On Switch Booting

Derek McAuley

15th March 1994

1 Introduction

This document covers two aspects of using Fairisle switches: internal switch booting and automatic generation of switch configurations. The two are described together, as the automatic tool sc also generates the information files for the switch booting.

2 Switch configuration

The process of building a switch configuration is as follows:

```
co -l sc.conf.c
do
        do
                <edit sc.conf.c>
                make sc
        while <compile/link errors>     # probably C syntax
        make auto
while <sc errors>                       # erroneous configuration
make install
ci -u sc.conf.c
```

This is normally performed in one place, currently /usr/groups/wanda/master/sc, as it is vital that all hostinfo information and booting information is consistent and generated from the same sc.conf.c.

The man page sc.8 gives some indication of what sc does; however, the Makefile in the directory does most of it for you.

What you need to do is modify the files sc.conf.c (under RCS control of course), build the sc program, build the auto files by running sc, then install the generated switch configuration and hostinfo files in the standard place (currently /usr/groups/wanda/boot/hostinfo). Separate targets sc, auto and install exist in the Makefile with appropriate dependencies, but (motherhood coming up) it is advised to do a "make auto" and ensure it worked before a "make install".

To build a configuration, use co -l to grab the lock on sc.conf.c, and build in THIS directory. Release lock when finished. IT IS VITAL we do not end up with different people using different versions of sc.conf.c. This is why this directory is outside the normal tools tree, to avoid multiple copies.

The format of sc.conf.c is fairly tedious, but a brief study should show it's main features – some example gloop below may help.

The sc program starts at the top level of the description (end of file) with a `struct world`, composed of switches, hosts and gateways; these in turn are arrays of `struct swtch`, `struct portc` (this includes hosts), and `struct gateway`; a `struct swtch` is an array of `struct portc`. The macros PD() and HS() are used to define ports and hosts respectively, which then should be included in a switch or the list of hosts as appropriate.

It might be nice if this structure was actually built by parsing some simpler text file at run time – volunteers required. Further the sc program (apart from main) is "OO" in that it is trivial to have it deal with a number of worlds.

sc builds a sub directory auto containing an automatically generated hostinfo file, <hostname>.auto, and switch configuration files, <switch>-wdbc.c. Summary files F_READABLE and F_ROUTES are also produced for human consumption. The old auto subdirectory is renamed auto.BCK.

2.1 sc.conf.c format

Some examples of the format of entries are given below, although only looking at sc.conf.c will make it obvious what is going on.

2.1.1 Hosts and port controllers

```
Port          FSN,    MSNL address,    line to, public gate,
PD(hake,"hake",22, 128,232,132,144, &p_bummalo, ETHER | ALL, 0, 0, RAP | LRAP);
```

Hake has a link to bummalo (N.B. p_ in &p_bummalo for port)., offers gateway to ETHER and ALL, acts a rarp server for switch (RAP) and also will rarp on its link (LRAP).

```
Port                      FSN,       MSNL address,   private gate,
PD( pilchard, "pilchard", 16, 128, 232, 132, 138, 0, ETHER | ALL, 0, 0, RAP);
```

Pilchard acts as rarp server for switch, has a private ETHER.

```
Port                      FSN,       MSNL address,  link to,
PD( snook,    "snook",    10, 128, 232, 132, 142, &h_hyale, 0, 0, 0, 0);
```

Snook has a link to host hyale (N.B. h_ in &h_hyale for host).

```
Port                     FSN,       MSNL address,line to,  link gate,
PD( bummalo,  "bummalo",  21, 128, 232, 132, 131, &p_hake, 0, 0, ALL, RAP);
```

Bummalo offers a gateway to some switch ports to ALL down its link – this also tells attached gateway (hake in this case) to route from ALL to switch.

```
Host                            MSNL address,      private gateways
HS( hyale,    "hyale",    128, 232, 132,  41, 0, ETHER | ALL | FAIR, 0);
```

Hyale is a host attached to ETHER, ALL and FAIR.

In these entries, the following flags are used:

```
        #define RAP  (1 << 0) /* Be a rarpd on a switch */
        #define WDBS (1 << 1) /* Be a WandaDumbBootServer */
        #define LRAP (1 << 2) /* Be a rarpd on a line */
        #define TTYS (1 << 3) /* Be a TTY Server */
        #define TTYC (1 << 4) /* Be a TTY Client */
```

A master port on a switch would normally be the mode with Ethernet attached and flags of `RAP | WDBS | TTYS`, slave ports `TTYC` and a slave port with attached host `TTYC | LRAP` (to cause it to act as rarpd for the attached host.

2.1.2 Gateways

Ports (hosts) that act as gateways can either be public (i.e. willing to route for anyone) or private (i.e. only route for machine to which they are attached); there are two separate fields for private and public gateways in the port and host data structures.

