System V Interface Definition, Fourth Edition
Volume 3

Novell, Inc.
122 East 1700 South
Provo, UT 84606
U.S.A.

IMPORTANT NOTE TO USERS

While every effort has been made to ensure the accuracy of all information in this document, Novell assumes no liability to any party for any loss of damage caused by errors or omissions or by statements of any kind in the *System V Interface Definition*, its updates, supplements, or special editions, whether such errors are omissions or statements resulting from negligence, accident, or any other cause. Novell further assumes no liability arising out of the application or use of any product or system described herein; nor any liability for incidental or consequential damages arising from the use of this document. Novell disclaims all warranties regarding the information contained herein, whether expressed, implied or statutory, including implied warranties of merchantability or fitness for a particular purpose.

Novell makes no representation that the interconnection of products in the manner described herein will not infringe on existing or future patent rights, nor do the descriptions contained herein imply the granting or license to make, use or sell equipment constructed in accordance with this description.

Novell reserves the right to make changes without further notice to any products herein to improve reliability, function, or design.

TRADEMARKS

Ann Arbor is a trademark of Ann Arbor Terminals, Inc.
Beehive is a trademark of Beehive International.
Concept is a trademark of Human Designed Systems, Inc.
HP is a trademark of Hewlett–Packard Co.
LSI is a trademark of Lear Siegler, Inc.
Micro–Term, ACT and MIME are trademarks of Micro–Term, Inc.
OSF/Motif is a trademark of the Open Software Foundation
PostScript is a trademark of Adobe Systems.
Tektronix and Tektronix 4010 are registered trademarks of Tektronix, Inc.
TeleVideo is a registered trademark of TeleVideo Systems, Inc.
Teleray is a trademark of Research, Inc.
Teletype is a registered trademark of AT&T.
The X Window System is a trademark of MIT.
UNIX is a registered trademark in the USA and other countries, licensed exclusively through X/Open Company, Ltd.
VT100 is a trademark of Digital Equipment Corporation.
X/Open is a trademark of X/Open Company Limited.

Volume 3 Table of Contents

FINAL COPY
June 15, 1995
File: MasterToc
svid

Page: 8

Auditing Introduction

The Auditing Extension provides a mechanism capable of recording and reporting on security-related operations that occur on the system. (See the Enhanced Security Extension Introduction for information concerning the relationship between security classifications and the auditing commands and libraries.) The type of operations to be audited are specified as **audit events**. In most cases these events must be explicitly set by the administrator to be audited. A certain subset of events deemed critical to the integrity of the audit subsystem and other events deemed necessary to maintain the traceability of these events, are always audited whenever auditing is enabled. These events are called *fixed* events. Other events are auditable at the discretion of the system administrator; these are called **selectable** events.

The audit event type is intended as a way of identifying all audit records that describe the same type of event; that is, events that differ only in the parameters supplied to an operation.

The audit event class is intended as a way of grouping audit event types in a way that is useful to an administrator.

The administrator sets selectable events to be audited with the **auditset** command. The set of selectable events so chosen will begin being audited the next time auditing is enabled. The administrator enables or disables auditing with the commands **auditon** and **auditoff**. When auditing is enabled with **auditon**, fixed events plus any selectable events chosen with **auditset** are audited. **auditset** may also be used after auditing is enabled to specify additional events to be audited or to de-select events that no longer require auditing. Once auditing is disabled by **auditoff**, the setting of selectable events for individual users is cleared; to re-select, the administrator must invoke **auditset** again just before (or after) re-enabling audit with **auditon**.

User-specific audit masks may be designated for each user by using the **useradd** or **usermod** commands. These masks are permanent - whenever auditing is enabled and the user is logged on, events specified in these masks will be audited. To temporarily audit additional events for a user, the **auditset** command can be used to select the desired additional events. The events selected with **auditset** apply only to the user's current login session; if the specified user is not logged on, these events will not be recorded.

Audit records are maintained in an audit log file that is accessible only by the administrator.

Each auditable event, when audited, generates an associated audit record; collected for each event audited are a time stamp, the user identity, object name, level of the process (subject) causing the event, privileges used, an identification of the type of event, and an indication of the success or failure of the event. The **auditrpt** command is used to print data from the log file.

Summary of Library Routines

The following routines are supported by the Auditing Extension. All of the routines in this section have been internationalized and may reference environment variables for localization information. [See envvar(BA_ENV)].

auditbuf	**auditctl**	**auditdmp**	**auditevt**	**auditlog**

SUMMARY OF COMMANDS AND UTILITIES

The following commands and utilities are supported by the Auditing Extension. All of the commands and utilities in this section have been internationalized and may reference environment variables for localization information. [See envvar(BA_ENV)].

auditcnv	**auditlog**	**auditoff**	**auditrpt**	**auditset**
auditfltr	**auditmap**	**auditon**		

Organization of Technical Information

The ‘‘Auditing Library Routines’’ chapter provides manual page descriptions of library routines supported by this extension.

The ‘‘Auditing Commands and Utilities’’ chapter provides manual page descriptions of commands and utilities supported by this extension.

Auditing Extension Library Routines

The following section contains the manual pages for AT_LIB routines.

NAME

auditbuf – manipulate the audit buffer

SYNOPSIS

#include <sys/audit.h>

int auditbuf(int *cmd***, struct abuf ****bufp***, int** *size***)**

DESCRIPTION

The **auditbuf** system call is used to get or set the high_water_mark (vhigh) and size (bsize) of the audit buffer(s). The high_water_mark limits the amount of memory that can be held within the audit buffer. Use of the **auditbuf** system call requires appropriate privileges.

The default **high_water_mark** is equal to the size of an audit buffer (**ADT_BSIZE**). The valid range of values for *vhigh* is greater than or equal to zero and less than or equal to **ADT_BSIZE**. The size of the audit buffer (**ADT_BSIZE**) is a tunable parameter found in **/etc/master.d/audit** and can not be modified by the **auditbuf** system call.

Two values for *cmd* are supported: **ABUFGET** and **ABUFSET**. When the specified *cmd* is **ABUFGET**, the value of the **high_water_mark** is returned in *vhigh*, and the size of the audit buffer is returned in *bsize*.

When the specified *cmd* is **ABUFSET**, the value of the **high_water_mark** is changed to *vhigh*, and the *bsize* of the audit buffer is ignored.

The *bufp* argument points to a structure of type **abuf** which contains the following elements:

int vhigh;	audit buffer high_water_mark
int bsize;	audit buffer size

The **size** argument is used to verify the size of the **abuf** structure being passed in order to determine the version of auditing.

RETURN VALUE

Upon successful completion, the system call **auditbuf** returns a value of **0**; otherwise, a value of **–1** is returned and **errno** is set to indicate an error.

ERRORS

Under the following conditions, **auditbuf** fails and sets **errno** to:

EPERM if the process does not have the appropriate privileges.

EINVAL if the size of **abuf** is not equal to **size**.

EINVAL if cmd is **ABUFSET** and the value of *vhigh* is less than zero or greater than *bsize*.

EINVAL if the *cmd* is invalid.

ENOPKG if the audit package is not installed.

SEE ALSO

auditevt(AT_LIB).

LEVEL

Level 1.

NAME

auditctl – control or report the status of auditing

SYNOPSIS

```
#include <sys/audit.h>

int auditctl(int cmd, struct actl *actlp, int size)
```

DESCRIPTION

The **auditctl** system call fills the appropriate audit control structures or reports the status of auditing, depending on the values of *cmd*. Three values of *cmd* are supported: **AUDITON**, **AUDITOFF**, and **ASTATUS**.

When the specified *cmd* is **AUDITON**, the **auditctl** system call performs the following actions:

- Copies in the offset in seconds from the Greenwich mean time.
- It initializes the **vnode** for the primary audit log file.
- It initializes the audit buffer and log control structures.
- It exempts system resident processes and /sbin/init from auditing.
- It writes a machine-specific header record.
- It sets the **auditon** flag to 1.

When the specified *cmd* is **AUDITOFF**, the **auditctl** system call sets the **auditon** field to zero; frees all process audit structures; and locks, flushes, and releases the audit buffers.

