SUN BLUEPRINTS APPROVED

[illegible]

Customizing the JumpStart™ Boot Image for Recovery

By John S. Howard - Enterprise Engineering

Sun BluePrints™ OnLine - March 2001

Sun microsystems
ENTERPRISE ENGINEERING

http://www.sun.com/blueprints

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300 fax 650 969-9131

Part No.: 816-0265-10
Revision 01, 02/12/01
Edition: March 2001

Customizing the JumpStart™ Boot Image Recovery

The boot image of the Solaris™ Operating Environment (OE) can easily be enhanced to provide a platform to be used to affect system recovery. This article provides techniques and recommendations for augmenting the miniroot boot image of the Solaris OE and the JumpStart™ framework to create a recovery platform.

This article will examine the boot and installation processes, demonstrating how to adapt those processes for system recovery. The following sections detail these techniques and aspects of the JumpStart framework:

- Recovery fundamentals
- Altering the boot process
- Challenges unique to the miniroot

This article is an excerpt from the forthcoming Sun BluePrints™ book titled "JumpStart™ Technology: Effective use in the Solaris™ Operating Environment." This book is scheduled for publication by Prentice Hall in the Summer of 2001 and will be available though `http://www.sun.com/books`, amazon.com, fatbrain.com and Barnes & Noble bookstores.

Recovery Fundamentals

The typical mechanism for attempting recovery on a failed system is to boot the host from the Solaris OE installation CD. This approach is the most direct method to bring the host to be recovered to a point where some form of corrective action can be initiated. However, this method has a major draw-back; because the unalterable nature of the CD restricts the available tools, the recovery procedures are restricted, and the recovery time may be adversely affected. Tools commonly used in the datacenter, such as Veritas Volume Manager (VxVM), Veritas NetBackup, or Solstice Backup™ software may be either completely unavailable, or very cumbersome to use while booted from the CD.

Booting from the network using JumpStart technology is virtually identical to booting from the CD-ROM. Although the mechanics of accessing system and data files may differ between a network boot and a CD boot, the operating system image that is loaded when booting from the network is identical to the operating system image that is loaded when booting from
CD-ROM. Further, booting from the network restricts the system administrator in the same fashion as booting from the CD itself; the root filesystem provided to the client is mounted read-only. However, since the JumpStart server's boot image originates on writable media, it may be modified at the server side, before the client is booted.

By modifying the client’s boot image, software tools commonly used in the datacenter can be installed into the JumpStart boot image, and configured for use by any system that uses that boot image.

The `$ROOTDIR`

To understand how to correctly modify or augment the client’s boot image, you must understand how the JumpStart server maps the filesystem that contains the boot image to the client’s view of the same filesystem. For example, to modify the network services available on the client, you may need to modify the `/etc/inetd.conf` file. If this is the case, you must locate and modify the client’s `/etc/inetd.conf` file on the JumpStart server.

This location information is controlled, in part, by the manner in which the JumpStart server was installed. The following command sample shows a typical installation and configuration of a JumpStart server named badtrap. `/jumpstart` is the filesystem that is used as the location for all JumpStart configuration information, profiles, boot images and software installation packages. `/jumpstart` is shared via NFS with read-only access rights and the effective UID of unknown users mapped to `root` (the `anon=0` option, see the `share_nfs(1M)` man page for additional details). setup_install_server and add_to_install_server copy the boot image and install packages from the Solaris OE Software CD-ROMs to the specified directory on the JumpStart server (/jumpstart/OS/Solaris_8_1000):

```
badtrap# share -F nfs -o ro,anon=0 /jumpstart
badtrap# mkdir /jumpstart/OS/Solaris_8_1000
badtrap# cd \
> /cdrom/sol_8_1000_sparc/s0/Solaris_8/Tools
badtrap# ./setup_install_server /jumpstart/OS/Solaris_8_1000
Verifying target directory...
Calculating the required disk space for the Solaris_8 product
Copying the CD image to disk...
Install Server setup complete
[ insert Solaris 8 Software cd 2 of 2 ]
badtrap# cd /cdrom/sol_8_1000_sparc_2/Solaris_8/Tools
badtrap# ./add_to_install_server /jumpstart/OS/Solaris_8_1000

The following Products will be copied to /jumpstart/OS/
Solaris_8_1000/Solaris_8/Product:

Solaris_2_of_2

If only a subset of products is needed enter Control-C
and invoke ./add_to_install_server with the -s option.

Checking required disk space...

Copying the Early Access products...
213481 blocks


Processing completed successfully.
```

