The Linux Proc File System for Embedded Systems Concepts and Programing

Nicholas Mc Guire
Opentech.at
Hollabrunn, Austria
der.herr@hofr.at, http://www.opentech.at

Abstract

Since the late 0.99.X releases of the Linux kernel the proc file system is included in many GNU/Linux systems. This virtual file system interface allows both the inspection of kernel internal data structures and the manipulation of these data structures without the need of additional non-volatile memory. In embedded systems with tight resource constraints, the file system footprint is a key issue, in addition to the requirement for run-time optimisation. For such systems, utilising the capabilities of the proc file system and the related sysctl functions in order to provide kernel related administrative information via proc, as well as resource-optimised control interfaces, can substantially ameliorate embedded systems performance. A further, often ignored aspect, is that the proc and sysctl interface allow very precise tuning of access permissions, both increasing the security of the embedded system's administrative interfaces and improving diagnostic precision, which is essential for efficient error detection and analysis.

1 Introduction

The proc interface is a well-established and widely used interface in the Linux kernel; beginning with the late 0.99.X releases of the Linux kernel it has been part of the official kernel releases. First versions focused on network issues, but additional subsystems quickly began using proc files in order to simplify administrative and debug tasks. With the early releases the API was fairly complex, but as of Linux kernel 2.4.X, the API for the proc interface has become very user-friendly. The main feature of the proc file systems can be summarised as follows.

- Simple API,
- Direct access to kernel internals,
- Simple access via file system-abstraction,
- POSIX compliant open/read/write/close interface,
- Kernel level security setting on a file-scope.

In this article an introduction to using the proc interface specifically for embedded an real-time Linux is given. This work is part of an on-going GPL project for Keymile AG Vienna, Austria, where the proc interface is applied to a series of embedded Linux based high-bandwidth Internet access devices.

2 Proc Basics

Before going into specifics of building an interface using the proc file system, a basic concept overview of this special file system is given.

There is a tight relation between proc and sysctl functions. In general, all sysctl functions are also represented under /proc/sys/ as a proc file system entry. proc file system entries are not stored on a non-volatile media such as a hard-drive, they are generated on the fly, i.e. every time the read-method for the associated file is invoked. This results in a large freedom in the way output is represented to the user without requiring to parse complex input formats just to stay user-friendly. The proc file system is a file system in the sense that it provides an interface to the user-space that resembles a normal virtual file system interface of any other file system allowing POSIX style access via open, read, write and close.

Two basic interface types exist in proc, character based text-mode interfaces, and binary interfaces. Most interfaces are text-mode, and in cases where binary interfaces are used, usually both types are implemented at the same time. For user-space applications it is generally simpler to interface to the binary version rather than to text-mode, since in the latter case parsing (or at least scanning fixed format input lines) would be required. On the other hand, binary interfaces are not well suited for direct interpretation by humans. As an example `/proc/pci` and `/proc/bus/pci/devices` basically contain the same information, interpreted and raw, respectively.

In this paper, the representation of the `sysctl` tree in `/proc/sys` is treated as part of the `proc` file system, since `sysctl` handlers are not covered, but only the `proc` related functions.

2.1 Performance

The main reason to consider the `proc` interface is the performance of some standard Linux tools. Running applications like `top` or the ps-utilities on embedded systems show that these tools simply have too high CPU-demands for the system. Analysing these problems unveils:

- System calls are expensive but heavily used by some tools,
- The executables are large because they are providing more functionality than required,
- File system utilisation issues arise if many small tools are built (`buzybox` solves that problem partially),
- Not all desired information is accessible easily.

Let's look at some of these issues in more detail, as they could be relevant for the analysis of other performance bottlenecks on embedded systems.

2.1.1 System calls

System calls are the preferred, standardised and safe way to cross the kernel-user-space boundary. But they are expensive if heavily used. A simple `./hello_world` performs about 30 system calls, `echo "beep"` is up to 42 (numbers may vary slightly on different systems); this constitutes the bottom line for more or less any user space application. Looking at some of the typical administration tools such as `top` makes the situation even clearer. `top` takes up to a few thousand system calls to build the output for a single page (SuSE 8.0 standard installation), and the default is to update the content once per second. Hence on a reasonably reduced system `top` causes one thousand system calls per second to output a single page. But basically `top` is only collecting information stored entirely in the Linux task structure. Running through this task structure using the `proc read` method and outputting the result to the console in a `top` like manner only takes one additional system call to what `echo beep` does, i.e. approximately 43!

2.1.2 Optimised file operations possible

The general file system layer provides a POSIX style interface to the programmer via `open`, `read`, `write` and `close`. Data blocks are a general data abstraction, an approach, which is very flexible, but sub-optimal if the data is very specific and especially if the data amounts are rather small. The `proc` file system has a different approach. File operations are split in *file system specific* `open/release`, and not file system specific but *file specific* `read/write`, allowing to optimise not only with respect to performance but also with respect to data representation. In the examples given later, it can be seen how to register a specific `read/write` method that allows to present kernel internal or driver specific data structures in a formatted manner as well as to perform data interpretation within the `read/write` methods.

2.1.3 File system overhead

General purpose file system have a certain overhead, since management objects such as `inodes` and `superblocks` are required to interface to the operating system. Data blocks are discrete, leading to fragmentation effects. The `proc` file system can build application or problem specific data "'blocks"' and thus optimise the file system layer, minimising memory usage and file system overhead without loosing the advantage of a standardised interface. The drawback though is that the `proc` file system support code in the Linux kernel itself is fairly large. It only makes sense if it is providing sufficient utility to an embedded system. The question if the `proc` file system overhead pays off is fairly specific to the appliance, but most systems which can be found have it enabled per default.

2.1.4 Module size vs. User-space App

One issue related to the file system overhead mentioned above is the size of a user-space application required to achieve a comparable representation of kernel internal data-structures if dedicated `proc` files are not used. Such user-space applications not only require storage area on a file system, but also the associated libraries must be taken into account. Comparison between a `proc` version of `top` and the usual top program are given later. Generally speaking, a kernel-module will be fairly small. Most of the `proc` applications built ended up being smaller than a stripped "'hello-world"' using shared libraries! So the small file system overhead of storing the module is definitely a clear advantage of this approach. Special user-space application are not required for accessing the files in `proc`, since there is no need to parse or format data if the content of the `proc` files is already prepared in a user-friendly manner. The user-space can thus be satisfied with `cat` and `echo`, which are considered to be part of the base file system.

2.2 Portability

Whenever time and effort is invested into designing an embedded device, the question of portability to other potentially interesting OS/RTOS and hardware platforms arises. It may well be the most significant reason not to go for a `proc` based administrative interface, since `proc` must be generally considered very non-portable.

