：该函数在中断上下文被调用。
- S\_objLib\_OBJ\_ID\_ERROR：pSem 参数输入错误。

备注：非 V2LIN 创建的任务不能使用 V2LIN 的信号量。

### 9.2.8 semDelete 接口

功能：删除信号量。

函数原型：

```
STATUS semDelete(V2LINsem_t * pSem)
```

参数说明：pSem 为需要删除的信号量 ID。

返回：成功，OK；失败，ERROR。

错误码：

- S\_objLib\_OBJ\_ID\_ERROR：信号量 ID 错误。

- S\_objLib\_OBJ\_UNAVAILABLE: 信号量无法被删除。
- ERR\_UF\_SEM\_UNAVAILABLE: ufipc 方式下, 信号量无法被使用。

与 VxWorks 相应接口的差异: VxWorks 错误码如下。

- S\_intLib\_NOT\_ISR\_CALLABLE: 该函数在中断上下文中被调用。
- S\_objLib\_OBJ\_ID\_ERROR: pSem 参数输入错误。
- S\_smObjLib\_NO\_OBJECT\_DESTROY: 不允许删除共享的信号量。

备注: 非 V2LIN 创建的任务不能使用 V2LIN 的信号量。当 M 信号量在被删除时, 用户需要保证该信号量已经被释放, 没有被占用, 否则删除失败。

## 9.3 消息队列接口

头文件: msgQLib.h

### 9.3.1 msgQCreate 接口

功能: V2LIN 适配库消息队列创建接口

结构:

```
MSG_Q_ID msgQCreate
(
    int maxMsgs,           /* max messages that can be queued */
    int maxMsgLength,     /* max bytes in a message */
    int options            /* message queue options */
)
```

输入参数:

- maxMsgs: 创建消息队列时需要指定最大消息数;
- maxMsgLength: 发送消息的最大消息长度;
- options: 创建消息队列的选项。

返回值: 0 成功; -1 失败。

### 9.3.2 msgQDelete 接口

功能: V2LIN 适配库消息队列删除接口

结构:

```
STATUS msgQDelete
(
    MSG_Q_ID msgQId          /* message queue to delete */
)
```

输入参数：msgQId 为需要删除的消息队列 ID。

返回值：0 成功；-1 失败。

### 9.3.3 msgQSend 接口

功能：V2LIN 适配库消息队列发送接口

结构：

```
STATUS msgQSend
(
    MSG_Q_ID msgQId,          /* message queue on which to send */
    char *   buffer,          /* message to send */
    UINT     nBytes,          /* length of message */
    int      timeout,          /* ticks to wait */
    int      priority          /* MSG_PRI_NORMAL or MSG_PRI_URGENT */
)
```

输入参数：

- msgQId：发送报文的消息队列 ID；
- buffer：存储发送报文的内容；
- nBytes：发送报文的长度；
- timeout：当资源不满足时的等待时间（单位：毫秒）；
- priority：发送报文的优先级。

返回值：0 成功；-1 失败。

### 9.3.4 msgQReceive 接口

功能：V2LIN 适配库消息队列接收接口

结构：

```
int msgQReceive
```

```
(  
    MSG_Q_ID msgQId,  
    char *    buffer,          /* buffer to receive message */  
    UINT      maxNBytes,      /* length of buffer */  
    int       timeout         /* ticks to wait */  
)
```

输入参数：

- msgQId: 需要接收的消息队列 ID;
- buffer: 接收报文时的消息队列 buffer;
- maxNbytes: buffer 的长度;
- timeout: 当资源不满足时的等待时间（单位：毫秒）。

返回值：0 成功；-1 失败。

### 9.3.5 msgQNumMsgs 接口

功能：V2LIN 适配库查看消息队列个数

结构：

```
int msgQNumMsgs  
(  
    MSG_Q_ID msgQId          /* message queue to examine */  
)
```

输入参数：msgQId 为消息队列 ID。

返回值：0 成功；-1 失败。

### 9.3.6 msgQShow 接口

头文件：msgQShow.h

功能：打印消息队列信息。

结构：

```
STATUS msgQShow  
(  
    MSG_Q_ID msgQId,          /* message queue to display */  
)
```



```
int level          /* 0 = summary, 1 = details */
)
```

输入参数:

- msgQId: 消息队列 ID。
- level: 消息显示等级, 0 基本信息; 1 详细信息。

返回: 成功, OK; 失败, ERROR。

## 9.4 TICK 值接口

头文件: tickLib.h

### 9.4.1 tick64Get 接口

功能: 获取当前的系统 tick 值。

结构:

```
unsigned long long tick64Get (void)
```

返回: 返回系统当前的 tick 时间, 该接口返回值为 64 位。

### 9.4.2 tickGet 接口

功能: 获取当前的系统 tick 值。

结构:

```
unsigned long tickGet (void)
```

说明:

返回: 返回系统当前的 tick 时间。该接口返回值为 32 位。

输入参数

## 9.5 TIMER 定时器接口

头文件: timerLib.h

### 9.5.1 timer\_create 接口

功能：创建定时器。

结构：

```
int timer_create(clockid_t clock_id, struct sigevent *evp, v2pt_timer_t *pTimer)
```

输入参数：

- clock\_id: 指定时钟 ID，目前该参数只接受传入 CLOCK\_REALTIME；
- evp: 传入 struct sigevent 结构体指针，该结构体由用户分配，指定定时器到期时发送信号的值；
- pTimer: 返回创建定时器的指针。

返回：成功，OK；失败，ERROR。

### 9.5.2 timer\_connect 接口

功能：为定时器设置用户态信号处理函数。

结构：

```
int timer_connect(v2pt_timer_t timer, VOIDFUNCPTR routine, int arg)
```

输入参数：

- timer: 进行设置的定时器指针；
- routine: 定时器到期时的信号处理函数；
- arg: 信号处理函数的参数。

返回：成功，OK；失败，ERROR。

### 9.5.3 timer\_settime 接口

功能：设定定时器超时时间。

结构：

```
int timer_settime  
(  
    v2pt_timer_t timer,  
    int flags,
```

```
const struct itimerspec *value,  
struct itimerspec *ovalue  
)
```

输入参数:

- **timer**: 进行设置的定时器指针;
- **flags**: 如果该值为 **TIMER\_ABSTIME**, 表示第三个参数为绝对时间; 如果不为 **TIMER\_ABSTIME**, 则表示第三个参数为距该接口调用时间开始的相对时间;
- **value**: 指定定时器的超时时间;
- **ovalue**: 返回该定时器距前一次设置的超时点的时间差; 如果该定时器已经超时, 则返回的时间差值为 0。

返回: 成功, OK; 失败, ERROR。

#### 9.5.4 timer\_gettime 接口

功能: 获取定时器剩余到期时间。

结构:

```
int timer_gettime(v2pt_timer_t timer, struct itimerspec *value)
```

输入参数:

- **timer**: 获取剩余时间的定时器指针;
- **value**: 返回定时器的剩余时间, 如果定时器已经到期, 返回时间为 0。

返回: 成功, OK, 失败, 其他。

#### 9.5.5 timer\_cancel 接口

功能: 取消定时器的计时。

结构:

```
int timer_cancel(v2pt_timer_t timer)
```

输入参数: **timer** 为取消定时的定时器的指针。

返回: 成功, OK; 失败, 其他。

### 9.5.6 timer\_delete 接口

功能：删除定时器。

结构：

```
int timer_delete(v2pt_timer_t timer)
```

输入参数：timer 为定时器的指针。

返回：成功，OK；失败，ERROR。

## 9.6 WD 定时器接口

头文件：wdLib.h

### 9.6.1 wdCreate 接口

功能：创建软件看门狗定时器。

结构：

```
v2pt_wdog_t *wdCreate(void)
```

返回：成功，返回 wd 定时器结构指针；失败，NULL。

### 9.6.2 wdDelete 接口

功能：删除软件看门狗定时器。

结构：

```
STATUS wdDelete(v2pt_wdog_t * wdId)
```

输入参数：wdId 为 wd 定时器结构指针。

返回：成功，OK；失败，ERROR。

### 9.6.3 wdStart 接口

功能：启动软件看门狗定时器。

结构：

```
STATUS wdStart(v2pt_wdog_t * wdId, int delay, int (*f) (int), int parm)
```

输入参数：

- wdId: wd 定时器结构体指针；
- delay: wd 定时器超时时间，单位 tick；
- f: wd 超时执行的超时回调函数；
- parm: 超时回调函数的参数。

