Configuration E

Introduction

This appendix contains information about configuring STREAMS drivers and modules into SunOS 5.x. This chapter discusses how to configure a driver and a module for the STREAMS framework only. For more in-depth information on the general configuration mechanism, see *Writing Device Drivers*.

This appendix also includes a list of STREAMS related tunable parameters and describes the autopush facility.

Configuring STREAMS Drivers and Modules

The following sections contain descriptions of the pointer relationships maintained by the kernel and the various data structures used in STREAMS drivers. For the kernel to access a driver, it uses a sequence of pointers in various data structures. Look first at the data structure relationship and then the entry point interface for loading the driver into the kernel and accessing the driver from the application level.

The order of data structure traversal the kernel uses to get to a driver is as follows:

1. `modlinkage` - contains the revision number and a list of drivers to dynamically load. It is used by mod_install in the _init() routine to load the module into the kernel. Points to a `modldrv` or `modlstrmod`.

2. `modldrv` - contains info about the driver being loaded, points to the devops structure
3. `modlstrmod` - points to an fmodsw structure (which points to a streamtab) Only used by STREAMS modules.
4. `devops` - contains list of entry points for a driver, such as identify, attach, and info. Also points to a cb_ops structure.
5. `cb_ops` - points to list of threadable entry points to driver, like open, close, read, write, ioctl. Also points to the streamtab
6. `streamtab` - points to the read and write queue init structures
7. `qinit` - points to the entry points of the STREAMS portion of the driver, such as put, srv, open, close, as well as the mod_info structure. These entry points only process messages.

Each STREAMS driver or module contains the linkage connections for the various data structures. This contains a list of pointers to `dev_ops` structures. Within each `dev_ops` structure, there is a pointer to the `cb_ops` structure. Within the `cb_ops` structure, there is a pointer named `streamtab`. If the driver in question is *not* a STREAMS driver, then `streamtab` will be `NULL`. If the driver *is* a STREAMS driver, then `streamtab` will point to a structure that contains initialization routines for the driver.

Data Structure Layout

Now look at the structures presented above and see what they consist of and how they fit together. Starting with the `modlinkage` structure, we will work through the chain of pointers connecting all the different data structures. We will then look at the *loadable module wrappers* (for regular device drivers) and STREAMS module wrappers.

modlinkage

This is the definition of `modlinkage`. See `modlinkage`(9S).

```
struct modlinkage {
    int         ml_rev;     /* rev of loadable modules system */
    void        *ml_linkage[4];/* NULL terminated list of
```

```
                                        * linkage structures */
};
```

modldrv

This is the definition of `modldrv`(9S).

```
struct modldrv {
    struct mod_ops              *drv_modops;
    char                        *drv_linkinfo;
    struct dev_ops              *drv_dev_ops;
};
```

modlstrmod

This is the definition of `modlstrmod`(9S). It does not access `devops` structures because modules can only be pushed onto an existing stream.

```
struct modlstrmod {
    struct mod_ops              *strmod_modops;
    char                        *strmod_linkinfo;
    struct fmodsw               *strmod_fmodsw;
};
```

dev_ops

The first structure is `dev_ops`(9S). It represents a specific class or type of device. Each `dev_ops` structure represents a unique device to the operating system. Each device has its own `dev_ops` structure. Within each dev_ops structure is a cb_ops.

```
struct dev_ops  {
  int       devo_rev;               /* Driver build version*/
  int       devo_refcnt;            /* device reference count*/
```

```
  int       (*devo_getinfo)(dev_info_t *dip,
               ddi_info_cmd_t infocmd, void *arg, void **result);
  int       (*devo_identify)(dev_info_t *dip);
  int       (*devo_probe)(dev_info_t *dip);
  int    (*devo_attach)(dev_info_t *dip, ddi_attach_cmd_t cmd);
  int    (*devo_detach)(dev_info_t *dip, ddi_detach_cmd_t cmd);
  int    (*devo_reset)(dev_info_t *dip, ddi_reset_cmd_t cmd);

  struct cb_ops*devo_cb_ops;      /* cb_ops ptr for leaf driver*/
  struct bus_ops   *devo_bus_ops; /* ptr for nexus drivers */
};
```

