Addendum to Programming in the DG/UX™ Kernel Environment

086-000426-00

This addendum updates manual 093-701083-00. See Updating Instructions on reverse.

Addendum to Programming in the DG/UX™ Kernel Environment

086-000426-00

Revision History:	Effective with:
Original Release – February 1992	DG/UX Release 5.4.1

Updating Instructions

This addendum updates *Programming in the DG/UX Kernel Environment* (093-701083-00) with a new appendix, Appendix D, that describes how to take advantage of the symmetric multiprocessing environment with the DG/UX STREAMS facility.

To update your copy of 093-701083-00, please remove the manual pages listed below and replace them with addendum pages as follows:

REMOVE	**INSERT**
Title/Notice Page	Title/Notice Page
xii	xii
-	D-1/D25

Insert this instruction sheet immediately behind the new Title/Notice page.

Programming in the DG/UX™ Kernel Environment

093-701083-00

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-701083

Revision 00, March 1991

Programming in the DG/UX™ Kernel Environment

093-701083-00

Revision History:	Effective with:
Original Release – March 1991	DG/UX Release 5.4
Addendum 083-000426 – Feb. 1992	DG/UX Release 5.4.1

Contents

Figures

Appendix D
Using STREAMS in the DG/UX Multiprocessor Environment

DG/UX STREAMS is fully compatible with industry standard STREAMS. As Chapter 1 notes, the *Programmer's Guide: STREAMS (UNIX System V Release 4)* describes how to write code for such industry standard STREAMS. You need only create a master file entry with default settings for each STREAMS module and driver to integrate industry standard code transparently into the DG/UX system.

However, DG/UX STREAMS also allows you to access some additional features of the DG/UX kernel that are not available under the basic industry standard STREAMS. Chapter 1 mentions that you can add some additional routines via a routines vector. The *Writing a Device Driver for the DG/UX System* manual describes how to create this routines vector and these additional routines. This appendix looks at a second, more important option under DG/UX STREAMS, how to take advantage of the symmetric multiprocessor environment without changing your code. The feature that allows you to do this is called *concurrency sets*. Concurrency sets are controlled solely through your master file entries.

This appendix describes what concurrency sets are and how to use them and then discusses how different concurrency sets affect performance.

What Are Concurrency Sets?

Industry standard STREAMS does not yet support symmetric multiprocessing and portability is very important in a STREAMS program. As we've seen, one of the keys to programming in a fully symmetric environment is guaranteeing that only one process has access to critical sections of code or shared data at the same time. Normally, to run concurrently you would have to add locks to your code to guarantee exclusivity when the process is accessing critical code or data. But adding locks makes the code less portable. Concurrency sets allow you to define which modules/drivers can run concurrently and which cannot by setting a flag in their master file entry. Thus, concurrency sets support multiprocessing STREAMS while maintaining complete portability.

A concurrency set is most easily understood as a set of modules that are associated with a common lock. Notice that different modules sharing a common lock means that modules in the same set cannot run at the same time because only one member of the set can hold the common lock at a time. On the other hand, modules that do not share the same lock can run at the same tiem. Thus, paradoxically members of the *same* concurrency set *cannot* run concurrently while members of *different* concurrency sets *can*.

The master file entry lets the kernel's STREAMS management code know which modules/drivers share a common lock. Whenever a message is passed between modules, this management code checks to see if the receiving module's lock is available. If it is, the new module can run. If it is not, the new module is put on a deferral list awaiting its lock's release.

This is the basic idea of concurrency sets. In order to describe their use we need to look more carefully at STREAMS modules and drivers in action. To do this let us first introduce some basic terminology.

Terminology

The three basic components of a stream are: a stream head, modules, and a driver.

- **Stream Head** — The stream head is the code segment at the top of the stream. It is the interface between user space and kernel space for a stream and provides synchronization between the stream and kernel/user space.

- **Module** — A module is a stream component that manipulates data. Users can push one or more modules onto the stream between the stream head and the driver using a special **ioctl** system call. Every module has a read queue and a write queue. The read queue holds data that is going up the stream and the write queue holds data that is going down the stream. A module's job is to respond to messages coming in on its own read/write queues and send messages to the read/write queues of modules above and below it on the stream.