2.2.1 proc functions

Employing the `proc` file system interfaces for application specific administrative functionality is also the most significant disadvantage. The application is not portable to other embedded operating systems. The portability over different Linux supported architectures is very good though. This is an important issue, since anybody who tried to cross-compile "'simple"' utilities knows that having cross-platform portability is a serious development advantage.

2.2.2 Bound to kernel release

The `proc` file system may even be non-portable between different kernel releases as internal data-structures change quite often.

2.3 Security

A key concern to embedded systems is security, especially now where every system needs full network access over standard protocols. Two issues out of many should be emphasised here with respect to introducing a `proc` based interface:

- Introducing kernel code is always a potential risk,
- Utilising advanced security mechanism in kernel space can improve security a lot.

Security, as usual, depends largely on the know-how of the programmers. Linux is not a secure operation system *per se*; it is though an operating system that has the potential to be configured and to be used in a secure manner.

2.3.1 Modifying kernel code

The idea of kernel-space user-space separation always was that kernel code is validated and safe; but errors in kernel-space often are fatal to the system. On the other hand user-space is considered un-trusted; errors are fatal to the application but not to the system. Introducing kernel code potentially breaks this trusted-code concept. If a decision is made to introduce kernel code in a project, carrying out a security evaluation is required, which again requires that a security policy is available. Since the kernel is one flat address space and it is non pre-emptive in principal, deadlock prevention is up to the programmer.

2.3.2 Utilisation of kernel capabilities on a file scope

The last paragraph might suggest that introducing kernel code is in principal a bad idea. The reason why this may not be the case is that the security mechanisms available in the Linux kernel are quite potent but have not really made there way into the file system designs. Since `proc` declares file operations on a per file basis, these file operations can be designed much more restrictive than a generalised virtual file system interface. In addition, full utilisation of kernel capabilities is possible on a per file basis which can lead to clearly enhanced security capabilities. As a simple example consider taking away privileges even from the root-user.

3 Proc API vor Kernel 2.4.X

In this section an introduction to the kernels `proc` API is given. It is not limited to the `proc` specific functions for building and maintaining the `proc` files but also gives the most commonly associated functions to allow actually working with `proc`.

3.1 Proc core structures

The core `proc` data structures are presented in the following.

3.1.1 proc_dir_entry

The most important data structure is

```
struct proc_dir_entry {
   const char *name;
   mode_t      mode;
```

The name should be selected in a meaningful way and should not contain any special characters or blanks as this makes it difficult to access them from shell scripts, which is a very common access method. The proper selection of the mode bits is also important. Especially setting these carelessly on writable files can create serious security problems (see man stat for details on the flags defined in linux/stat.h).

```
struct file_operations *proc_fops;
```

The file operations structure proc_fops: these file operations are not on a file system scope like with usual file systems but on a file scope. The proc_fops are limited to read and write.

```
get_info_t *get_info;
```

The get_info method is a special read method that is not part of the file operations. It is more restrictive than the general read method.

```
struct module *owner;
```

The owner is used to associate modules with each other in order to prevent race conditions caused by module unloading. If the module name is set to THIS_MODULE (see linux/modules.h) then the kernel module is independent of all others and will unload if not in use. By setting it to a different module it will not be unloaded as long as this other module is loaded even if no common symbols are shared.

```
struct proc_dir_entry *next, *parent, *subdir;
```

Linked list of proc directories; the proc file system does not distinguish between regular files and directories directly, a regular file is everything that does not have any subdirectories, this is why directories and files are created with the same functions (the flags do differ though).

```
void *data;
```

Data returned by the proc read methods; this can be passed directly or assigned to a specific variable if only this should be returned (see create_proc_read_entry below).

```
   read_proc_t  *read_proc;
   write_proc_t *write_proc;
   ...
};
```

proc file operations; this is the generic read/write method interface; it is up to the programmer to make sure that these functions don't open any security holes into the system.

3.1.2 proc file operation

Although the proc file system integrates into POSIX environments, it does not define all possible file operations. User-defined file operations are limited to

- get_info_t get_info
- read_proc_t proc_read
- write_proc_t proc_write

whereby get_info is not part of the fops structure associated with the proc file system and does not follow POSIX read/write interface standard in its parameter list. The prototypes of these functions are declared in linux/proc_fs.h as

```
typedef int (read_proc_t)(
   char   *page,
   char  **start,
   off_t   off,
   int     count,
   int    *eof,
   void   *data);

typedef int (write_proc_t)(
   struct file   *file,
   const char    *buffer,
   unsigned long  count,
   void          *data);

typedef int (get_info_t)(
   char   *buffer,
   char  **start,
   off_t   offset,
   int     length);
```

3.2 General proc functions

The proc interface has a small and simple API. Unfortunately, it tends to change over time with kernel versions (at least with major releases, rarely with minor releases). The following list is the general proc API, it provides methods that have no format restriction on them, i.e. checking and validation of passed data is up to the programmer.

3.2.1 create_proc_entry

The function

```
struct proc_dir_entry *create_proc_entry(
   const char            *name,
   mode_t                 mode,
   struct proc_dir_entry *parent);
```

creates an entry in the proc file system with the string in the first field as file name. Generally it is a bad idea to have blanks in a filename; it is not forbidden but most UNIX-users don't expect blanks in file names. Thus, *file_name* is to be preferred over *file name*. For the mode bits see stat.h, for a description of the flags see man stat. The last argument is the directory in which the proc file is to appear, the value &proc_root is /proc itself.

After having created the entry in /proc the file operations for this file must be assigned by assigning the functions to the appropriate function pointers in the proc_dir_entry structure returned by the call to proc_dir_entry. Example:

```
struct proc_dir_entry *proc_file;

   proc_file = create_proc_entry(
      "file_name",
      S_IFREG | S_IWUSR,
      &proc_root);

   proc_top->read_proc=list_tasks;
```

Predefined directories in proc are proc_root (/proc), proc_root_driver (proc/drivers), proc_root_fs (/proc/fs), proc_net (/proc/net), proc_bus (/proc/bus).

3.2.2 create_proc_read_entry

The convenience function

```
static inline
struct proc_dir_entry * create_proc_read_entry(
   const char * name,
   mode_t mode,
   struct proc_dir_entry * base,
   read_proc_t * read_proc,
   void * data);
```

is basically a wrapper to proc_create_entry:

```
static inline
struct proc_dir_entry *create_proc_read_entry(
   const char            *name,
   mode_t                 mode,
   struct proc_dir_entry *base,
   read_proc_t           *read_proc,
   void                  *data)
{
   struct proc_dir_entry *res = create_proc_entry(
      name, mode, base);
   if (res) {
      res->read_proc=read_proc;
      res->data=data;
   }
   return res;
}
```

There is no principal difference in calling proc_create_read_entry and proc_create_entry plus the additional setups done explicitly.