The add_install_client command is then executed to configure the server to accept requests from the client (in this example, the client is named wasabi). add_install_client updates, or creates, if necessary, the JumpStart server's /etc/bootparams file containing the information for the specified client.

Additionally, specifying the `-e` and `-i` options instruct `add_install_client` to update the JumpStart server's `/etc/ethers` and `/etc/hosts`, respectively, if necessary.

```
badtrap# cd /jumpstart/OS/Solaris_8_1000/Solaris_8/Tools
badtrap#./add_install_client \
> -i 129.153.47.6 -e 8:0:20:7c:ff:d0 \
> -p badtrap:/jumpstart/Sysidcfg/Solaris_8 \
> -c badtrap:/jumpstart \
> wasabi\
> sun4u
```

Examining `/etc/bootparams` shows how badtrap's view (the JumpStart server view) of the client's root filesystem is mapped to wasabi's view (the client's view):

```
wasabi  root=badtrap:/jumpstart/OS/Solaris_8_1000/Solaris_8/
Tools/Boot install=badtrap:/jumpstart/OS/Solaris_8_1000
boottype=:in sysid_config=badtrap:/jumpstart/Sysidcfg/Solaris_8
install_config=badtrap:/jumpstart rootopts=:rsize=32768
```

The parameter `root` is set to `badtrap:/jumpstart/OS/Solaris_8_1000/Solaris_8/Tools/Boot`. This is the NFS filesystem wasabi will mount as its root filesystem when booting over the network. As the JumpStart environment variable `$ROOTDIR` is set to this value, the JumpStart server's view of this directory is commonly referred to as `$ROOTDIR`. In this example, `$ROOTDIR` would be set to `/jumpstart/OS/Solaris_8_1000/Solaris_8/Tools/Boot.` Locating or placing a file into the client's filesystem on the JumpStart server simply becomes a matter of pre-pending `$ROOTDIR` to the file name. For example, if the client needs to have a file placed in its `/etc` directory, that file must be placed in `$ROOTDIR/etc` on the JumpStart server.

Altering the Boot Process

This section demonstrates how to augment the client's boot image to be suitable for use as a platform for system recovery operations. This section explains how to:

- Modify the option and argument processing during boot.
- Provide services and daemons.
- Provide an interactive shell.

It is important to note that making the following modifications to the client's boot image allows support for additional functionality. The default functionality of the client's boot image will remain unchanged and the client's boot image can still be used to install the Solaris OE.

The first challenge in transforming the install boot image into a boot image that is suitable for recovery operations is to change the boot process, or start-up process, of the client's boot image. The client should come up to multi-user mode, yet not enter the default action of beginning the installation process. To subsume this default boot process, the scripts run at startup time on the client must be modified.

Option and Argument Processing During Boot

The first task in converting the install boot image into a recovery boot image is to augment the boot proces with a "recover" mode. This recover mode will be entered if the correct parameter is specified when the system is booted. The addition of the `recover` parameter is achieved by locating and altering the boot parameter processing logic in the start-up scripts. For example, the OpenBoot PROM (OBP) boot command normally used to perform a JumpStart (network) boot and installation is `boot net - install`. The "`install`" argument is passed through the kernel and on to `init` as the system startup scripts are executed. The remainder of this section describes the steps necessary to locate this argument parsing and adding the logic to recognize a `recover` parameter.

