Reprinted from the

Proceedings of the Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.

Incrementally Improving the Linux SCSI Subsystem

James E.J. Bottomley
SteelEye Technology, Inc.
http://www.steeleye.com
James.Bottomley@steeleye.com

Abstract

This paper tackles two issues in the current SCSI subsystem: init time probing using the new hotplug infrastructure and improvements to the current error handler. We also include an appendix sketching the operation of the SCSI subsystem and another one listing other outstanding problems not covered in this paper.

1 Introduction

Obviously, the scope of potential incremental improvements to the SCSI subsystem is enormous. In order to narrow the field quite a bit, we will concentrate on just two particular examples.

1.1 Device Scanning and Inquiry

The first addresses the current weaknesses in the device probing and inquiry code. As things stand today, the SCSI subsystem will scan all targets (up to 15) and, depending on compile and run time variables, try to scan all LUNs on those targets. There is also a compiled in exception table stored in `scsi_scan.c` which can cope with the idiosyncrasies of certain devices. The principle disadvantages of this system are

1. It is extremely inflexible and rigid. New devices that need exceptions have to be patched into the kernel table and the kernel recompiled before it does the correct thing, and
2. The exception table is cumbersome and does not cover all cases. For example, how certain devices are probed can depend on the SCSI host adaptor they are attached to (mercifully, this is becoming extremely rare).

It is the thesis of this paper that such complex rules based logic should be abstracted entirely from the kernel and placed in user land, where it can easily be altered and extended without even rebooting; and furthermore, that such a system can be grafted on to the existing code fairly easily.

1.2 Fixing the Error Handler

This topic is also broad, particularly as there are several bio related errors in the 2.5 series kernel that make error handling especially problematic (see appendix B). However, the

object here is to concentrate on the SCSI specific region of code in `scsi_error.c` and assume that the kinks in the bio system will be worked out as the kernel evolves.

The essential problem in `scsi_unjam_host()` is that it tends to execute on one command at once, and have limited facilities for understanding that error recovery on one command may affect a large numbers of others. This is particularly acute when drives start misbehaving because they usually have the maximum number of commands queued when error recovery begins. Our presentation will be essentially to clean up the error handler actions and to restart command execution slowly and gently (throttling) instead of slamming an entire set of outstanding commands down again.

1.3 A Historical Context

Years ago, when the first monolithic UNIX kernels were emerging into the light of day it was recognised that you could draw a neat line around most of the code used to boot the system and configure its devices. Further, that this code was never used again in the entirety of the operation of the kernel.

Back in the mists of time, Linux began to separate this initialisation code into a different compiler section and release it after the system had completed booting. This worked well for a while but as modular drivers came along, less of the core kernel code which was used by the boot process could be discarded because it might be used by a module to initialise its devices.

Much later, the concept of hotplugging devices came along, and the Linux hotplug [1] project was started.

2 Hotplug

The essence of the hotplug system and its utility has been described elsewhere [2]. Hotplug is primarily intended for computers whose configurations change on the fly, obvious examples of which are the laptop PCMCIA system, a USB daisy chain, firewire and so on. It was recognised fairly early on in this project that the problem of adding a device to a running system is substantially similar to that of configuring a device at boot up, except that the operating system may not be completely initialised (and thus the hotplug system may not be available) when boot probing is done. For this reason, the boot probe issue is called “coldplugging”.

2.1 Avoiding Coldplug

In general, the coldplug problem is similar to most bootstrap problems. However, there is a fairly neat way to avoid the difficulty for several subsystems (SCSI being among them, fortunately): by using an initial ramdisk. As long as the hotplug system is built into the initial ramdisk, the coldplug bootstrap problem is completely eliminated for any subsystem which can be inserted entirely as modules, since it would be handled as a genuine hotplug event by the initial ramdisk hotplug system.

2.2 Hotplug and Device Scanning

A number of the devices that can be hot plugged are also effectively “bridges”, that is units which make onward connections to other busses which may contain other devices. SCSI Host Bus Adaptor (HBA) cards are a classic example of this, since all they really do is bridge the computer bus (often PCI) to an ex-

ternal or internal SCSI bus.