- **Driver** — The driver is a special case of a module; it is the module at bottom of the stream. Because it is at the bottom, it serves as the external interface for the stream. Drivers often address hardware devices, but they may also address pseudodevices.

- **Messages** — Messages are the items that are passed along the stream. They contain data and other state information about the type of message etc. Messages can be passed downstream (from the stream head towards the driver) or upstream (from the driver towards the stream head). Modules perform various operations on message information as it is passed up and down the stream.

Figure D-1 shows a basic stream. The stream head is at the top, the driver is at the bottom, and in between you have some number of optional modules.

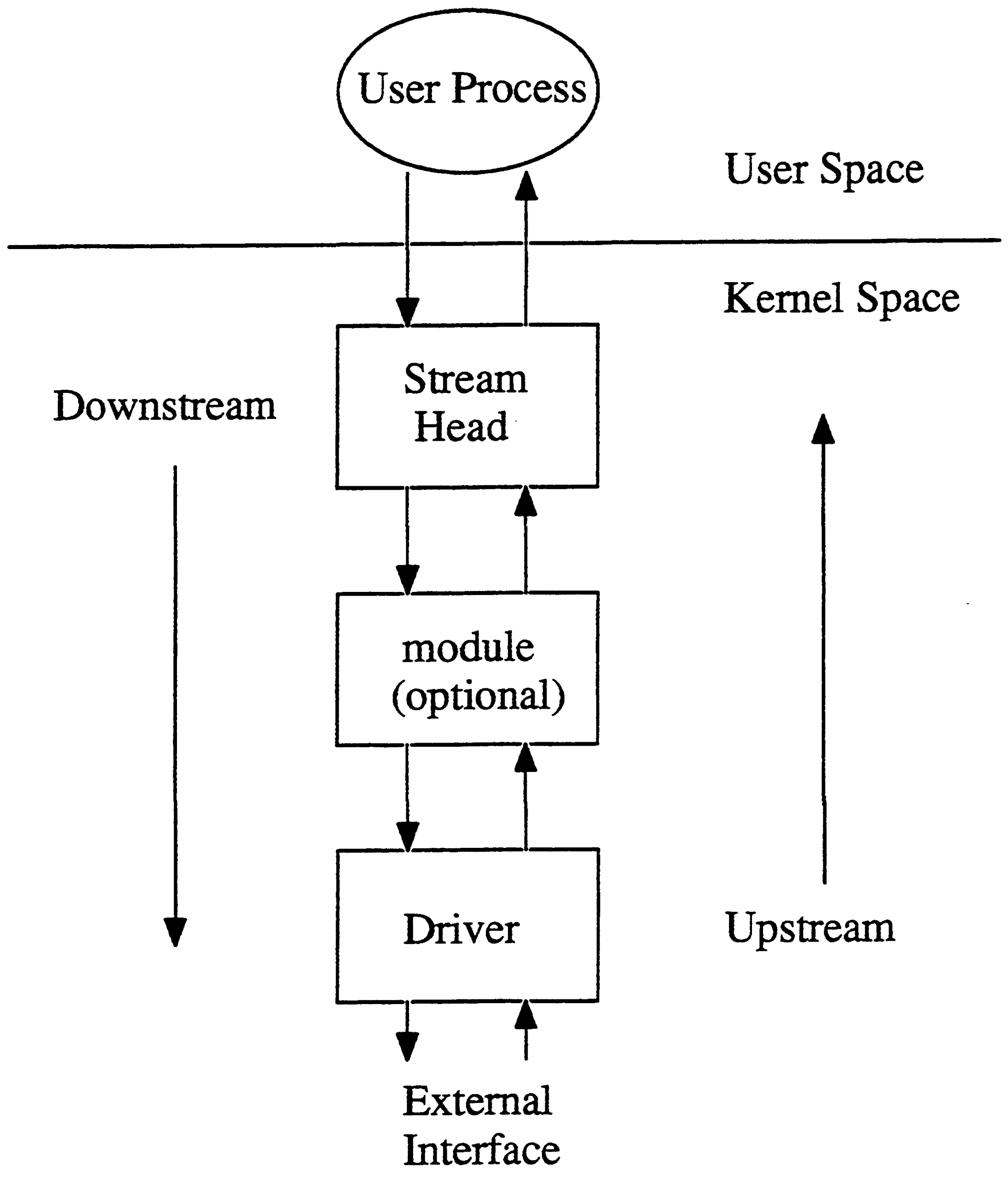

Figure D-1 *Basic STREAM Layout*

Figure D-2 shows an example of a more complicated set of streams, the type that frequently arise in real world applications. This example shows multiple streams feeding into a TCP/IP communications driver.

