Sunrise
DRAFT

Anthony Sorace
a.9srv.net
Strand 1 Technologies

ABSTRACT

DRAFT Our lab includes a variety of older Sun hardware

1. Introduction

I have a collection of older Sun systems which I would like to use. This set currently includes:

sysname	model	RAM
arabica	JavaStation	
kona	JavaStation	
bluemountain	JavaStation	
solar	Ultra 5	
	SPARCstation 20 MP	256 MB
	SPARCstation 2	64 MB

All of these systems lack integrated storage; the JavaStations have no capability for internal storage, the Ultra 5 has a bad disk, and the SPARCstations were obtained without hard disks. I would like to be able to boot each of these over the network and run different systems on each. This era of Sun systems have good capabilities for network booting, although the procedure differs quite a bit from more modern systems.

2. Get an IP address

All of these systems are able to boot over the network; they begin this process using *bootp* to obtain an IP address (they will also use the address of the responding server later). This relies on some information stored in NVRAM, but all of this hardware is old enough that the NVRAM battery is dead. Unfortunately, this era of Sun hardware all used NVRAM chips where the battery is integrated; replacing the battery involves some degree of surgery on the chip itself. The configuration stored by the NVRAM can be entered at the serial console, but will be lost upon each power cycle[1]. To avoid manually re-entering this information on each boot, a new script, *bootscript*, provides it. Invoked as `bootscript <sysname> <device>`, it will look up the IP address for

[1] Settings will sometimes be preserved across very short power off/on cycles, but the period is too unpredictable and too short to be useful.

sysname in *ndb*, attach to a serial console on *device*, and prompt the user to power on the system. It will then monitor the output from the console and provide the needed settings.

In the representative session below, **:;** is the prompt and lines following that are user input.

```
:; cat /dev/eia2status
b28800 c0 d1 e0 l8 m0 pn r1 s1 i0
dev(2) type(0) framing(13637) overruns(0) berr(0) serr(0) cts
:; echo i99 > /dev/eia2ctl
:; :; echo f > /dev/eia2ctl
:; echo k > /dev/eia2ctl
:; echo h > /dev/eia2ctl
:; echo b9600 > /dev/eia2ctl
:; cat /dev/eia2status
b9600 c0 d1 e0 l8 m0 pn r0 s1 i99
dev(2) type(0) framing(22070) overruns(0) berr(0) serr(0) cts
:; bootscript arabica /dev/eia2
08 00 20 c0 ff ee c0ffee mkpl
Power on arabica now.
Starting real time clock...
Incorrect configuration checksum;
Setting NVRAM parameters to default values.
Setting diag-switch? NVRAM parameter to true
Probing CPU FMI,MB86904

The IDPROM contents are invalid

Initializing  40 megs of memory at addr        0
Boot device: /iommu/sbus/ledma@4,8400010/le@4,8c00000  File and args: /kona
Internal loopback test -- Did not receive expected loopback packet.

Can't open boot device

ok set-defaults
Setting NVRAM parameters to default values.
ok setenv diag-switch? false
diag-switch? =        false
ok 17 0 mkp
ok 08 00 20 c0 ff ee c0ffee mkpl
ok
ok
Target configured. To boot: 'echo reset > /dev/eia2'
:; echo reset > /dev/eia2
:;
```

The device was manually powered on when instructed.

At the moment, *bootscript* stops short of instructing the Sun to actually boot, but provides the instructions to reboot using the new configuration.

2.1. Configuring the serial port

In most cases, *bootscript* is directly controlling a uart device. In such cases, there are two configuration steps which should be considered. First, all of these systems come up with their serial port configured to communicate at 9600 baud with 8 data bits, no parity, and 1 stop bit (often written `9600,8,n,1`). Serial ports on Plan 9 all come up with 8 data bits, no parity, and 1 stop bit by default, but the baud rate cany vary. Set it before proceeding with `echo b9600 > /dev/eia2ctl`, (replacing `/dev/eia2ctl` with the appropriate control file for your serial console).

Second, several of the serial consoles on our Sun systems are connected to a Raspberry Pi using a tll-rs232 adapter as described in Setting up RS-232 on a Raspberry Pi. We do not have hardware flow control configured for these ports. If the serial port overruns its buffer, garbage characters can be printed; while this clears itself when typing on the console interactively, it can cause problems for *expect(1)* when used by *sunrise* to automate the process.

This process was first started using the `uartmini` driver, which controls an older, simpler, and less full-featured uart on the Raspberry Pi. With this uart, I was unable to find a configuration the driver supported with proved "safe" for *bootscript*. Instead, a new program, *trickle*, was written to simply copy the uart's input and output at a slower rate.

This was effective, but complicates the procedure. Instead, I'm now using the `pl011` driver, added to the Raspberry Pi kernel earlier this year. With that driver, enabling the device FIFOs at a high level seems to be sufficent. To set the FIFOs to their maximum level and flush any data in the queue, run:

```
echo i99 > /dev/eia2ctl
echo f > /dev/eia2ctl
echo h > /dev/eia2ctl
```

(again replacing `/dev/eia2ctl` as appropriate).

In the future, *bootscript* should do some of this configuration on its own, but that is not in place. Since *bootscript* also allows connecting to a console over an existing *consolefs(4)* connection; in those cases, we would expect *consolefs* to be responsible for setting up the uart correctly. It is probably safe for *bootscript* to check for the existence of a `<device>ctl` file and configure it appropriatly only if such a file exists.

3. Load a Kernel

Having obtained their IP address, these SUN systems will attempt to load a kernel using *tftp*. They will ask for a boot file reflecting the IP address and architecture of the system. To support this without having to maintain a kernel image per system, Plan 9's tftpd offers some extra support. tftpd(8) states:

> All requests for files with non-rooted file names are served starting at this directory with the exception of files of the form xxxxxxxx.SUNyy. These are Sparc kernel boot files where xxxxxxxx is the hex IP address of the machine requesting the kernel and yy is an architecture identifier. Tftpd looks up the file in the network database using ipinfo (see ndb(2)) and responds with the boot file specified for that particular machine.

Unfortunately, a bug introduced several years ago prevents that from working. To correct it and restore the documented functionality, change the line

```
if(suffix-name != 8 || (strcmp(suffix, "") != 0 && strcmp(suffix, ".SUN") != 0))
```

to

```
        if(suffix−name != 8 || (strcmp(suffix, "") != 0 && strncmp(suffix, ".SUN", 4) != 0))
```

and recompile tftpd. Then ensure that your ndb database contains en entry for the Sun’s IP address with a `bootf` value pointing to the desired kernel (it needn’t be in `/lib/tftpd`; anywhere in the namespace tftpd can see is fine).