Whenever any type of bus bridge is added to the system, logic must be invoked to scan the devices beyond the bridge and add them into the system. Usually, all this scanning is performed inside the kernel; however, for this paper we investigate transferring they scanning logic to the user level hotplug system.

2.3 Bridge Insertion Events

In general, hotplug events are designed to allow the system to configure the particular device which has just been inserted and the interaction between the programs executed during the hotplug event processing are designed to perform this configuration. Scanning and configuration of devices beyond the bridge device should obviously not be begun until the bridge device is properly configured and fully functional. It therefore makes sense to fire a separate "bridge scan" hotplug event after bridge configuration and bus sanitisation to trigger probing on the actual bus beyond the bridge.

Following the initiation of scanning beyond the bridge, the job of the scanning routine should be to notify the kernel of the existence of the new devices, but allow the kernel to configure them, or better yet trigger another hotplug event to configure them.

One of the issues in bridge/device configuration that require the bridge to be configured before the scan are the setting or collecting of intrinsic bus properties: things like bus speed, width and configuration, all of which must be known before the devices on the bus may be probed. For instance, the SCSI bus can be configured for various widths (wide or narrow). The width is usually governed by the HBA, but nothing prevents a wide device being connected to a narrow only HBA. Thus, the device configuration is dependent on the parameter rage of the bus, which are controlled by the bridge (the HBA).

2.4 Bridge Configuration

We need a mechanism for making available to both the user and kernel the parameters detected and set during the prior bridge hotplug event. There is an evolving infrastructure in the new driver model[3] that may ultimately be capable of storing this information in a usefully abstracted form. However, for the time being, we opted for a completely opaque bridge programming model so that the bridge hotplug event handler needs exact bridge programming knowledge. Basically, we elected to place the SCSI bus parameters in a SCSI specific field which can be queried by `ioctl`s.

3 Replacing SCSI Scan/Inquiry

This project essentially builds on top of the ideas and code provided by the scsimon [4] project. Although scsimon was designed to provide notifications for device insertions, the code it supplies and the design basics are essentially reusable in the scan/inquiry replacement.

3.1 What `scsi_scan.c` does now

This entire file of code is dedicated to scanning busses and detecting and configuring devices. It is driven entirely from a static table (called `device_list`) which contains inquiry strings matching devices for which special actions are taken. Most of the actions are

geared to LUN scanning. Here is an example of some of the flags:

- `BLIST_FORCELUN`: scan for LUNS even if kernel is compiled not to.
- `BLIST_NOLUN`: Never scan LUNS on this device.
- `BLIST_SINGLELUN`: only allow I/O to one LUN at a time.
- `BLIST_NOTQ`: Device claims to support Tag Command Queueing but in reality cannot.
- . . .

Other flags deal with device type misidentification and so forth. All of which can easily be accommodated in user land, with the addition of two extra ioctls: one to set or clear the tag bit (`tagged_supported`) and one to alter the device type.

3.2 Adding the Bridge Insertion Hook

In the SCSI subsystem, the easiest way to add the bus insertion hook is right at the end of `scsi_register_host()`. We eliminate the code in `scan_scsis` except for the hard coded entry used by `add-single-device`.

The Bus insertion hook is now used to begin scanning the targets (at LUN zero) using the `add-single-device` command (the scsi and channel numbers being passed up from the hotplug event).

It is certainly open to debate whether it is worthwhile moving this functionality into user land. However, in principle we could also set up bus transfer parameters or actually opt to use the SCSI-3 report LUNs command instead for the scanning, so it still provides an arguably much more flexible system. This will become particularly important as newer standards are adopted and the process of scanning for devices changes.

3.3 Adding Device Insertion Hooks

Just as the majority of the work is done in `scsis_scan_single()`, so most of the work will be done in the code running after the device insertion hotplug event. The correct place for this is after the initial inquiry command, so that when the hotplug event is called we know the inquiry parameters and can pass them as event parameters.

The internal `device_list` table may now be laced into a flat configuration file (which is thus easily customised) which matches inquiry strings and triggers the appropriate actions. Since we now have more powerful tools at our disposal, the matching can be much more finely controlled using regular expressions. The actions may also be much more dynamic than simply sending parameters down to the kernel: indeed, the system may now be designed to be completely extensible so that we could execute a vendor supplied script, installed in the system, whenever that vendor's device is detected.