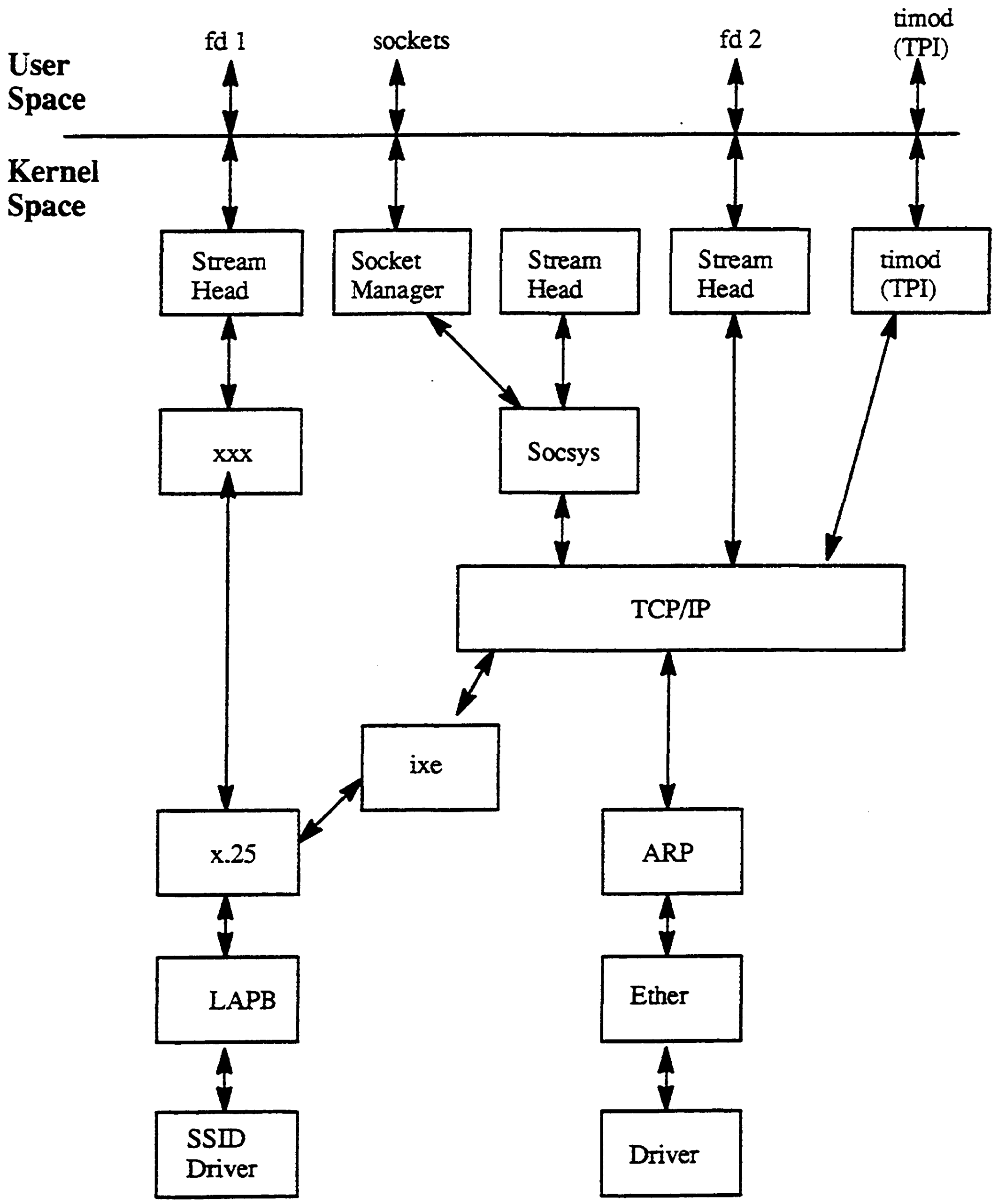

Figure D-2 *Multiplexor STREAM*

Because several streams end at TCP/IP, it is a driver. But the TCP/IP module also branches out into several other streams. A module that maps one or more upper streams to one or more lower streams is called a multiplexor.

