Trusted Solaris 2.5 Man Pages: 9FTSOL Kernel Functions

Sun Microsystems Federal, Inc.
A Sun Microsystems, Inc. Business
901 San Antonio Road, MS USJC01-201
Palo Alto, CA 94303
U.S.A.

Preface

In the Trusted Solaris Reference Manual, each collection of information on a particluar topic is called a man page, even though a man *page* may actually consist of *many pages* of text.

A man page is intended to answer concisely the question “What does it do?”. The man pages are not intended to be a tutorial. Depending what you are trying to do, refer to the other Trusted Solaris user, developer, and administrator manuals for when and why to use a command or other features described in the man pages.

ACCESSING MAN PAGES

The man pages that make up the reference manual may be accessed in three ways.

Note: The following discussion of man page viewing options uses the term **package**, which is a unit of software that is typically delivered on Sun’s product CDs. Installing the documentation packages is optional, because they are not required for operations. Each customer’s administrators decides whether or not the documentation packages are installed and made available.

The first means of accessing the man pages is through the use of the **man**(1) command. When the contents of the man page package, SUNWman, are available on the local system, anyone with a login account, plus a terminal emulator (such as **cmdtool**(1), **shelltool**(1), or **dtterm**(1)) and the **man**(1) command in one of the account’s execution profiles can view a man page on-line. (For more about Trusted Solaris execution profiles and user accounts, see the Trusted Solaris user and administrator documentation.) To view a man page, enter the **man** command followed by the name of the man page. For example, to view the **ls**(1) man page that describes the command used to print out a directory's contents, a user enters the command: **man**ls.

The second way to read man pages is in the printed Trusted Solaris Reference Manual. The reference manual is in the Trusted Solaris documentation set, and it may be ordered in hardcopy form from Sun by using part number: 805-8005-10.

The third means of reading the man pages is by viewing them in AnswerBook format. When the Trusted Solaris AnswerBook package, SUNWtab, is available on the local system, anyone with a login account and with the **answerbook**() command and a terminal emulator in an execution profile can display the Trusted Solaris reference manual and the other user documentation. For Trusted Solaris 2.5, the Trusted Solaris documentation AnswerBook is shipped on a separate documentation CD, but it may be bundled on the same CD with the Trusted Solaris software in future releases.

Trusted Solaris man pages are identified with a TSOL suffix in the section name. The TSOL suffix is used for man pages that are either new to Trusted Solaris or modified from the base man pages from the Solaris, CDE, or Solstice products that are bundled into Trusted Solaris. The man pages are organized alphabetically by section.

- Section 1TSOL describes new or modified user commands available with the Trusted Solaris operating system.

- Section 1BTSOL describes printer commands adapted for Trusted Solaris from the Berkeley Software Distribution (BSD) print subsystem, which are used chiefly for printing administration.

  Note: Use of the equivalent System V print commands is recommended (such as **lp**(1TSOL)instead of **lpr**(1BTSOL)) because although the BSD commands are included for compatability, they will be removed in future releases.

- Section 1MTSOL describes Trusted Solaris system maintenance and administration commands.

- Section 2TSOL describes Trusted Solaris system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value.

- 3*TSOL subsections describe functions found in various Trusted Solaris libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2TSOL.

Subsections include: 3CTSOL, 3NTSOL, 3RTSOL, 3TSOL, and 3X11TSOL.

- Section 4TSOL outlines the formats of various files. The C structure declarations for the file formats are given where applicable.
- Section 5TSOL contains miscellaneous documentation such as Trusted Solaris macros.
- 7*TSOL subsections describe various special files that refer to specific hardware peripherals and device drivers.

  Subsections include: 7DTSOL and 7TSOL.
- 9*TSOL subsections provide reference information for writing device drivers in the kernel operating system environment.

  Subsections include: 9FTSOL and 9TSOL.

Following is a generic list of headings on each man page. The man pages of each manual section include only the headings they need. For example, if there are no bugs to report, there is no BUGS section. See the intro pages for more information and detail about each section, and **man**(1) for more information about man pages in general.

NAME

This section gives the names of the commands or functions documented, followed by a brief description of what they do.

SYNOPSIS

This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full pathname is shown. Literal characters (commands and options) are in **bold** font and variables (arguments, parameters and substitution characters) are in *italic* font. Options and arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.

The following special characters are used in this section:

[ ]	The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument *must* be specified.

. . .	Ellipses. Several values may be provided for the previous argument, or the previous argument can be specified multiple times, for example, '*filename* . . .'.
|	Separator. Only one of the arguments separated by this character can be specified at time.
{ }	Braces. The options and/or arguments enclosed within braces are interdependent, such that everything enclosed must be treated as a unit.

PROTOCOL

This section occurs only in subsection 3R to indicate the protocol description file. The protocol specification pathname is always listed in **bold** font.

AVAILABILITY

This section briefly states any limitations on the availabilty of the command. These limitations could be hardware or software specific.

A specification of a class of hardware platform, such as **x86** or **SPARC**, denotes that the command or interface is applicable for the hardware platform specified.

In Section 1TSOL and Section 1MTSOL, **AVAILABILITY** indicates which package contains the command being described on the manual page. In order to use the command, the specified package must have been installed with the operating system. If the package was not installed, see **pkgadd**(1) for information on how to upgrade.

MT-LEVEL

This section lists the **MT-LEVEL** of the library functions described in the Section 3 manual pages. The **MT-LEVEL** defines the libraries' ability to support threads. See **Intro**(3TSOL) for more information.

DESCRIPTION

This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES. Interactive commands, subcommands, requests, macros, functions and such, are described under USAGE.

IOCTL

This section appears on pages in Section 7TSOL only. Only the device class which supplies appropriate parameters to the **ioctl**(2) system call is called **ioctl** and generates its own heading. **ioctl** calls for a specific device are listed alphabetically (on the man page for that specific device). **ioctl** calls are used for a particular class of devices all of which have an **io** ending, such as **mtio**(7).

OPTIONS

This lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option and where appropriate default values are supplied.

OPERANDS

This section lists the command operands and describes how they affect the actions of the command.

OUTPUT

This section describes the output - standard output, standard error, or output files - generated by the command.

RETURN VALUES

If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or –1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared as **void** do not return values, so they are not discussed in RETURN VALUES.

ERRORS

On failure, most functions place an error code in the global variable **errno** indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code.