There has been recent concern [5] about certain devices not respecting the SCSI standards with regard to inquiry parameter lengths. This could probably be handled either by allowing the maximum inquiry length to be a bus parameter set by the bridge insertion event, or by having the initial inquiry only be the minimum length and allowing the device hotplug event to use the `sg` device to formulate a second inquiry to get all the parameters it needs. The counter

argument: that the low level drivers need to snoop the inquiry data to set up their parameters could be avoided by allowing the device insertion hotplug to communicate the relevant parameters to the bridge.

3.4 Device and Bridge Interaction

If we managed to create the correct abstraction of the SCSI bus, there would be no need for special ioctls to be sent to inform the bridge of device bus characteristics, nor would the bridge need to snoop data (like inquiry returns) being passed over the bus to obtain this information.

However, until such an ideal state of affairs is reached, it is possible that the bridge will need to be made aware of extra parameters in the device. For this reason, we permit a "bridge callout" to be done at the end of the device hotplug event script (essentially the hotplug engine checks to see if the bridge wishes to be informed of device insertion events and executes the script provided by the bridge if it does). This should allow for arbitrary setting of bridge/device parameters to suit the bus environment.

In the current implementation, it is the responsibility of the device/bridge script to scan for additional LUNs (if necessary). This leads to the unwanted side effect that `add-single-device` for LUN 0 will now trigger a complete LUN scan, which may not be what was intended. This can be solved by making "have scanned for LUNs" a property of the device and passing it up on the device insertion event.

3.5 Event Flow

The flow of events described above is shown in figure 1 with time moving from left to right in the diagram.

4 Error Handling

The current error handler thread begins when a fatal error is detected (see Appendix A for an operational sketch) it then quiesces the device and proceeds, one command at a time, through its abort and reset sequence. It checks the progress of error recovery at most points by sending down a test unit ready (TUR) command. However, there are common SCSI driver problems that the mid-layer ignores and others that cause it to malfunction.

4.1 Who Owns the Tag Starvation Problem?

When Tag Command Queueing (TCQ) is enabled on a device, we pretty much (although not always) use "unordered" tags. This leaves it entirely within the province of the device firmware to determine command execution order. In theory, the device firmware has a mode page which lays out guarantees about the maximum times it will take for the device to process any given tag. However in practise, most (particularly older) devices govern tag execution by closest head stepping times and thus some I/Os to different parts of the disc surface can find themselves ignored—a phenomenon called tag starvation.

Since the problem can occur on almost every parallel SCSI card, every driver that does TCQ has to be aware of it and evolve a strategy to cope. Therefore, the tag starvation problem is