cb_ops

The `cb_ops`(9S) structure is the SunOS 5.x version of the `cdevsw` and `bdevsw` tables from previous versions of System V. It contains character and block device information. It contains the driver entry points for non-STREAMS drivers.

```
struct cb_ops  {
 int    *cb_open)(dev_t *devp, int flag, int otyp, cred_t *credp);
 int   (*cb_close)(dev_t dev, int flag, int otyp, cred_t *credp);
 int      (*cb_strategy)(struct buf *bp);
 int        (*cb_print)(dev_t dev, char *str);
 int (*cb_dump)(dev_t dev, caddr_t addr,daddr_t blkno, int nblk);
 int    (*cb_read)(dev_t dev, struct uio *uiop, cred_t *credp);
 int    (*cb_write)(dev_t dev, struct uio *uiop, cred_t *credp);
 int    (*cb_ioctl)(dev_t dev, int cmd, int arg, int mode,
              cred_t *credp, int *rvalp);
 int    (*cb_devmap)(dev_t dev, dev_info_t *dip,
            ddi_devmap_data_t *dvdp, ddi_devmap_cmd_t cmd,
            off_t offset, unsigned int len, unsigned int prot,
            cred_t *credp);
 int    (*cb_mmap)(dev_t dev, off_t off, int prot);
 int    (*cb_segmap)(dev_t dev, off_t off, struct as *asp,
         caddr_t *addrp, off_t len, unsigned int prot,
         unsigned int maxprot, unsigned int flags, cred_t *credp);
 int    (*cb_chpoll)(dev_t dev, short events, int anyyet,
            short *reventsp, struct pollhead **phpp);
```

```
 int    (*cb_prop_op)(dev_t dev, dev_info_t *dip,
            ddi_prop_op_t prop_op, int mod_flags,
            char *name, caddr_t valuep, int *length);

 struct streamtab *cb_str;           /* streams information */

    /*
     * The cb_flag fields are here to tell the system a
     * bit about the device. The bit definitions are
     * in <sys/conf.h>.
     */
 int        cb_flag;           /* driver compatability flag */
};
```

streamtab

The streamtab structure contains pointers to the structures that hold the routines that actually initialize the reading and writing for module. These definitions are also in Appendix A, “STREAMS Data Structures".

If `streamtab` is `NULL`, then it signifies no STREAMS routines and the entire driver is treated as though it were a regular driver. The `streamtab` is used to indirectly identify the appropriate `open`, `close`, `put`, `service`, and administration routines. These driver and module routines should generally be declared `static`.

```
struct streamtab {
    struct qinit          *st_rdinit;   /* defines read queue */
    struct qinit          *st_wrinit;   /* defines write queue */
    struct qinit          *st_muxrinit; /* for multiplexing */
    struct qinit          *st_muxwinit; /* drivers only */
};
```

qinit

The `qinit` structure (also shown in Appendix A) contains pointers to the STREAMS entry points. These routines are called by the module loading code in the kernel.

```
struct qinit {
    int                 (*qi_putp)();      /* put procedure */
    int                 (*qi_srvp)();      /* service procedure */
    int             (*qi_qopen)();/*called on each open or push*/
    int             (*qi_qclose)();/*called on last close or pop*/
    int             (*qi_qadmin)();/* reserved for future use */
    struct module_info *qi_minfo;/* info struct */
    struct module_stat *qi_mstat; /*stats struct (opt)*/
};
```

Entry Points

As described in Chapter 9, “Drivers", and as also seen in the previous data structures, there are four types of entry points:

1. Kernel module loading - `_init`(9E), `_fini`(9E), `_info`(9E)
2. `dev_ops` - `identify`(9E), `attach`(9E), `getinfo`(9E).
3. `cb_ops` - `xxopen`(), `xxclose`(), `xxread`(), `xxwrite`(), `xxioctl`().
4. `streamtab` - `xxput`(), `xxsrv`().