USAGE

This section is provided as a *guidance* on use. This section lists special rules, features and commands that require in-depth explanations. The subsections listed below are used to explain built-in functionality:

Commands
Modifiers
Variables
Expressions
Input Grammar

EXAMPLES

This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command line entry and machine response is shown. Whenever an example is given, the prompt is shown as

example%

or if the user must be in an administrative role,

example#

Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS and USAGE sections.

ENVIRONMENT

This section lists any environment variables that the command or function affects, followed by a brief description of the effect.

EXIT STATUS

This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion and values other than zero for various error conditions.

FILES

This section lists all filenames referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation.

SEE ALSO

This section lists references to other man pages, in-house documentation, and outside publications.

DIAGNOSTICS

This section lists diagnostic messages with a brief explanation of the condition causing the error. Messages appear in **bold** font with the exception of variables, which are in *italic* font.

WARNINGS

This section lists warnings about special conditions which could seriously affect your working conditions — this is not a list of diagnostics.

NOTES

This section lists additional information that does not belong anywhere else on the page. It takes the form of an *aside* to the user, covering points of special interest. Critical information is never covered here.

BUGS

This section describes known bugs and wherever possible suggests workarounds.

SUMMARY OF TRUSTED SOLARIS CHANGES

On base man pages that have Trusted Solaris modifications, this section summarizes the changes in a single easy-to-find place on the man page.

NAME

Intro, intro – introduction to Trusted Solaris kernel functions

DESCRIPTION

Section 9FTSOL describes the new Trusted Solaris kernel functions and modified Solaris kernel functions available for use by Trusted Solaris device drivers. Base Solaris kernel functions described in **Intro**(9F) also may be used in Trusted Solaris device drivers. If a function has been modified for Trusted Solaris device drivers, a modified version of the man page exists in section 9FTSOL, and you should follow the 9FTSOL version of the man page. If a function with a **tsol_** suffix has an equivalent with the same name without the **tsol_** suffix, (for example **tsol_linkb**(9FTSOL) has an equivalent in **linkb**(9FTSOL), then the Trusted Solaris version should be used if the driver is to be trusted, so that the driver knows about Trusted Solaris attributes and enforces policy when necessary, and the base function can still be used by new or existing drivers that do not enforce policy. **Intro**(9TSOL) describes how Trusted Solaris man pages are included in section 9FTSOL.

See the RETURN VALUES, and CONTEXT sections of **Intro**(9F) for information that also applies to the use of the kernel functions described in the 9FTSOL man pages. As is stated on the base **Intro**(9F) man page: Every driver MUST include **<sys/ddi.h>** and **<sys/sunddi.h>**, in that order, and as the last files the driver includes.

Name	**Description**
copyb(9FTSOL)	copy a message block
copymsg(9FTSOL)	copy a message
dupb(9FTSOL)	duplicate a message block descriptor
dupmsg(9FTSOL)	duplicate a message
insq(9FTSOL)	insert a message into a queue
linkb(9FTSOL)	concatenate two message blocks
put(9FTSOL)	call a STREAMS put procedure
putbq(9FTSOL)	place a message at the head of a queue
putctl(9FTSOL)	send a control message to a queue
putctl1(9FTSOL)	send a control message with a one-byte parameter to a queue
putnext(9FTSOL)	send a message to the next queue
putnextctl(9FTSOL)	send a control message to a queue
putnextctl1(9FTSOL)	send a control message with a one-byte parameter to a queue
putq(9FTSOL)	put a message on a queue
tsol_get_strattr(9FTSOL)	get security attributes of a message
tsol_set_strattr(9FTSOL)	See **tsol_get_strattr**(9FTSOL)
tsol_linkb(9FTSOL)	concatenate two message blocks

tsol_putctl(9FTSOL)	send a control message to a queue
tsol_putctl1(9FTSOL)	send a control message with a one-byte parameter and a set of attributes to a queue
tsol_putnextctl(9FTSOL)	send a control message and a set of attributes to a queue
tsol_putnextctl1(9FTSOL)	send a control message with a one-byte parameter and a set of attributes to a queue

NAME

copyb – Copy a message block

SYNOPSIS

#include <sys/stream.h>

mblk_t *copyb(mblk_t ***bp*);**

ARGUMENTS

bp Pointer to the message block from which data is copied.

INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI).

DESCRIPTION

copyb( ) allocates a new message block and copies into it the data from the block that *bp* denotes. The new block will be at least as large as the block being copied. **copyb** uses the **b_rptr** and **b_wptr** members of *bp* to determine how many bytes to copy.

RETURN VALUES

If successful, **copyb** returns a pointer to the newly allocated message block containing the copied data. upon failure, **copyb** returns a **NULL** pointer.

CONTEXT

copyb can be called from user or interrupt context.

EXAMPLES

For each message in the list, use the **canputnext**(9F) function (line 21) to test whether the downstream queue is full. If it is not full, use **copyb** to copy a header message block, and **dupmsg**(9F) to duplicate the data to be retransmitted. If either operation fails, reschedule a timeout at the next valid interval.

Update the new header block with the correct destination address (line 34), link the message to it (line 35), and send it downstream (line 36). At the end of the list, reschedule this routine.

```
 1  struct retrans {
 2          mblk_t              *r_mp;
 3          long                r_address;
 4          queue_t             *r_outq;
 5          struct retrans      *r_next;
 6  };
 7
 8  struct protoheader {
        . . .
 9      long                    h_address;
        . . .
10  };
11
12  mblk_t *header;
13
14  void
15  retransmit(struct retrans *ret)
16  {
        mblk_t *bp, *mp;
        struct protoheader *php;

        while (ret) {
            if (!canputnext(ret->r_outq)) {          /* no room */
                  ret = ret->r_next;
                  continue;
            }
            bp = copyb(header);                      /* copy header msg. block */
            if (bp == NULL)
                  break;
            mp = dupmsg(ret->r_mp);                  /* duplicate data */
            if (mp == NULL) {                        /* if unsuccessful */
                  freeb(bp);                         /* free the block */
                  break;
            }
            php = (struct protoheader *)bp->b_rptr;
            php->h_address = ret->r_address;         /* new header */
            bp->bp_cont = mp;                        /* link the message */
            putnext(ret->r_outq, bp);                /* send downstream */
            ret = ret->r_next;
        }
        /* reschedule */
        (void) timeout(retransmit, (caddr_t)ret, RETRANS_TIME);
}
```

SUMMARY OF TRUSTED SOLARIS CHANGES

This routine will assign correct attributes to the message it copies.