3.2.3 create_proc_info_entry

The convenience function

```
static inline *create_proc_info_entry(
   const char *name,
   mode_t      mode,
   get_info_t *get_info);
```

is a wrapper to proc_create_entry again:

```
static inline
struct proc_dir_entry *create_proc_info_entry(
   const char            *name,
   mode_t                 mode,
   struct proc_dir_entry *base,
   get_info_t            *get_info)
{
   struct proc_dir_entry *res = create_proc_entry(
      name, mode, base);
   if (res) res->get_info=get_info;
   return res;
}
```

The get_info method itself is a special read method that is not a part of the fops (file operations), it is special in that it is bounded at create time to a defined length; this is visible in the get_info_t type.

3.2.4 proc_mkdir

This creates a directory in the proc file system. Before setting up the own directory in the top level /proc it should be considered putting the new entry into one of the available categories as people used to Linux would probably search there first. Generally, it's a bad idea to put things in proc_root as it is already fairly cluttered due to the PID directories.

```
extern struct proc_dir_entry *proc_mkdir(
   const char             *dir_name,
   struct proc_dir_entry *parent);
```

3.2.5 proc_symlink

This function creates a `symlink`, a symbolic link. The only real use of this function is for compatibility reasons to systems that are changing. One probably should not use symlinks when designing a new `proc` interface.

```
extern struct proc_dir_entry *proc_symlink(
   const char             *file_name,
   struct proc_dir_entry *parent,
   const char             *symlink_name);
```

3.2.6 proc_mknod

`proc_mknod` is used to create device special files below `/proc`. Basically it can be used just like `mknod` is used. The only real usage is that creating device files below `/proc` is convenient if the file system was a read-only file system (`romfs` or the like). Originally this seems to have been introduced in order to provide an initial console device via `/proc` during system startup, probably because `devfs` provides this in a cleaner way. For embedded systems enabling `devfs` is fairly expensive (`vmlinuz` size of increase of 11k), so this somewhat brutal substitution can be interesting for resource constraint systems.

```
extern struct proc_dir_entry *proc_mknod(
   const char             *name,
   mode_t                  mode,
   struct proc_dir_entry *parent,
   kdev_t                  rdev);
```

It is possible to create all device files for an embedded system below `/proc` and link `/dev` to `/proc`. This eliminates the file system overhead of `/dev` and allows to set the permissions tightly (it's not trivial to modify the permissions of these files even as root). Generally it's probably better to use `devfs` for dynamically created device files than to misuse `/proc`.

```
   if (register_chrdev(SIMPLE_MAJOR,
      "simple_dev", &simple_dev_fops) == 0)
   {
      printk("driver (major %d) registered\n",
         SIMPLE_MAJOR);

      /* create the device */
      mydevice = proc_mknod(
         "simple_dev", S_IFCHR | 0666,
         &proc_root, MKDEV(17, 0));
      if (mydevice == NULL) {
         ret=-ENODEV;
      }

      mydevice->owner = THIS_MODULE;
      return ret;
   }
```

Now the user-space can access this character device via `/proc/simple_dev`, just as it would via `/dev`. For embedded systems this might be a way to "'emulate"' `devfs` without requiring the full size overhead of `devfs`.

3.2.7 remove_proc_entry

A very nice way to get a system into trouble is to forget removing a `proc` file in the `cleanup_module` of a kernel module. So anything created with any of the calls above needs a `remove_proc_entry` in `cleanup_module`. The symptom of forgetting this is that `ls -l /proc` results in a segmentation fault.

```
extern void *remove_proc_entry(
   const char             *name,
   struct proc_dir_entry *parent);
```

`remove_proc_entry` returns void. There seems to be no simple way to detect failures of `remove_proc_entry`. `proc` entries have to be removed in reverse order to creation.

3.3 Subsystem specific wrapper functions

Some subsystems make heavy use of `/proc`. Therefore, some wrapper functions have been introduced to simplify programming.

3.3.1 proc_net_create

`proc_net_create` is a wrapper for creating an info entry routed at `/proc/net`. It is intended for the network subsystem. It is probably a bad idea to use it for anything else but the network subsystem.

```
static inline
struct proc_dir_entry *proc_net_create(
   const char *name,
   mode_t      mode,
   get_info_t *get_info)
{
   return create_proc_info_entry
      (name,mode,proc_net,get_info);
}
```

3.3.2 proc_net_remove

As to be expected this is a wrapper to remove_proc_entry for the networking subsystem.

```
static inline void proc_net_remove(
   const char *name)
{
   remove_proc_entry (name,proc_net);
}
```

3.4 /proc/sys Sysctl functions list

The sysctl related functions have type conversions integrated. So they provide the safer way of building a proc interface but more restricted. The type conversions are performed in a way that ensures that if incorrect types are passed (e.g. abc to proc_dointvec) then nothing is passed on at all. There is no error or warning though, so checking for invalid null data is left to the programmer. Note that the proc mirroring of sysctl table entries is a side effect of sysctl and not vice-versa. So one can disable the mirroring of any sysctl related setups by passing a NULL string in the procname field. Mode fields are valid for access via /proc as well as accessing via sysctl.

3.4.1 register_sysctl_table

register_sysctl_table registers sysctl names and the mapping to there associated functions via the ctl_tables. These ctl_tables are passed as NULL terminated arrays, and inserted at the sysctl_head passed.

```
struct ctl_table_header * register_sysctl_table(
   ctl_table  *table,
   int         insert_at_head);
```

The sysctl head is declared as:

```
   static struct ctl_table_header
      *somename_sysctl_header;
```

3.4.2 unregister_sysctl_table

In cleanup_module the sysctl functions need to be unregistered. To do this unregister_sysctl_table is called with the ctl_table_head.

```
void unregister_sysctl_table(
   struct ctl_table_header *table);
```

3.4.3 ctl_table

Before going on with the available proc related callback functions the ctl_table is introduced as it is the core structure used to build sysctl interfaces.

```
struct ctl_table
{
   int         ctl_name;
   const char *procname;
```

The ctl_name is an enumeration of the files in a given directory. If the integration of a ctl_table entry into an existing /proc/sys/* directory is intended, then it has to be ensured that there is no conflict with existing entries (see linux/sysctl.h for defined values). If new directories are created, then functions should simply be enumerated as shown later in the example for real time threads controlled via sysctl interface.