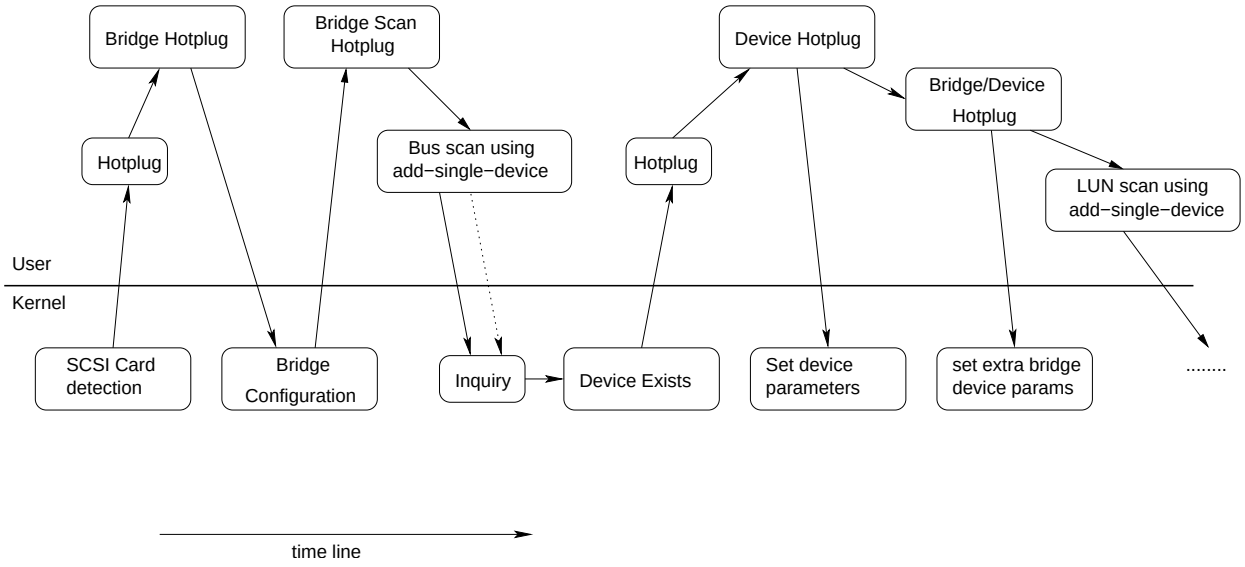

Figure 1: Time line for Hotplug Insertion Events

pretty much owned by the low level drivers, which is a pity, since it means code duplication and lots of extra testing.

The two most general ways tag starvation is handled are: sending an ordered tag with the next command, or not sending down any more commands until the starved tag is processed. These are both extremely easy to implement inside the mid-layer and would relieve the low level drivers of a sizable amount of duplicated code.

4.2 What Kinds of Errors Occur?

The SCSI operation model is essentially a giant state machine. There are a wide variety of error conditions which can occur but which are recognised as particular states in the model. The SCSI mid-layer translates these model states into actions in `scsi_decide_disposition()`. However, anything that gets into this code is pretty much part of the state model, since it is invoked using a SCSI return code. Pretty much everything other than an unrecognised return code for a command still in progress will be handled without troubling the error recovery thread.

The point is that the error handler thread is usually only invoked when the state model has failed[1] or a command has timed out (which is pretty much the same thing, since it means we sent a command to the device and it got lost), so the remedies it applies have usually got to be a drastic kick to get the device back into a state the model recognises and can resume processing from.

4.3 Why Abort?

The first action the error handler thread tries is to issue an abort to the problem command. Abort is a SCSI message that informs the device to discard a particular tag at whatever point it has reached in processing it. It is a command designed to fit inside the SCSI state model and has several uses, particularly for stopping linked commands which have en-

[1] Here remember that the SCSI state model defined by the standards is much larger than the one *implemented* in the mid-layer.

countered errors. However, its use as the first port of call for the error handler thread trying to recover a device is worse than useless, since it is applying a course of action within the state model to something already outside of it. Following this logic, the abort sequence and its associated driver hook may simply be removed (or at least deprecated) in the SCSI driver model.

4.4 Flavours of Reset

The next courses of actions, if abort failed, will be to begin a series of resets culminating in the complete reset of the HBA. The SCSI protocols actually support three levels of reset: LUN, device and bus. The former is a message addition from SCSI-3 and is only relevant for devices with multiple LUNs. It does everything that a device reset does, but operates only on a single LUN—a device reset operates on all LUNs. A device reset only operates on a single target (but all of its LUNs) and a bus reset operates on every device on a physical bus.

Resets are designedly very intrusive to device operation. A reset basically causes the device to drop everything on the floor and re-initialise itself; it is allowed to spend quite a bit of time (measured in seconds) on this re-initialisation and is entitled to respond "not ready" to any command received during this period.

The error handler, meanwhile, should be aware of the extent of the potential disruption resets cause to the device in question, particularly with regard to losing all outstanding commands. It should pull all commands affected by the reset from both the pending and error queues, cancelling the timers on the pending commands and place them all in abeyance until the error handling completes (it might make sense at this point to push them back into the bio queues so that they can be merged if necessary, but hang on to the command we were initially trying to recover).

After a reset has been sent, it should keep the device quiesced and back off for a while (probably a second) before probing with a TUR—we should also loop in this mode, probing every second or so, until the TUR comes back as not "not ready".

Once we know the device is ready to accept commands again, we should feed the first command from the error thread, wait for it to complete and remove the device quiesce if it returns successfully. We should probably also lower the tag queue depth (if it had one) on the assumption that the error may have been triggered by over feeding.