SEE ALSO

allocb(9F), **canputnext**(9F), **dupmsg**(9F)

NAME

copymsg – Copy a message

SYNOPSIS

#include <sys/stream.h>

mblk_t *copymsg(mblk_t ***_mp_);**

ARGUMENTS

_mp_	Pointer to the message to be copied

INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI)

DESCRIPTION

copymsg( ) forms a new message by allocating new message blocks and copying (using the **copyb** function) the contents of the message to which _mp_ refers. **copymsg** returns a pointer to the new message.

RETURN VALUES

If the copy is successful, **copymsg** returns a pointer to the new message. Upon failure, **copymsg** returns a **NULL** pointer.

CONTEXT

copymsg can be called from user or interrupt context.

EXAMPLES

The routine **lctouc**( ) converts all the lowercase ASCII characters in the message to uppercase. If the reference count is greater than one (line 8), then the message is shared and must be copied before changing the contents of the data buffer. If the call to the **copymsg**(9F) function fails (line 9), return **NULL** (line 10), otherwise, free the original message (line 11). If the reference count was equal to **1**, the message can be modified. For each character (line 16) in each message block (line 15), if the character is a lowercase letter, convert it to an uppercase letter line 18). A pointer to the converted message is returned (line 21).

```
 1  mblk_t *lctouc(mp)
 2          mblk_t *mp;
 3  {
 4          mblk_t *cmp;
 5          mblk_t *tmp;
 6          unsigned char *cp;
 7
 8          if (mp->b_datap->db_ref > 1) {
 9                  if ((cmp = copymsg(mp)) == NULL)
10                          return (NULL);
11                  freemsg(mp);
12          } else {
13                  cmp = mp;
14          }
15          for (tmp = cmp; tmp; tmp = tmp->b_next) {
16                  for (cp = tmp->b_rptr; cp < tmp->b_wptr; cp++) {
17                          if ((*cp <= 'z') && (*cp >= 'a'))
```

```
                        *cp -= 0x20;
                }
        }
        return(cmp);
}
```

SUMMARY OF TRUSTED SOLARIS CHANGES

This routine will assign correct attributes to the message it copies.

SEE ALSO

allocb(9F), **copyb**(9F), **msgb**(9S)

NAME

dupb – Duplicate a message-block descriptor

SYNOPSIS

#include <sys/stream.h>

mblk_t ∗**dupb(mblk_t** ∗*bp***);**

ARGUMENTS

bp	Pointer to the message block to be duplicated **mblk_t** is an instance of the **msgb**(9S) structure.

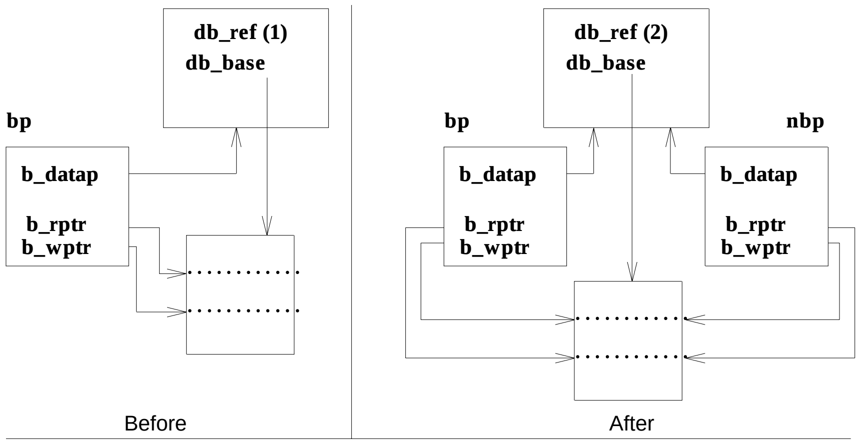
INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI)

DESCRIPTION

dupb( ) creates a new **mblk_t** structure to reference the message block to which *bp* points. Unlike **copyb**(9F), **dupb** does not copy the information in the data block but creates a new structure to point to the message block.

The following figure shows how the **db_ref** field of the **dblk_t** structure has been changed from **1** to **2**, reflecting the increase in the number of references to the data block. The new **mblk_t** contains the same information as the first. Note that **b_rptr** and **b_wptr** are copied from *bp*, and that **db_ref** is incremented.



nbp=dupb(bp);

RETURN VALUES

If successful, **dupb** returns a pointer to the new message block. Upon failure, **dupb** returns a **NULL** pointer.

CONTEXT

dupb can be called from user or interrupt context.

EXAMPLE

This **srv**(9E) (service) routine adds a header to all **M_DATA** messages before passing them along. The message block for the header was allocated elsewhere. For each message on the queue that is a priority message, pass it along immediately (lines 9–10). Otherwise, if the message is anything other than an **M_DATA** message (line 11) and if it can be sent along (line 12), then do so (line 13). Otherwise, put the message back on the queue and return (lines 15–16). For all **M_DATA** messages, first check to see if the stream is flow-controlled (line 19). If it is, put the message back on the queue and return (line 22); if it is not, the header block is duplicated (line 20). If **dupb** fails, the service routine is rescheduled in one-tenth of a second with **timeout** and then we return (lines 23–24). If **dupb** succeeds, link the **M_DATA** message to it (line 26) and pass it along (line 27). **dupb** can be used here instead of **copyb**(9F) because the contents of the header block are not changed.

Note that this example ignores issues related to cancelling outstanding timeouts at close time.

```
 1  xxxsrv(q)
 2      queue_t *q;
 3  {
 4          mblk_t *mp;
 5          mblk_t *bp;
 6          extern mblk_t *hdr;
 7
 8          while ((mp = getq(q)) != NULL) {
 9                  if (mp->b_datap->db_type >= QPCTL) {
10                          putnext(q, mp);
11                  } else if (mp->b_datap->db_type != M_DATA) {
12                          if (canputnext(q))
13                                  putnext(q, mp);
14                          else {
15                                  putbq(q, mp);
16                                  return;
17                          }
18                  } else {  /* M_DATA */
19                          if (canputnext(q)) {
20                                  bp = dupb(hdr);
21                                  if (bp == NULL) {
22                                          putbq(q, mp);
23                                          timeout(qenable, (long)q, drv_usectohz(100000));
24                                          return;
25                                  }
26                                  linkb(bp, mp);
27                                  putnext(q, bp);
28                          } else {
29                                  putbq(q, mp);
30                                  return;
31                          }
```

```
                }
        }
}
```

SUMMARY OF TRUSTED SOLARIS CHANGES

This routine will preserve the attributes of the message manupulated.