The procname is a string that will be used to represent this control function via the /proc/sys interface, if it should not be available via proc then a NULL string has to be passed here.

```
   void         *data;
   int           maxlen;
   mode_t        mode;
   ctl_table    *child;
   proc_handler *proc_handler;
```

data is a pointer to variable returned by the sysctl/proc read call. The data passed is limited to maxlen at compile time. So this interface is fairly restrictive, or inflexible compared to proc_read/proc_write methods, but allows for more security. The mode again is typical UNIX rwxrwxrwx and will be honored both via sysctl and /proc/sys access.

proc_handler is the pointer to the function of type proc_handler_t that is called to produce the data returned. In this handler further restrictions, being beyond usual UNIX rwx, can be imposed using kernel capabilities.

```
   ctl_handler            *strategy;
   struct proc_dir_entry  *de;
   void                   *extra1;
   void                   *extra2;
};
```

The elements ctl_handler is is not proc related. This is outside of the scope of this document and is mentioned here for completeness. The value of *de does NOT need to be set. This is taken care of by sysctl_register_table. The two void pointer entries are used for the minmax sysctl handlers to store the minimum and maximum arrays, respectively. These two pointers can be misused for anything!

The proc_handler_t type is defined in linux/sysctl.h to

```
typedef int proc_handler(
   ctl_table   *ctl,
   int          write,
   struct file *filp,
   void        *buffer,
   size_t      *lenp);
```

3.4.4 ctl_table hirarchy

The struct ctl_table_header is used to maintain dynamic lists of ctl_table trees. These trees are then "'translated"' to /proc/sys/ based directory structures.

```
struct ctl_table_header
{
   ctl_table        *ctl_table;
   struct list_head  ctl_entry;
};
```

An example of a file using a self-defined proc callback handler:

```
enum {
   DEV_SIMPLE_INFO=1,
   DEV_SIMPLE_DEBUG=2
};
...
/* files named "info" and "debug" */
ctl_table simple_table[] = {
   {DEV_SIMPLE_INFO, "info",
      &simple_sysctl_settings.info,
      INFO_STR_SIZE, 0444, NULL,
      &simple_sysctl_info},
   {DEV_SIMPLE_DEBUG, "debug",
      &simple_sysctl_settings.debug,
      sizeof(int), 0644, NULL,
      &simple_sysctl_handler},
   {0}};
```

Setup a simple sub-directory:

```
ctl_table simple_simple_table[] = {
   {DEV_SIMPLE_INFO, "simple", NULL,
      0, 0555, simple_table},
   {0}};
```

To create a directory below /proc/sys/dev in order to put the simple device related files into, a further table needs to be created. Checking in linux/sysctl.h gives:

```
/* CTL_DEV names: */
enum {
   DEV_CDROM   = 1,
   DEV_HWMON   = 2,
   DEV_PARPORT = 3,
   DEV_RAID    = 4,
   DEV_MAC_HID = 5
};
```

Even if the system might only show one directory (say cdrom) in /proc/sys/dev, the number to use cannot be chosen freely. This is a somewhat irritating problem. In the general proc interface, it's sufficient to chose a unique name for the directories and files; for sysctl interfaces it is up to the programmer to ensure that there are no conflicts with predefined names!

```
#define DEV_SIMPLE 6
...
ctl_table simple_simple_table[] = {
   {DEV_SIMPLE, "simple", NULL,
      0, 0555, simple_table},
   {0}};
```

The simple device directory is now put into /proc/sys/dev as this seems to be the appropriate place for a device related sysctl. CTL_DEV is defined in linux/sysctl.h again.

```
ctl_table simple_root_table[] = {
   {CTL_DEV, "dev", NULL,
      0, 0555, simple_simple_table},
   {0}};
```

If a new file should be created in /proc/sys, then a value that is not yet in use has to be selected. In most cases it should be possible to fit it into the existing structure, which should ensure that /proc/sys does not clutter up. As of kernel 2.4.19 the list is:

```
enum
{
  CTL_KERN = 1, /* General kernel info and control */
  CTL_VM   = 2, /* VM management */
  CTL_NET  = 3, /* Networking */
  CTL_PROC = 4, /* Process info */
  CTL_FS   = 5, /* file systems */
  CTL_DEBUG= 6, /* Debugging */
  CTL_DEV  = 7, /* Devices */
  CTL_BUS  = 8, /* Busses */
  CTL_ABI  = 9, /* Binary emulation */
  CTL_CPU  =10  /* CPU stuff (speed scaling, etc) */
};
```

3.5 basic proc_handlers

In many cases it is not necessary to write up a complex proc_handler callback function. The sysctl implementation provides a number of proc_handlers. They are fairly restrictive but on the other hand they are well tested. Before writing up proc callback handler, it should be checked if the task can't be done with one of these. In order to use one of the predefined handlers, a table entry has to be defined like:

```
proc_callback_type var[]=...;

ctl_table simple_table[] = {
   {ENUMERATION, "procname", &var,
      sizeof(var), UNIX_MODE, NULL,
      &proc_calback,...},
   {0}};
```

syctl can be used, or open/read/write/close on the file "'procname"', and it will be bounded in type by the callback function, and in size by the initial variable size. For many of the sysctl needed, this will be sufficient.

The predefined proc callback handlers all have the same prototype:

```
extern int proc_xxxxxxxxx(
   ctl_table   *table,
   int          direction,
   struct file *filep,
   void        *data_buffer,
   size_t      *lenth);
```

The integer direction is TRUE if this is a write to the sysctl table, FALSE other wise. The other values should be clear.

3.6 Predefined proc callbacks

Above proc callback handlers were introduced, being set up and mapped to some function. In many, if not most, cases the set of predefined proc callback handlers will be sufficient.

3.6.1 proc_dostring

read/write strings callback proc handler. If this callback receives non-string data, it simply will set the buffer to NULL.

```
static char somestring[]="the initial string";
...
ctl_table simple_table[] = {
   {DEV_SIMPLE_SOMESTRING, "somestring",
      &somestring, sizeof(somestring),
      0644, NULL, &proc_dostring},
   {0}};
```

Passing an oversized string by writing to the proc file will be truncated to sizeof(somestring) as set at compile time.

3.6.2 proc_dointvec

read/write a set of integer values to the file, the list of integers is white space separated. To use an integer array it is necessary to declare it; the proc_dointvec callback handler is used to read/write to it.

```
static int someintvec[]={0,0,0,0};
...
ctl_table simple_table[] = {
   {DEV_SIMPLE_SOMEINTVEC, "someintvec",
      &someintvec, sizeof(someintvec),
      0644, NULL, &proc_dointvec},
   {0}};
```

A somewhat special behavior that can be confusing with these handlers is their way of managing excess data elements. If the example above is taken as a reference, then cat /proc/sys/dev/simple/someintvec will originally return 0 0 0 0. If 1 2 3 4 is written to it with echo 1 2 3 4 > someintvec, then it will show 1 2 3 4; if echo 1 2 3 4 5 > someintvec is written to it again, then this will wrap around and will show 5 2 3 4. This can be quite confusing during debugging of sysctl entries. So sysctl arrays can be viewed as FIFOs with respect to there behavior on write.