When the specified *cmd* is **ASTATUS**, the **auditctl** system call returns the current status of auditing. A zero value for **auditon** in the **actl** structure indicates that auditing is disabled, and a value of one indicates that auditing is enabled.

The *actlp* argument points to a structure of type **actl** which contains the following elements:

int auditon;	audit status variable
char version[ADT_VERLEN];	audit version
long gmtsecoff;	GMT offset in seconds

The **size** argument is used to verify the size of the **actl** structure being passed in order to determine the version of auditing.

Auditing must be installed on the system for this system call to be used. The use of the **auditctl** system call requires the appropriate privilege.

RETURN VALUE

The **auditctl** system call returns zero on success. When unsuccessful, **auditctl** returns a value of –1 and sets **errno** to indicate the error.

ERRORS

Under the following conditions, **auditctl** fails and set **errno** to:

EINVAL The size of **actl** is not equal to **size**.

EINVAL An attempt was made to disable auditing while it was already disabled.

EINVAL An attempt was made to enable auditing while it was already enabled.

EINVAL The *cmd* is invalid.

ENOENT It is not possible to access the primary event log path.

EPERM The invoking subject does not have the required privilege.

ENOPKG The audit package is not installed.

EEXIST All the possible log files exist when attempting to enable auditing.

EROFS The primary audit log file resides within a file system that is mounted read-only.

EIO An I/O error occurred while performing a write to the audit log file.

SEE ALSO

auditbuf(AT_LIB), auditdmp(AT_LIB), auditevt(AT_LIB), auditlog(AT_LIB)

LEVEL

Level 1.

NAME

auditdmp – write audit record to audit buffer

SYNOPSIS

#include <sys/audit.h>

int auditdmp(struct arec ***_arecp_, int** _size_**)**

DESCRIPTION

The **auditdmp** system call is used to write an audit record to the audit buffer. Use of **auditdmp** system call requires the appropriate privileges. Upon successful completion, a record is written to the audit buffer. Trusted user-level commands with the appropriate privilege append user-level event records to the audit buffer. Privileged applications append only records of type **misc** to the audit buffer.

The _arecp_ argument points to a structure of type **arec** which contains the following elements:

int _rtype_;	audit record event type
int _rstatus_;	audit record event status
int _rsize_;	audit record size of argp
char *_argp_;	audit record data

The _rtype_ element of the **arec** structure specifies the event type of the audit record. If the _rtype_ argument is valid (one of the user-level events) and if its corresponding bit is set in the process **emask** for the invoking process, the system generates an audit record. The _rstatus_ element of the **arec** structure is the status of the user-level event, zero for success, non-zero for failure. The _rsize_ element of the **arec** structure specifies the size of memory required to record the data to be written. The _argp_ element of the **arec** structure is a character pointer to the audit data.

The _size_ argument is used to verify the size of the **arec** structure being passed in order to determine the version of auditing.

RETURN VALUE

The **auditdmp** system call returns zero on success. It will also return zero if the _rtype_ is not currently being audited and no record is written. When unsuccessful, **auditdmp** returns a value of –1 and sets **errno** to indicate the error.

ERRORS

Under the following conditions, **auditdmp** fails and sets **errno** to:

EINVAL	if the system call is invoked while auditing is disabled.
EINVAL	if the size of **arec** is not equal to **size**.
EINVAL	if **rtype** is invalid.
EPERM	if the process does not have the appropriate privileges.
ENOPKG	if the audit package is not installed.

SEE ALSO

auditbuf(AT_LIB), auditctl(AT_LIB), auditevt(AT_LIB), auditlog(AT_LIB)

LEVEL

Level 1.

NAME

auditevt – get or set auditable events

SYNOPSIS

#include <sys/types.h>
#include <sys/audit.h>

int auditevt(int *cmd*, **struct aevt** **aevtp*, **int** *size***)**

DESCRIPTION

The **auditevt** system call gets or sets auditable events, depending on the value of *cmd*. The following values of *cmd* are supported: **AGETSYS**, **ASETSYS**, **AGETUSR**, **AGETME**, **ASETME**, **†AGETLVL**, **†ACNTLVL**, **†ASETLVL**, **ASETUSR**, **AYAUDIT**, and **ANAUDIT**. The auditable event bit mask (*emask*) is represented by a hexadecimal number. The value of *uid* in the **aevt** structure is used to identify users to be audited on the system.

The *aevtp* argument points to a structure of type **aevt** which contains the following elements:

adtemask_t *emask*;	event mask to be set or retrieved
uid_t *uid*;	user's event mask to be set or retrieved
uint *flags*;	event mask flags
uint *nlvls*;	size of the individual object level table
level_t **lvl_minp*;	minimum object level range criteria
level_t **lvl_maxp*;	maximum object level range criteria
level_t **lvl_tblp*;	individual object level table

When the specified *cmd* is **AGETSYS**, the system-wide event mask (**adt_sysemask**) is copied to *emask* in the **aevt** structure, and the entire structure is returned. All elements of the **aevt** structure except *emask* are ignored.

When the specified *cmd* is **ASETSYS**, the value of *emask* in the **aevt** structure is OR'ed with the fixed auditable events and then copied into the system wide event mask. If auditing is enabled, then every process audit structure is updated to reflect the change. All elements in the **aevt** structure except *emask* are ignored.

When the specified *cmd* is **AGETUSR**, the active process list is searched for a process that belongs to the *uid* given in the **aevt** structure. If one is located, the value of the user's *emask* is copied into the **emask** field in the **aevt** structure, and the entire structure is returned. All elements of the structure except for *emask* and *uid* are ignored.

When the specified *cmd* is **AGETME**, the invoking process user's *emask* is retrieved and copied into the *emask* field in the **aevt** structure. All elements of the structure except for *emask* are ignored.

When the specified *cmd* is **ASETME**, the value of *emask* is copied into the user's event mask field of the user's process audit structure and then combined by a bitwise OR with the system wide event mask to create a new process event mask for the invoking process only. All elements of the structure except for *emask* are ignored.

When the specified *cmd* is **ASETUSR**, the active process list is searched for every process belonging to the given *uid*. When a valid active process is located, the value of *emask* is copied into the user's event mask field of the process audit structure and then combined by a bitwise OR with the system wide event mask to create a new process event mask. This processing continues until it finds and sets every valid active process belonging to the specified *uid*. All elements of the structure except for *emask* and *uid* are ignored.

When the specified *cmd* is **ANAUDIT**, the current process and any subsequent forked process is exempt from auditing. All elements of the structure are ignored.

When the specified *cmd* is **AYAUDIT**, the current process is made auditable again. All elements of the structure are ignored.

†When the specified *cmd* is **ACNTLVL**, the number of individual object levels is copied into the **nlvls** field and the entire **aevt** structure is returned. All elements of the structure except for **nlvls** are ignored. The Mandatory Access Control Module (MAC) must be installed for this value of *cmd* to be used.

†When the specified *cmd* is **AGETLVL**, the object level event mask is retrieved and copied into the **emask** field. The object level flags are copied into the **flags** field, and the number of individual object levels is copied into the **nlvls** field. If the object level range criteria was set (indicated by a flag setting of **ADT_RMASK**), then the *lvl_minp* and *lvl_maxp* fields are filled. If any individual object level criteria were set (indicated by a flag setting of **ADT_LMASK**), then the *lvl_tblp* field is filled. (Note that the amount of storage space for the *lvl_tblp* must be allocated by the invoking process. The amount of space is calculated by multiplying the value of **nlvls** by the size of a level_t. The value of **nlvls** is obtained from **ACNTLVL**.) The entire **aevt** structure is returned; only the *uid* element is ignored. The Mandatory Access Control Module must be installed for this value of *cmd* to be used.

†When the specified *cmd* is **ASETLVL**, the object level audit criteria is set. Object level auditing may be performed on either a single level or a range of levels, neither of which can be specified unless an object level event mask has been previously set or is currently being set. If the object level event mask flag is specified (indicated by a flag setting of **ADT_OMASK**), then the object level event mask is modified to reflect the value of the *emask* field. The Mandatory Access Control Module must be installed for this value of *cmd* to be used.