SEE ALSO

copyb(9F), **msgb**(9S)

NAME

dupmsg – Duplicate a message

SYNOPSIS

#include <sys/stream.h>

mblk_t ∗**dupmsg(mblk_t** ∗*mp***);**

INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI)

ARGUMENTS

mp	Pointer to the message

DESCRIPTION

dupmsg( ) forms a new message by copying the message-block descriptors to which *mp* points and linking them. **dupb**(9FTSOL) is called for each message block. The data blocks themselves are not duplicated.

RETURN VALUES

Upon success, **dupmsg** returns a pointer to the new message block. Upon failure, **dupmsg** returns a **NULL** pointer.

CONTEXT

dupmsg can be called from user or interrupt context.

EXAMPLE

See **copyb**(9FTSOL) for an example using **dupmsg**.

SUMMARY OF TRUSTED SOLARIS CHANGES

This routine will preserve the attributes of the message it manipulates.

SEE ALSO

copyb(9FTSOL), **copymsg**(9FTSOL), **dupb**(9FTSOL), **datab**(9S)

NAME

insq – Insert a message into a queue

SYNOPSIS

#include <sys/stream.h>

int insq(queue_t **q*, **mblk_t** **emp*, **mblk_t** **nmp***);**

ARGUMENTS

q	Pointer to the queue containing message *emp*
emp	Enqueued message before which the new message is to be inserted. **mblk_t** is an instance of the **msgb**(9S) structure.
nmp	Message to be inserted

INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI)

DESCRIPTION

insq( ) inserts a message into a queue. The message to be inserted, *nmp*, is placed in *q* immediately before the message *emp*. If *emp* is **NULL**, the new message is placed at the end of the queue. The queue class of the new message is ignored. All flow-control parameters are updated. The service procedure is enabled unless S.B "QNOENB" is set.

RETURN VALUES

Upon success, **insq** returns **1**. Upon failure, **insq** returns **0**.

CONTEXT

insq can be called from user or interrupt context.

EXAMPLE

This routine illustrates the steps a transport provider may take to place expedited data ahead of normal data on a queue. (Assume all **M_DATA** messages are converted into **M_PROTO T_DATA_REQ** messages.) Normal **T_DATA_REQ** messages are just placed on the end of the queue (line 16). However, expedited **T_EXDATA_REQ** messages are inserted before any normal messages already on the queue (line 25). If there are no normal messages on the queue, **bp** will be **NULL** and we fall out of the **for** loop (line 21). **insq** acts like **putq**(9F) in this case.

```
 1  #include <sys/tihdr.h>
 2  #include <sys/stream.h>
 3
 4  static int
 5  xxxwput(queue_t *q, mblk_t *mp)
 6  {
 7          union T_primitives *tp;
 8          mblk_t *bp;
 9          union T_primitives *ntp;
10
11          switch (mp->b_datap->db_type) {
12          case M_PROTO:
13                  tp = (union T_primitives *)mp->b_rptr;
14                  switch (tp->type) {
15                  case T_DATA_REQ:
```

```
                putq(q, mp);
                break;

            case T_EXDATA_REQ:
                freezestr(q);
                for (bp = q->q_first; bp; bp = bp->b_next) {
                  if (bp->b_datap->db_type == M_PROTO) {
                    ntp = (union T_primitives *)bp->b_rptr;
                    if (ntp->type != T_EXDATA_REQ)
                      break;
                 }
                }
                (void) insq(q, bp, mp);
                unfreezestr(q);
                break;
            . . .
      }
    }
}
```

SUMMARY OF TRUSTED SOLARIS CHANGES

This routine will try to assign attribute structures to the **mblk**s of a message that does not have one. The first attribute stucture found will be used. If a stream module illegally combined messages, **mblk**s can have different attribute structures; in that case, the message will be dropped by this routine unless overridden by the **TSOL_STR_LINKB** flag.

SEE ALSO

freezestr(9F), **msgb**(9S), **putq**(9F), **unfreezestr**(9F), **rmvq**(9F)

NAME

linkb – Concatenate two message blocks

SYNOPSIS

#include <sys/stream.h>

void linkb(mblk_t ∗*mp1*, **mblk_t** ∗*mp2***);**

ARGUMENTS

mp1 The message to which *mp2* is to be added. **mblk_t** is an instance of the **msgb**(9S) structure.

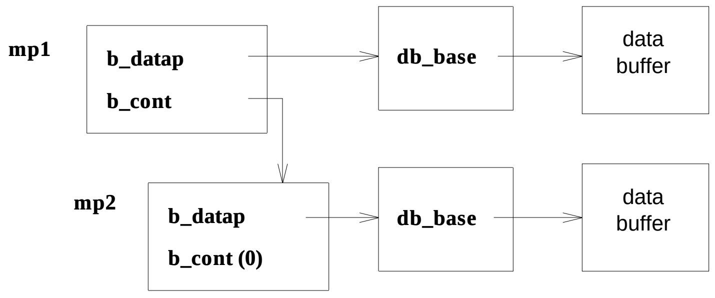
mp2 The message to be added

INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI)

DESCRIPTION

linkb( ) creates a new message by adding *mp2* to the tail of *mp1*. The continuation pointer, **b_cont**, of the first message is set to point to the second message:



linkb(mp1, mp2);

CONTEXT

linkb can be called from user or interrupt context.

EXAMPLE

See **dupb**(9F) for an example of using **linkb**.

SUMMARY OF TRUSTED SOLARIS CHANGES

This function can fail because of mismatched attributes attached to the messages. If the attributes of the two messages differ only in their information labels, the linked message will be assigned a label derived from conjoining the information labels from both messages. If any other attributes are different, the second message if discarded. Module writers should check the attributes before calling this function or use the **tsol_linkb**(9FTSOL)

SEE ALSO

unlinkb(9FTSOL)

tsol_linkb(9FTSOL)

NAME

msgpullup – Concatenate bytes in a message

SYNOPSIS

#include <sys/stream.h>

mblk_t ∗**msgpullup(mblk_t** ∗*mp*, **int** *len***);**

ARGUMENTS

mp	Pointer to the message whose blocks are to be concatenated
len	Number of bytes to concatenate

INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI)

DESCRIPTION

msgpullup( ) concatenates and aligns the first *len* data bytes of the message to which *mp* points, copying the data into a new message. Any remaining bytes in the remaining message blocks will be copied and linked onto the new message. The original message is unaltered. If *len* equals –**1**, all data is concatenated. If *len* bytes of the same message type cannot be found, **msgpullup** fails and returns **NULL**.