The OBP will pass options to the kernel, such as the -s (boot to single-user mode) and -r (initiate a device tree rebuild and kernel reconfiguration) options. The OBP will also pass arguments along to the kernel. Since the kernel will not process command line arguments (it will only process options it recognizes), the arguments are ignored by the kernel and passed along to `init`. To prevent confusion, kernel switches and arguments are separated by a lone dash, which is why the space following the '-' is crucial in the following command: `boot net - install`

In turn, `init` passes all arguments on to the scripts that it calls. By using `$ROOTDIR/etc/inittab` as a road map to the start-up processing, you can determine that the argument processing takes place in `$ROOTDIR/sbin/rcS`. The portion of `$ROOTDIR/sbin/rcS` that is relevant to the argument processing is:

```
set -- ""
set -- ‘/sbin/getbootargs 2>/dev/null‘
if [ $# -gt 0 ] ; then
        while [ $# -gt 0 ] ; do
                case $1 in
                FD=*)
# at end of script, save root dev in /tmp/.preinstall
# this is an unambiguous indication of stub boot
                        FJS_BOOT="yes"
                   From=‘(IFS="="; set -- $1; echo "$2 $3 $4 $5" )‘
                        break
                        ;;

                browser)
                        cat < /dev/null > /tmp/.install_boot
                        cat < /dev/null > /tmp/.smi_boot
                        shift
                        ;;

                install)
                        INSTALL_BOOT="yes"
                        cat < /dev/null > /tmp/.install_boot
                        shift

dhcp)
                        TRY_DHCP="yes"
                        shift
                        ;;

                mansysid)
                        cat < /dev/null > /tmp/.manual-sysid
                        shift
                        ;;
                w)
                        cat < /dev/null > /tmp/.nowin
                        shift
                        ;;
                *)
                        shift
                        ;;
                esac
        done
fi
```

It is now relatively straightforward to add recognition and processing for the ‘recover’ argument by adding another word to the case statement. The modified section, shown in bold face, of $ROOTDIR/sbin/rcS is:

```
set -- ""
set -- ‘/sbin/getbootargs 2>/dev/null‘
if [ $# -gt 0 ] ; then
        while [ $# -gt 0 ] ; do
                case $1 in
                FD=*)
# at end of script, save root dev in /tmp/.preinstall
# this is an unambiguous indication of stub boot
                        FJS_BOOT="yes"
                   From=‘(IFS="="; set -- $1; echo "$2 $3 $4 $5" )‘
                        break
                        ;;
browser)
                        cat < /dev/null > /tmp/.install_boot
                        cat < /dev/null > /tmp/.smi_boot
                        shift
                        ;;
recover)
cat < /dev/null > /tmp/._recover_startup
shift
;;
install)
                        INSTALL_BOOT="yes"
                        cat < /dev/null > /tmp/.install_boot
                        shift
dhcp)
                        TRY_DHCP="yes"
                        shift
                        ;;
                mansysid)
                        cat < /dev/null > /tmp/.manual-sysid
                        shift
                        ;;
                w)
                        cat < /dev/null > /tmp/.nowin
                        shift
                        ;;
                *)
                        shift
                        ;;
                esac
        done
fi
```

It is important to note that almost no action is performed within the case statement or $ROOTDIR/sbin/rcS. As in the case of the install argument, the processing of the recover argument consists only of creating a state file (or flag file). Using this mechanism makes the decision process in scripts executed later easier, and there is no need to require all scripts to parse the arguments directly. If recovery scripts or tools need to be started automatically on a recovery boot, or if scripts need to determine the state of the system and take appropriate action, these scripts only need to check for the existence of the file /tmp/._recover_startup to determine the system state. Because this method of modifying $ROOTDIR/sbin/rcS reduces complexity and minimizes the opportunity for human error in modifying the crucial $ROOTDIR/sbin/rcS script, it is strongly recommended that the same or similar approach be used whenever adding start-up processing.