If auditing is to be performed on single levels, the value of *flags* is set to **ADT_LMASK**, and the levels specified by *lvl_tblp* will be set. To clear the individual levels, the *flags* value is set to **ADT_LMASK**, and list of null levels is specified by *lvl_tblp*.

If auditing is to be performed on a level range, the value of *flags* is set to **ADT_RMASK**, and the range of levels specified by *lvl_maxp* and *lvl_minp* will be set. In this case, *lvl_maxp* must dominate *lvl_minp*. To clear the level range, the value of *flags* is set to **ADT_RMASK**, and the values of *lvl_maxp* and **lvl_minp** are both null.

The **size** argument is used to verify the size of the **aevt** structure being passed in order to determine the version of auditing.

Auditing must be installed on the system for this system call to be used. Use of the **auditevt** system call requires the appropriate privilege.

RETURN VALUE

The **auditevt** system call returns zero on success. When unsuccessful, **auditevt** returns a value of –1 and sets **errno** to indicate the error.

ERRORS

Under the following conditions, **auditevt** fails and sets **errno** to:

EINVAL The size of **aevt** is not equal to **size**.

EINVAL Either **lvl_minp** or **lvl_maxp** points to an invalid level.

†EINVAL The *cmd* is **ASETLVL**, the *flags* field is **ADT_RMASK**, and **lvl_maxp** does not dominate **lvl_minp**.

†EINVAL The *cmd* is **ASETLVL**, the *flags* field is **ADT_RMASK**, and **lvl_maxp** and **lvl_minp** are not both NULL.

EINVAL The *cmd* is invalid.

†ENOPKG The *cmd* is **ACNTLVL**, **AGETLVL**, or **ASETLVL**, and the MAC feature is not installed.

EPERM The invoking subject does not have the required privilege.

ESRCH No process can be found corresponding to that specified by the uid.

SEE ALSO

auditbuf(AT_LIB), auditctl(AT_LIB), auditdmp(AT_LIB), auditlog(AT_LIB).

FUTURE DIRECTIONS

The ACNTLVL cmd value is designated Level 2 as of July 1992. A new command value will be added that will not require that a structure be passed in order to return the number of auditing levels.

The concept of Object Level Auditing will not be supported in the future. The NCSC's Orange Book makes no specific references to this for a B2 system. In association with removing the concept of "Object Level Auditing" from the SVID, the AGETLVL, and ASETLVL "cmd" values and related descriptions and error conditions are designated Level 2 for removal effective May 1993.

The ACNTLVL, AGETLVL, and ASETLVL cmd values and associated descriptions will be removed from the SVID when their three year waiting period has been completed.

LEVEL

Level 1.

ACNTLVL "cmd" value has been designated Level 2, effective July 1992.

AGETLVL, ASETLVL "cmd" values are designated Level 2, effective May 1993.

NAME

auditlog - get or set audit log file attributes

SYNOPSIS

```
#include <limits.h>
#include <sys/types.h>
#include <sys/audit.h>

int auditlog(int cmd, struct alog *alogp, int size)
```

DESCRIPTION

The **auditlog** system call is used to get or to set the audit log file attributes, depending on whether *cmd* is **ALOGGET** or **ALOGSET**. Use of the **auditlog** system call requires the appropriate privilege. The *alogp* argument points to a structure of type **alog** which contains the following elements:

int *flags*;	log file attributes
int *onfull*;	action on log file full
int *onerr*;	action on log file error
int *maxsize*;	maximum log file size
int *seqnum*;	log file sequence number 001-999
char *mmp*[ADT_DATESZ];	current month time stamp
char *ddp*[ADT_DATESZ];	current day time stamp
char *pnodep*[ADT_NODESZ];	optional primary log file node name
char *anodep*[ADT_NODESZ];	optional alternate log file node name
char **ppathp*;	optional primary log file pathname
char **apathp*;	optional alternate primary log file pathname
char **progp*;	optional program to run during log file switch
char **defpathp*;	default primary log file pathname
char **defnodep*;	default primary log file node name
char **defpgmp*;	default program to run during log file switch
int *defonfull*;	default action on log file full

The following values for *flags* in the **alog** structure are supported and may be modified or retrieved:

PPATH	set primary audit log file pathname (ppathp)
PNODE	set primary audit log file node name (pnodep)
APATH	set alternate audit log file pathname (apathp)
ANODE	set alternate audit log file node name (anodep)
PSIZE	set maximum size for primary audit log file
PSPECIAL	set primary audit log file to a character special device

ASPECIAL	set alternate audit log file to a character special device

The following values for *onfull* in the **alog** structure are supported and may be modified or retrieved: **ASHUT**, **AALOG**, **APROG**, and **ADISA**. (**APROG** is valid only if **AALOG** is also specified.) The following values of *onerr* are supported and may be modified or retrieved: **ASHUT** and **ADISA**.

ASHUT	shutdown the system when audit log file is full or an audit log file error occurs
AALOG	switch to alternate audit log file when current log file is full
APROG	run optional binary or shell program when audit log file is full
ADISA	disable auditing subsystem when audit log file is full or an audit log file error occurs

In addition to the ones listed above, the following elements in the **alog** structure may also be modified or retrieved: *maxsize*, *pnodep*, *anodep*, *ppathp*, *apathp*, and *progp*.

The following elements in the **alog** structure may only be retrieved because they can only be set internally: *seqnum*, *mmp*, and *ddp*.

The following elements in the **alog** structure may only be set because the defaults are read from the /etc/default directory: *defpathp*, *defnodep*, *defpgmp*, and *defonfull*.

The value of *maxsize* in the **alog** structure must be greater than or equal to the size of the audit buffer, **ADT_BSIZE**. The absolute pathnames to the primary audit log file (*ppathp*) and to the alternate audit log file (*apathp*) must be valid and be either of type directory or character special file. The absolute pathname to the optional program to be run during log switch (*progp*) must be less than **PATH_MAX** - 15. A seven-character node name may be appended to both the primary audit log file (*pnodep*) and the alternate audit log file (*anodep*).

seqnum is the log sequence number that is to be retrieved. **seqnum** can range from 001–999. **mmp** is the character pointer to the current month time stamp that is to be retrieved. **ddp** is the character pointer to the current day time stamp that is to be retrieved.

When the specified value of *cmd* is **ALOGGET**, the return values of the call are all the elements of the **alog** structure. Note that the space required for the *ppathp*, *apathp*, and *progp* must be allocated by the user.

When the value of *cmd* is **ALOGSET**, the elements of the **alog** structure determine what actions are to be performed.

The *size* argument is used to verify the size of the **alog** structure being passed in order to determine the version of auditing.

RETURN VALUE

When invoked successfully, the **auditlog** system call returns zero and sets the appropriate audit log file attributes. When unsuccessful, **auditlog** returns a value of –1 and sets **errno** to indicate the error.

ERRORS

Under the following conditions, **auditlog** fails and sets **errno** to:

EACCES The *cmd* is **ALOGSET**, and *ppathp*, *apathp*, or *progp* cannot be accessed.

EAGAIN It is not possible to allocate memory for the *alogp*.

EAGAIN The *cmd* is **ALOGSET**, and it is not possible to allocate memory for various elements used to fill in the **alog** structure.

EINVAL The size of **alog** does not equal **size**.

EINVAL The value of *cmd* is invalid.

EINVAL The *cmd* is **ALOGSET**, and the value of *onfull* is not either **ASHUT**, **AALOG**, **ADISA**, or **AALOG**|**APROG**.

EINVAL The *cmd* is **ALOGSET**, and the value of *onerr* is not either **ASHUT** or **ADISA**.

EINVAL The *cmd* is **ALOGSET** and the value of *maxsize* is not equal to zero and less than the size of the audit buffer (**ADT_BSIZE**).

EINVAL The *cmd* is **ALOGSET**, and the *flags* field contains **PPATH** or **NODE** when auditing is enabled.

ENOENT The *cmd* is **ALOGSET** and the pathname to the primary log file, alternate log file, or program to be run during a log switch does not exist.

ENAMETOOLONG The *cmd* is **ALOGSET**, and the *ppathp*, *apathp*, or *defpathp* fields are longer than **PATH_MAX** - 15.