RETURN VALUES

Upon success, **msgpullup** returns a pointer to the new message . Upon failure, **msgpullup** returns **NULL**.

CONTEXT

msgpullup can be called from user or interrupt context.

SUMMARY OF TRUSTED SOLARIS CHANGES

This routine will try to assign attribute structures to the **mblk**s of amessage that does not have one. The first attribute stucture found will be used. If a stream module illegally combined messages, **mblk**s can have different attribute structures; in that case, the message will be dropped by this routine unless overridden by the **TSOL_STR_LINKB** flag.

SEE ALSO

srv(9E), **allocb**(9F), **msgb**(9S)

NAME

pullupmsg – Concatenate bytes in a message

SYNOPSIS

#include <sys/stream.h>

int pullupmsg(mblk_t ∗*mp*, **int** *len*);

ARGUMENTS

mp	Pointer to the message whose blocks are to be concatenated. **mblk_t** is an instance of the **msgb**(9S) structure.
len	Number of bytes to concatenate

INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI)

DESCRIPTION

pullupmsg ( ) tries to combine multiple data blocks into a single block. **pullupmsg** concatenates and aligns the first *len* data bytes of the message to which *mp* points. If *len* equals **-1**, all data is concatenated. If *len* bytes of the same message type cannot be found, **pullupmsg** fails and returns **0**.

RETURN VALUES

Upon success, **pullupmsg** returns **1**. Upon failure, **pullupmsg** returns **0**.

CONTEXT

pullupmsg can be called from user or interrupt context.

EXAMPLE

This is a driver write **srv**(9E) (service) routine for a device that does not support scatter / gather DMA. For all **M_DATA** messages, the data will be transferred to the device with DMA.

First, try to pull up the message into one message block with the **pullupmsg** function (line 12). If successful, the transfer can be accomplished in one DMA job. Otherwise, the transfer must be done one message block at a time (lines 19–22). After the data has been transferred to the device, free the message and continue processing messages on the queue.

```
 1 xxxwsrv(q)
 2     queue_t *q;
 3 {
 4         mblk_t *mp;
 5         mblk_t *tmp;
 6         caddr_t dma_addr;
 7         int dma_len;
 8
 9         while ((mp = getq(q)) != NULL) {
10                 switch (mp->b_datap->db_type) {
11                 case M_DATA:
12                         if (pullupmsg(mp, -1)) {
13                                 dma_addr = vtop(mp->b_rptr);
14                                 dma_len = mp->b_wptr - mp->b_rptr;
15                                 xxx_do_dma(dma_addr, dma_len);
```

```
                                freemsg(mp);
                                break;
                        }
                        for (tmp = mp; tmp; tmp = tmp->b_cont) {
                                dma_addr = vtop(tmp->b_rptr);
                                dma_len = tmp->b_wptr - tmp->b_rptr;
                                xxx_do_dma(dma_addr, dma_len);
                        }
                        freemsg(mp);
                        break;
        . . .
                }
        }
}
```

SUMMARY OF TRUSTED SOLARIS CHANGES

This routine will try to assign attribute structures to the **mblk**s of a message that does not have one. The first attribute stucture found will be used. If a stream module illegally combined messages, **mblk**s can have different attribute structures; in that case, the message will be dropped by this routine unless overridden by the **TSOL_STR_LINKB** flag.

SEE ALSO

srv(9E), **allocb**(9F), **msgpullup**(9F), **msgb**(9S)

NAME

put – Call a STREAMS put procedure

SYNOPSIS

#include <sys/stream.h>
#include <sys/ddi.h>

void put(queue_t **q*, **mblk_t** **mp***);**

ARGUMENTS

q	Pointer to a STREAMS queue
mp	Pointer to message block being passed into queue

INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI)

DESCRIPTION

put calls the put procedure [**put**(9E) entry point] for the STREAMS queue specified by *q*, passing it the message block to which *mp* refers. **put** is typically used by a driver or module to call its own put procedure.

CONTEXT

put can be called from a STREAMS module or driver put or service routine, or from an associated interrupt handler, timeout, bufcall, or esballoc call-back. In the latter cases, the calling code must guarantee the validity of the *q* argument.

Because *put* may cause re-entry of the module (as it is intended to do), mutexes or other locks should not be held across calls to it because of the risk of single-party deadlock.

NOTES

The caller cannot have the stream frozen [see **freezestr**(9F)] when calling this function.

DDI / DKI-conforming modules and drivers are no longer permitted to call put procedures directly but must call through the appropriate STREAMS utility function [such as **put**(9E), **putnext**(9F), **putctl**(9F), and **qreply**(9F)]. This function is provided as a DDI / DKI-conforming replacement for a direct call to a put procedure.

SUMMARY OF TRUSTED SOLARIS CHANGES

This routine will try to assign attribute structures to the **mblk**s of a message that does not have one. The first attribute stucture found will be used. If a stream module illegally combined messages, **mblk**s can have different attribute structures; in that case, the message will be dropped by this routine unless overridden by the **TSOL_STR_LINKB** flag.

SEE ALSO

put(9E), **putctl**(9F), **putctl1**(9F), **putnext**(9F), **putnextctl**(9F), **putnextctl1**(9F), **qreply**(9F)

NAME

putbq – Place a message at the head of a queue

SYNOPSIS

#include <sys/stream.h>

int putbq(queue_t **q*, **mblk_t** **bp*);

ARGUMENTS

q	Pointer to the queue
bp	Pointer to the message block

INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI)

DESCRIPTION

putbq( ) places a message at the beginning of the appropriate section of the message queue.There are always sections for high-priority and ordinary messages.If other priority bands are used, each will have its own section of the queue, in priority-band order, after high-priority messages and before ordinary messages. **putbq** can be used for ordinary, priority-band, and high-priority messages. However, unless precautions are taken, using **putbq** with a high-priority message is likely to lead to an infinite loop of putting the message back on the queue, being rescheduled, pulling it off, and putting it back on.

This function is usually called when **bcanput**(9F) or **canput**(9F) determines that the message cannot be passed to the next stream component. The flow-control parameters are updated to reflect the change in the queue's status. If **QNOENB** is not set, the service routine is enabled.

RETURN VALUES

Upon success, **putbq** returns **1**. Upon failure, **putbq** returns **0**.