Providing Services For Recovery

The default boot process does not start any service daemons that are not required by the Solaris OE installation process. However, some of these daemons may be needed during a system recovery operation. For example, the Internet service daemon, inetd, is not needed for installation; however, a recovery operation is greatly facilitated by having inetd start automatically on a recovery boot.

Unfortunately, daemons can be started from many scripts, and there is no reference or mapping to the location where a particular daemon may be started. Most of the standard Solaris OE daemons are present, but commented out (*stubbed*), in the start-up scripts executed by init. The only way to include additional daemons is to follow the execution flow of the scripts launched by init, identifying which scripts start which services, and uncommenting the required daemons. For example, inetd is present, but commented out, in $ROOTDIR/sbin/sysconfig. To have inetd started during a recovery boot, modify $ROOTDIR/sbin/sysconfig at the point where inetd start-up is commented out as follows:

```
if [ -f /tmp/._recover_startup ] ; then
    /usr/sbin/inetd -s
fi
```

It is crucial to keep in mind that these changes were made on the JumpStart server and that all clients booting off of this JumpStart server will have access to these added functions.

If a service or daemon is required at startup, but it is not stubbed in one of the start-up scripts, you must add the invocation to the start-up processing. If it is necessary to add a call to a script or the execution of a command, make the addition as late as possible in the startup sequence to avoid impacting dependencies or set-up work that services may have on other services or daemons. For example, many network services require that `sysidnet` will have been run which implies the presence of a name service or `sysidcfg` file.

Providing an Interactive Shell

The presence of the `install` parameter causes the boot image to enter the installation utility (`suninstall`) at the end of its start-up processing. To allow you to perform any manual recover operations that may be necessary, the presence of the recover parameter will start an interactive shell at the end of its start-up processing. To accomplish this, add the following lines to the end of `$ROOTDIR/sbin/sysconfig:`

```
if [ -f /tmp/._recover_startup ] ; then
    echo “Starting interactive Korn shell...“
    exec /bin/ksh -o vi
fi
```

Adding Utilities and Manual Pages

The methodology of the previous section can also be applied to the task of augmenting the client’s boot image with tools, utilities, and documentation. Most of the tools to be added to the client’s boot image will not require special configuration. In most cases, you only need to copy the utility or tool to its expected location under `$ROOTDIR` to make it accessible and usable by the client after a `boot net - recover`. However, it is important to note that device drivers, kernel extensions, and utilities expecting a writable `/var/tmp` are a special case and will be examined later in this section.

Adding a Recovery Tool

The example presented in this section adds Solstice Backup software to a client’s boot image. Solstice Backup software installs its client side utilities in `/usr/bin/nsr` and `/usr/lib/nsr`. To provide the client portions of Solstice Backup software to the client’s recovery boot image, it is only necessary to replicate, or relocate, the `/usr/bin/nsr` and `/usr/lib/nsr` directory hierarchies on the JumpStart server to `$ROOTDIR/usr/bin/nsr` and `$ROOTDIR/usr/lib/nsr`, respectively. This relocation can be accomplished by using the root-path option (`-R`) of pkgadd, or by simply copying the directory hierarchies from an installed Solstice Backup software Client to $ROOTDIR/usr/bin/nsr and $ROOTDIR/usr/lib/nsr.

Adding Device Drivers

As with system services and daemons, the device drivers provided in the default client's boot image are only those device drivers that may be necessary for installing the Solaris OE. Providing the client's recovery boot image with additional device drivers, such as FDDI, ATM, or some other high-band width network driver may be useful or required in system recovery situations.

Consider the FDDI SBus card. While the default client's boot image does not provide this device driver, many datacenters use a dedicated FDDI network for backups and restores. During a recovery operation, the system administrator would benefit from having the recovery boot image access the high bandwidth network to recover data from the backup server.