ENAMETOOLONG The *cmd* is **ALOGSET**, and *progp* or *defpgmp* are longer than **PATH_MAX**.

ENOTBLK The *cmd* is **ALOGSET**, the *flags* field contains **PSIZE**, and the *maxsize* value is not zero.

EPERM The invoking subject does not have the required privilege.

ENOPKG The audit package is not installed.

SEE ALSO

auditbuf(AT_LIB), **auditctl**(AT_LIB), **auditdmp**(AT_LIB), **auditevt**(AT_LIB)

LEVEL

Level 1.

Auditing Extension Commands And Utilities

The following section contains the manual pages for the AT_CMD routines.

NAME

auditcnv – create audit mask file

SYNOPSIS

auditcnv

DESCRIPTION

The **auditcnv** shell-level command allows an administrator with the appropriate privileges to create an audit mask file for the user login interface. The **/etc/passwd** and **/etc/default/useradd** files are used to assign an initial default audit mask for every user on the system. When the **auditcnv** command is invoked and completes successfully, the following message is displayed:

/etc/security/ia/audit created

FILES

/etc/passwd
/etc/default/useradd
/etc/security/ia/audit

USAGE

Administrator.

SEE ALSO

defadm(BU_CMD), useradd(AU_CMD), usermod(AU_CMD).

LEVEL

Level 1.

NAME

auditfltr – convert audit log file for inter-machine portability

SYNOPSIS

auditfltr [[**–iN**] [**–oX**]] | [**–iX –oN**]

DESCRIPTION

The **auditfltr** command is used to convert audit log files from native machine format into XDR (External Data Representation) format and vice versa. These conversions allow you to transport audit log files from one machine to another for processing with **auditrpt**. **auditfltr** does not need to be used in all instances. If the two machines are of the same architecture and are running the same version of auditing, the log files can simply be copied from the source machine to the destination machine. If the two machines are of different architecture, or if they are not running the same version of auditing, **auditfltr** must be used as part of the copying procedure.

The **auditfltr** command has the following options:

–i This option specifies the type of the input file, which is always standard input.

–o This option specifies the type of the output file, which is always standard output.

The type of format may be **N**, for native machine format, or **X**, for XDR format. If no options are specified it is assumed the input file is in native machine format and the output file is in XDR format.

The procedure for transferring an audit log file from one machine to another has basically three steps. First, the audit log is converted from native machine format to the portable XDR format, using a command like the following:

```
cat /var/audit/1125103 | auditfltr –iN –oX > \
/var/tmp/1125103.xfer
```

Second, the file is transferred to another machine. This can be done by transferring the file to magnetic media on one with **tcpio** and then restoring it with the same command on the other. Third, the file is converted back to machine format. To avoid confusion with the destination machine's own audit log files, a subdirectory import under /var/audit is created. The file can then be converted with a command like the following:

```
cat /var/tmp/1125103.xfer | auditfltr –iX –oN > \
/var/audit/import/1125103
```

The **auditfltr** command accepts only audit log files as input.

FILES

/var/audit/*MMDD###*

USAGE

Administrator.

SEE ALSO

auditmap(AT_CMD), auditrpt(AT_CMD)

LEVEL

Level 1

NAME

auditlog – display or set audit event log file attributes

SYNOPSIS

auditlog [**-P** *path*] [**-p** *node*] [**-v** *high_water*] [**-x** *max_size*]
[**-s** | **-d** | **-A** *next_path* [**-a** *next_node*] [**-n** *pgm*]
| **-a** *next_node* [**-n** *pgm*]]

DESCRIPTION

The **auditlog** shell-level command allows an administrator with the appropriate privileges to manipulate the path to the event log file, the value for the high water mark of the audit buffer, the maximum size of the event log file, and the action taken when event log file is full. It also allows administrators to display information on all these settings and to display the action taken after an event log file error. While auditing is enabled, execution of this command results in an audit record being written to the log file via the **auditdmp** system call.

Without any options or arguments, **auditlog** displays the following information. (The default options are displayed first.)

Current Status of Auditing:	**OFF| ON**
Current Event Log In Use:	**/var/audit/MMDD###| [***path***]***MMDD###***[***node***]**
Current Audit Buffer High Water Mark:	**[***ADT_BSIZE***] bytes| high_water bytes**
Current Maximum File Size Setting:	**none|** *max_size* **blocks**
Action To Be Taken Upon Full Event Log:	**auditing disabled |**
	system shutdown |
	log switch
Action To Be Taken Upon Error:	**auditing disabled| system shutdown**
Next Event Log To Be Used:	**none| [***next_path***]***MMDD###***[***next_node***]**
Program to Run When Event Log Is Full:	**none|** *pgm*

The system reverts to the default values when auditing is stopped and subsequently restarted.

The **auditlog** *command has the following options:*

-P *path* — The **-P** option specifies the absolute pathname to the primary event log. The *path* may be either a directory or character special file message is printed. If auditing is enabled, the **-P** option is invalid.

If the argument to **-P** is a directory, **auditon** will create a regular file in the directory *path*, based upon the current month and day, followed by a 3 digit sequence number (for example, 1225001 means December 25, first log). The maximum path allowed is (PATH_MAX - 15). If the path exceeds this value, an error message is printed.

The valid range of sequence numbers is 001 to 999 per day, and the default event log file to be used is the regular file **/var/audit/***MMDD###* which is maintained by the audit subsystem. If the file exists when the system attempts its initial write, the sequence number is incremented and another attempt is made.

If the argument to **-P** is a special file, **auditon** will use the device as the file name. The administrator is responsible for attaching a physical label to the media.

-p *node* The **-p** option provides the ability to append an additional seven characters to the system-generated event log file name. For example, the command

auditlog -p abcdefg

will create the audit log file **/var/audit/***MMDD*###**abcdefg**. If the *node* is larger than seven characters or if it contains a slash, an error message is displayed. If the event log is a character special device the **-p** option argument is ignored.

-v *high_water* The **-v** option specifies the **high_water_mark** of the audit buffer. This setting is a byte value that is initialized to the size of the audit buffer (**ADT_BSIZE**) and can be dynamically changed in order to vary the relative frequency at which records are written to disk. Since this value is checked before an audit record is written to the buffer, large records will bypass the buffer and be written directly to disk. The **high_water_mark** must be either **0** (zero) or a positive integer less than or equal to the size of the audit buffer (**ADT_BSIZE**). If the value is not valid, an error message is displayed.

A setting of **0** forces all records to be written directly to the log file. The **0** value is required for the audited data to be displayed immediately (via the **auditrpt -w** command).

-x *max_size* The **-x** option specifies the maximum file size setting in 512 byte blocks for all event logs that are regular files. If this option is used with event logs that are not regular files, a warning message is displayed, and the option is ignored.

max_size must be greater than or equal to the size of the audit buffer tunable parameter **ADT_BSIZE**. The maximum size of the current event log is immediately reset and remains in effect until **auditlog -x** is invoked again, a log switch occurs, auditing is disabled, or the system is rebooted. An event log is considered full either when a regular log file reaches *max_size* (if specified), when the file system that the log resides in runs out of space, or when a special file (for example, a tape) cannot hold any more data.

-s The **-s** option specifies that the system will be shutdown when the primary event log is full. If this action is chosen and the log becomes full, the system is brought down immediately.

-d The **-d** option specifies that auditing will be disabled when the primary event log becomes full.

-A *next_path* The **-A** option specifies the absolute pathname to the alternate event log that will be used when the current primary event log becomes full. When a log switch occurs, all previous settings are reset to the default values. If the *next_path* to either a directory or character special file does not exist, an error message is displayed.

If the argument to **-A** is a directory, **auditon** creates a regular file relative to *next_path*, based upon the current month and day, followed by a three digit sequence number (for example, 1231002). The maximum path allowed is (PATH_MAX - 15). If the path exceeds this value, an error message is printed.

The valid range of sequence numbers is 001 to 999, and the default event log file is the regular file **/var/audit/***MMDD*###. If the file exists when the system attempts its initial write, the sequence number is incremented and another attempt is made.

-a *next_node* The **-a** option provides the ability to append an additional seven characters to the system-generated alternate event log file name. For example, the command

auditlog -a abcdefg

will create the file **/var/audit/***MMDD*###**abcdefg** when a log switch occurs.