pts example

Now look at a real example taken from SunOS 5.x. This is the driver pts, which is the pseudo terminal slave driver.

```
/*
 * Slave Stream Pseudo Terminal Module
 */

#include <sys/types.h>
#include <sys/param.h>
```

```
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/stat.h>
#include <sys/errno.h>
#include <sys/debug.h>
#include <sys/cmn_err.h>
#include <sys/modctl.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

static int ptsopen (queue_t*, dev_t*, int, int, cred_tstatic int
ptsclose (queue_t*, int, cred_t*);
static int ptswput (queue_t*, mblk_t*);
static int ptsrsrv (queue_t*);
static int ptswsrv (queue_t*);

static int pts_devinfo(dev_info_t *dip, ddi_info_cmd_t infocmd,
      void *arg,void **result);

static struct module_info pts_info = {
    0xface,
    "pts",
    0,
    512,
    512,
    128
};

static struct qinit ptsrint = {
    NULL,
    ptsrsrv,
    ptsopen,
    ptsclose,
    NULL,
    &pts_info,
    NULL
};

static struct qinit ptswint = {
    ptswput,
    ptswsrv,
    NULL,
    NULL,
    NULL,
```

```
    &pts_info,
    NULL
};

static struct streamtab ptsinfo = {
    &ptsrint,
    &ptswint,
    NULL,
    NULL
};

static int pts_identify(dev_info_t *devi);
static int pts_attach(dev_info_t *devi, ddi_attach_cmd_t cmd);
static int pts_detach(dev_info_t *devi, ddi_detach_cmd_t cmd);
static dev_info_t *pts_dip;                 /* private copy of
devinfo ptr */

extern kmutex_t pt_lock;
extern pt_cnt;
static struct       cb_ops   cb_pts_ops = {
       nulldev,     /* cb_open */
       nulldev,     /* cb_close */
       nodev,       /* cb_strategy */
       nodev,       /* cb_print */
       nodev,       /* cb_dump */
       nodev,       /* cb_read */
       nodev,       /* cb_write */
       nodev,       /* cb_ioctl */
       nodev,       /* cb_devmap */
       nodev,       /* cb_mmap */
       nodev,       /* cb_segmap */
       nochpoll,    /* cb_chpoll */
       ddi_prop_op, /* cb_prop_op */
       &ptsinfo,    /* cb_stream */
       D_MP         /* cb_flag */
};

static struct       dev_ops  pts_ops = {
       DEVO_REV,    /* devo_rev */
       0,           /* devo_refcnt */
       pts_devinfo, /* devo_getinfo */
       pts_identify, /* devo_identify */
       nulldev,     /* devo_probe */
       pts_attach,   /* devo_attach */
       pts_detach,   /* devo_detach */
```

```
        nodev,          /* devo_reset */
        &cb_pts_ops,    /* devo_cb_ops */
        (struct bus_ops*) NULL    /* devo_bus_ops */
};

/*
 * Module linkage information for the kernel.
 */

static struct modldrv modldrv = {
    &mod_driverops, /* Type of module: a pseudo driver */
    "Slave Stream Pseudo Terminal driver'pts'",
    &pts_ops,/* driver ops */
};

static struct modlinkage modlinkage = {
    MODREV_1,
    (void *)&modldrv,
    NULL
};

int
_init(void)
{
    return (mod_install(&modlinkage));
}

int
_fini(void)
{
    return (mod_remove(&modlinkage));
}

int
_info(struct modinfo *modinfop)
{
    return (mod_info(&modlinkage, modinfop));
}

static int
pts_identify(dev_info_t *devi)
{
    if (strcmp(ddi_get_name(devi), "pts") == 0)
            return (DDI_IDENTIFIED);
    else
```

```
            return (DDI_NOT_IDENTIFIED);
}

static int
pts_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
{
    int i;
    char name[5];

    if (cmd != DDI_ATTACH)
            return (DDI_FAILURE);

    for (i = 0; i < pt_cnt; i++) {
            (void) sprintf(name, "%d", i);
            if (ddi_create_minor_node(devi, name, S_IFCHR,
                i, NULL, 0) == DDI_FAILURE) {
                        ddi_remove_minor_node(devi, NULL);
                        return (DDI_FAILURE);
            }
    }
    return (DDI_SUCCESS);
}

static int
pts_detach(dev_info_t *devi, ddi_detach_cmd_t cmd)
{
    ddi_remove_minor_node(devi, NULL);
    return (DDI_SUCCESS);
}

static int
pts_devinfo (dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg,
         void **result)
{
           int error;

         switch (infocmd)   {
         case DDI_INFO_DEVT2DEVINFO:
                 if (pts_dip == NULL) {
                         error = DDI_FAILURE;
                 } else {
                         *result = (void *) pts_dip;
                         error = DDI_SUCCESS;
                 }
                 break;
```

```
        case DDI_INFO_DEVT2INSTANCE:
                *result = (void *) 0:
                error = DDI_SUCCESS;
                break;
        default:
                error = DDI_FAILURE;
        }
        return (error);
}

/* the open, close, wput, rsrv, and wsrv routines are presented
 * here solely for the sake of showing how they interact with the
 * configuration data structures and routines. Therefore, the
 * bulk of their code is not included.
 */
static int
ptsopen(rqp, devp, oflag, sflag, credp)
    queue_t *rqp; /* pointer to the read side queue */
    dev_t   *devp;            /* pointer to stream tail's dev */
    int  oflag;               /* the user open(2) supplied flags
*/
    int sflag;                /* open state flag */
    cred_t  *credp;/* credentials */
{
    qprocson(rqp);
    return (0);
}

static int
ptsclose(rqp, flag, credp)
    queue_t  *rqp;
    int      flag;
    cred_t   *credp;
{
    qprocsoff(rqp);
    return (0);
}

static int
ptswput(qp, mp)
    queue_t *qp;
    mblk_t  *mp;
{
    return (0);
}
```

```
static int
ptsrsrv(qp)
    queue_t *qp;
{
    return (0);
}

static int
ptswsrv(qp)
    queue_t *qp;
{
    return (0);
}
```