In principle, device drivers are added in much the same way as standard files are added to the JumpStart server. Device drivers or kernel extensions are merely copied to $ROOTDIR/kernel/drv. Unfortunately, a device driver may need to modify several other configuration files in order to function properly. The Solaris add_drv command is the mechanism by which the required files can be updated in a controlled fashion. For example, to add the FDDI driver to the recovery boot image, the following commands would be executed:

```
badtrap# ROOTDIR=/jumpstart/OS/Solaris_8_1000/Solaris_8/Tools/
Boot ; export ROOTDIR
badtrap# cd /kernel/drv
badtrap# cp fddi fddi.conf $ROOTDIR/kernel/drv
badtrap# add_drv -b $ROOTDIR $ROOTDIR/kernel/drv
Warning.  Major number based on server, not client.
```

Note – Before installation, consult the manufacturer's or vendor's instructions for all device drivers.

It is important to note that some device drivers, such as those that create or require entries in /dev upon boot, may require further configuration by hand. Some device drivers may require modifications to files such as:

- /etc/devlink.tab
- /etc/driver_aliases
- /etc/system

After modifying these files, make the appropriate modifications to the respective files under `$ROOTDIR` to affect the client's boot image. Consult the installation documentation for the device driver for information about the specific modifications that may be necessary.

For example, the Veritas Volume Manager (VxVM) product includes the device drivers `vxspec`, `vxio`, and `vxdmp`. Because these device drivers must be forceloaded at boot, you must add the following directives to `$ROOTDIR/etc/system`:

```
forceload drv/vxdmp
forceload drv/vxspec
forceload drv/vxio
```

Additionally, theses VxVM device drivers require device entries to be created in `/dev` according to a specific template in `/etc/devlink.tab`.

Challenges Unique to the Miniroot

While the miniroot provides most of the services and facilities found in the full Solaris OE, there are some restrictions and challenges to working in the miniroot. This section examines these challenges and provides recommendations and work-arounds for meeting these challenges.

Read-Only Media

While it is obvious that the boot image on a Solaris OE Software CD is read-only, it may not be obvious that the client's boot image served by the JumpStart server is also read-only. As such, all systems booted from this image for recovery or installation will mount their filesystems read-only. Whether booted from a CD or over the network from a JumpStart server, the `/tmp` memory mapped `tmpfs` filesystem is provided as the only writable space for any utility or service that requires writable media.

Because the Solaris OE `tmpfs` filesystem is fashioned from the virtual memory in the client, all of the contents will be lost when the client reboots. Files or directories on the filesystems other than `/tmp` that must behave as if they are writable must be redirected to `/tmp` by means of a symbolic link. For example, the `/etc/vfstab` file must be writable; however, `/etc` is on a read-only filesystem. `/etc/vfstab` is actually a symbolic link, created at the client's boot, to a writable `vfstab` file in

/tmp. All such symbolic links are specified on the read-only CD image or on the client's boot image on the JumpStart server. The targets of these symbolic links are then created in /tmp when the miniroot is booted.

The same mechanism may be used to provide writable space for tools added to the recovery boot image. The following commands provide a writable log space for NetBackup in $ROOTDIR/usr/openv/logs:

```
badtrap# cd \
/jumpstart/OS/Solaris_8_1000/Solaris_8/Tools/Boot/usr/openv
badtrap# ln -s /tmp/_openv_logs ./logs
```

However, this only addresses half of the problem. The target of the symbolic link, in /tmp, must still be populated at boot. Any required link targets in /tmp can be populated using either of the following methods:

- Create a start script that executes: mkdir /tmp/_openv_logs.
- Utilize the built-in script for populating /tmp.

The first method is a brute force approach, but it is effective on a small or limited scale. The second method takes advantage of the existing method that the miniroot uses at boot to populate /tmp.

A prototype for /tmp is maintained on the miniroot, which is then copied into /tmp very early in the boot process. This prototype is the directory $ROOTDIR/.tmp_proto; very early in the startup sequence (in $ROOTDIR/sbin/rcS), cpio is used to populate /tmp from this prototype. These will become the targets that are referenced by symbolic links throughout the miniroot image.