3.6.3 proc_dointvec_bset

proc_dointvec_bset is a specially restricted version of proc_dointvec for setting of kernel capabilities (cap_bset). It is a good example of how to use a sysctl interface to set up access to security critical data structures in a simple but still safe way (see linux/kernel/sysctl.c for details). To protect this data structure not only the tight limits imposed by proc_dointvec are used, but also kernel capabilities:

```
int proc_dointvec_bset(
   ctl_table   *table,
   int          write,
   struct file *filp,
   void        *buffer,
   size_t      *lenp)
{
   if (!capable(CAP_SYS_MODULE)) {
      return -EPERM;
```

```
    }
    return do_proc_dointvec(
        table,write,filp,buffer,lenp,1,
        (current->pid == 1) ? OP_SET : OP_AND);
}
```

The assignment of the callback function is as expected (from linux/kernel/sysctl.c:

```
extern kernel_cap_t cap_bset;

ctl_table kernel_table[] = {
    ...
    {KERN_CAP_BSET, "cap-bound", &cap_bset,
        sizeof(kernel_cap_t), 0600, NULL,
        &proc_dointvec_bset},
    ...
    {0}};
```

It is no recommended to use proc_dointvec_bset for other variables. This should be seen as a sample implementation to build specific proc callback functions for security critical variables.

3.6.4 proc_dointvec_minmax

All the minmax variants of the proc callbacks use the values stored in table->extra1 and table->extra2 (min, max respectively) to verify that the passed data on write is in the permitted range. If not, then no change occurs (but there also is no warning message in the log files). The min/max values are valid for the corresponding elements in the integer vector passed, i.e if an integer array is passed with 2 elements but the min/max value is an "'array"' with only one value then the first element is bounded, the second is not bounded and can't be changed. To make both element bounded the min and max variables must both be of size two in the below example, if only duty_cycle_min where an array of size two then the second value still would not be changeable.

```
static int duty_cycle[]={50,45};
static int duty_cycle_min=10;
static int duty_cycle_max=90;
    ...
    {DEV_SIMPLE_DUTYCYCLE, "dyty_cycle",
        &duty_cycle, sizeof(int), 0644, NULL,
        &proc_dointvec_minmax, NULL, NULL,
        &duty_cycle_min, &duty_cycle_max},
```

In this example the second value can't be changed and the first is limited between 10 and 90. Setting the ctl_handler (the 8th parameter) to NULL makes this table accessible via proc but inaccessible via sysctl).

3.6.5 proc_dointvec_jiffies

This function treats the input as seconds and converts it to jiffies.

3.6.6 proc_doulongvec_minmax

Same as proc_dointvec_minmax, just for long integers not integers, and bounded.

3.6.7 proc_doulongvec_ms_jiffies_minmax

Same as proc_dointvec_minmax just for unsigned long integers passed, interpreted as milliseconds which are converted to jiffies.

3.7 Using regular Library Functions

System libraries (such as libc, libm, etc.) that are available to user-space programmers are unavailable to kernel programmers. When a process is being loaded, the loader will automatically load any dependent libraries into the address space of the process. None of this mechanism is available to kernel programmers. Libraries can be linked statically to kernel modules. This is in fact useful for math functions, but generally not a good thing to do. To statically include libm something like the Makefile entry below should be used.

```
my_mod.o: my_mod.c
    $(CC) ${INCLUDE} ${CFLAGS} -c -o tmp.o my_mod.c
    $(LD) -r -static tmp.o -o my_mod.o -L/usr/lib -lm
    rm -f tmp.o
```

Naturally this can have side effects, as library functions are not designed to run in kernel context. Therefore it has to be verified what the functions in the library included are doing.

The standard libc code can be used instead as basis for kernel re-implementation, as there might be significant problems with stack handling (the kernel is limited to a small amount of stack space, while user-space programs don't have this limitation) causing random memory corruption. Many of the commonly requested functions have already been implemented in the kernel, sometimes in "'lightweight"' versions that aren't as featureful as their user-land counterparts. Therefore, usage should not be based on the man-pages of the corresponding libc functions. The headers for any functions to be used can be "'grepped"' before writing kernel versions from scratch. If such functions are written, they should be contributed back to the Linux community. Some of the most commonly used ones are in include/linux/string.h.

Whenever a library function is needed, it should be considered in the design phase whether all the code can be moved into user-space instead, or to limit the functions to those available in the kernel. Generally modifying and adding to the kernel should be limited to the really necessary.

3.8 Kernel 'libc' Functions

The kernel internally available `libc` function set is limited to what kernel developers need and identified as so general that it was extracted into generally available functions. In general, they behave like the functions from the C-library. These are currently the memory and string functions:

3.8.1 memory functions

memcpy	memset	memmove
memscan	memcmp	memchr

3.8.2 string functions

strcpy	strncpy	strsep
strcat	strncat	strcmp
strncmp	strchr	strrchr
strlen	strstr	strtok
simple_strtol	strpbrk	sprintf

3.8.3 type-conversion functions

As noted in `linux/include/linux/ctype.h`. NOTE: This `ctype` does not handle `EOF` like the standard C library.

isalnum(c)	isalpha(c)	iscntrl(c)
isdigit(c)	isgraph(c)	islower(c)
isprint(c)	ispunct(c)	isspace(c)
isupper(c)	isxdigit(c)	isascii(c)
toascii(c)	tolower(c)	toupper(c)

Some of these are architecture specific, therefore all cannot be expected to be available on all platforms. In order to find out, the kernels symbol table can be inspected either by running `ksyms -a` or by checking the `System.map`. Aside from these `libc` functions in kernel space, any of the kernel internal functions provided there symbol exported can be used.

Of course the programmer is not limited to using these string functions even if `proc` input and output is a character basis and these are most commonly needed; any other exported kernel function can be used. Note though that many functions can have security relevant side-effects aside from the ability to hard-lock up the system if used incorrectly. Generally speaking, some sanity checks on any values users may pass have to be performed.

3.9 Related Kernel functions

The more common cases of kernel functions are listed here for convenience.

3.9.1 copy_from_user

`copy_from_user` is still widely used although it is actually only a function for backwards compatibility to 2.3 and 2.2 kernel releases. The function behind `copy_from_user` is `memcpy` (`memcpy_fromfs`). `memcpy` takes the same arguments as `copy_from_user` (`to`, `from`, `count`). See `/linux/compatmac.h` for more information.

3.9.2 MKDEV

`MKDEV` just wraps up a major and minor number to create a `kdev_t` type, a "'device file"' hook in the kernel. It is declared in `linux/kdev_t.h` along with a number of further useful macros to extract major and minor number from `kdev_t` types.