STREAMS Module Configuration

Here are the structures if you are working with a module instead of a driver. Notice that it is an `modlstrmod` used in `modlinkage` and `fmodsw` points to `streamtab` instead of going through `devops`.

```
extern struct streamtab pteminfo;

static struct fmodsw fsw = {
    "ptem",
    &pteminfo,
    D_NEW | D_MP
};

/*
 * Module linkage information for the kernel.
 */
extern struct mod_ops mod_strmodops;

static struct modlstrmod modlstrmod = {
    &mod_strmodops, "pty hardware emulator", &fsw
};

static struct modlinkage modlinkage = {
    MODREV_1, (void *)&modlstrmod, NULL
};
```

Compilation

Here are some compile, assemble and link lines for an example driver with two C modules and an assembly language module.

```
cc -D_KERNEL -c example_one.c
cc -D_KERNEL -c example_two.c
as -P -D_ASM -D_KERNEL -I. -o example_asm.o example_asm.s
ld -r -o example example_one.o example_two.o example_asm.o
```

Kernel Loading

See *Writing Device Drivers* for more information on the sequence of installing and loading device drivers. The procedures are to copy your driver or module to `/kernel/drv` or `/kernel/strmod`, respectively, and for drivers run `add_drv`(1M).

Checking module type

Next, see the code that enables a driver to determine if it is running as a regular driver, a module, or a cloneable driver. The open routine returns the *sflag* which is checked.

```
    if (sflag == MODOPEN)
            /* then the module is being pushed */
    else if (sflag == CLONEOPEN)
            /* then its being opened as a cloneable driver */
    else
            /* its being opened as a regular driver */
```

Tunable Parameters

Certain system parameters referenced by STREAMS are configurable when building a new operating system (see the file `/etc/system` and the *SunOS User's Guide to System Administration* for further details). These parameters are:

`nstrpush`
Maximum number (should be at least 8) of modules that may be pushed onto a single Stream.

`strmsgsz`
Maximum number of bytes of information that a single system call can pass to a Stream to be placed into the data part of a message (in `M_DATA` blocks). Any `write`(2) exceeding this size will be broken into multiple messages. A `putmsg`(2) with a data part exceeding this size will fail with `ERANGE`. If `STRMSGSZ` is set to 0, then the number of bytes passed to a Stream is effectively infinite.

`strctlsz`
Maximum number of bytes of information that a single system call can pass to a Stream to be placed into the control part of a message (in an M_PROTO or M_PCPROTO block). A `putmsg`(2) with a control part exceeding this size will fail with ERANGE.

Autopush Facility

The Autopush facility (see `autopush`(1M)) is a general mechanism that configures the list of modules for a STREAMS device. It automatically pushes a pre-specified list (`/etc/iu.ap`) of modules onto the Stream when the STREAMS device is opened and the device is not already open.