返回：成功，OK；失败，ERROR。

#### 9.6.4 wdCancel 接口

功能：取消软件看门狗定时器的计时。

结构：

```
STATUS wdCancel(WDOG_ID wdId)
```

输入参数：wdId 为 wd 定时器结构体指针。

返回：成功，OK；失败，ERROR。

#### 9.6.5 wdShow 接口

头文件：wdShow.h

功能：打印指定看门狗定时器的信息。

结构：

```
STATUS wdShow (WDOG_ID wdId)
```

返回：成功，OK。

输入参数：wdId 为 wd 定时器结构体指针。

### 9.7 END 接口

头文件：endLib.h

### 9.7.1 mib2Init 接口

功能：初始一个 MIB 结构。

结构：

```
STATUS mib2Init
(
    M2_INTERFACETBL *pMib,          /* struct to be initialized */
    long ifType,                     /* ifType from m2Lib.h */
    UCHAR * phyAddr,                 /* MAC/PHY address */
    int addrLength,                  /* MAC/PHY address length */
    int mtuSize,                     /* MTU size */
    int speed                         /* interface speed */
)
```

输入参数：

- pMib: 指向需要初始化的 M2\_INTERFACETBL 结构体；
- ifType: 网络接口类型。一般为 M2\_ifType\_ethernetCsmacd（以太网\_CSMA/CD）；
- phyAddr: 物理地址（MAC 地址）；
- addrLength: 物理地址长度；
- mtuSize: 网络接口的 mtu 大小；
- speed: 网络接口的速度。

返回：成功，OK；失败，ERROR。

### 9.7.2 mib2ErrorAdd 接口

功能：增加 MIB 结构中的错误信息统计数。

结构：

```
STATUS mib2ErrorAdd
(
    M2_INTERFACETBL * pMib,
    int errCode,
    int value
)
```

输入参数：

- pMib: 指向 M2\_INTERFACETBL 结构体;
- errCode: 错误码;
- value: 该项错误码的错误统计增加的数量。

返回: 成功, OK; 失败, ERROR。

### 9.7.3 endObjInit 接口

功能: 初始化一个 END\_OBJ 结构。

结构:

```
STATUS endObjInit
(
    END_OBJ *   pEndObj,          /* object to be initialized */
    DEV_OBJ*    pDevice,          /* ptr to device struct */
    char *      pBaseName,        /* device base name, for example, "In" */
    int         unit,              /* unit number */
    NET_FUNCS * pFuncTable,        /* END device functions */
    char*       pDescription
)
```

输入参数:

- pEndObj: 指向需要初始化的 END\_OBJ 结构体;
- pDevice: 指向一个 DEV\_OBJ 结构体, 将 END 与 DEV 联系起来;
- pBaseName: 设备名;
- unit: 设备编号;
- pFuncTable: END 驱动的回调函数表;
- pDescription: 设备的描述字符串。

返回: 成功, OK; 失败, ERROR。

### 9.7.4 endObjFlagSet 接口

功能: 设置 END 结构的 flags 标志位。

结构:

```
STATUS endObjFlagSet
(
```

```
END_OBJ * pEnd,  
UINT flags  
)
```

输入参数：

- pEnd: 指向 END\_OBJ 结构体；
- flags: 新设置的标志位的值。

返回：成功，OK；失败，ERROR。

### 9.7.5 endEtherAddressForm 接口

功能：设置网络数据包的链路层源地址、目的地址。

结构：

```
M_BLK_ID endEtherAddressForm  
(  
    M_BLK_ID pMblk,      /* pointer to packet mBlk */  
    M_BLK_ID pSrcAddr,   /* pointer to source address */  
    M_BLK_ID pDstAddr,   /* pointer to destination address */  
    BOOL bcastFlag       /* use link-level broadcast? */  
)
```

返回：返回增添了链路层源地址、目的地址的 mBlk 指针。

输入参数：

- pMblk: 指向 mBlk 结构体的指针，表示网络数据包；
- pSrcAddr: 设置的链路层源地址；
- pMblk: 设置的链路层目的；
- bcastFlag: 是否使用链路层广播地址作为目的地址。

### 9.7.6 endEtherPacketDataGet 接口

功能：剥离链路层包头的信息。

结构：

```
STATUS endEtherPacketDataGet  
(
```



```

M_BLK_ID      pMblk,
LL_HDR_INFO *  pLinkHdrInfo
)

```

输入参数：

- **pMblk**: 指向 mBlk 结构体的指针，表示网络数据包。在该函数操作完成后，该指针指向剥离链路层包头收的数据；
- **pLinkHdrInfo**: 将从 pMblk 指向的数据包中获取的链路层包头信息，放到该参数指向的 LL\_HDR\_INFO 结构体中。

返回：成功，OK；失败，ERROR。

### 9.7.7 endEtherPacketAddrGet 接口

功能：获取 pMblk 参数指向的数据包中的链路层地址信息。

结构：

```

STATUS endEtherPacketAddrGet
(
    M_BLK_ID pMblk, /* pointer to packet */
    M_BLK_ID pSrc,  /* pointer to local source address */
    M_BLK_ID pDst,  /* pointer to local destination address */
    M_BLK_ID pESrc, /* pointer to remote source address (if any) */
    M_BLK_ID pEDst  /* pointer to remote destination address (if any) */
)

```

输入参数：

- **pMblk**: 指向 mBlk 结构体的指针，表示网络数据包；
- **pSrc**: 获取的链路层源地址；
- **pDst**: 获取的链路层目的地址；
- **pESrc**: V2LIN 中，该参数获取值与 pESrc 一致；
- **pEDst**: V2LIN 中，该参数获取值与 pEDst 一致。

返回：成功，OK；失败，ERROR。

## 9.8 网络缓存池接口

头文件：netBufLib.h

### 9.8.1 netBufLibInit 接口

功能：初始化 netBufLib。

结构：

```
STATUS netBufLibInit (void)
```

返回：成功，OK；失败，ERROR。

### 9.8.2 netPoolInit 接口

功能：初始化 netPool。

结构：

```
STATUS netPoolInit
(
    NET_POOL_ID      pNetPool,      /* pointer to a net pool */
    M_CL_CONFIG * pMclBlkConfig,    /* pointer to a mBlk configuration */
    CL_DESC *      pClDescTbl,      /* pointer to cluster desc table */
    int            clDescTblNumEnt,  /* number of cluster desc entries */
    POOL_FUNC *    pFuncTbl         /* pointer to pool function table */
)
```

输入参数：

- pNetPool: netPool 结构指针，指向需初始化的 netPool。
- pMclBlkConfig: 指向一个 mBlk 的初始化配置。
- pClDescTbl: 指向一个 cluster 描述列表。
- clDescTblNumEnt: cluster 列表中表项的数量。
- pFuncTbl: netPool 相关函数列表。

返回：成功，OK；失败，ERROR。

### 9.8.3 netPoolDelete 接口

功能：删除指定的 netPool。

结构：

```
STATUS netPoolDelete
(
    NET_POOL_ID  pNetPool           /* pointer to a net pool */
)
```

输入参数：pNetPool 为 netPool 结构指针，指向需删除的 netPool。

返回：成功，OK；失败，ERROR。

### 9.8.4 netMblkFree 接口

功能：释放一个 mBlk 到指定的 netPool。

结构：

```
void netMblkFree
(
    NET_POOL_ID      pNetPool,       /* pointer to the net pool */
    M_BLK_ID         pMblk           /* mBlk to free */
)
```

输入参数：

- pNetPool: netPool 结构指针，指向 netPool；
- pMblk: 指向需要释放的 mBlk。

返回：无。

### 9.8.5 netCIBlkFree 接口

功能：释放一个 mBlk 到指定的 netPool。

结构：

```
void netCIBlkFree
(
    NET_POOL_ID      pNetPool,       /* pointer to the net pool */

```

```

        CL_BLK_ID    pClBlk                /* pointer to the clBlk to free */
    )

```

输入参数:

- pNetPool: netPool 结构指针, 指向 netPool;
- pClBlk: 指向需要释放的 clBlk。

返回: 无。

### 9.8.6 netClFree 接口

功能: 释放一个 cluster 到指定的 netPool。

结构:

```

void netClFree
(
    NET_POOL_ID    pNetPool,                /* pointer to the net pool */
    UCHAR *        pClBuf                  /* pointer to the cluster buffer */
)