If the *next_node* is larger than seven characters or if it contains a slash, an error message is displayed.

If the alternate log is a character special device the **-a** option is ignored.

-n *pgm* The **-n** option specifies either a shell file or binary executable (*pgm*) that will be run when a log switch occurs. The **-n** option may be used with at least one of the **-a** or **-A** options.

FILES

/etc/default/audit
/etc/master.d/audit
/var/audit/*MMDD*###

USAGE

Administrator.

SEE ALSO

auditbuf(AT_LIB), auditctl(AT_LIB), auditdmp(AT_LIB), auditevt(AT_LIB), auditlog(AT_LIB), auditoff(AT_CMD), auditon(AT_CMD).

LEVEL

Level 1.

NAME

auditmap – create and write the audit map files

SYNOPSIS

auditmap [**-m** *dirname*]

DESCRIPTION

The **auditmap** shell-level command creates and writes the audit map data to a set of files. To execute this command, a user must have the appropriate privileges. This command is invoked by the **auditon** command whenever auditing is started. The administrator may also invoke this command directly.

The **auditrpt** command uses the audit map files to translate numeric data in the log (for example, user ids) into character strings (for example, login names). The default name of the directory containing the audit map files is **/var/audit/auditmap**. The audit map file contains the following information:

- file identification, consisting of audit software version, timezone information, privilege mechanism information, system name, machine node name, operating system release and version, and machine type.
- all **/etc/passwd** login names and their corresponding UIDs
- all **/etc/group** names and their GIDs
- all event type names and their corresponding event type numbers
- all event classes defined in **/etc/security/audit/classes** and their corresponding event types
- all privilege names and their corresponding numbers
- all system call names and their corresponding numbers

If the Enhanced Security Extension is implemented a copy of the Level Translation Database (LTDB) is created in addition to the auditmap file. The LTDB consists of the following four separate files: ltf.class, ltf.cat, ltf.alias, and lid.internal.

You can specify a name for the audit map directory with the **-m** option. The specified directory must exist to be writable. The audit map files are readable only by users with appropriate privileges. Access controls should be set appropriately for the directory that contains the map files.

FILES

/etc/security/audit/classes
/var/audit/auditmap/auditmap
/var/audit/auditmap/ltf.class
/var/audit/auditmap/ltf.cat
/var/audit/auditmap/ltf.alias
/var/audit/auditmap/lid.internal
/etc/security/mac/ltf.class
/etc/security/mac/ltf.cat
/etc/security/mac/ltf.alias
/etc/security/mac/lid.internal

USAGE

Administrator.

SEE ALSO

auditon(AT_CMD), auditrpt(AT_CMD).

LEVEL

Level 1.

NAME

auditoff – disable auditing

SYNOPSIS

auditoff

DESCRIPTION

The **auditoff** shell-level command allows an administrator with the appropriate privileges to disable auditing. When auditing is disabled by **auditoff**, auditable events currently in progress will not have a record logged because they will not complete while auditing is enabled. While auditing is enabled, execution of this command causes the **auditdmp** system call to write an audit record to the log file. If auditing is already disabled, and **auditoff** is executed, no record is written.

RETURN VALUE

Upon successful completion of **auditoff**, the following informational message is displayed:

Auditing disabled

If **auditoff** is invoked while auditing is already disabled, an error status is returned and the following informational message displayed:

Auditing already disabled

USAGE

Administrator.

SEE ALSO

auditctl(AT_LIB), auditdmp(AT_LIB), auditlog(AT_CMD).

LEVEL

Level 1.

NAME

auditon – enable auditing

SYNOPSIS

auditon

DESCRIPTION

The **auditon** shell-level command allows an administrator with the appropriate privileges to enable auditing.

When auditon is invoked, it initializes the following with default values from the /etc/default/audit file:

AUDIT_LOGERR	log full conditions. May have the values "DISABLE" or "SHUTDOWN".
AUDIT_LOGFULL	log error conditions. May have the values "DISABLE", "SHUTDOWN" or "SWITCH".
AUDIT_DEFPATH	log file path name.
AUDIT_NODE	log file node name.
AUDIT_PGM	program to be executed during log switch.

The auditlog command can be used to override the values specified in /etc/default/audit for AUDIT_LOGFULL, AUDIT_DEFPATH, AUDIT_NODE, and AUDIT_PGM. If access to the **/etc/default/audit** file is denied or if a variable name is either missing or invalid, a warning message is printed. The AUDIT_NODE and AUDIT_PGM parameters are not evaluated unless the value of AUDIT_LOGFULL is SWITCH.

When auditing is enabled, the fixed events and any selectable events set by previous execution of **auditset** command take effect. When the **auditon** command is invoked successfully, the following message is displayed:

```
Auditing enabled
```

If the event log path cannot be accessed **auditon** prints an error message.

While auditing is enabled, execution of **auditon** results in an audit record being written to the log file via the *auditdmp* system call. The **auditmap** command is also invoked to write information to the audit map files. Any event being audited that completes while auditing is enabled will generate an event log record.

The auditing criteria remain in effect until one of the following occurs:

- When the system is shutdown both the event criteria and the log file attributes are lost.
- When auditing is disabled the system, object-level, and user event criteria are maintained but the log file attributes return to their default settings.
- When a log switch occurs the system, object-level, and user event criteria are maintained but the log file attributes return to their default settings.
- When the **auditlog** or **auditset** command is invoked the appropriate criteria is changed.

Auditing remains enabled until the **auditoff** command is executed, or until the log full condition of DISABLE or SHUTDOWN occurs, or until a log error is encountered. If the system is shutdown, the **auditlog** and **auditset** commands may need to be executed to reset any specified auditing criteria before invoking the **auditon** command.

FILES

/etc/default/audit
/var/audit/*MMDD*###

USAGE

Administrator.

SEE ALSO

auditctl(AT_LIB), auditdmp(AT_LIB), auditevt(AT_LIB), auditlog(AT_LIB), auditlog(AT_CMD), auditmap(AT_CMD), auditoff(AT_CMD), auditset(AT_CMD), defadm(BU_CMD).

LEVEL

Level 1.

NAME

auditrpt – display recorded information from audit trail

SYNOPSIS

auditrpt [-o] [-i] [-b | -w]
[-e[!]*event***[,. . .]] [-u** *user***[,. . .]] [-f** *object_id***[,. . .]]**
[-t *object_type***[,. . .]] [-l** *level* **| -r** *levelmin-levelmax***]**
[-s *time***] [-h** *time***] [-a** *outcome***] [-m** *map***]**
[-p all | *priv***[,. . .]] [***log* **[. . .]]**

DESCRIPTION

The **auditrpt** shell level command allows the administrator with the appropriate privileges to selectively display the contents of audit log files. Note that if the log files are presented as standard input that only one log file may be presented at a time. If more than one log file is presented in this manner, **auditrpt** will fail when it encounters data from the second log file. Specify the file names on the command line if you wish to process multiple log files. The privileges required are **audit** and **setplevel**.

The following options are available:

-o Display the events that correspond to the union of the specified auditing criteria.

-i Take input audit records from standard input.

-b Display the events in reverse chronological order (backwards). This option cannot be used with the **-w** option.

-w Display the events as they are being written to the event log file. This option cannot be used with the **-b** option.

-e[!] *event*[,. . .] Display the selected event types or event classes. If **!** is specified, all the events except those listed are displayed. Event classes, which are aliases for groups of events, are defined in the **/etc/security/audit/classes** file.

-u *user*[,. . .] Display all the recorded events for the specified real and effective uids and/or login names.

-f *object_id*[,. . .] Display all the recorded events for the specified *object_ids*. The *object_id* must be the full pathname of a regular file, special file, directory, or a named pipe, or the ID of an IPC object or loadable module.

-t *object_type*[,. . .]
Display all the recorded events for the specified *object_types*. Valid arguments are **f** (regular file), **c** (character special file), **l** (links), **d** (directories), **p** (named pipes or unnamed pipes), **s** (semaphores), **h** (shared memory), and **m** (messages).