CONTEXT

putbq can be called from user or interrupt context.

EXAMPLE

See the **bufcall**(9F) function page for an example of **putbq**.

SUMMARY OF TRUSTED SOLARIS CHANGES

This routine will try to assign attribute structures to the **mblk**s of amessage that does not have one. The first attribute stucture found will be used. If a stream module illegally combined messages, **mblk**s can have different attribute structures; in that case, the message will be dropped by this routine unless overridden by the **TSOL_STR_LINKB** flag.

SEE ALSO

bcanput(9F), **bufcall**(9F), **canput**(9F), **getq**(9F), **putq**(9F)

NAME

putctl – Send a control message to a queue

SYNOPSIS

#include <sys/stream.h>

int putctl(queue_t **q*, **int** *type***);**

ARGUMENTS

q	Queue to which the message is to be sent
type	Message type (must be control, not data type)

INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI)

DESCRIPTION

putctl( ) tests the *type* argument to make sure a data type has not been specified, and then attempts to allocate a message block. **putctl** fails if *type* is **M_DATA**, **M_PROTO**, or **M_PCPROTO**, or if a message block cannot be allocated. If successful, **putctl** calls the **put**(9E) routine of the queue to which *q* points with the newly allocated and initialized messages.

RETURN VALUES

Upon success, **putctl** returns **1**. If *type* is a data type or if a message block cannot be allocated, **putctl** returns **0**.

CONTEXT

putctl can be called from user or interrupt context.

EXAMPLE

The **send_ctl** routine is used to pass control messages downstream. **M_BREAK** messages are handled with **putctl**( ) (line 11). **putctl1**(9F) (line 16) is used for **M_DELAY** messages, so that *parm* can be used to specify the length of the delay. In either case, if a message block cannot be allocated, a variable recording the number of allocation failures is incremented (lines 12, 17). If an invalid message type is detected, **cmn_err**(9F) panics the system (line 21).

```
 1  void
 2  send_ctl(wrq, type, parm)
 3      queue_t *wrq;
 4      unchar type;
 5      unchar parm;
 6  {
 7          extern int num_alloc_fail;
 8
 9          switch (type) {
10          case M_BREAK:
11                  if (!putctl(wrq->q_next, M_BREAK))
12                          num_alloc_fail++;
13                  break;
14
15          case M_DELAY:
16                  if (!putctl1(wrq->q_next, M_DELAY, parm))
```

```
                num_alloc_fail++;
            break;

        default:
            cmn_err(CE_PANIC, "send_ctl: bad message type passed");
            break;
        }
}
```

SUMMARY OF TRUSTED SOLARIS CHANGES

This routine puts an unlabeled message on the stream. Use of this routine is strongly discouraged. Programmers should use **tsol_putctl**(9FTSOL) instead. If this routine is used on a network-type stream, the message will be dropped by the STREAMS head.

SEE ALSO

put(9E), **cmn_err**(9F), **datamsg**(9F), **putctl1**(9FTSOL), **putnextctl**(9FTSOL), **tsol_putctl**(9FTSOL)

NAME

putctl1 – Send a control message with a one-byte parameter to a queue

SYNOPSIS

#include <sys/stream.h>

int putctl1(queue_t **q*, **int** *type*, **int** *p*);

ARGUMENTS

q	Queue to which the message is to be sent
type	Type of message
p	One-byte parameter

INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI)

DESCRIPTION

putctl1( ), like **putctl**(9F), tests the *type* argument to make sure a data type has not been specified, and attempts to allocate a message block. The *p* parameter can be used, for example, to specify how long the delay will be when an **M_DELAY** message is being sent. **putctl1** fails if *type* is **M_DATA**, **M_PROTO**, or **M_PCPROTO**, or if a mesage block cannot be allocated. If successful, **putctl1** calls the **put**(9E) routine of the queue to which points with the newly allocated and initialized message.

RETURN VALUES

Upon success, **putctl1** returns **1**. **putctl1** returns **0** if *type* is a data type or if a message block cannot be allocated.

CONTEXT

putctl1 can be called from user or interrupt context.

EXAMPLE

See the **putctl**(9F) function page for an example of **putctl1**.

SUMMARY OF TRUSTED SOLARIS CHANGES

This routine puts an unlabeled message on the stream. The use of this routine is strongly discouraged. Programmers should use **tsol_putctl1**(9FTSOL) instead. If this routine is used on a network-type stream, the message will be dropped by the streams head.

SEE ALSO

put(9E), **allocb**(9F), **datamsg**(9F), **putctl**(9F), **putnextctl1**(9F), **tsol_putctl1**(9FTSOL)

NAME

putnext – Send a message to the next queue

SYNOPSIS

#include <sys/stream.h>
#include <sys/ddi.h>

int putnext(queue_t **q*, **mblk_t** **mp***);**

ARGUMENTS

q	Pointer to the queue from which the message *mp* will be sent
mp	Message to be passed

INTERFACE LEVEL

Architecture-independent level 1 (DDI/DKI)

DESCRIPTION

putnext( ) is used to pass a message to the **put**(9E) routine of the next queue in the stream.

RETURN VALUES

None

CONTEXT

putnext can be called from user or interrupt context.

EXAMPLE

See **allocb**(9F) for an example of using **putnext**.

SUMMARY OF TRUSTED SOLARIS CHANGES

This routine will try to assign attribute structures to the **mblk**s of amessage that does not have one. The first attribute stucture found will be used. If a stream module illegally combined messages, **mblk**s can have different attribute structures; in that case, the message will be dropped by this routine unless overridden by the **TSOL_STR_LINKB** flag.

SEE ALSO

allocb(9F), **put**(9E)

NAME

putnextctl – seNd a control message to a queue

SYNOPSIS

#include <sys/stream.h>

int putnextctl(queue_t **q*, **int** *type*);

ARGUMENTS

q	Queue to which the message is to be sent
type	Message type (must be control, not data type)

INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI)

DESCRIPTION

putnextctl( ) tests the *type* argument to make sure a data type has not been specified, and then attempts to allocate a message block. **putnextctl** fails if *type* is **M_DATA**, **M_PROTO**, or **M_PCPROTO**, or if a message block cannot be allocated. If successful, **putnextctl** calls the **put**(9E) routine of the queue to which *q* points with the newly allocated and initialized messages.