```

输入参数:

- pNetPool: netPool 结构指针, 指向 netPool;
- pClBuf: 指向需要释放的 cluster。

返回: 无。

### 9.8.7 netMblkClFree 接口

功能: 释放一个 mBlk-clBlk-cluster 结构 1。

结构:

```

M_BLK_ID netMblkClFree
(
    M_BLK_ID    pMblk                /* pointer to the mBlk */
)

```

输入参数: pMblk 为指向需要释放的 mBlk。

返回: 成功, 如果指定释放的 mBlk 为某个 mBlk 链中的一个, 函数返回下一个 mBlk 的指针;

失败，NULL。

### 9.8.8 netMblkClChainFree 接口

功能：释放一个 mBlk-clBlk-cluster 结构 2。

结构：

```
void netMblkClChainFree
(
    M_BLK_ID      pMblk          /* pointer to the mBlk */
)
```

输入参数：pMblk 为指向需要释放的 mBlk。

返回：成功，如果指定释放的 mBlk 为某个 mBlk 链中的一个，函数返回下一个 mBlk 的指针；失败，NULL。

### 9.8.9 netMblkGet 接口

功能：从指定的 netPool 池中获取一个 mBlk 结构。

结构：

```
M_BLK_ID netMblkGet
(
    NET_POOL_ID  pNetPool,        /* pointer to the net pool */
    int  canWait,                  /* M_WAIT/M_DONTWAIT */
    UCHAR  type                    /* mBlk type */
)
```

输入参数：

- pNetPool：指向 netPool 内存池；
- canWait：可取以下两值；
  - ✧ M\_WAIT：如果没有可以获取的 mBlk，并且初始化了\_pNetBufCollect 函数指针，将会调用\_pNetBufCollect 指向的函数，执行一次垃圾收集后，在重新获取一次；
  - ✧ M\_DONTWAIT：如果没有可以获取的 mBlk，则直接返回。
- type：设置获取的 mBlk 的类型。

返回：成功，获取的 mBlk 指针；失败，NULL。

### 9.8.10 netCIBlkGet 接口

功能：从指定的 netPool 池中获取一个 cIBlk 结构。

结构：

```
CL_BLK_ID netCIBlkGet
(
    NET_POOL_ID    pNetPool,    /* pointer to the net pool */
    int            canWait      /* M_WAIT/M_DONTWAIT */
)
```

输入参数：

- pNetPool: 指向 netPool 内存池；
- canWait: 可取以下两值；
  - ✧ M\_WAIT: 如果没有可以获取的 cIBlk，并且初始化了\_pNetBufCollect 函数指针，将会调用\_pNetBufCollect 指向的函数，执行一次垃圾收集后，在重新获取一次；
  - ✧ M\_DONTWAIT: 如果没有可以获取的 cIBlk，则直接返回。

返回：成功，获取的 cIBlk 指针；失败，NULL。

### 9.8.11 netClusterGet 接口

功能：从指定的 netPool 池的一个 cluster pool 中获取一个 cluster buffer。

结构：

```
char * netClusterGet
(
    NET_POOL_ID    pNetPool,    /* pointer to the net pool */
    CL_POOL_ID     pCIPool      /* ptr to the cluster pool */
)
```

输入参数：

- pNetPool: 指向 netPool 内存池；
- pCIPool: 指向 netPool 中的一个 cluster pool。

返回：成功，获取的 cluster 指针；失败，NULL。

### 9.8.12 netMblkClGet 接口

功能：从指定的 netPool 池中分配 clBlk-cluster 链。

结构：

```
STATUS netMblkClGet
(
    NET_POOL_ID    pNetPool,    /* pointer to the net pool */
    M_BLK_ID       pMblk,       /* mBlk to embed the cluster in */
    int            bufSize,      /* size of the buffer to get */
    int            canWait,      /* wait or dontwait */
    BOOL           bestFit       /* TRUE/FALSE */
)
```

返回：成功，OK；失败，NULL。

输入参数：

- pNetPool：指向 netPool 内存池；
- pMblk：分配的 clBlk-cluster，将连接到该 mBlk 上；
- bufSize：需要分片的 buffer 数据大小；
- canWait：可取以下两值；
  - ✧ M\_WAIT：如果没有可以获取的 clBlk，并且初始化了 \_pNetBufCollect 函数指针，将会调用 \_pNetBufCollect 指向的函数，执行一次垃圾收集后，在重新获取一次；
  - ✧ M\_DONTWAIT：如果没有可以获取的 clBlk，则直接返回。
- bestFit：可取以下两值；
  - ✧ TRUE：如果与 bufsize 精确一致的 cluster 无法获取，则该函数将自动获取一个更大的 cluster，如果更大的 cluster 无法获得，怎返回失败；
  - ✧ FALSE：如果与 bufsize 精确一致的 cluster 无法获取，则该函数自动获取一个更大或者更小的 cluster。

### 9.8.13 netTupleGet 接口

功能：从指定的 netPool 池中，分配一个 mBlk-clBlk-cluster 链。

结构：

```
M_BLK_ID netTupleGet
(
    NET_POOL_ID    pNetPool,    /* pointer to the net pool */
    int            bufSize,      /* size of the buffer to get */
    int            canWait,      /* wait or dontwait */
    UCHAR          type,         /* type of data */
    BOOL           bestFit       /* TRUE/FALSE */
)
```

返回：成功，获取的 mBlk 指针；失败，NULL。

输入参数：

- pNetPool：指向 netPool 内存池；
- bufSize：需要分片的 buffer 数据大小；
- canWait：可取以下两值；
  - ✧ M\_WAIT：如果没有可以获取的 mBlk 或者 cBlk，并且初始化了 \_pNetBufCollect 函数指针，将会调用 \_pNetBufCollect 指向的函数，执行一次垃圾收集后，在重新获取一次；
  - ✧ M\_DONTWAIT：如果没有可以获取的 mBlk 或者 cBlk，则直接返回。
- type：分配 buffer 的数据类型；
- bestFit：可取以下两值；
  - ✧ TRUE：如果与 bufsize 精确一致的 cluster 无法获取，则该函数将自动获取一个更大的 cluster，如果更大的 cluster 无法获得，怎返回失败；
  - ✧ FALSE：如果与 bufsize 精确一致的 cluster 无法获取，则该函数自动获取一个更大或者更小的 cluster。

#### 9.8.14 netCIBlkJoin 接口

功能：把一个 cluster 连接到一个指定的 cIBlk 上。

结构：

```
CL_BLK_ID netCIBlkJoin
(
    CL_BLK_ID    pCIBlk,        /* pointer to a cluster Blk */
    char *       pCIBuf,        /* pointer to a cluster buffer */
    int          size,           /* size of the cluster buffer */
)
```



```

FUNCPTR      pFreeRtn,    /* pointer to the free routine */
int          arg1,        /* argument 1 of the free routine */
int          arg2,        /* argument 2 of the free routine */
int          arg3         /* argument 3 of the free routine */
)

```

输入参数：

- pClBlk: 指定连接的 clBlk;
- size: cluster buffer 的大小;
- pFreeRtn: 初始化 pClBlk 中 pClFreeRtn 的值。在释放 clBlk 时调用, 用于释放相关联的 cluster buffer;
- arg1, arg2, arg3: pFreeRtn 的参数。

返回: 成功, 返回 pClBlk 指针; 失败, NULL。

### 9.8.15 netMblkClJoin 接口

功能: 把一个 mBlk 连接到一个指定的 ClBlk-cluster 结构上。

结构:

```

M_BLK_ID netMblkClJoin
(
    M_BLK_ID      pMblk,    /* pointer to an mBlk */
    CL_BLK_ID      pClBlk   /* pointer to a cluster Blk */
)

```

输入参数:

- pMblk: 指定连接的 mBlk;
- pClBlk: 指向连接的 ClBlk-cluster 结构。

返回: 成功, 返回 mBlk 指针; 失败, NULL。

### 9.8.16 netClPoolIdGet 接口

功能: 根据 bufsize 参数, 返回 cluster pool 指针。

结构:

```

CL_POOL_ID netClPoolIdGet

```

```
(  
    NET_POOL_ID    pNetPool,    /* pointer to the net pool */  
    int            bufSize,      /* size of the buffer */  
    BOOL           bestFit       /* TRUE/FALSE */  
)
```