In the previous example, creating the directory $ROOTDIR/.tmp_proto/_openv_logs ensures that /tmp/_openv_logs will be created when the client miniroot is started.

Files in /var

The /var directory tree is an extension of the writable directory problem outlined in the previous section. Most facilities in the Solaris OE utilize /var/tmp, or write temporary files into some subdirectory of /var. Because so many utilities expect /var to be writable, the entire /var filesystem is provided as writable by means of a symbolic link to /tmp/var. This is accomplished using the same mechanism as described in the previous section; the prototype for the /var hierarchy is in $ROOTDIR/.tmp_proto/var. To relocate a file directory into the client's /var, the file or directory must be the $ROOTDIR/.tmp_proto/var prototype.

The *path_to_inst* File

The Solaris OE device instance number file, path_to_inst, is a special case. Because this file must be created from scratch each time a system boots from the miniroot, the path_to_inst file must be writable; however, it cannot simply link to /tmp.

The miniroot cannot possibly have a valid device tree and instance number file for each client that might boot from it. The path_to_inst file, and the /dev and /devices directory trees must be created by the miniroot each time a system boots from the CD or JumpStart boot image. The OBP is responsible for probing the bus and inventorying all of the devices in the system. The OBP hands off the result of this inventory (the “OBP device tree”) to the kernel when it begins loading. The OBP device tree is used as the basis for the kernel to map the device location (path) to the device driver instance numbers. This mapping is what is stored in the path_to_inst file. For a boot from read-only media (i.e., from CD, or from a JumpStart server) the newly created path_to_inst file is generated and written to /tmp/root/etc/path_to_inst early in the miniroot boot process.

Because a valid /etc/path_to_inst is required by the kernel very early in the boot cycle, using a writable surrogate in /tmp is not possible, as /tmp will not have been populated when the instance number file is needed.

To redirect the kernel to the “real” path_to_inst file that was written in /tmp/root/etc/path_to_inst, *two* path_to_inst files are used. The “real” path_to_inst, which contains valid device paths mapped to the correct instance names, is created in /tmp/root/etc/path_to_inst. The bootstrap version, required by the kernel, but invalidated after the kernel loads, is in /etc. This bootstrap version of
$ROOTDIR/etc/path_to_inst consists only of the line:

```
#path_to_inst_bootstrap_1
```

This bootstrap path_to_inst may cause problems for utilities that are hardcoded to reference /etc/path_to_inst for the device instance number file.

System utilities (such as tools like SunVTS™ diagnostic software or older versions of StorTools™ diagnostic software) that must build internal tables of the devices attached to the system typically read /etc/path_to_inst directly, rather than obtaining the device instance information from the kernel.

Unfortunately, if these utilities hard code the pathname for /etc/path_to_inst into their object modules, libraries, or binaries, it is not easily changed. This will prevent the use of such utilities while the client is booted from the JumpStart server or CD.

Summary

This article has examined the start-up processing performed by the miniroot. Methodologies and techniques were provided to augment start-up processing. Techniques for adding tools to the client's boot image were also detailed.

Finally, the constraints of the miniroot, as well as techniques for working within these constraints, were examined.

Author's Bio: John S. Howard

John S. Howard has over 19 years experience in software engineering and systems administration. As a Staff Engineer in the Enterprise Engineering group at Sun Microsystems, he is currently working on projects for enhancing system availability and serviceability.

Prior to his position in Enterprise Engineering, John worked as an Area System Support Engineer with Sun Enterprise Services. As an ASSE, he was responsible for providing escalation management and backline system support for problem isolation and resolution in the areas of clustered systems, the storage subsystem, and the Solaris kernel. In addition to these support functions, he developed and performed Reliability, Accessibility, and Serviceability (RAS) studies of customer datacenters.