A call to **putnextctl(***q***,***type)* is an atomic equivalent of **putctl(***q*->*q_next***,***type)***.** The STREAMS framework provides whatever mutual exclusion is necessary to ensure that dereferencing *q* through its **q_next** field and then invoking **putctl**(9F) proceeds without interference from other threads.

putnextctl should always be used in preference to **putctl**(9F)**.**

RETURN VALUES

Upon success, **putnextctl** returns **1**. If *type* is a data type or if a message block cannot be allocated, **putnextctl** returns **0**.

CONTEXT

putnextctl can be called from user or interrupt context.

EXAMPLE

The **send_ctl** routine is used to pass control messages downstream. messages are handled with **putnextctl** (line 8). **putnextctl1**(9F) (line 13) is used for **M_DELAY** messages, so that *parm* can be used to specify the length of the delay. In either case, if a message block cannot be allocated, a variable recording the number of allocation failures is incremented (lines 9, 14). If an invalid message type is detected, **cmn_err**(9F) panics the system (line 18).

```
 1  void
 2  send_ctl(queue_t *wrq, u_char type, u_char parm)
 3  {
 4          extern int num_alloc_fail;
 5
 6          switch (type) {
 7          case M_BREAK:
 8              if (!putnextctl(wrq, M_BREAK))
 9                  num_alloc_fail++;
10              break;
11
```

```
        case M_DELAY:
                if (!putnextctl1(wrq, M_DELAY, parm))
                        num_alloc_fail++;
                break;

        default:
                cmn_err(CE_PANIC, "send_ctl: bad message type passed");
                break;
        }
}
```

SUMMARY OF TRUSTED SOLARIS CHANGES

This routine will put an unlabeled message on the stream. Use of this routine is strongly discouraged. Programmers should use **tsol_putnextctl**(9FTSOL) instead. If this routine is used on a network-type stream, the message will be dropped by the STREAMS head.

SEE ALSO

put(9E), **cmn_err**(9F), **datamsg**(9F), **putctl**(9FTSOL), **putnextctl1**(9FTSOL), **tsol_putnextctl**(9FTSOL)

NAME

putnextctl1 – Send a control message with a one-byte parameter to a queue

SYNOPSIS

#include <sys/stream.h>

int putnextctl1(queue_t **q*, **int** *type*, **int** *p*);

ARGUMENTS

q	Queue to which the message is to be sent
type	Type of message
p	One-byte parameter

INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI)

DESCRIPTION

putnextctl1( ), like **putctl1**(9F), tests the *type* argument to make sure a data type has not been specified, and attempts to allocate a message block. The *p* parameter can be used, for example, to specify how long the delay will be when an **M_DELAY** message is being sent. **putnextctl1** fails if *type* is **M_DATA**, **M_PROTO**, or **M_PCPROTO**, or if a message block cannot be allocated. If successful, **putnextctl1** calls the **put**(9E) routine of the queue to which *q* points with the newly allocated and initialized message.

A call to **putnextctl1(***q***,***type,***p)** is an atomic equivalent of **putctl1(***q*->*q_next***,***type,***p)**. The STREAMS framework provides whatever mutual exclusion is necessary to ensure that dereferencing *q* through its **q_next** field and then invoking **putctl1**(9F) proceeds without interference from other threads.

putnextctl1 should always be used in preference to **putctl1**(9F)**.**

RETURN VALUES

Upon success, **putnextctl1** returns **1**. If *type* is a data type or if a message block cannot be allocated, **putnextctl1** returns **0**

CONTEXT

putnextctl1 can be called from user or interrupt context.

EXAMPLE

See the **putnextctl**(9F) function page for an example of **putnextctl1**.

SUMMARY OF TRUSTED SOLARIS CHANGES

This routine will put an unlabeled message on the stream. Use of this routine is strongly discouraged. Programmers should use **tsol_putnextctl1**(9FTSOL) instead. If this routine is used on a network-type stream, the message will be dropped by the STREAMS head.

SEE ALSO

put(9E), **allocb**(9F), **datamsg**(9F), **putctl1**(9FTSOL), **putnextctl**(9FTSOL), **tsol_putnextctl1**(9FTSOL)

NAME

putq – Put a message on a queue

SYNOPSIS

#include <sys/stream.h>

int putq(queue_t *∗q*, **mblk_t** *∗bp***);**

INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI)

ARGUMENTS

q	Pointer to the queue to which the message is to be added
bp	Message to be put on the queue

DESCRIPTION

putq( ) is used to put messages on a driver's queue after the module's put routine has finished processing the message. The message is placed after any other messages of the same priority, and flow control parameters are updated. If **QNOENB** is not set, the service routine is enabled. If no other processing is done, **putq** can be used as the module's put routine.

RETURN VALUES

Upon success, **putq** returns **1**. Upon failure, **putq** returns **0**.

CONTEXT

putq can be called from user or interrupt context.

EXAMPLE

See the **datamsg**(9F) function page for an example of **putq**.

SUMMARY OF TRUSTED SOLARIS CHANGES

This routine tries to assign attribute structures to the **mblk**s of a message that does not have any. The first attribute stucture found is used. If a stream module illegally combined messages, **mblk**s can have different attribute structures; in that case, the message might be dropped by this routine unless overridden by the **tsol_str_linkb** flag.

SEE ALSO

datamsg(9F), **putbq**(9FTSOL), **qenable**(9F), **rmvq**(9F)

NAME

tsol_get_strattr, tsol_set_strattr – Get security attributes of a message

SYNOPSIS

#include <sys/stream.h>
#include <sys/tsol/tsteam.h>

tsol_str_attr *tsol_get_strattr(mblk_t **mp* **);**
void tsol_set_strattr(mblk_t **mp*, **tsol_strattr_t** **strattr* **);**

INTERFACE LEVEL

Architecture-independent, Trusted Solaris only

ARGUMENTS

mp	Pointer to message block
strattr	Pointer to a STREAMS attributes structure

DESCRIPTION

tsol_get_strattr is called by a STREAMS routine to find the attributes attached to a STREAMS message. **tsol_get_strattr** returns a pointer to the first attibute structure found. The attribute-structure reference count will be increased to prevent the structure's being released. A module using this routine must call the **SATTR_RELE**( ) macro to free the attribute structure. If no attribute structure is found in a message, **NULL** is returned.

tsol_set_strattr replaces the current set of STREAMS attributes in the message to which *mp* points with new attributes supplied by *strattr*. If either *mp* or *strattr* is set to **NULL**, the function exits without changing *mp*.