输入参数：

- pNetPool: 指向分配的内存池 netPool;
- bufSize: 获取 cluster 的 buffer 大小;
- bestFit: 去以下两值。
  - ✧ TRUE: 如果与 bufsize 精确一致的 cluster 无法获取, 则该函数将自动获取一个更大的 cluster pool;
  - ✧ FALSE: 如果与 bufsize 精确一致的 cluster 无法获取, 则该函数自动获取一个任意大小可获取的 cluster pool。

返回: 成功, 返回 cluster pool 指针; 失败, NULL。

### 9.8.17 netMblkToBufCopy 接口

功能: 从 mBlk 中复制数据到 pbuf。

结构:

```
int netMblkToBufCopy  
(  
    M_BLK_ID      pMblk,        /* pointer to an mBlk */  
    char *        pBuf,          /* pointer to the buffer to copy */  
    FUNCPTR       pCopyRtn      /* function pointer for copy routine */  
)
```

输入参数：

- pMblk: 指向数据源 mBlk;
- pBuf: 拷贝目的地址;
- pCopyRtn: 复制操作调用的函数, 如果 pCopyRtn 为 NULL, 则调用默认的复制函数, V2LIN 中该函数为 bcopy。

返回: 成功, 复制 data 的长度 (字节); 失败, 0。

### 9.8.18 netMblkDup 接口

功能：复制一个 mBlk 结构。

结构：

```
M_BLK_ID netMblkDup
(
    M_BLK_ID  pSrcMblk,      /* pointer to source mBlk */
    M_BLK_ID  pDestMblk     /* pointer to the destination mBlk */
)
```

输入参数：

- pSrcMblk: 指向源 mBlk;
- pDestMblk: 指向目的 mBlk。

返回：成功，目的 mBlk 的指针；失败，NULL。

### 9.8.19 netMblkChainDup 接口

功能：复制一个 mBlk-clBlk-cluster 链结构。

结构：

```
M_BLK_ID netMblkChainDup
(
    NET_POOL_ID  pNetPool,    /* pointer to the pool */
    M_BLK_ID  pMblk,          /* pointer to source mBlk chain*/
    int         offset,       /* offset to duplicate from */
    int         len,          /* length to copy */
    int         canWait       /* M_DONTWAIT/M_WAIT */
)
```

返回：成功，目的 mBlk 的指针；失败，NULL。

输入参数：

- pNetPool: 复制的 mBlk-clBlk-cluster 链结构从该内存池中分配;
- pMblk: 指向源 mBlk;
- offset: 从源 mBlk 的 buffer 中，offset 的位置开始复制;
- len: 复制数据的长度;

- canWait: 取以下两值。
  - ✧ M\_WAIT: 如果没有可以获取的 mBlk 或者 clBlk, 并且初始化了 \_pNetBufCollect 函数指针, 将会调用 \_pNetBufCollect 指向的函数, 执行一次垃圾收集后, 在重新获取一次;
  - ✧ M\_DONTWAIT: 如果没有可以获取的 mBlk 或者 clBlk, 则直接返回。

## 9.9 LIST 接口

头文件: listLib.h

### 9.9.1 lstLibInit 接口

功能: 初始 lstLib 各种操作需要的资源。

结构:

```
void lstLibInit (void)
```

### 9.9.2 lstInit 接口

功能: 初始化一个链表描述符。

结构:

```
void lstInit
(
    FAST LIST *pList          /* ptr to list descriptor to be initialized */
)
```

输入参数: pList 为指向需要初始化的链表描述符结构。

返回: 无。

### 9.9.3 lstAdd 接口

功能: 添加一个节点到链表尾部。

结构:

```
void lstAdd
(
```

```
LIST *pList,           /* pointer to list descriptor */
NODE *pNode            /* pointer to node to be added */
)
```

输入参数：

- pList:指向需要添加新节点的链表描述符；
- pNode:指向新添加的节点结构。

返回：无。

#### 9.9.4 lstConcat 接口

功能：将源链表中的所有节点依次添加到目的链表的尾部。

结构：

```
void lstConcat
(
    FAST LIST *pDstList,      /* destination list */
    FAST LIST *pAddList      /* list to be added to dstList */
)
```

输入参数：

- pDstList: 目的链表描述符；
- pAddList: 源链表描述符。

返回：无。

#### 9.9.5 lstCount 接口

功能：获取当前链表中元素的个数。

结构：

```
int lstCount
(
    LIST *pList              /* pointer to list descriptor */
)
```

输入参数：pList 为指向需要进行元素个数统计的链表描述符。

返回：指定链表中元素的个数。

### 9.9.6 lstDelete 接口

功能：从指定的链表上删除指定的链表元素。

结构：

```
void lstDelete
(
    FAST LIST *pList,          /* pointer to list descriptor */
    FAST NODE *pNode          /* pointer to node to be deleted */
)
```

输入参数：

- pList：指向需要进行元素个数统计的链表描述符；
- pNode：指向需要删除的元素。

返回：无。

### 9.9.7 lstExtract 接口

功能：从指定的链表上分解得到一个子链表。

结构：

```
void lstExtract
(
    FAST LIST *pSrcList,       /* pointer to source list */
    FAST NODE *pStartNode,     /* first node in sublist to be extracted */
    FAST NODE *pEndNode,       /* last node in sublist to be extracted */
    FAST LIST *pDstList        /* ptr to list where to put extracted list */
)
```

输入参数：

- pSrcList：指向需要进行子链表分解的链表描述符；
- pStartNode：指向分解子链表的开始元素；
- pEndNode：指向分解子链表的结束元素；
- pDstList：指向分解得到的子链表的链表描述符。

返回：无。

### 9.9.8 lstFirst 接口

功能：获取指定链表的第一个元素。

结构：

```
NODE *lstFirst
(
    LIST *pList      /* pointer to list descriptor */
)
```

输入参数：pList 为指向链表描述符。

返回：无。

### 9.9.9 lstGet 接口

功能：从指定链表中取下第一个元素，并返回该元素结构地址。

结构：

```
NODE *lstGet
(
    FAST LIST *pList /* ptr to list from which to get node */
)
```

输入参数：pList 为指向链表描述符。

返回：获取的元素结构体指针。

### 9.9.10 lstInsert 接口

功能：将一个元素插入到指定链表的某个元素之后。

结构：

```
void lstInsert
(
    FAST LIST *pList,      /* pointer to list descriptor */
    FAST NODE *pPrev,      /* pointer to node after which to insert */
)
```

```
FAST NODE *pNode      /* pointer to node to be inserted */  
)
```

输入参数:

- pList: 指向链表描述符;
- pList: 指向链中某个元素的结构。新元素将被插入到该元素之后;
- pNode: 指向被插入的链表元素。

返回: 无。

### 9.9.11 lstLast 接口

功能: 获取指定链表的最后一个元素

结构:

```
NODE *lstLast  
(  
    LIST *pList      /* pointer to list descriptor */  
)
```

输入参数: pList 为指向链表描述符。

返回: 指向该链表中最后一个链表元素。

### 9.9.12 lstNext 接口

功能: 获取指定链表元素的下一个链表元素。

结构:

```
NODE *lstNext  
(  
    NODE *pNode      /* ptr to node whose successor is to be found */  
)
```

输入参数: pNode 为指向链表元素结构。

返回: 指向下一个链表元素。



### 9.9.13 lstNext 接口

功能：获取指定链表中第 n 个链表元素结构。

结构：

```
NODE *lstNth
(
    FAST LIST *pList,          /* pointer to list descriptor */
    FAST int nodenum           /* number of node to be found */
)
```

输入参数：

- pList：指向指定链表描述符；
- nodenum：表示获取第 n 个链表元素结构。

返回：获取得到的第 n 个链表元素结构。

### 9.9.14 lstPrevious 接口

功能：获取指定链表元素的前一个链表元素。

结构：

```
NODE *lstPrevious
(
    NODE *pNode               /* ptr to node whose predecessor is to be found */
)
```

输入参数：pNode 为指向链表元素结构。

返回：指向前一个链表元素。

### 9.9.15 lstNStep 接口

功能：获取距指定链表元素 nStep 距离的链表元素指针。

结构：

```
NODE *lstNStep
(
    FAST NODE *pNode,         /* the known node */
)
```

```
int nStep          /* number of steps away to find */  
)
```

输入参数:

- pNode: 指向链表元素结构;
- nStep: 指定获取的链表元素的距离。

功能: 获取距指定链表元素 nStep 距离的链表元素指针。

### 9.9.16 lstFind 接口

功能: 从指定链表中查找指定链表元素是否存在。

结构:

```
int lstFind  
(  
    LIST *pList,          /* list in which to search */  
    FAST NODE *pNode      /* pointer to node to search for */  
)
```

输入参数:

- pList: 指向查找的链表描述符。
- pNode: 指定需要查找的链表元素指针。

返回: 成功, 其他, 表示该链表元素在指定链表上的序号; 失败, NULL。

### 9.9.17 lstFree 接口

功能: 释放整个链表。

结构:

```
void lstFree  
(  
    LIST *pList          /* list for which to free all nodes */  
)
```

输入参数: pList 为指向查找的链表描述符。

返回: 无。

## 9.10 事件接口

头文件：eventLib.h

### 9.10.1 eventReceive 接口

功能：接收事件。

结构：

```
STATUS eventReceive
(
    UINT32 events,
    UINT8 options,
    int timeout,
    UINT32 *eventsReceived
)
```

返回：成功，OK；失败，ERROR。

输入参数：

- events：任务等待的事件；
- options：事件等待选项；
  - ✧ EVENTS\_WAIT\_ANY：所有等待的事件中，只要其中之一发生，则该函数被唤醒；
  - ✧ EVENTS\_WAIT\_ALL：只有所有等待的事件都已经发生，该函数才被唤醒；
  - ✧ EVENTS\_RETURN\_ALL：当函数被唤后，通过 eventsReceived 参数，将所有已经收到的事件返回给用户，并清空任务的事件表；
  - ✧ EVENTS\_KEEP\_UNWANTED：当函数被唤后，通过 eventsReceived 参数，将已经收到的事件中，属于参数 events 范围内的事件返回给用户，并清空任务事件表中该部分事件，保留不属于参数 events 范围内的事件；
  - ✧ EVENTS\_FETCH：当配置了该参数后，函数 events、timeout 参数，以及 options 参数的其他选项将被忽略，函数调用将立即返回，通过 eventsReceived 参数，将当前收到的所有事件返回给用户。
- timeout：函数等待事件的超时时间；
  - ✧ NO\_WAIT：函数等待，直接返回，即使是没有任何事件发生；

- ✧ WAIT\_FOREVER: 函数一直等待, 直到期望的事件发生;
- ✧ 其他整数: 函数等待的 tick 数。
- eventsReceived: 指向一个无符号整型变量, 用来保存函数返回的事件表。

### 9.10.2 eventSend 接口

功能: 发送事件。

结构:

```
STATUS eventSend
(
    int taskId,
    UINT32 events
)
```

输入参数:

- taskId: 发送事件的任务对象;
- events: 发送的事件列表。

返回: 成功, OK; 失败, ERROR。

### 9.10.3 eventClear 接口

功能: 清除当前任务的事件列表。

结构:

```
STATUS eventClear (void)
```

返回: 成功, OK; 失败, ERROR。

## 9.11 HOST 接口

头文件: hostLib.h

### 9.11.1 hostTblInit 接口

功能: 初始化 hostLib 函数接口需要的资源。

结构：

```
void hostTblInit (void)
```

### 9.11.2 hostAdd 接口

功能：向 hostTbl 表中添加主机名-ip 地址项。

结构：

```
STATUS hostAdd(  
    char *hostName,  
    char *hostAddr  
)
```

输入参数：

- hostName：主机名；
- hostAddr：主机地址。

返回：成功，OK；失败，ERROR。

### 9.11.3 hostDelete 接口

功能：从 hostTbl 表中删除主机名-ip 地址项。

结构：

```
STATUS hostDelete  
(  
    char *hostName,  
    char *hostAddr  
)
```

输入参数：

- hostName：主机名；
- hostAddr：主机地址。

返回：成功，OK；失败，ERROR。

### 9.11.4 hostGetByName 接口

功能：根据主机名查找 IP 地址。

结构：

```
int hostGetByName
(
    char *name          /* name of host */
)
```

输入参数：name 为主机名。

返回：成功，查找得到的 ip 地址；失败，ERROR。

### 9.11.5 hostGetByAddr 接口

功能：根据 IP 地址查找主机名。

结构：

```
STATUS hostGetByAddr
(
    int addr,           /* inet address of host */
    char *name          /* buffer to hold name */
)
```

输入参数：

- addr: 查找的 ip 地址；
- name: 指向 buffer，保存获取到的设备名称。

返回：成功，OK；失败，ERROR。

## 9.12 MUX 接口

头文件：muxLib.h

### 9.12.1 muxLibInit 接口

功能：初始化 muxLib。

结构:

```
STATUS muxLibInit (void)
```

返回: 成功, OK; 失败, NULL。

### 9.12.2 muxDevLoad 接口

功能: 加载网络驱动。

结构:

```
void * muxDevLoad
(
    int unit,                        /* unit number of device */
    END_OBJ * (*endLoad) (char*, void*), /* load function of the driver */
    char * pInitString,             /* init string for this driver */
    BOOL loaning,                   /* we loan buffers */
    void * pBSP                     /* for BSP group */
)
```

输入参数:

- unit: 设备编号;
- endLoad: 指向网络驱动 endLoad()接口, 用来加载设备;
- pInitString: 网络驱动的初始化字符串, 作为 endLoad()接口的参数;
- loaning: 未使用;
- pBSP: endLoad()接口的另一个参数。

返回: 成功, 加载的新设备的 cookie; 失败, NULL。

### 9.12.3 muxDevStart 接口

功能: 调用驱动中的 start()回调函数。

结构:

```
STATUS muxDevStart
(
    void * pCookie /* a pointer to cookie returned by muxDevLoad() */
)
```

输入参数：pCookie 为设备的 Cookie 号，一般由 muxDevLoad()函数返回。

返回：成功，OK；失败，ERROR。

### 9.12.4 muxBind 接口

功能：将网络协议栈与 END 设备绑定起来。

结构：

```
void * muxBind
(
    char * pName,           /* interface name, for example, ln, ei,... */
    int unit,              /* unit number */
    BOOL (*stackRcvRtn) (void*, long, M_BLK_ID, LL_HDR_INFO *, void*),
                           /* receive function to be called. */
    STATUS (*stackShutdownRtn) (void*, void*),
                           /* routine to call to shutdown the stack */
    STATUS (*stackTxRestartRtn) (void*, void*),
                           /* routine to tell the stack it can transmit */
    void (*stackErrorRtn) (END_OBJ*, END_ERR*, void*),
                           /* routine to call on an error. */
    long type,             /* protocol type from RFC1700 and many */
                           /* other sources (for example, 0x800 is IP) */
    char * pProtoName,     /* string name for protocol */
    void * pSpare          /* per protocol spare pointer */
)
```

输入参数：

- pName: end 设备名；
- unit: end 设备编号；
- stackRcvRtn: MUX 数据接收回调函数；
- stackShutdownRtn: MUX 关闭 end 设备回调函数。在调用 muxDevUnload()时，该函数将会调用。在关闭某个 end 设备前，应该向该设备上所有绑定的协议发送一个关闭消息；
- stackTxRestartRtn: 在调用 muxTxRestart()时，该函数将会被调用。用来通知 end 设备上的所有协议，重启之前被挂起的数据发送操作。比如，等待资源释放；
- stackErrorRtn: 当数据传输发生错误时将被调用，对错误进行处理；
- type: 绑定的协议类型；



- pProtoName: 设定绑定的协议名;
- pSpare: 指向一个协议栈定义的数据, 作为绑定回调函数的参数。

返回: 成功, 与绑定设备相关的 cookie; 失败, NULL。

### 9.12.5 muxSend 接口

功能: 通过指定 end 设备发送数据。

结构:

```
STATUS muxSend
(
    void* pCookie,          /* cookie that identifies a network interface */
                           /* (returned by muxBind()) */
    M_BLK_ID pNBuf          /* data to be sent */
)
```

输入参数:

- pCookie: end 设备 Cookie;
- pNBuf: 包含发送数据的 mBlk 指针。

返回: 成功, OK; 失败, ERROR。

### 9.12.6 muxReceive 接口

功能: 为 mux 提供的回调函数。

结构:

```
STATUS muxReceive
(
    void *   pCookie,       /* device identifier from driver's load routine */
    M_BLK_ID pMblk          /* buffer containing received frame */
)
```