3.9.3 SET_MODULE_OWNER

If the `proc` functionality of a application is put into a separate kernel module then this module often needs to be protected against race conditions that occur during unloading of stacked kernel modules. To associate a `proc` file entry with the network subsystem the net structure can be grabbed via the device and assigned to the module owner with the `SET_MODULE_OWNER` macro (`linux/modules.h`. If no association is necessary, i.e. the module may be unloaded independently of any other modules, then the module owner is set to `THIS_MODULE` (also declared in `linux/modules.h`.

```
struct net_device       *net;
...
net = &dev->net;
SET_MODULE_OWNER (net);
```

4 Managment Interfaces via proc

On many embedded systems the user space is primarily responsible to provide a dedicated HMI (Human-Machine-Interface), an administrative interface, and rarely a full-fledged user space as expected on a regular Linux desk-top system. The HMI in most cases is very application specific, but at the same time utilises typical user space interfaces (libs, device files, etc.). The administrative interface on the other hand will require tools to inspect system status that are Linux specific and fairly independent of the specific application.

4.1 Available tools

During Linux development a large variety of administrative tools has evolved. Many of these are distribution specific (`linuxconf`, `yast`, etc.) and are of no interest for embedded systems as they are tailored to the demands of the distribution and not really adaptable to the needs of a dedicated device. They are too heavy-weighted for most embedded platforms. A large set of distribution independent administrative tools is also available, many of which integrate into the POSIX2 specs (`ifconfig`, `route`, etc.). But some of these tools, notably those that relied on the `proc` file system are too heavy weight for embedded systems due to the large number of system calls required to access information in `/proc`. Some typical examples of this overhead are listed below:

Command	Options	Nr of syscalls
sysctl	-a	2750
top	-n1	1350
who	-u	174
route		135
ifconfig		88

Number of system calls on a normal desktop for some typical admin tools. Results naturally will vary on different systems.

Looking at the tools in detail the overhead is produced due to a few factors:

- Highly configurable,
- Large number of possible options,
- General-purpose, that is, application independent interface,
- CPU usage is not a key issue during design of these apps.

The first three issues listed might not seem like disadvantages if they did not have a dramatic influence on the resource demands - but this is not quite correct. The issues of configurability and abundance of runtime options is critical, and any product-hot-line will be able to tell this, because the maintenance personnel often will not be trained to a level to manage all of these options leading to misinterpretation and consequent errors. The goal of an administrative interface is to provide all necessary data to the administrative personnel in a well-documented manner and with a minimum complexity. Taking the above list - the admin interface demands can be sketched out as

- Minimum set of required options
- Configured to the needs of the specific appliance
- Highly application specific especially with respect to error messages
- CPU usage and data representation is a key issue.

What does this all have to do with the `proc` interface? Administrative interfaces primarily are monitoring system operations, user space tasks are looked at from the kernels perspective, i.e. administration interfaces are primarily interested in kernel-space data-structures and thus it is very expensive to put these administrative interfaces in user space.

4.2 /proc/top a comparison

As a comparison a top-like interface is presented that has de-facto zero-configurability, runs in kernel-space, has a minimum file system foot-print and a minimum runtime-resource demand and at the same time outputs exactly what is needed in a top-like manner. Naturally the 'exactly what is needed' will vary but the variance can normally be set at compile time and need not be runtime-configurable (although that is possible even with this approach).

4.2.1 Comparing resources

To give an idea of what resource advantage such a dedicated proc interface may provide here is a (somewhat unfair) comparison between standard `top` and a `/proc/top`.

- top - libncurses 289k stripped
- top - file system size 350k (54k /usr/bin/top + 298k for the libs)
- top - number of system calls is approximately 1500 to produce a single page of output
- proc_top - no libs
- proc_top - file system size 2k (2028 bytes on linux-2.4.20)

- proc_top - number of system calls 43 (one more than echo takes)

From this comparison it seems fairly clear that at least in some areas building dedicated /proc interface for embedded systems really can pay-off the effort invested.

4.2.2 proc_top.c

/proc/top
simply run through the task list of linux for a light weight "top" cat /proc/top to get a list of PID, NICE, UserTime, SYStemTime and Command, this should be enough for most embedded systems . /proc/ifconfig display network information in the form you would expect from ifconfig.

```
#include <linux/kernel.h>  /* printk level */
#include <linux/module.h>  /* kernel version etc. */
#include <linux/proc_fs.h> /* all the proc stuff */
#include <linux/fs.h>
#include <linux/errno.h>

/* don't forget to make it GPL...*/
MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Der Herr Hofrat");
MODULE_DESCRIPTION("Embedded top");

struct proc_dir_entry *proc_top;

int
list_tasks(char *page,
   char **start,
   off_t  off,
   int    count,
   int   *eof,
   void  *data)
{
   int size = 0;
   struct task_struct *p;
   char state;
   size+=sprintf(page+size,
      "%5s%7s%7s%7s%7s%7s%7s  %s\n\n",
      "PID","UID","PRIO","NICE",
      "STATE","USERt","SYSt","COMMAND");
}
```

Admittedly, this is not very elegant code, but the issue is too simple to allow for elegant solutions. Therefore simply the tasklist_lock is grabbed and the task list walked through, printing data of interest. The only datum that needs some interpretation is task-state as the numeric values would not tell anybody anything. Basically the goal is to have output that will be close enough to the output of top to allow administrators to interpret it properly.

```
   read_lock(&tasklist_lock);
   for_each_task(p){
      switch((int)p->state){
         case -1: state='Z'; break;
         case 0: state='R'; break;
         default: state='S'; break;
      }
      size+=sprintf(page+size,
         "%5d%7d%7d%7d%7c%7d%7d  %s\n",
         (int)p->pid,
         (int)p->uid,
         (int)p->rt_priority,
         (int)p->nice,
         state,
         (int)p->times.tms_utime,
         (int)p->times.tms_stime,
         p->comm);
   }
   read_unlock(&tasklist_lock);
   return (size);
}
```

init_module and cleanup_module just need to take care of setting up and deleting the proc file system entry.

```
int
init_module(void)
{
   proc_top = create_proc_entry("top",
      S_IFREG | S_IWUSR,
      &proc_root);
   proc_top->read_proc = list_tasks;
   return 0;
}

void
cleanup_module(void)
{
   remove_proc_entry("top", &proc_root);
   printk("out of here\n");
}
```

By invoking cat /proc/top the typical output would be something like:

```
  PID    UID   PRIO   NICE  STATE  USERt   SYSt  COMMAND

    1      0      0      0      S      1   5848  init
    2      0      0      0      S      0      0  keventd
    3      0      0     19      S      0      3  ksoftirqd_CPU0
    4      0      0      0      S      0      0  kswapd
```

```
   5       0       0       0       S       0       0   bdflush
 ....
 661       0       0       0       S      18       7   sshd
 662       0       0       0       S      24      12   bash
 671       0       0       0       R       0       2   cat
```