CONTEXT

This routine can be called from the interrupt level. The routine will not sleep. The reference count is protected by a short-term spin-lock mutex.

NOTES

These interfaces are uncommitted; although not expected to change between minor releases of Trusted Solaris systems, these interfaces may change.

NAME

tsol_linkb – Concatenate two message blocks

SYNOPSIS

#include <sys/stream.h>

int tsol_linkb(mblk_t *∗mp1***, mblk_t** *∗mp2***);**

ARGUMENTS

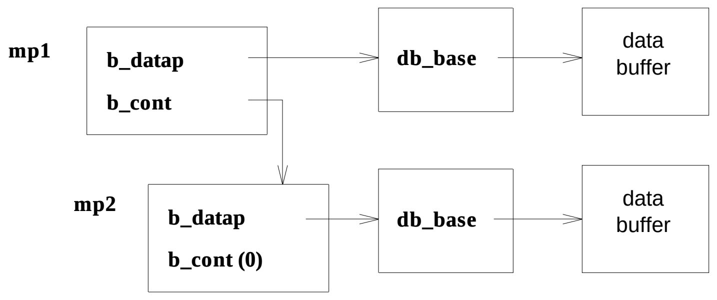
mp1	The message to which *mp2* is to be added. **mblk_t** is an instance of the **msgb**(9S) structure.
mp2	The message to be added

INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI), Trusted Solaris only

DESCRIPTION

tsol_linkb( ) is a replacement for **linkb**(9FTSOL). **tsol_linkb** will return an error if the linking of two messages fails because of mismatched attributes; other wise, **tsol_linkb** performs like **linkb**(9FTSOL). **tsol_linkb** creates a new message by adding *mp2* to the tail of *mp1*. The continuation pointer, **b_cont**, of the first message is set to point to the second message:



tsol_linkb(mp1, mp2);

CONTEXT

tsol_linkb can be called from user or interrupt context.

SEE ALSO

unlinkb(9FTSOL)

NAME

tsol_putctl – Send a control message to a queue

SYNOPSIS

#include <sys/stream.h>

int putctl(queue_t **q*, **int** *type*, **str_attr_t** **attrs*

INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI)

ARGUMENTS

q	Queue to which the message is to be sent
type	Message type (must be control, not data type)
attrs	Pointer to the security attributes for the control message

DESCRIPTION

tsol_putctl( ) is a replacement for **putctl**(9F). **tsol_putctl** allows a set of attributes to be sent with the message via *attrs*. Otherwise, **tsol_putctl** performs the same as **putctl**.

tsol_putctl should always be used in preference to **putctl**(9F)**.**

RETURN VALUES

Upon success, **tsol_putctl** returns **1.** If *type* is a data type or if a message block cannot be allocated, **tsol_putctl** returns **0**.

CONTEXT

tsol_putctl can be called from user or interrupt context.

SEE ALSO

put(9E), **cmn_err**(9F), **datamsg**(9F), **putctl1**(9F), **putnextctl**(9F) **tsol_putctl1**(9F)

NAME

tsol_putctl1 – Send a control message with a one-byte parameter and a set of attributes to a queue

SYNOPSIS

#include <sys/stream.h>

int tsol_putctl1(queue_t **q***, int** *type***, int** *p***, str_attr_t** **attrs***);**

ARGUMENTS

q	Queue to which the message is to be sent
type	Type of message
p	One-byte parameter
attrs	Pointer to the security attributes for the control message

INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI), Trusted Solaris only

DESCRIPTION

tsol_putctl1( ) is a replacement for **putctl**(9F), **tsol_putctl1** allows a set of attributes to be sent with the message via *attrs*. Otherwise, **tsol_putctl1** performs the same as putctl1.

tsol_putctl1 should always be used in preference to **putctl1**(9F)**.**

RETURN VALUES

Upon success, **tsol_putctl1** returns **1**. If *type* is a data type or if a message block cannot be allocated, **tsol_putctl1** returns **0**.

CONTEXT

tsol_putctl1 can be called from user or interrupt context.

SEE ALSO

put(9E), **allocb**(9F), **datamsg**(9F), **putctl**(9F), **putnextctl1**(9F)

NAME

tsol_putnextctl – Send a control message and a set of attributes to a queue

SYNOPSIS

#include <sys/stream.h>

int tsol_putnextctl(queue_t **q*, **int** *type*, **str_attr_t** **attrs*

INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI)

ARGUMENTS

q	Queue to which the message is to be sent
type	Message type (must be control, not data type)
attrs	Pointer to the security attributes for the control message

DESCRIPTION

tsol_putnextctl( ) is a replacement for **putnextctl**(9F). **tsol_putnextctl** allows a set of attributes to be sent with the control message via *attrs*. Otherwise, **tsol_putnextctl** performs the same as **putnextctl**.

tsol_putnextctl should always be used in preference to **putnextctl**.

RETURN VALUES

Upon success, **tsol_putnextctl** returns **1**. If *type* is a data type or if a message block cannot be allocated, **tsol_putnextctl** returns **0**.

CONTEXT

tsol_putnextctl can be called from user or interrupt context.

SEE ALSO

cmn_err(9F), **datamsg**(9F), **putctl**(9F), **putnextctl1**(9F) **tsol_putnextctl1**(9F)

NAME

tsol_putnextctl1 – Send a control message with a one-byte parameter and a set of attributes to a queue

SYNOPSIS

#include <sys/stream.h>

int tsol_putnextctl1(queue_t **q*, **int** *type*, **int** *p*, **str_attr_t** **attrs*);

INTERFACE LEVEL

Architecture-independent level 1 (DDI / DKI), Trusted Solaris only

ARGUMENTS

q	Queue to which the message is to be sent
type	Type of message
p	One-byte parameter
attrs	Pointer to the security attributes for the control message

DESCRIPTION

tsol_putnextctl1( ) is a replacement for **putnextctl1**(9F), **tsol_putnextctl1** allows a set of attributes to be sent with the control message via *attrs*. Otherwise it performs the same as putnextctl1().

tsol_putnextctl1 should always be used in preference to **putctl1**(9F)**.**

RETURN VALUES

Upon success, **tsol_putnextctl1** returns **1**. If *type* is a data type or if a message block cannot be allocated, **tsol_putnextctl1** returns **0**.

CONTEXT

tsol_putnextctl1 can be called from user or interrupt context.

SEE ALSO

put(9E), **allocb**(9F), **datamsg**(9F), **putctl1**(9F), **tsol_putnextctl**(9F)

Index