How Different Types of Concurrency Sets Work

A concurrency set associates a lock with a set of modules. It provides mutual exclusion for all the modules belonging to the set. A single lock protects all the modules in the set from colliding with each other.

There are four types of concurrency sets, that is, four ways that you can associate a lock with a module. The four types are: per-stream, per-module, set and default concurrency. Note that the label "none" is reserved for future use and should not be used as a concurrency set name.

You define a module/driver's concurrency in its master file entry by entering "stream" for per-stream, "module" for per-module, a set name for set concurrency and "default" for default concurrency. Thus, in the example shown below, **hken** is per-stream, **sd** is per-module, the **xdev** driver belongs to a set concurrency set named "george" and the **cird** driver used "default" concurrency.

```
#-------------------------------------------------------
# Disks:
#  Name              Restriction    Concurrency
#  Prefix            Flags          Set
#  ----------        --------       ------------
   hken              n              stream
   inen              n              module
   ttcodpat          n              george
   ldterm            n              default
#
#-------------------------------------------------------
```

If a module or driver has per-stream concurrency, it uses the same lock as its stream head. Figure D-3 shows per-stream concurrency.

Per–stream Concurrency

* Within each stream, the stream–head, module, and driver all share a common lock.

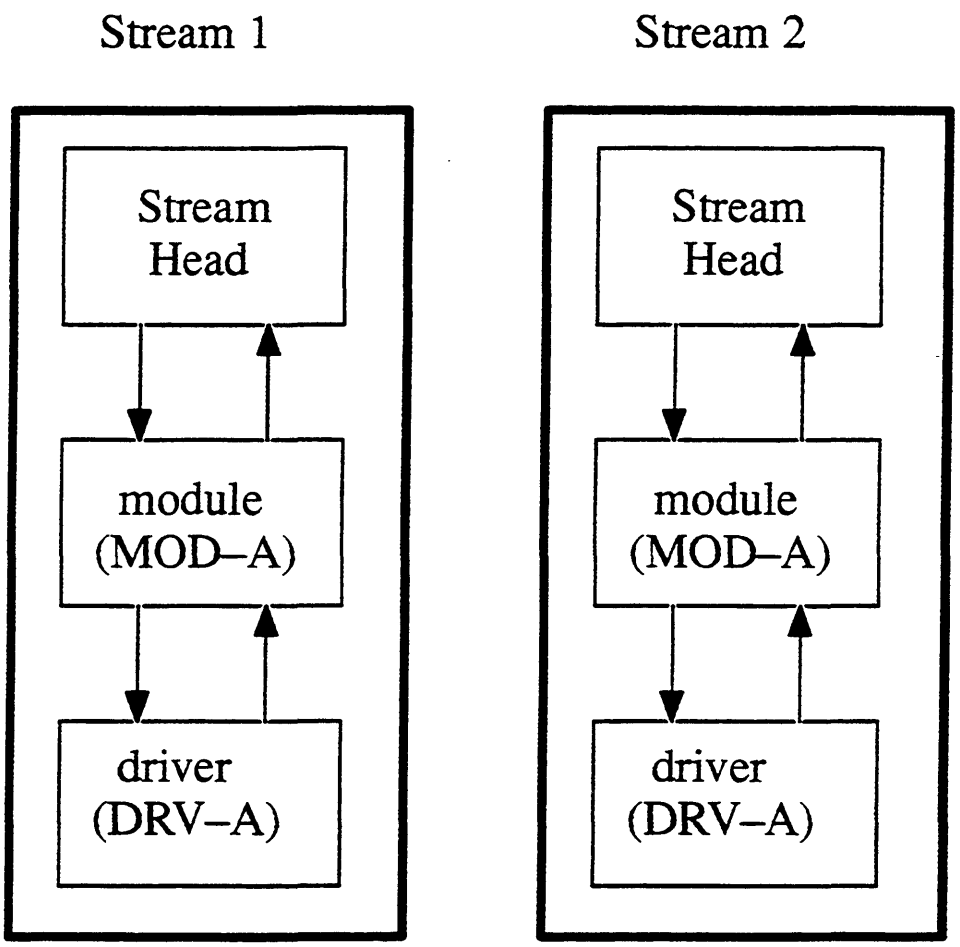

Figure D-3 *Per-stream Concurrency*