Fig.1 output of `cat /proc/top`

4.2.3 proc_ifconfig.c

The second typical admin tool that is introduced here is a proc based version of ifconfig. The motivation for this was that busybox' ifconfig, by default, allows setting of network parameters but not displaying them. So an "'inexpensive"' way of displaying the settings in a human-readable form was anticipated. Basically all of the output produced by this proc_ifconfig.o module via cat /proc/ifconfig could be extracted from all ready available proc file system entries. But it would hardly be reasonable to assume that untrained personnel would be very happy with the output of /proc/net/* if they are trying to locate a network problem. Therefore, the goal is to mimic the regular output of ifconfig.

```
extern struct
net_device *dev_get_by_index(int idx);

struct proc_dir_entry *proc_ifcfg;

int
list_netdev(char *page,
   char **start,
   off_t off,
   int count,
   int *eof,
   void *data)
{
   int size = 0;
   int type = 0;
   char *hw_types[]={"Ethernet",
      "Local Loopback",
      "Other"};
   struct net_device *dev;
   unsigned long addr;
   unsigned long bcast;
   unsigned long mask;
   /* netdevices start counting at 1 */
   int i=1;

   while((dev=dev_get_by_index(i)) != NULL){
      struct net_device_stats *stats =
      (dev->get_stats ? dev->get_stats(dev): NULL);
      struct in_device *in_dev = dev->ip_ptr;
      addr=in_dev->ifa_list->ifa_address;
      bcast=in_dev->ifa_list->ifa_broadcast;
      mask=in_dev->ifa_list->ifa_mask;

     /* only check ethernet and
      * loopback for now
      */
      switch(dev->type){
         case 1: type=0; break;
         case 772: type=1; break;
         default: type=2; break;
      }
/*  other code */
   }
}
```

The work is done in the top part of the loop above. All that needs to be done is to grab the appropriate pointers to some kernel structures of interest and then run through the list of network devices. The actual sprintf code, an endless long list of structure elements from the different network related core structures, is not shown here.

```
      size+=sprintf(page+size,....
         ...
         dev->irq,
         dev->base_addr&0x0ffff);
      i++;
   }
   return (size);
}
```

Finally the mandatory init_module/cleanup_moduleis shown in the following.

```
int
init_module(void)
{
   proc_ifcfg=create_proc_entry(
      "ifconfig",
      S_IFREG | S_IWUSR,
      &proc_root);
   proc_ifcfg->read_proc=list_netdev;
   return 0;
}

void
cleanup_module(void)
{
   remove_proc_entry("ifconfig",
      &proc_root);
}
```

The output of a cat /proc/top Fig.2 call is fairly close to what one would expect from calling /sbin/ifconfig Fig.3.

```
lo    Link encap:Local Loopback HWaddr 00:00:00:00:00:00
      inetd addr:127.0.0.1 Bcast:0.0.0.0 Mask:255.0.0.0
      MTU:16436
      RX packets:38 errors:0 dropped:0 overruns:0 frame:0
      TX packets:38 errors:0 dropped:0 overruns:0 carier:0
      collisions:0 txqueuelen:0
      Interrupt:0 Base address:0

eth0  Link encap:Ethernet HWaddr 00:02:b3:2c:9a:d2
      inetd addr:192.168.1.31 Bcast:192.168.1.255 Mask:255.255.255.0
      MTU:1500
      RX packets:7130 errors:0 dropped:0 overruns:0 frame:0
      TX packets:3801 errors:0 dropped:0 overruns:0 carier:0
      collisions:0 txqueuelen:100
      Interrupt:10 Base address:b000
```

Fig.2 output of `cat /proc/ifconfig`

```
eth0      Link encap:Ethernet  HWaddr 00:02:B3:2C:9A:D2
          inet addr:192.168.1.31  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8466 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4540 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          Interrupt:10 Base address:0xb000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:38 errors:0 dropped:0 overruns:0 frame:0
          TX packets:38 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
```

Fig.3 output of `sbin/ifconfig`

For trouble shooting it is fairly simply to take apart the details of the different errors that `/sbin/ifconfig` ads up, or add other network related values of interest.

5 RT-Interfaces via proc

When setting up a real-time task there are a number of issues where using the `proc` file system can help. Notably starting or stopping rt-threads, reporting status of the rt-system, or rt-applications as well as some of the security issues related to managing rt-threads.

5.1 Task control via /proc

Inserting modules requires root privileges. When setting up an embedded system with RTLinux then commonly some way to launch an rt-thread is neded without giving the operator root privileges. Setting the `SETUID` bit for `insmod` is a unacceptably insecure way, as this would allow inserting a trivial module to gain full control of the system. A common method used is to insert the rt-modules at system startup and have the application modules loaded in an inactive state. Later on, a unprivileged user starts the rt-thread by sending a start command via real-time FIFO, but this does requires to give the `/dev/rtf#` write access for unprivileged users, thus also opening some potential problems. An alternative is to use a `/proc` file and protect these files via kernel capabilities if needed. The advantage of the proc based solution is that the read/write methods are file specific and not file system specific or tied to the major number of a device with access control restricted to virtual file systems capabilities, which are generally insufficient. These file specific file operations allow very restricted access to kernel space. File operations for proc files not only map to a very specific read/write method but also have statically, compile time defined, virtual file systems permissions preventing runtime modifications, and allow a very application specific check of passed data.

```
pthread_t thread;
hrtime_t start_nsec;

static int running=1;
struct proc_dir_entry *proc_th_stat;
```

This rt-thread is launched on insmod (running is initialised to 1) and stops by exiting the while(running) loop when running is set to 0 via /proc/thread_status, it also allows monitoring the status of this thread by inspecting the /proc/thread_status simply by running cat /proc/thread_status.

```
void *
start_routine(void *arg)
{
   int i=0;
   struct sched_param p;
   hrtime_t elapsed_time,now;
   p . sched_priority = 1;
   pthread_setschedparam(pthread_self(),
      SCHED_FIFO, &p);

   pthread_make_periodic_np(pthread_self(),
      gethrtime(), 500000000);

   while (running) {
      pthread_wait_np ();
      now = clock_gethrtime(CLOCK_REAL-TIME);
      elapsed_time=now-start_nsec;
      rtl_printf("elapsed_time = %Ld\n",
         (long long)elapsed_time);
      i++;
   }
   return (void *)i;
}
```