4.5 Choosing a Reset

Often, the simplest reset for any device driver writer to use is the bus reset. This is because all the other resets are actually phrased as SCSI messages and thus need special processing. The bus reset is activated simply by pulling the reset line on the SCSI bus low for 25ms and is usually triggerable using a simple chip register flip.

In choosing bus reset, one thing to beware of is the SCSI standard soft reset alternative (see section 6.2.2 of the SCSI-2 standard [6]). What this does is allow the device to essentially ignore some of the most useful aspects of the bus reset (i.e. dropping everything and starting over from a clean state). The mid layer picks the flag indicating soft reset alternative out of the inquiry data and sets the `soft_reset` device flag in this case. We have never come across one of these devices, but if we did it would certainly cause huge problems for low

level drivers that rely solely on bus resets.

4.6 Device Offlining

The last response of the error handler, if it fails to get the device to accept commands once more is to place it offline. All outstanding commands should be immediately failed with I/O errors. However, the mid layer should continue to accept commands for this device, but should just immediately fail them with I/O errors. This should break out of the unfortunate condition where offlining a device with a huge outstanding bio queue can leave lots of processes stuck in D wait (see appendix B.2).

4.7 Multi-Initiator Scenarios

Previously, multi-initiator (where more than one initiator, or HBA, is connected to the same bus, so multiple machines may be talking to the same devices) was a fairly esoteric configuration primarily limited to clustering environments. With the advent of Fibre Channel, this all changed and shared busses are becoming much more prevalent.

In the classic multi-initator scenario, a reset from another initiator, that you don't see, causes all of your outstanding commands to be lost without trace. This can be particularly nasty in the case where LUN reset is not implemented, because you could be quietly processing exclusively on LUN15 of an array only to be reset because another initator was having issues with LUN3.

About the best way to handle this is to take special action whenever the signature for a reset occurs (which is a check condition followed by unit attention sense on the next command to be sent down to the device). On detecting this condition, we should immediately proceed as though we were the ones resetting the device as part of error recovery: collect all the outstanding commands, cancel the timers probe with a TUR and start feeding them back down again when the device is ready to accept them.

4.8 Testing Error Handler Changes

Once changes are made to error handling, one of the main problems is actually testing them. Most modern SCSI devices really don't actually ever cause the error handler to be invoked. Even transmission line conditions or other problems which cause the SCSI bus to go marginal aren't exactly very useful since there is little chance of correct recovery from them. What is needed is a method for simulating errors in the SCSI subsystem and gauging what happens next.

One particularly useful tool is the debug driver of the Linux SCSI subsystem [7]. Although it currently only comes with the ability to simulate a medium error, persuading it to drop a SCSI command silently (and thus trigger a timeout and error handling) isn't that much of a difficult problem. Once this enhancement is made, it is comparatively easy to trigger a recoverable error an watch how the system behaves.

For those people with access to genuinely malfunctioning devices (my favourite being an old HP C3255 device which seems just to stop working occasionally with high tag queue depths), it is extremely nice to be able to plug them in an watch the system cope (or not, as the case may be).

A A Sketch of How the Current SCSI Subsystem Works

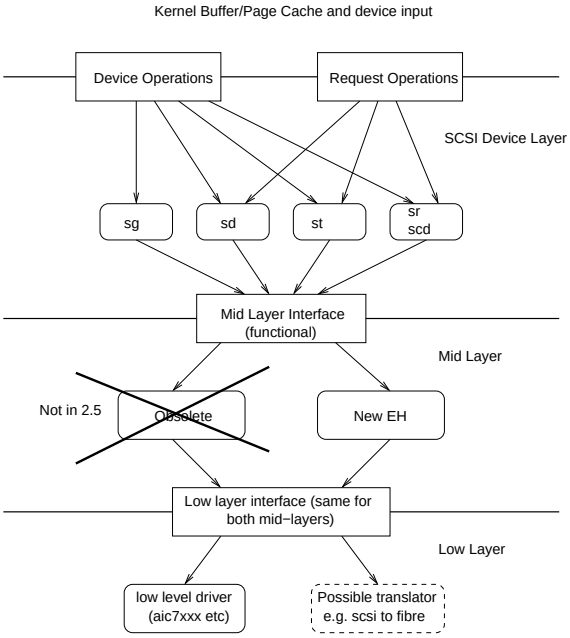

Figure 2: Block Diagram of the SCSI subsystem

A complete block diagram of the SCSI subsystem is shown in figure 2. The error handler comprises a very small portion of the mid-layer (shown as new EH—although for 2.5, this is the only error handler). It's code is entirely in `scsi_error.c` and it is invoked from the bottom half handler routine activated by `scsi_done()`.