The MSNL network associated with a particular gateway is defined by the `struct gateway gates[]` array. besides the obvious name, MSNL network and mask fields, there is a unique bit for each network (hence current `sc` tool is limited to 32 networks).

```
#define FAIR    (1 << 0)
#define CL      (1 << 1)
#define CFR     (1 << 2)
#define CBN     (1 << 3)
#define ETHER   (1 << 4)
#define PANDORA (1 << 5)

struct gateway gates[] = {
/* Own network here */
  {  ONE, "fair",    FAIR,    {128, 232, 128, 0}, {255, 255, 240, 0} },
/* Remainder which we gateway to here */
  {  ONE, "cl",      CL,      {128, 232,   0, 0}, {255, 255, 240, 0} },
  { MANY, "ether",   ETHER,   {128, 232,  64, 0}, {255, 255, 240, 0} }
};
```

The remaining field describe how the network is attached and how routeing should be done.

```
#define ONE 1          /* Only one gateway to this network */
#define MANY 0         /* Many gateways to this network */
```

In the first case, anything attached to the network will attempt to route via the port / host. In the second case (i.e. Ethernet on every switch) routeing in generated only for the host or port controllers on the switch – hence in the case of multiple Ethernet cards-on-a-switch, one should be public, one private.

2.1.3 A Switch

A switch entry consists of a (unique) name, MSNL base address if positional RARP (POSNRARP) is to be used , a null terminated list of port controllers (i.e. their structures)

```
static struct swtch b_switch = {
  "b",                          /* name */
  0,0,0,0,                      /* switch base address for POSNRARP */
  {
    &p_bass, &p_bloater, &p_bream, &p_bummalo,
    0                           /* null terminated array of ports */
  },
  0
};

static struct swtch grange_switch = {
  "grange",
  128,232,133,96,
  {
    &p_grange00, &p_grange01, &p_grange02, &p_grange03,
    &p_grange04, &p_grange05, &p_grange06, &p_grange07,
    &p_grange08, &p_grange09, &p_grange10, &p_grange11,
    &p_grange12, &p_grange13, &p_grange14, &p_grange15, 0
  },
  0
};
```

3 Wanda Dumb Boot Service

WDBS is a Wanda application that implements the subset of the standard Wanda Bootserver. A WDBS is a Wanda application(see directory .../master/wapps/wboot) built containing the kernel and hostinfo files for the systems to be booted and a trivial directory structure to find these files based on the requestors MSNL address.

The wbtool (see wbtool.8) takes a set of binary files (it strips the a.out header from these), and a list of text files and generates an assembler file from hell (order of 1Mbyte ASCII) – this provides part of the interface defined in .../wapps/wboot/wdbf.h:

```
extern int   wdbf_num;
extern char *wdbf_names[];
extern long  wdbf_len[];
extern char *wdbf_data[];
```

These are the obvious things: number of files, array of names, array of length, base pointers to data.

The directory is provided by the `struct wdbf_dir` structure:

```
#define WDB_MAX_IMAGES 16
#define WDB_END (WDB_MAX_IMAGES + 1)

typedef struct wdbf_dir {
  unsigned char msnl_addr[4];
  int  bufs[WDB_MAX_IMAGES];
} WDBF_DIR;

extern int    wdbf_dir_len;
extern struct wdbf_dir wdbf_dirs[];
```

This can be constructed by hand, but for switches this is done automatically by the sc program, in the generated file <switch>-wdbc.c. The WDBS the uses this structure to lookup the files that should be a-wdbf.appended together for booting according to the MSNL address; the files are provided by the list of indices (up to 15) into the `wdbf_names`, `wdbf_len`, and `wdbf_data` arrays, terminated with a WDB_END.

The normal use of wbtool is to load one kernel and separate hostinfo files for each machine. A script hi can be used to generate the hostinfo files. Hence a plausible sequence of events is:

```
hi city00 > city00 #sc
hi city01 > city01 #sc
hi city02 > city02 #sc
hi city03 > city03 #sc
wbtool -a city_fpcNgr -h city00 city01 city02 city03 > city-wdbf.s #sc
```

The `-a` argument tells wbtool to generate ARM assembler, then a single binary file city_fpcNgr, the kernel, then the four hostinfo files.

In fact, the city-wdbc.c file:

```
/*
 * Generated by sc tool v$Id: sc.c,v 1.14 1993/12/06 16:39:58 dm Exp $
```

```
 * Generated by sc tool at Wed Feb  9 13:46:53 1994
 *
 * These lines magic

hi city00 > city00 #sc
hi city01 > city01 #sc
hi city02 > city02 #sc
hi city03 > city03 #sc
wbtool -a city_fpcNgr -h city00 city01 city02 city03 > city-wdbf.s #sc

 *
 */

#include "wdbf.h"

struct wdbf_dir wdbf_dirs[] = {
  { {128, 232, 133,  48}, {0, 1, WDB_END, } },
  { {128, 232, 133,  49}, {0, 2, WDB_END, } },
  { {128, 232, 133,  50}, {0, 3, WDB_END, } },
  { {128, 232, 133,  51}, {0, 4, WDB_END, } }
};

int wdbf_dir_len = sizeof (wdbf_dirs) / sizeof (struct wdbf_dir);
```

contains the magic needed for the WDBS directory, together with the required lines for generating the WDBS file system – the standard Makefile will extract these lines and execute them in response to make <switch> (in this case make city).