One of the nice things about the proc files being generated on the fly is that the read method can output the values in a nice user-friendly manner while the write method does not need to bother with any parsing as would be required with a configuration file.

```
int
get_status(char *page, char **start,
   off_t off, int count, int *eof,
   void *data)
{
   int size = 0;

   MOD_INC_USE_COUNT;

   size+=sprintf(page+size,"Thread State:%d\n",
      (int)running);

   MOD_DEC_USE_COUNT;
   return(size);
}
```

As the proc interface receives character input, one needs to convert input values to the appropriate internal data types. In this example a brute-force atoi is done, which also only takes the first passed character into account. Generally one needs to ensure that ANY write method in proc checks data passed to not open security holes in the kernel.

```
static int
set_status(struct file *file,
   const char *user_buffer,
   unsigned long count,
   void *data)
{
   MOD_INC_USE_COUNT;

   /* brute force atoi */
   running=(int)*user_buffer-'0';

   MOD_DEC_USE_COUNT;
   return count;
}

int init_module(void) {
   int retval;
   start_nsec=clock_gethrtime( CLOCK_REAL-TIME);
   retval = pthread_create( &thread, NULL,
      start_routine, 0);
   if(retval){
      printk("pthread create failed\n");
      return -1;
   }
   proc_th_stat=create_proc_entry( "thread_status",
      S_IFREG | S_IWUSR, &proc_root);

   /* the file specific operations */
   proc_th_stat->read_proc=get_status;
   proc_th_stat->write_proc=set_status;
   return 0;
}

void cleanup_module(void) {
   void * ret_val;
   pthread_cancel(thread);
```

```
    pthread_join( thread, &ret_val);
    printk("Thread terminated (%d)\n",
       (int)ret_val);
    remove_proc_entry("thread_status",
       &proc_root);
}
```

5.2 Exporting RTLinux-internals via /proc

A critical issue for real-time systems is the ability to monitor status of the system with a minimum overhead. Periodically logging to the system logs is one of the possibilities. This is somewhat limited though as the data-volume would become very large and it is often hard to say a-priori what values are going to be relevant for monitoring. Therefore periodic monitoring needs to by adjustable. To make it adjustable a large spectrum of kernel/rt internal values must be reachable with low processing overhead. For this purpose the `proc` and `sysctl` interfaces are clearly a most suitable approach. The current `/proc` file system gives a snap shot of the status of the kernel. But more important for systems that need to exhibit fault-tolerance qualities is the analysis of system tendencies. Roughly this means that the developments of values are more important than the values themselves. With the current concept behind `/proc` there are two possibilities.

- Save status locally and periodically compare it to current values,
- Log status to a remote system and leave complex, and computational intensive, work to a appropriately powerful server system.

With the limited resources of embedded system the first option more or less is not suitable as it would potentially request log or analysis related processing efforts at the same time that the system is in a high load situation due to error handling. Thus the data needs to be analysed as far as possible at low-load situations. This can be best achieved by delegating the data interpretation to the system's idle task. In order to minimise processing overhead this task is performed in kernel-space and the results are then presented via `sysctl` or `proc`.

Here is an example of making RTLinux internal data available by simply dumping the `hrtime` variable to user-space via `/proc/hrtime`. This allows user-space applications direct access to RTLinux internal data structures via `open/read/close` on `proc` files or as shown here make it available in a "‘formated"’ way to allow use of `cat /proc/hrtime` to read the RTLinux internal clock.

```
/* /proc/hrtime "file-descriptor"
 */
struct proc_dir_entry *proc_hrtime;

/* /proc/hrtime read method - just
 * dump the dynamic syscall number
 * in a human readable manner
 */
int
dump_stuff(char *page, char **start,
   off_t off, int count, int *eof,
   void *data)
{
   int size = 0;
   MOD_INC_USE_COUNT;

   size+=sprintf(page+size,"RT-Time:%llu\n",
      (unsigned long long)gethrtime());

   MOD_DEC_USE_COUNT;
   return(size);
}

int
init_module(void)
{
   /* set up a proc file in /proc */
   proc_hrtime=create_proc_entry("hrtime",
        S_IFREG | S_IWUSR, &proc_root);

   /* assign the read method of
    * /proc/hrtime to dump the number
    */
   proc_hrtime->read_proc=dump_stuff;
   return 0;
}

void
cleanup_module(void)
{
   /* remove the proc entry */
   remove_proc_entry("hrtime", &proc_root);
}
```

5.3 Security Issues

There are some general security issues involved with modules. Commonly on embedded systems, everything is statically compiled into the kernel to eliminate the problem of requiring privileges to load modules at runtime. In cases where this is not possible – and RTLinux is one of them – some way to permit usage of dynamically loaded kernel modules in a safe way is needed. For RTLinux a common strategy is to load all RTLinux modules at system startup time (RTLinux core modules + application specific modules), and have the application specific modules in an inactive state (suspended). This way the only thing left to do is to start or stop the rt-threads, which can be done safely via a `proc` interface.

6 Conclusion

Standard tools are designed for desktop systems/server-systems and are too heavy weighted for embedded systems. A large amount of the administrative tasks comprises inspecting kernel internal structures. Using the `proc` interface of the Linux kernel lightweight variants of standard admin tools can be built.

Embedded systems have a higher security demand than standard desktop systems as they must operate in a very autonomous fashion. They have a higher demand on the monitoring of kernel internals to allow reacting to arising problems on time. To do this, a safe and light-wait method for accessing kernel/rt-context internal structures is necessary. The `proc` interface is both capable of providing the necessary secure access as well as providing the basic functions to allow human-readable output via `/proc` files.

7 Acknowledgement

This project is part of ongoing development work for Keymile AG, Vienna `http://www.keymile.com` as part of design of a new generation of telecom-access devices. This GPL project has been made available at `http://www.opentech.at/projec` in order to develop a proc based embedded utility set. This project is also available via cvs, `cvs -d :pserver:anoncvsopente` `login` (password: `anoncvs`), `cvs -d :pserver:anoncvsopentech.at:/home/gpl co proc_utils`. Feedback to `der.herrhofr.at` is always appreciated.

References

[GNU] GNU not UNIX,
`http://www.gnu.org/`, `ftp://ftp.gnu.org/`.

[linux] Linux Kernel Home-Page,
`http://www.kernel.org/`, `ftp://ftp.kernel.org/`.

[rtlinux] RTLinux/GPL Home-Page,
`http://www.rtlinux.org/`, `ftp://ftp.rtlinux.at/`.

[proc-howto] The proc-howto,
Terrehon Bowden `terrehonpacbell.net`, Bodo Bauer `bbricochet.net`,
Jorge Nerin `comandantezaralinux.com`,
`Documentations/proc.txt`, `/linux/Documentation/filesystems/proc.txt`.

[LDD] Allesandro Rubini, Linux Device Drivers,
O'Reilly & Associates, `http://www.xml.com/ldd/chapter/book/index.html`