The STREAMS Administrative Driver (SAD) (see `sad`(7)) provides an interface to the autopush mechanism. System administrators can open the SAD driver and set or get autopush information on other drivers. The SAD driver caches the list of modules to push for each driver. When the driver is opened, if not already open, the Stream head checks the SAD's cache to see if the device opened has been configured to have modules pushed automatically. If an entry is found, the modules are pushed. If the device has already been opened but has not yet been closed, another open would not cause the list of the pre-specified modules to be pushed again.

Three options are available to configure the module list:

- Configure for each minor device - that is, a specific major and minor device number.
- Configure for a range of minor devices within a major device.
- Configure for all minor devices within a major device.

When the configuration list is cleared, a range of minor devices has to be cleared as a range and not in parts.

User Interface

The SAD driver can be accessed via the node `/dev/sad/admin` or `/dev/sad/user`. After the device is initialized, a program can be run to perform any needed autopush configuration. The program should open the SAD driver, read a configuration file to find out what modules are needed to be configured for which devices, format the information into `strapush` structures, and perform the necessary SAD_SAP `ioctl`s. See `sad`(7) for more information.

All autopush operations are performed through an `ioctl`(2) command to set or get autopush information. Only the superuser may set autopush information, but any user may get the autopush information for a device.

The `ioctl` is a form of `ioctl(fd, cmd, arg)`, where *fd* is the file descriptor of the SAD driver, *cmd* is either SAD_SAP (set autopush information) or SAD_GAP (get autopush information), and *arg* is a pointer to the structure `strapush`.

The structure `strapush` is defined as:

```
/*
 * maximum number of modules that can be pushed on a
 * Stream using the autopush feature should be no greater
 * than nstrpush
 */
 #define MAXAPUSH 8

 /* autopush information common to user and kernel */

 struct apcommon {
    uint apc_cmd;       /* command - see below */
    long apc_major;     /* major device number */
    long apc_minor;     /* minor device number */
    long apc_lastminor; /* last minor dev # for range */
    uint apc_npush;     /* number of modules to push */
 };
```

```
 /* ap_cmd - various options of autopush */
 #define SAP_CLEAR               0 /* remove configuration list
*/
 #define SAP_ONE                 1 /* configure one minor device
*/
 #define SAP_RANGE               2 /* config range of minor
devices */
 #define SAP_ALL                 3 /* configure all minor
devices */

 /* format of autopush ioctls */
 struct strapush {
    struct apcommon sap_common;
    char sap_list[MAXAPUSH] [FMNAMESZ + 1]; /* module list */
 };

 #define sap_cmd                          sap_common.apc_cmd
 #define sap_major                        sap_common.apc_major
 #define sap_minor                        sap_common.apc_minor
 #define sap_lastminor
sap_common.apc_lastminor
 #define sap_npush                        sap_common.apc_npush
```

A device is identified by its major device number, *sap_major*. The `SAD_SAP ioctl` (*sap_cmd*) can take the following options:

- `SAP_ONE` configures a single minor device, *sap_minor*, of a driver.
- `SAP_RANGE` configures a range of minor devices from *sap_minor* to *sap_lastminor*, inclusive.
- `SAP_ALL` configures all minor devices of a device.
- `SAP_CLEAR` clears the previous settings by removing the entry with the matching *sap_major* and *sap_minor* fields.

The list of modules is specified as a list of module names in *sap_list*. The maximum number of modules to push automatically is defined by MAXAPUSH.

A user may query the current configuration status of a given major/minor device by issuing the SAD_GAP `ioctl` with *sap_major* and *sap_minor* values of the device set. On successful return from this system call, the `strapush`

structure will be filled in with the corresponding information for that device. The maximum number of entries the SAD driver can cache is determined by the tunable parameter `NAUTOPUSH` found in the SAD driver's master file.

The following is an example of an autopush configuration file in `/etc/iu.ap`:

```
# /dev/console and /dev/contty autopush setup
#
#   major      minor          lastminor      modules

    wc         0              0              ldterm ttcompat
    zs         0              1              ldterm ttcompat
    ptsl       0              15             ldterm ttcompat
```

The first line represents the configuration for a single minor device whose major name is `wc` and minor numbers start at 0 and end at 0, thus creating only one minor number. The modules automatically pushed are `ldterm` and `ttcompat`. The second line represents the configuration for the zs driver. The minor device numbers will be 0 and 1, and automatically pushing the modules in the list. The last line allows minor device numbers to be used from 0 to 15 for the ptsl driver.

E