-l *level* Display all the recorded events involving objects at the specified level. Only one level may be specified. Level information is recorded only if the Mandatory Access Control (MAC) feature was installed on the system that generated the audit log. This option cannot be used with the **-r** option.

-r *levelmin–levelmax*
: Display all recorded events involving objects whose security level falls within the range defined by *levelmin* and *levelmax*. Only one level range may be specified, and the level specified by *levelmax* must dominate *levelmin*. Level information is recorded only if the MAC feature was installed on the system that generated the audit log. This option cannot be used with the **-l** option.

-s *time*
: Display all the events occurring at or after the specified *time*. The *time* should be specified in the format used by the **date** command. The following are valid values for times: for hours, 00 to 23; for minutes, 00 to 59; for days, 01 to 31; for months, 01 to 12; and for years, 00 to 99.

  When both **-s** and **-h** are specified without the **-o** option, the start time (**-s**) must be earlier than the end time (**-h**).

-h *time*
: Display all the events existing at or before the specified *time*. Format and valid values for *time* are the same as the **-s** option.

-a *outcome*
: Display all the recorded events for the specified *outcome*: **s** (success) or **f** (failure).

-m *map*
: Specify the path (absolute or relative) of the auditmap directory.

-p *all* | *priv*[,. . .]
: Display the recorded events that use the specified privilege(s). If the word **all** follows the **-p** option, display all recorded events that use any privilege.

log[. . .]
: Name (absolute or relative pathname) of the audit log(s) to use.

OUTPUT

The first part of the output of **auditrpt** consists of the command line entered by the administrator. For each log file, the output consists of two parts. First, **auditrpt** displays audit log file and system identification information to verify that the correct log file was specified. This includes the internal identification of the audit log file, the version of the audit software that produced the log file, and the identification of the machine that produced the log file. Second, all records that meet the selection criteria are displayed one record per line. Records are displayed in the following format:

time,event,pid,outcome,user,group(s),session,subj_lvl, ****
(obj_id:obj_type:obj_lvl:device:maj:min:inode:fsid)(. . .)[,pgm_prm]

The meanings of the fields are as follows:

time
: The time is printed as hour:minute:second:day:month:year. For example, **10:30:00:15:04:91** is 10:30am of April 15, 1991.

event
: The event type.

pid
: The process ID number of the process that triggered the event, preceded by the letter **P**.

outcome
: The outcome of the event is either **s** for success or **f(***exit value***)** for failure.

user Real and effective user names are displayed. User names are separated by a colon (that is, *real_user_name*:*effective_user_name*).

group(s) Real and effective groups are displayed, followed by a list of supplementary groups, if any. Groups are separated by a colon (that is, *real_grp*:*effective_grp*:*suppl_grp1*:*suppl_grp2*: . . .).

session_id The session ID number, preceded by the letter **S**.

subj_lvl Subject level information is recorded only if the MAC feature was installed on the system that generated the audit log file. This field will be blank otherwise.

(obj_id:obj_type:obj_lvl:device:maj:min:inode:fsid)
This field contains file identification information, enclosed in parentheses. If multiple objects are accessed in a single event, the field is repeated. This field contains the following subfields:

obj_id The the name of a regular file, special file, directory, named pipe, or the id of an IPC object. If the full pathname of a file system object cannot be determined, the partial pathname will be printed with an asterisk (*) as a prefix.

obj_type The object type, using the codes described in the description of the **-t** option.

obj_lvl Object level information is recorded only if the MAC feature was installed on the system that generated the audit log file. This field will be blank otherwise.

device The object's device number.

maj The major number component of the object's device.

min The minor number component of the object's device.

inode The object's **inode** number.

fsid The object's file system ID number.

pgm_prm This field is specific to each audit event and may be composed of several subfields. The subfields described for each event will be displayed in the order shown below and will be separated by commas, unless otherwise specified.

The *pgm_prm* field can be one of the following:

For the **audit_ctl/audit_evt/audit_log/audit_map** events when generated by the audit user level commands **auditon**, **auditoff**, **auditset**, **auditlog**, **auditmap**, respectively: the entire command line.

For the **add_grp/add_usr/add_usr_grp/mod_grp/mod_usr** events: the entire command line.

For the **tfadmin** event: the entire command line.

For the **chg_times/date** events: the new date. For the **chg_times** event only, the file name is also given.

For the **fork** event: the child process ID.

For the **init** event: if generated by the user level command **init(1M)**, the entire command line. If generated by the **init** process (''process 1''):

current state: *state1* **new_state:** *state2*

The old init state is represented by *state1*, and the new init state by *state2*.

For the **kill** event: the signal and a list of pids to which the signal was posted.

For the **set_uid** event: new user.

For the **set_gid** event: the new group.

For the **set_pgrps** event: the name of the system call that generated the event (**setpgrp** or **setpgid**). In addition, if generated by the **setpgid** system call, the process ID and process group ID passed to the system call.

For the **set_grps** event: the supplementary group access list.

For the **link** event: the pathname of the target file.

For the **dac_own_grp** event: if the owner was changed, the new user ID (preceded by user:) or if the group was changed, the new group ID (preceded by group:). In addition, for the **chown** system call, the file name.

For the **dac_mode** event: the new mode.

For the **msg_ctl/msg_get/msg_op/sem_ctl/sem_get/sem_op/shm_ctl/shm_get/shm_op** events: the operation code, flag and command value. If a subfield does not pertain to an event type, a zero will be displayed.

For the **login/bad_auth** events, the terminal identification (tty), user, and group, of the user attempting to log on (if valid). In addition, for the **bad_auth** event: the error message (**LOGIN**, **PASWD** or **AUDIT**)

For the **passwd** event: the user whose password is being changed (if valid).

For the **pm_denied** event: the requested privilege, system call name, and maximum set of privileges.

For the **cron** event: user's effective uid, user's effective gid, user's level (enclosed in double quotes), and cron job name. User refers to the user that cron is running on behalf of. Note that the level subfield will be blank if the Enhanced Security Utilities were not installed and running on the audited system.

For the **open_rd/open_wr** events: the file descriptor.

For the **file_lvl** event: new security level (enclosed in double quotes). In addition, for the **flvlfile** system call, the file name.

For the **disp_attr/set_attr** events: the release flag (**persistent**, **last-close**, or **system**), device mode (**static** or **dynamic**), low_level (enclosed in double quotes), high_level (enclosed in double quotes) and device state (**private** or **public**). In addition, for the **disp_attr** event: the inuse flag (**inuse** or **unused**). For the **fdevstat** system call, the file descriptor.

For the **file_acl** event: all ACL entries and the file name.

For the **ipc_acl** event: the ipc type, the ipc id and all ACL entries.

For the **ulimit** event: the new limit.

For the **setrlimit** event: the resource (**RLIMIT_CORE**, **RLIMIT_CPU**, **RLIMIT_DATA**, **RLIMIT_FSIZE**, **RLIMIT_NOFILE**, **RLIMIT_STACK**, **RLIMIT_VMEM**), soft limit and hard limit.

For the **sched_lk** event: the action (**PROCLOCK**, **TXTLOCK**, **DATLOCK**) if generated by the **plock** system call. The page mapping attributes (**PRIVATE**, or **SHARED**) and page protection attributes (one or more of the following: **PROT_READ**, **PROT_WRITE**, **PROT_EXEC**) if generated by the **memctl** system call.

For the **sched_fp/sched_ts/sched_fc** events: If generated by the **priocntl** system call with the **PC_ADMIN** command, the function name (**FP_SETDPTBL**, **FC_SETDPTBL**, **RT_SETDPTBL** or **TS_SETDPTBL**), global priority and time quantum. In addition, if **TS_SETDPTBL** or **FC_SETDPTBL**, the time-sharing dispatcher parameters: tqexp, slpret, maxwait and lwait. If generated by the **priocntl** system call with the **PC_SETPARMS** command, the function name (**RT_NEW**, (**FP_NEW**, **FC_NEW**, **TS_NEW**, **RT_PARMSET**, **FP_PARMSET**, **FC_PARMSET**, **TS_PARMSET**), process id and user priority. In addition, if the **sched_ts** or **sched_fc** event, user priority limit. If **sched_fp** event, the seconds in time quantum.