输入参数:

- pCookie: end 设备 Cookie;
- pNBuf: 包含接收数据的 mBlk 指针。

返回: 成功, OK; 失败, ERROR。

### 9.12.7 muxIoctl 接口

功能：为 mux 提供的回调函数。

结构：

```
STATUS muxIoctl
(
    void * pCookie,      /* service/device binding from muxBind()/muxTkBind() */
    int    cmd,          /* command to pass to ioctl */
    caddr_t data /* data need for command in cmd */
)
```

输入参数：

- pCookie: end 设备 Cookie;
- cmd: 控制字命令;
- data: 控制字命令所带参数或者命令返回值。

返回：成功，OK；失败，ERROR。

### 9.12.8 muxUnbind 接口

功能：解除协议栈与 end 设备的绑定。

结构：

```
STATUS muxUnbind
(
    void * pCookie,      /* binding instance from muxBind() or muxTkBind() */
    long   type,         /* type passed to muxBind() or muxTkBind() call */
    FUNCPTR stackRcvRtn /* pointer to stack receive routine */
)
```

输入参数：

- pCookie: end 设备 Cookie;
- type: 解除绑定的协议类型;
- stackRcvRtn: 解除绑定的回调函数。

返回：成功，OK；失败，ERROR。

### 9.12.9 muxDevUnload 接口

功能：从 mux 层上卸载一个 end 设备。

结构：

```
STATUS muxDevUnload
(
    char * pName,          /* a string containing the name of the device */
                          /* for example, ln or ei */
    int    unit            /* the unit number */
)
```

输入参数：

- pName: end 设备名;
- unit: end 设备编号。

返回：成功，OK；失败，ERROR。

### 9.12.10 endFindByName 接口

功能：查找 END\_OBJ 结构。

结构：

```
END_OBJ * endFindByName
(
    char * pName,          /* device name to search for */
    int    unit
)
```

输入参数：

- pName: end 设备名;
- unit: end 设备编号。

返回：成功，返回获取的 END\_OBJ 结构指针；失败，ERROR。

### 9.12.11 muxDevExists 接口

功能：判断 end 设备是否存在。

结构:

```
BOOL muxDevExists
(
    char* pName,      /* string containing a device name (ln, ei, ...)*/
    int unit          /* unit number */
)
```

输入参数:

- pName: end 设备名;
- unit: end 设备编号。

返回: 成功, TRUE; 失败, FALSE。

## 9.13 中断接口

头文件: intArchLib.h

### 9.13.1 intLock 接口

功能: 关闭中断。

结构:

```
int intLock (void)
```

返回: 成功, 返回 intKey 值, 供 intUnlock 使用; 失败, ERROR。

### 9.13.2 intUnlock 接口

功能: 开中断。

结构:

```
int intUnlock
(
    int lockKey      /* lock-out key returned by preceding intLock() */
)
```

返回: 无。

输入参数：lockKey 为 intLock()调用的返回值。

### 9.13.3 intEnable 接口

功能：使能某个用户态中断。

结构：

```
int intEnable
(
    int level
)
```

输入参数：level 为使能用户态中断的中断号。

返回：无。

### 9.13.4 intDisable 接口

功能：关闭某个用户态中断。

结构:

```
int intDisable
(
    int level
)
```

输入参数：level 为使能用户态中断的中断号。

返回：无。

### 9.13.5 intConnect 接口

功能：挂接用户态中断处理函数。

结构：

```
STATUS intConnect
(
    VOIDFUNCPTR *vector, /* interrupt vector to attach to */
    VOIDFUNCPTR routine, /* routine to be called */
```

```
int parameter      /* parameter to be passed to routine */  
)
```

返回：成功，OK；失败，ERROR。

输入参数：

- vector：用户态中断的中断号；
- routine：挂接的用户态中断处理函数；
- vector：用户态中断处理函数的参数。

### 9.13.6 intLocalLock 接口

功能：关闭本核中断。

结构：

```
int intLocalLock (void)
```

返回：成功，返回 intKey 值，供 intLocalUnlock 使用；失败，ERROR。

### 9.13.7 intLocalUnlock 接口

功能：开本核中断。

结构：

```
int intLocalUnlock  
(  
    int lockKeys  
)
```

输入参数：lockKey 为 intLocalLock()调用的返回值。

返回：无。

## 9.14 异常钩子函数接口

头文件：excLib.h

### 9.14.1 excHookAdd 接口

功能：设置用户态异常处理钩子函数。

结构：

```
void excHookAdd
(
    FUNCPTR excepHook /* routine to call when exceptions occur */
)
```

输入参数：excepHook 为异常处理钩子函数。

返回：无。

## 9.15 网络库接口

头文件：ifLib.h

### 9.15.1 ifUnnumberedSet 接口

功能：设置 Unnumbered 网络接口。

结构：

```
STATUS ifUnnumberedSet
(
    char *pIfName,      /* Name of interface to configure */
    char *pDstIp,       /* Destination address of the point to point link */
    char *pBorrowedIp,  /* The borrowed IP address/router ID */
    char *pDstMac       /* Destination MAC address */
)
```

输入参数：

- pIfName：配置的网络接口名；
- pDstIp：对端使用的地址；
- pBorrowedIp：设置一个本端其他端口的 ip 为本端口的借用 ip；
- pDstMac：对端使用的 MAC 地址。

返回：成功，OK；失败，ERROR。

### 9.15.2 ifAddrAdd 接口

功能：为网络接口添加 IP 地址、广播地址、子网掩码。

结构：

```
STATUS ifAddrAdd
(
    char *interfaceName,      /* name of interface to configure */
    char *interfaceAddress,   /* Internet address to assign to interface */
    char *broadcastAddress,   /* broadcast address to assign to interface */
    int  subnetMask           /* subnetMask */
)
```

输入参数：

- interfaceName：网络接口名；
- interfaceAddress：添加的 ip 地址；
- broadcastAddress：添加的广播地址；
- subnetMask：添加的子网掩码。

返回：成功，OK；失败，ERROR。

### 9.15.3 ifAddrSet 接口

功能：为网络接口设置 ip 地址。

结构：

```
STATUS ifAddrSet
(
    char *interfaceName,      /* name of interface to configure, i.e. ei0 */
    char *interfaceAddress    /* Internet address to assign to interface */
)
```

输入参数：

- interfaceName：网络接口名；
- interfaceAddress：添加的 ip 地址。

返回：成功，OK；失败，ERROR。



#### 9.15.4 ifAddrDelete 接口

功能：删除指定接口上的指定 ip 地址。

结构：

```
STATUS ifAddrDelete
(
    char *interfaceName,      /* name of interface to delete addr from */
    char *interfaceAddress    /* Internet address to delete from interface */
)
```

输入参数：

- interfaceName：网络接口名；
- interfaceAddress：添加的 ip 地址。

返回：成功，OK；失败，ERROR。

#### 9.15.5 ifAddrGet 接口

功能：获取指定网络接口上的 ip 地址。

结构：

```
STATUS ifAddrGet
(
    char *interfaceName,      /* name of interface, i.e. ei0 */
    char *interfaceAddress    /* buffer for Internet address */
)
```

返回：成功，OK；失败，ERROR。

输入参数：

- interfaceName：网络接口名；
- interfaceAddress：指向事先分配的缓冲区，用来保存返回的接口地址。

#### 9.15.6 ifBroadcastSet 接口

功能：设置指定网络接口的广播地址。

结构：

```
STATUS ifBroadcastSet
(
    char *interfaceName,      /* name of interface to assign, i.e. ei0 */
    char *broadcastAddress    /* broadcast address to assign to interface */
)
```

输入参数：

- interfaceName：网络接口名；
- broadcastAddress：设置的广播地址。

返回：成功，OK；失败，ERROR。

### 9.15.7 ifBroadcastGet 接口

功能：设置指定网络接口的广播地址。

结构：

```
STATUS ifBroadcastGet
(
    char *interfaceName,      /* name of interface, i.e. ei0 */
    char *broadcastAddress    /* buffer for broadcast address */
)
```

输入参数：

- interfaceName：网络接口名；
- broadcastAddress：指向事先分配的缓冲区，用来保存返回的广播地址。

返回：成功，OK；失败，ERROR。

### 9.15.8 ifDstAddrSet 接口

功能：设置指定网络接口的对端地址。

结构：

```
STATUS ifDstAddrSet
(
    char *interfaceName,      /* name of interface to configure, i.e. ei0 */
    char *dstAddress          /* Internet address to assign to destination */
)
```

```
)
```