A stream-head is by definition in per-stream concurrency. This cannot be changed. In this example, the module, MOD-A, and the driver, DRV-A, have also been defined as per-stream concurrency in their master file entries. Notice that both streams have instances of MOD-A and DRV-A. Per-stream concurrency means that the instances of MOD-A and DRV-A share a lock with their respective stream heads. The big box around each stream shows that the stream's components are in the same concurrency set and in a different concurrency set from the neighboring stream. This means that STREAM-1 and STREAM-2 can run concurrently because they use different locks. Conversely, the stream-head, MOD-A, and DRV-A in each stream run separately because they all share the same lock.

In per-module concurrency, all instances of a module or driver have the same lock. Figure D-4 shows per-module concurrency.

Per–module Concurrency

* All instances of a module share a common lock.

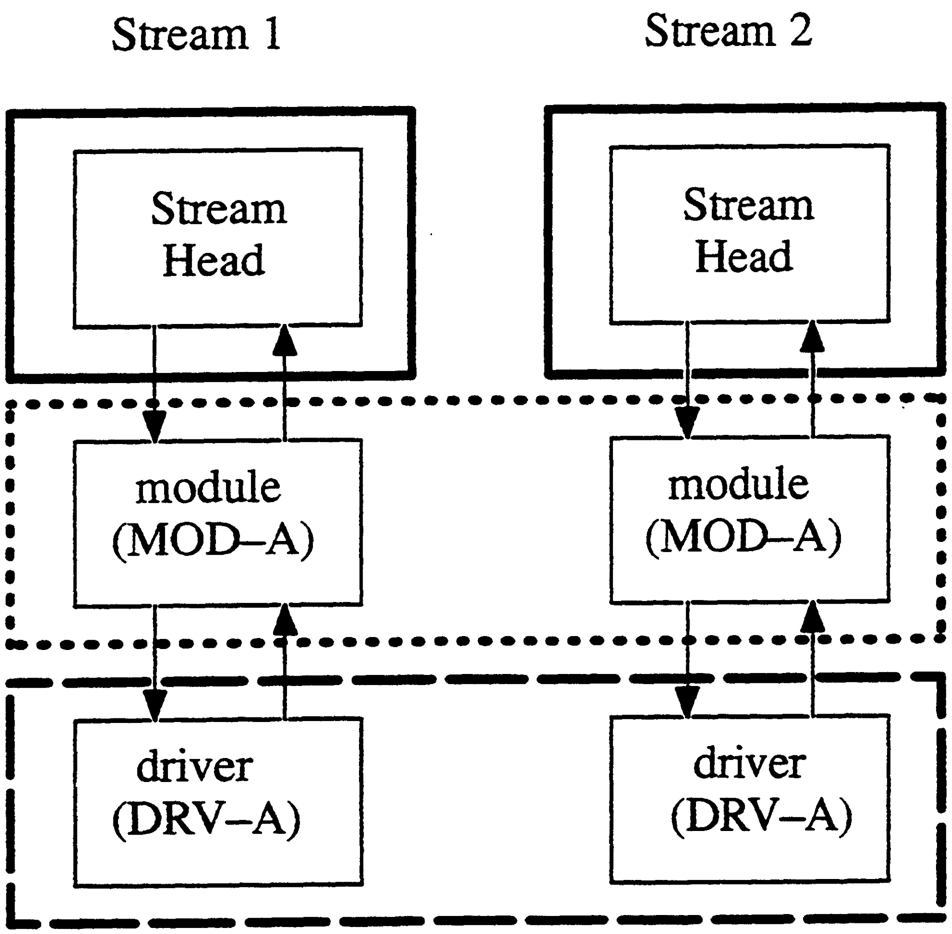

Figure D-4 *Per-module Concurrency*

The module, MOD-A and the driver, DRV-A have been defined as having per-module concurrency. Both STREAM-1 and STREAM-2 contain instances of the module, MOD-A. The box around these two instances of MOD-A show that they share the same lock, which means each instance will run exclusive of the other. The same logic holds for the two instances of DRV-A as well. The stream heads are, as always, per-stream concurrency and thus use a different lock from both MOD-A and DRV-A. Consequently, the stream-head, MOD-A, and DRV-A of STREAM-1 can all run concurrently. In this example, exclusivity holds only between instances of MOD-A and DRV-A.