For the **modadm** event: the module type (**character device**, **block device**, **streams**, **filesystem**, **misc**, **none**), the command (**register**), and the module name. Also, module type specific data as follows: if module type is **character device** or **block device**, the major number; if module type is **filesystem**, the file system name; if module type is **misc** or **none**, no specific data is displayed.

For the **modload** event: the loadable module id.

For the **modpath** event: the absolute pathname added to the loadable module search path or NULL if the default search path is set.

For the **iocntl** event: the command argument id passed to the system call, the flags found in the file table entry, if any (separated by colons), (**FOPEN**, **FREAD**, **FWRITE**, **FNDELAY**, **FAPPEND**, **FSYNC**, **FNONBLOC**, **FMASK**, **FCREAT**, **FTRUNC**, **FEXCL**, **FNOCTTY**, **FASYNC**, **FNMFS**).

For the **fcntl** event: the command argument passed to the system call. If command is **F_SETFD**, close-on-exec flag (0 or 1). If command is **F_SETFL**, status flags (separated by colons) (**O_APPEND**, **O_NONBLOCK**, **O_SYNC**). If a **struct flock** was passed to the system call: the command argument passed to the system call, (**F_ALLCOSP**, **F_FREESP**, **F_SETLCK**, **F_SETLKW**, **F_RSETLCK**, **F_RSETLKW**) and the following structure members: l_type, l_whence, l_start, l_len.

For the **mount** event: the flags passed to the system call and one or more of the following: **RDONLY** (read-only), **FSS** (old (4-argument) mount), **DATA** (6-argument mount), **NOSUID** (setuid disallowed), **REMOUNT** (remount), **NOTRUNC** (return **ENAMETOOLONG** for long file names).

For the **file_priv** event: all information in the **priv_t** masks passed to the system call, in the following format:

priv_type1:*priv_name*[:*priv_name*],*priv_type2*:. . .

priv_type will be the name of the privilege type, if it is recognized by the privilege mechanism of the audited system. If it is not recognized, it will be the character representation of the first byte of the **priv_t** mask (for example, **i** for inheritable).

For the **recvfd** event: the receiver's process ID, and the LWP ID.

For the **misc** event: the free form string provided by the application.

For the **audit_buf** event: the high water mark value.

For the **audit_ctl** event when generated by the **auditctl** system call: the action taken (**AUDITON or AUDITOFF**).

For the **audit_log** event when generated by the **auditlog** system call: all information passed in the **alog** structure to the system call. This will include: log file attributes (**PPATH:PNODE:APATH:ANODE:PSIZE :ASPECIAL:PSPECIAL**), the action taken when the log is full (**ASHUT,ADISA,AALOG, AALOG:APROG**), the action taken when there is an audit error (**ASHUT** or **ADISA**), the maximum log size, the primary node name, the alternate node name, the primary log pathname, the alternate log pathname and the program to be run during a log switch.

For the **audit_dmp** event when generated by the **auditdmp** system call: the event type and the status (if success: **SUCCESS**, if failure: **FAILURE**(status)).

For the **audit_evt** event when generated by the **auditevt** system call: all information passed in the **aevt** structure to the system call. This will include: command argument (**ASETME,ASETSYS,ASETUSR, ANAUDIT,AYAUDIT**). If the command is **ASETME**, the new user event mask for the invoking process. If the command is **ASETSYS**, the new system event mask. If the command is **ASETUSR**, the user whose mask has been modified, the new user event mask.

For most events generated from file descriptor based system calls, file information is returned in the file identification information field.

All the commas in the output line, except possibly the last one (if *pgm_prm* is empty), will be displayed as place holders. For all the output fields, null will be displayed if the field is not appropriate for the event type being displayed. For example, the *date* event has no objects related to it, so the *obj_id:obj_type:obj_lvl:device:maj:min:inode:fsid* fields will be null (only the comma separator will be displayed for these fields). Also, in a base system the MAC level fields will be null.

The **auditrpt** command will use the audit map to translate users, groups,security levels, privileges, events and system calls from IDs(numbers) to names. If the information for translating a number to a name is not found in the map, raw data (ASCII representation of the numeric value) will be displayed for the corresponding field.

All numeric values are displayed in decimal representation unless preceded by **0x**, which indicates hexadecimal representation.

If a field is appropriate for an event but its value is "invalid," a **?** will be displayed. For example, if a *login* event fails because the logname used is unknown to the system (cannot be translated into a UID in the log record), the *user* will be flagged as "invalid" and a **?** will be displayed.

Files

/var/audit/*MMDD###*
/var/audit/auditmap/auditmap
/var/audit/auditmap/lid.internal
/var/audit/auditmap/ltf.alias
/var/audit/auditmap/ltf.cat
/var/audit/auditmap/ltf.class

SEE ALSO

auditfltr(AT_CMD), **auditlog**(AT_CMD), **auditmap**(AT_CMD), **auditoff**(AT_CMD), **auditon**(AT_CMD), **auditset**(AT_CMD)

LEVEL

Level 1.

NAME

auditset – select or display auditing criteria

SYNOPSIS

auditset [–d [–u *user***[,...]** | **–a]** [**-m]]**
auditset [–s [*operator***]***event***[,...]]**
†**[[–u** *user***[,...]** | **–a] –e [***operator***]***event***[,...]]**
†**[–o [***operator***]***event***[,...]] [–l [+|–]** *level***]**
†**[–r [–]***levelmin–levelmax*]

DESCRIPTION

The **auditset** shell-level command allows an administrator with the appropriate privileges to set or display the dynamic auditing criteria. The **-m**, **-o**, **-l**, and **-r** options are valid only if the Mandatory Access Control (MAC) subsystem is installed. If it is not installed and these options are used, an error message is printed. While auditing is enabled, execution of this command will result in an audit record being written to the log file by the **auditdmp** system call.

When invoked without options, **auditset** displays the current system, user, and object-level audit criteria. Each category is separated by a blank line, in the following format:

System Audit Criteria:
system: all | *events***[,...]**

User Audit Criteria:
*user***: all** | **none** | *events***[,...]**

†**Object Level Audit Criteria: all** | **none** | *events***[,...]**
levelmin – levelmax
level

The **-s**, **-e**, **-o** options take an event or a list of events as arguments to the option. The event input list must be separated by commas and can be the name of an event class or event type. The event classes are defined in the modifiable system file **/etc/security/audit/classes**. One of three operators can precede the event or list of events. The operators define the action taken with the event list. (Only one operator may be used for a list of events; the operator affects every event on the list.) The following are the valid operators:

[*no operator*] Replace the current auditable event(s), level, or level range with the specified input.

+ Add the specified auditable event(s) or level to the current audit criteria.

– Delete the specified auditable event(s), level, or level range from the current audit criteria.

! All events except those specified replace the current auditable events.

Additionally the words `all` or `none` may be used as event keywords. They may not be used in conjunction with any other events or keywords.

The **auditset** command takes the following options:

-d Display the current audit criteria. If no other options are supplied, the system audit criteria are displayed. When combined with either the **-a** or **-u** options, **-d** displays audit criteria for either all active users or the specified active users, respectively. The login name is displayed (instead of the UID), and the events are listed in alphabetical order. When combined with the **-m** option, **-d** displays the audited object events in alphabetical order, followed by a list of levels and/or level range to which the criteria apply.

–m When combined with **–d**, this option causes **auditset** to display the system audit criteria and the object-level audit criteria.

-u *user***[,...]** Set (when combined with the **-e** option) the auditing criteria for any number of *users* that are currently logged on or display (when combined with the **-d** option), the system audit criteria and the auditing criteria for any number of *users* that are currently logged on. The *user* can be identified by either login name or UID number. If more than one *user* is specified, each login name or UID in the input list should be separated by commas. This option cannot be combined with the **-a** option. If any of the given users are invalid or not active, a warning message is printed.

-a Set (when combined with the **-e** option) the auditing criteria for any number of *users* that are currently logged on or display (when combined with the **-d** option), the system audit criteria and the auditing criteria for all users that are currently logged on. This option cannot be combined with the **-u** option.