输入参数：

- **interfaceName**：网络接口名；
- **dstAddress**：设置的对端地址。

返回：成功，OK；失败，ERROR。

### 9.15.9 ifDstAddrGet 接口

功能：获取指定网络接口的对端地址。

结构：

```
STATUS ifDstAddrGet
(
    char *interfaceName,      /* name of interface, i.e. ei0 */
    char *dstAddress          /* buffer for destination address */
)
```

输入参数：

**interfaceName**：网络接口名；

**dstAddress**：指向事先分配的缓冲区，用来保存返回的对端地址。

返回：成功，OK；失败，ERROR。

### 9.15.10 ifMaskSet 接口

功能：设置指定接口的子网掩码。

结构：

```
STATUS ifMaskSet
(
    char *interfaceName,      /* name of interface to set mask for, i.e. ei0 */
    int netMask                /* subnet mask (e.g. 0xff000000) */
)
```

输入参数：

- **interfaceName**：网络接口名；

- netMask: 设置的子网掩码。

返回: 成功, OK; 失败, ERROR。

### 9.15.11 ifMaskGet 接口

功能: 获取指定接口的子网掩码。

结构:

```
STATUS ifMaskGet
(
    char *interfaceName,      /* name of interface, i.e. ei0 */
    int  *netMask             /* buffer for subnet mask */
)
```

输入参数:

- interfaceName: 网络接口名;
- netMask: 指向 buffer, 用来保存获取的子网掩码。

返回: 成功, OK; 失败, ERROR。

### 9.15.12 ifFlagChange 接口

功能: 设置指定网络接口的 flag 标志。

结构:

```
STATUS ifFlagChange
(
    char *interfaceName,      /* name of the network interface, i.e. ei0 */
    int  flags,               /* the flag to be changed */
    BOOL  on                  /* TRUE=turn on, FALSE=turn off */
)
```

返回: 成功, OK; 失败, ERROR。

输入参数:

- interfaceName: 网络接口名;
- flags: 设置的标志内容;
- on: 取以下指。

- ✧ TRUE: 将 flags 中的内容设置到指定的网络接口;
- ✧ FALSE: 将 flags 从指定的网络接口的标志中去除。

### 9.15.13 ifFlagSet 接口

功能: 将 flag 标志设置到指定网络接口。

结构:

```
STATUS ifFlagSet
(
    char *interfaceName,      /* name of the network interface, i.e. ei0 */
    int  flags                /* network flags */
)
```

输入参数:

- interfaceName: 网络接口名;
- flags: 设置的标志内容。

返回: 成功, OK; 失败, ERROR。

### 9.15.14 ifFlagGet 接口

功能: 获取指定网络接口上设置的标志。

结构:

```
STATUS ifFlagGet
(
    char *interfaceName,      /* name of the network interface, i.e. ei0 */
    int  *flags               /* network flags returned here */
)
```

输入参数:

- interfaceName: 网络接口名;
- flags: 指向 buffer, 用来保存获取的子网掩码。

返回: 成功, OK; 失败, ERROR。

### 9.15.15 ifunit 接口

功能：查找与设备相关的 ifnet 结构。

结构：

```
struct ifnet *ifunit
(
    register char *ifname    /* name of the interface */
)
```

输入参数：ifname 为网络接口名。

返回：成功，其他，获取的 ifnet 结构指针；失败，NULL。

### 9.15.16 ifNameToIfIndex 接口

功能：查找与设备相关的 ifnet 结构在 ifnet 链表中的 index。

结构：

```
unsigned short ifNameToIfIndex
(
    char * ifName    /* a string describing the full interface name. */
                    /* e.g., "fei0" */
)
```

返回：返回对于的 ifnet 结构在 ifnet 链表中的位置。

输入参数：ifname 为网络接口名。

### 9.15.17 ifIndexToIfName 接口

功能：查找设备的设备名。

结构：

```
STATUS ifIndexToIfName
(
    unsigned short ifIndex, /* Interface index */
    char * ifName    /* Where the name is to be stored */
)
```

输入参数：

- ifIndex：与设备相关的 ifnet 结构在 ifnet 链表中的位置；
- ifName：指向 buffer，存放获取的设备名。

返回：成功，OK；失败，ERROR。

### 9.15.18 ipAttach 接口

头文件：ipProto.h

功能：将 end 设备绑定到 tcp/ip 协议栈。

结构：

```
int ipAttach
(
    int unit,          /* Unit number */
    char *pDevice      /* Device name (i.e. ln, ei etc.). */
)
```

输入参数：

- unit：end 设备编号；
- unit：end 设备名。

返回：成功，OK；失败，ERROR。

### 9.15.19 ipDetach 接口

头文件：ipProto.h

功能：解除一个 end 设备与 tcp/ip 协议栈的绑定。

结构：

```
STATUS ipDetach
(
    int unit,          /* Unit number */
    char *pDevice      /* Device name (i.e. ln, ei etc.). */
)
```

输入参数：

- unit: end 设备编号;
- unit: end 设备名。

返回：成功，OK；失败，ERROR。

## 9.16 错误状态接口

头文件：errnoLib.h

### 9.16.1 errnoGet 接口

功能：获取当前任务的错误状态。

结构：

```
int errnoGet (void)
```

参数说明：无。

返回：当前任务的错误状态。

备注：当前任务将错误状态值保存在全局变量 `errno` 中，该接口直接返回全局变量 `errno` 的值。

### 9.16.2 errnoOfTaskGet 接口

功能：获取指定任务的错误状态。

结构：

```
int errnoOfTaskGet (int taskId)
```

参数说明：taskId 为获取错误状态的任务 id。

返回：ERROR，获取错误状态失败；其他，指定任务的错误状态。

错误码：S\_objLib\_OBJ\_ID\_ERROR：表示 taskId 无效，指定的任务不存在。

备注：当 taskId 为 NULL 时，该接口功能与 `errnoGet()` 一致。

### 9.16.3 errnoSet 接口

功能：设置当前任务的错误状态。

结构：



```
STATUS errnoSet(int errorValue)
```

参数说明：errorValue 为设置的错误状态值。

返回：OK，错误状态设置成功。

与 VxWorks 相应接口的差异：该接口对全局变量 `errno` 赋值，只返回 OK。

#### 9.16.4 errnoOfTaskSet 接口

功能：设置指定任务的错误状态。

结构：

```
STATUS errnoOfTaskSet
(
    int taskId,
    int errorValue
)
```

参数说明：

- taskId：设置错误状态的任务 id；
- errorValue：设置的错误状态值。

返回：ERROR，设置错误状态失败；OK，设置错误状态成功。

错误码：S\_objLib\_OBJ\_ID\_ERROR：表示 taskId 无效，指定的任务不存在。

备注：当 taskId 为 NULL 时，该接口功能与 `errnoSet()` 一致。

### 9.17 网络校验和接口

头文件：cksumLib.h

#### 9.17.1 checksum 接口

功能：计算网络校验和。

结构：

```
u_short    checksum
(
```

```
u_short * pAddr,  
int len  
)
```

输入参数：

- pAddr: 要进行校验的数据 buffer 起始地址；
- len: 要进行校验的数据长度。

返回值：校验和。

# 第 10 章

## CGEL4.x 与 CGEL3.x 差异

### 10.1 PROC 及 SYS 目录差异

#### 10.1.1 weight

```
/sys/class/net/eth0/weight
```

CGEL3.0 有此项，而 CGEL4.0 没有这项。

#### 10.1.2 stat

```
/proc/stat
```

CGEL4.0 新增字段名称及含义：

- guest: vcpu 的运行时间，与 kvm 虚拟机相关的概念，如未使用 kvm 虚拟机，无需关心该字段；
- softirq: 总的软中断次数和分 cpu 的软中断次数。

#### 10.1.3 meminfo

```
/proc/meminfo
```

CGEL4.0 新增字段名称及含义：

- Active(anon): 活动的匿名内存大小；

- Inactive(anon): 非活动的匿名内存大小;
- Active(file): 活动的文件映射内存大小;
- Inactive(file): 非活动的文件映射内存大小;
- Unevictable: 4.x 内核的新特性, 把 ramfs、SHM\_LOCK, VM\_LOCK 的页面加入 Unevictable LRU 链表, 避免 vmscan 时扫描这些不可回收的页面带来的耗时, 这里的 Unevictable 就表示具有上述三个特征的内存大小;
- Mlocked: 具有 VM\_LOCK 特征的内存大小;
- Shmem: 共享内存大小;
- KernelStack: 内核堆栈占用的内存大小;
- WritebackTmp: 回写时使用的临时 buffer 内存大小;
- HugePages\_Surp: 巨页池中最多能增加的巨页的大小。