A.1 I/O in

All requests come in from the upper layer device drivers through `scsi_request_fn()` which loops over all pending requests. If the device is `in_recovery` or plugged, it returns.

Otherwise, it proceeds as follows:

1. Dequeue the request.
2. Copy the request into a scsi request structure and release the bio request.
3. Call `scsi_init_io()` to set up the scatter/gather list on the request structure.
4. Call the device specific init command to set up the appropriate SCSI command.
5. Initialise the error handler components (mainly zero out sense and set up the timeout).
6. Call `scsi_dispatch_command()` to begin. This sets up the serial number and pid, adds the timer and calls the host `queuecommand()` if it `can_queue` otherwise calls `command()`.

A.2 I/O out

All finished commands come in from the low layer through `scsi_done()` with the queue lock held. They are then added to the bottom half (BH) queue and the BH is notified.

The BH handler (`scsi_bottom_half_handler()`) runs later picking work off the SCSI BH queue until none remains. It calls `scsi_device_disposition()` which returns four possibilities:

- `SUCCESS`: immediately complete the command by calling `scsi_finish_command()`.
- `NEEDS_RETRY`: send the command to `scsi_retry_command()` which will send it immediately down to the lower layer unless the retry count has been exceeded.
- `ADD_TO_MLQUEUE`: call `scsi_mlqueue_insert()` to send the command back to its elevator queue.

- FAILED: set in_recovery, plug the elevator queue and wake the error handler.

A.3 I/O Error

Once the error handler thread is awoken, it calls the template eh_strategy_handler() if it exists, otherwise goes into scsi_unjam_host.

scsi_unjam_host() loops over all pending commands and looks at their state field. Really, it is only interested in the FAILED or TIMEOUT states.

Essentially, it loops over every failed or timed out command and runs through first abort, then device reset, then bus reset and finally host reset, sending a TUR to test the device if one of these succeeded. If it gets all the way to the end and still has failed commands, it offlines the device.

B Unresolved Issues in the Mid-Layer

This section is really a collection of issues on my todo list, but obviously as I haven't got around to doing them yet, if anyone else wants to step into the crease, they're more than welcome.

B.1 queuecommand busy failure

The template hook queuecommand() is allowed to return 1 if the command has not been queued. This can be for a variety of reasons, but most commonly because of either tag starvation or static resource exhaustion. What is supposed to happen is that the unqueued command goes back into the bio elevator and is resubmitted at a later time.

It looks like the scsi_mlqueue_insert() function or the bio is failing somehow, because on most 2.5.x, as soon as queuecommand() returns 1, the buffer hangs forever in D wait.

B.2 Device Offline Failure

After an initial complete failure of the error handler, leading to a device being taken offline, processes trying to use buffers on the device often end hung in D wait. This is indicating that the code which returns I/O errors on all the outstanding I/O requests is missing some. I suspect there may be a problem prizing the rest of the I/O out of the bio, since it seems that the code in the error handler to fail the I/O that has reached the mid-layer is fairly bullet proof.

References

[1] http://sourceforge.net/projects/linux-hotplug

[2] Greg Kroah-Hartman *Hotpluggable Devices and the Linux Kernel* Ottawa Linux Symposium 2001

[3] Patrick Mochel *The (New) Linux Kernel Driver Model* Documentation/driver-model.txt

[4] Doug Gilbert *Scsimon Driver for Linux* http://www.torque.net/scsi/scsimon.html

[5] Martin Wilck *Hack to make Datafab KECF-USB work* http://marc.theaimsgroup.com

/?l=linux-usb-devel
&m=101304393027774

[6] X3T9.2 Project 375D *Information Technology—Small Computer System Interface 2*
ftp://ftp.t10.org/t10
/drafts/s2/s2-r10l.pdf

[7] Originally by Eric Youngdale, but now Maintained and enhanced by Doug Gilbert
http://www.torque.net/sg
/sdebug.html