Note that module concurrency for drivers is handled on a major number basis. Thus, if you have the same driver with different major numbers for it, then only instances with the same major number will have the same lock.

The final type of concurrency is called set concurrency. In set concurrency, modules belong to the set named in the master file. The set name can be any legitimate UNIX name string. Figure D-5 shows set concurrency in which modules MOD-A and MOD-B are both part of the the "george" concurrency set.

Set Concurrency

* All instances of one or more modules share a common lock.

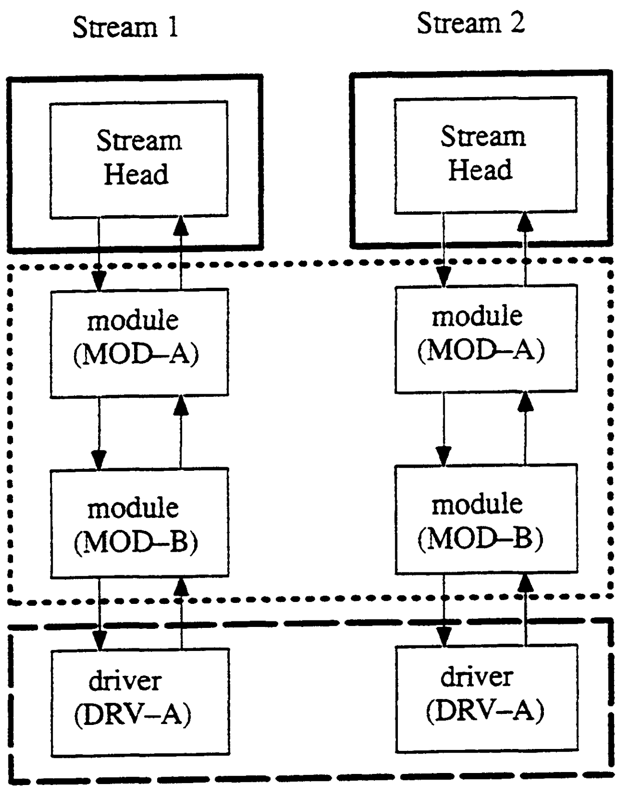

Figure D-5 *Set Concurrency*

Default concurrency is actually a special case of set concurrency. All modules with default concurrency belong to the same set. Thus, members of the default set run exclusively of each other and concurrently with other sets.

Note that if you define a driver to be in set concurrency, all instances of that driver will have the same lock regardless of the major number. Only in per-module concurrency does the major number make a difference.

Recommendations on How to Use Concurrency Sets.

We recommend you define all modules/drivers as per-stream concurrency except in the three special case conditions listed below. Using per-stream concurrency provides more concurrency than the default concurrency set and it avoids complicated overhead costs that can arise in other forms of concurrency. Note that stream heads are always defined as per-stream concurrency.

The three exceptions to this recommendation are as follows:

- All multiplexors must be defined as per-module, set or default concurrency. They cannot be defined as per-stream concurrency because they don't belong exclusively to any one stream-head (they map multiple upper streams to multiple lower streams).
- Because you must provide mutual exclusion for shared data, instances of modules that share data, or multiple modules that share data must belong to the same concurrency set.
- If you must be guaranteed that a **putnext** procedure runs immediately when called, then adjacent modules must be in the same concurrency set. The need to guarantee that the **putnext** runs immediately sometimes arises in state driven situations; for example, a module needs to know that a particular action has occurred when it thinks it has occurred, and it relies on the return value of the **putnext**. Thus, if you set up the **putnext** procedure to return an integer and it must come back accurately, then two adjacent modules must be in the same set.

Notes on Creating STREAMS Code on the DG/UX System

STREAMS kernel programming is generally the same as other kernel programming. If you are creating STREAM code from scratch and portability is not an issue, you can use fine-grained locks, eventcounters and most of the other facilities described in the body of this manual. However, there are a few kernel facilitites, such as buffer vectors and select management routines, that are not appropriate for STREAMS code. STREAMS restrictions, if any, are noted in the chapters that cover particular topics. Be sure to create master file entries for all STREAMS modules and drivers. Stream-heads do not require master file entries because their concurrency set is always per-stream concurrency. Use the system file entry to request either a default routines vector or your own. See *Writing a Standard Device Driver for the DG/UX System* for more information on how to create your own routines vector.