#### 10.1.4 stat

```
/proc/<pid>/task/<tid>/stat
```

CGEL4.0 新增字段名称及含义:

- gtime: 进程在 vcpu 上的运行时间, 与 kvm 虚拟机相关的概念, 如未使用 kvm 虚拟机, 无需关心该字段;
- cgtime: 进程的子进程在 vcpu 上的运行时间, 与 kvm 虚拟机相关的概念, 如未使用 kvm 虚拟机, 无需关心该字段。

#### 10.1.5 smaps

```
/proc/<pid>/smaps
```

CGEL4.0 新增字段名称及含义:

- Pss: Pss 是 Proportion set size 的缩写, 指进程所拥有内存页的数量, 比如进程 A 有 1000 个私有内存页, 有 1000 个共享内存页是与进程 B 共享, 那进程 A 的 Pss 应该为  $1000+1000/2=1500$ ;
- Referenced: 进程拥有的 referenced 和 accessed 标志的页的数量;
- Swap: 进程拥有的交换页的数量;
- KernelPageSize: 内核页的大小, 一般的情况是 PAGE\_SIZE, 如果是巨页的话, 该值反映真实的巨页大小;
- MMUPageSize: MMU 支持的页的大小, 一般的情况是 PAGE\_SIZE, 如果 MMU 支持大的页, 返回实际的页大小。

### 10.1.6 status

```
/proc/<pid>/status
```

CGEL4.0 新增字段名称及含义：

- **CapBnd**：进程的权限边界集，属于访问控制模块的概念，是进程允许保留的权限，默认情况下权限边界集的所有位都是打开的。

### 10.1.7 sched

```
/proc/<pid>/task/<tid>/sched
```

CGEL4.0 新增字段名称及含义：

- **se.avg\_overlap**：该字段参考意义不大，统计不精确，高版本已经去除；
- **se.avg\_wakeup**：该字段参考意义不大，统计不精确，高版本已经去除；
- **se.avg\_running**：平均单次执行时间；
- **se.wait\_sum**：累计等待时长，公平调度考虑了该字段，实时调度未使用；
- **se.wait\_count**：累计等待次数，公平调度考虑了该字段，实时调度未使用；
- **se.iowait\_sum**：io 等待总时长；
- **se.iowait\_count**：io 等待次数；
- **sched\_info.bkl\_count**：进程持有大内核锁发生调度的次数；
- **se.nr\_migrations**：进程发生 CPU 间迁移的次数；
- **se.nr\_migrations\_cold**：内核只定义未使用该字段，默认为 0；
- **se.nr\_failed\_migrations\_affine**：进程设置了亲和性，迁移失败的次数；
- **se.nr\_failed\_migrations\_runnin**：进程处于 R 状态，迁移失败的次数；
- **se.nr\_failed\_migrations\_hot**：在 cache hot 的情况下，进程被迁移失败的次数；
- **se.nr\_forced\_migrations**：在 cache hot 的情况下，进程被强制迁移的次数；
- **se.nr\_wakeups**：被唤醒的次数；
- **se.nr\_wakeups\_sync**：被同步唤醒的次数；
- **se.nr\_wakeups\_migrate**：被唤醒时任务发生迁移的次数；
- **se.nr\_wakeups\_local**：被本地唤醒的次数；
- **se.nr\_wakeups\_remote**：非本地唤醒的次数；
- **se.nr\_wakeups\_affine**：考虑了任务的 cache 亲和性的唤醒次数，只有公平调度考虑了该字段，实时调度未考虑；

- `se.nr_wakeups_affine_attempts`: 尝试进行考虑了任务的 cache 亲和性的唤醒次数,, 只有公平调度考虑了该字段, 实时调度未考虑;
- `se.nr_wakeups_passive`: 内核只定义未使用该字段, 默认为 0;
- `se.nr_wakeups_idle`: 内核只定义未使用该字段, 默认为 0;
- `avg_atom`: 总的运行时间除以切换次数, 平均每次切换的运行时间;
- `avg_per_cpu`: 总的运行时间除以迁移次数, 平均每次迁移的运行时间;
- `nr_voluntary_switches`: 主动切换次数;
- `nr_involuntary_switches`: 被动切换次数。

## 10.2 API 差异

### 10.2.1 sched\_setaffinity

在 CGEL3.0 中此接口可以对所有进程 (线程, 内核线程) 设置 CPU 亲和性; 但是在 CGEL4.0 中, 此接口不能对内核线程设置 CPU 亲和性。CGEL4.0 使用自研的“智能迁移功能”来对内核线程设置 CPU 亲和性, 进行绑定。

# 第 11 章

## 开源 LICENSE 说明

### 11.1 GPL 简介

GPL 是 GNU 通用公共许可证的简称，为大多数的 GNU 程序和超过半数的自由软件所采用，Linux 内核就是基于 GPL 协议而开源。GPL 许可协议对在 GPL 条款下发布的程序以及基于程序的衍生著作的复制与发布行为提出了保留著作权标识及无担保声明、附上完整、相对应的机器可判读源码等较为严格的要求。GPL 规定，如果将程序做为独立的、个别的著作加以发布，可以不要求提供源码。但如果作为基于源程序所生著作的一部分而发布，就要求提供源码。

成都研究所 OS 平台基于开源社区研发提供适用于各产品线的嵌入式 Linux 产品及相关工具，从总体而言，主要应该遵循 GPL 许可协议的条款。Linux 是基于 GPL2.0 发布的开源软件，CGEL 是基于 Linux 内核进行修改完成的新作品，因此，根据 GPL 协议第 2 条的规定，CGEL 属于修改后的衍生作品，并且不属于例外情况。因此，CGEL 在发布时需开放源代码，具体形式需要符合 GPL 协议第 3 条的规定。

### 11.2 开发指导

为尽可能地保护公司在 Linux 方面的自有研发成果，特别是产品线的应用程序，同时顺从于 GPL 协议条款的规定，这里为大家提供一些开发指导原则供参考，以便有效地规避由于顺从 GPL 条款所带来的风险。

#### ■ 应用程序尽可能运行于用户态

应用程序尽量放到用户态运行，应用程序对于内核的访问、功能使用主要通过信号、数据、文件系统、系统调用，基本属于松耦合，而且绝大部分系统调用都是通过 C 库封装进行，能比较充分

地证明应用的独立性、可移植性，避免开源。

### ■ 以动态链接方式链接开源库

以 GLIBC 库为例，应用程序对于库的链接方式应尽量采用动态链接，这样可遵循 LGPL，避免公开源码。如果需要静态链接，则可以设计一个封装容器，与开源库静态链接在一起，这样只需开放封装容器源码。其余应用程序以动态链接方式与封装容器链接，只需要提供二进制文件。

### ■ 内核应用采用模块加载方式

内核应用可以采用静态链接、模块加载方式加入内。

静态链接方式是将所有的模块（包括应用）都编译到内核中，形成一个整体的内核映像文件。这种情况下，比较难以鉴别应用程序与内核的独立性、可移植性，按照 GPL 的规定，就很可能被要求开发源代码。

模块加载方式是指将上层应用独立地编译连接成标准 linux 所支持的模块文件（\*.mod）。运行时，首先启动 Linux 内核，然后通过 insmod 命令将各模块按照彼此间的依赖关系插入到内核。该方式下应用程序相对独立于 Linux 内核，有可能被证明为独立的作品，与 OS 无关，认为是纯粹的聚集行为，从而可以不用开放源码。

因此，对于运行于内核态的驱动和应用软件应尽量采用模块加载方式进行独立编译，通过模块的可动态装载、卸载，以及跨 OS 的可移植性来证明应用与 OS 的独立性。

## 11.3 源码获取方式

广东中兴新支点所发布的 CGEL 操作系统是基于开源软件 Linux 2.6 版本开发的，遵从 GPL 协议开放源代码。如果你需要源码，请发送源码申请邮件到 [cgel@gd-linux.com](mailto:cgel@gd-linux.com)。