In addition, you should observe the following guidelines:

- Use the standard STREAMS utilities to manipulate queues and pass messages to other modules. Using your own routines to traverse the stream and manipulate queues is particularly dangerous in a multiprocessing environment. In particular, always use **putnext**() to pass messages between modules. Do not create your own **queue_t** structures and manipulate them with standard STREAMS utilities.
- Use only the **putq** and **qenable** STREAMS utilities in your interrupt handler at interrupt level. No other STREAMS utilites may be used.
- Explicitly protect shared data in interrupt handlers.
- Don't make assumptions about major number assignments for device drivers.

Notes on Porting STREAMS Code to the DG/UX System

The list of programming guidelines given above holds for ported code as well as newly created code. Check your program for violations of these guidelines before attempting to port. In particular, note the restrictions on STREAMS utilities at interrupt level.

In addition, there are four AT&T STREAMS utilities that do not hold on the DG/UX system. These are: **splstr**(), **splx**(), **sleep**(), and **wakeup**(). **Splstr**() and **splx**() are used to enable and disable interrupts. In AT&T's single-processor environment, such routines can be used to provide mutual exclusion. However, on a symmetric multiprocessor system, disabling interrupts does not provide exclusivity (as Chapter 1 describes). **Sleep**() and **wakeup**() do not work in the DG/UX environment for similar multiprocessing reasons.

Remove instances of **splstr**() and **splx**(). Review the code to see if the exclusion that these functions provided must be replaced through concurrency sets or by adding your own locks using the kernel-supplied locking routines. Replace instances of **wakeup**() with **su_str_wakeup**(). Replace instances of **sleep**() with **su_str_get_next_event**() and **su_str_sleep**() as shown in the coding examples below. Be sure to use **su_str_get_next_event**() to get the event before using **su_str_sleep**() to await it. The first example shows the AT&T version of the code, followed by the DG/UX equivalent.

```
                    AT&T CODE

n = splstr();
while ( condition )

      {
      if (sleep(resource, PCATCH))
            {
            /*a signal was detected*/
            }
       }

[optionally perform mutually exclusive operations
after event has occurred.]

splx ( n );
```

```
                    DG/UX CODE

next_event = su_str_next_event ( resource );
while ( condition )

      {
      if (su_str_sleep(resource, next_event))
            {
            /*a signal was detected*/
            }
       }
 [mutual exclusion is now guaranteed.]
```

The routines used in these examples are described on the next several pages.

su_str_sleep

Syntax

```
int        su_str_sleep(

caddr_t              resource,       /* READ ONLY*/
vp_ec_value_type     next_ec_value   /* READ ONLY*/
                     )
```

Summary

This routine awaits an event specified by the eventcount value **next_ec_value** or the eventcounter specified by **resource** or a process interrupt event.

Parameters

resource — A pointer to the resource being awaited.

next_ec_value — The value of the eventcount to await.

Description

This routine awaits a **su_str_wakeup(resource)** operation that advances the eventcounter associated with **resource** to **next_ec_value**. This routine should be used only by a module or driver open/close routine.

Return Values

0 — sleep was awakened by wakeup

1 — sleep was awakened by an interrupt signal (process is either being terminated or ut must handle a signal).

su_str_wakeup

Syntax

```
void        su_str_wakeup(

caddr_t     resource        /* READ ONLY*/
                            )
```

Summary

This routine causes the eventcount associated with **resource** to be advanced.

Parameters

resource — A pointer to a resource being freed.

Description

See Summary.

Return

None.

Exceptions

None.

su_str_next_event

Syntax

```
vp_ec_value_type      su_str_next_event(

caddr_t    resource    /* READ ONLY*/
```

Summary

This routine returns the next value of an eventcount associated with **resource**.

Parameters

resource — A pointer to a resource being sought.

Description

See Summary.

Return

The value that the eventcount associated with **resource** will attain when it is next advanced is returned.

Exceptions

None.