-s [*operator*]*event***[,...]**
Set the system-wide auditing criteria. Any valid event type or event class will be recorded, regardless of user or object levels involved.

-e [*operator*]*event***[,...]**
This option must be combined with either the **-u** *user* or the **-a** options to set user audit criteria.

-o [*operator*]*event***[,...]**
Set object-level auditing criteria. The event types specified (types or classes) are those to be audited for the *levels* currently in effect.

-l [+|–] [*level*] Set object-level audit criteria for the specified level. When combined with **–o**, it sets object-level audit criteria for the specified level. All events specified by the **–o** option are audited if they involve objects at the specified level. Only one level may be specified; to audit more than one level, repeat the **–l** option. A valid level is one which is defined to the system (see **lvlname(ES_CMD)**). The maximum number of individual levels which may be audited is system tunable.

If a minus sign precedes the level, delete the level from the levels used for object-level auditing. If a plus sign precedes the level, add the level to the levels used for object-level auditing.

-r [–]*levelmin–levelmax*
Set object-level audit criteria for all levels in the level range enclosed by *levelmin* and *levelmax*. The level range is inclusive therefore levelmin and levelmax are audited. The maximum level (*levelmax*) must dominate the minimum level (*levelmin*). If a minus sign (**–**) precedes the level range, delete audit criteria for the specified level range.

The **auditset** command sets audit criteria for users dynamically. When you set audit criteria for a user with the **-u** option, the criteria are in effect only for that login session. If the user logs out or logs in from another terminal, the criteria are no longer in effect. If you wish to set audit criteria for all of a user's login sessions, use either the **useradd** or **usermod** commands.

FILES

/etc/security/audit/classes

USAGE

Administrator.

SEE ALSO

auditctl(AT_LIB), auditdmp(AT_LIB), auditevt(AT_LIB), auditlog(AT_CMD), auditoff(AT_CMD), auditon(AT_CMD), defadm(BU_CMD), useradd(AU_CMD), usermod(AU_CMD).

FUTURE DIRECTIONS

The concept of Object Level Auditing will not be supported in the future. The NCSC's Orange Book makes no specific references to this for a B2 system.

Associated with this, the -m, -o, -r, -l options and the Object Level Audit Criteria have been moved to Level 2. They will be removed from the SVID when the three year waiting period has been completed.

LEVEL

Level 1. The -m, -o, -r, -l options are Level 2, effective May 1993.

Enhanced Security Introduction

The Enhanced Security Extension provides advanced interfaces to support a secure system. This need has been reflected in the recent publication of several security guidelines designed to specify a secure operating system. The need to protect files and directories from unauthorized user access, via the Enhanced Security Extension features, enhances the security of the system by preventing both unauthorized disclosure and unauthorized change.

Security Criteria

In 1983 the Department of Defense (DoD) published the definitive guideline to secure operating systems, the Trusted Computer System Evaluation Criteria (TCSEC). The TCSEC defined the criteria a system must meet to be certified as meeting multilevel security standards. The TCSEC defines four security divisions, D, C, B, and A, with multiple levels in all but the D division. From least to most secure, the levels are D, C1, C2, B1, B2, B3, and A1. Each level's requirements build on those of the previous level.

In 1989 the German Federal Office for Security in Information Technology (BSI) published the ZSIEC, defining the German security criteria. The ZSIEC is based on the TCSEC with the main difference being the separation of functionality and assurance. In 1990 France, the Federal Republic of Germany, the Netherlands, and the United Kingdom combined their criteria into a single set of harmonized criteria, the Information Technology Security Evaluation Criteria (ITSEC). The ITSEC follows the model of the German ZSIEC in that it also separates functionality from assurance. Both the ZSIEC and ITSEC provide clear mappings to TCSEC, as follows:

TCSEC Level	**ITSEC Level**	**ZSIEC Level**
C1	F1/E1	F1/Q1
C2	F2/E2	F2/Q2
B1	F3/E3	F3/Q3
B2	F4/E4	F4/Q4
B3	F5/E5	F5/Q5
A1	F5/E5	F5/Q5

If the base SVID (without extensions) were evaluated, it would likely be classified as C1, not fully meeting the requirements of any higher level, although it would fulfill selected criteria at the C2 level. (Note, the SVID definition has not been evaluated and therefore, is considered unrated.)

Security Classes

The Enhanced Security Extension may be configured for various classes of security. These classes, as defined in Trusted Computer Systems Evaluation Criteria, are C2, B1, and B2. The following table lists, for each Enhanced Security feature area, the associated commands and libraries that must be included to attain a C2, B1, or B2 class system.

	Class C2		Class B1		Class B2	
Enhanced Security Feature	Commands	Libraries	Commands	Libraries	Commands	Libraries
Audit	**auditcnv, auditlog, auditmap, auditoff, auditon, auditrpt, auditfltr, auditset**	**auditbuf, auditdmp, auditctl, auditevt, auditlog**	**auditcnv, auditlog, auditmap, auditoff, auditon, auditrpt, auditfltr, auditset**	**auditbuf, auditdmp, auditctl, auditevt, auditlog**	**auditcnv, auditlog, auditmap, auditoff, auditon, auditrpt, auditfltr, auditset**	**auditbuf, auditdmp, auditctl, auditevt, auditlog**
Trusted Import Export			**tcpio**		**tcpio**	
Trusted Path			**defsak**		**defsak**	

	Class C2		Class B1		Class B2	
Enhanced Security Feature	Commands	Libraries	Commands	Libraries	Commands	Libraries
Mandatory Access Control			**admalloc, chlvl, devattr, devstat, getdev, lvlname, lvldelete, lvlprt, mldmode, putdev**	**devalloc, devdeal-loc, devstat, lvldom, lvlequal, lvlin, lvlipc, lvlvalid, lvlout, lvlvfs, lvlproc, mkmld, mldmode**	**admalloc, chlvl, devattr, devstat, get-dev, lvlname, lvldelete, lvlprt, mldmode, put-dev**	**devalloc, devdeal-loc, devstat, lvldom, lvlequal, lvlin, lvlipc, lvlvalid, lvlout, lvlvfs, lvlproc, mkmld, mldmode**
Discre-tionary Access Control					**setacl, getacl**	**acl, aclipc, aclsort**
Trusted Facility Manage-ment					**adminrole, adminuser, filepriv, tfadmin**	**filepriv, procpriv, procprivl**

Background

Prior to the Enhanced Security Extension any attempt to execute a sensitive system service (for example, mount a file system) required the use of a "privilege". In System V there has been traditionally one such privilege, commonly called "root" or "superuser" which is signified by a process whose effective user id is 0. In the Enhanced Security Extension this single superuser privilege is subdivided into a finer grain set of privileges designed to assure that sensitive system services execute with the minimum amount of privilege required to execute the task. This feature is currently restricted to use by an administrative (or "trusted") user.

Enhanced Security Extension

The Enhanced Security Extension consists of the following features:

- Enhanced DISCRETIONARY ACCESS CONTROL (DAC) - The DAC access mechanism provides a controlling method which enhances the existing file permission bits mechanism by use of ACCESS CONTROL LISTS (ACLs). Each ACL consists of a list of named individuals and named groups of individuals, with their respective access permissions. This enhanced mechanism allows the ability to grant or deny permission access to the granularity of a single user.

- MandATORY ACCESS CONTROL (MAC) - The MAC access mechanism provides a controlling method by the assignment of sensitivity levels. The assignment of a security sensitivity level to every process and file/IPC on the system is the basis of this feature. A level includes both a hierarchical classification (e.g., "secret") and non-hierarchical categories (e.g., "projxyz"). The levels are the basis for all mandatory access control decisions.

  Unlike DAC where sharing (i.e., granting permissions) is at the owner's discretion, MAC sharing is mediated by the administrator, and enforced by the system. The MAC policy can be summarized as "no read up" and "write equal". This implies that a process at a given level (e.g., secret) can not read information at a higher level (e.g., top secret) and any process at a given level can only write information at its own level. The access decision is computed as a dominance/equality relation.

  When an access is attempted, both MAC and DAC checks are performed. If both checks pass, access is granted (see "Access Algorithm Changes" section below for more details).

- IDENTIFICATION and AUTHENTICATION (I&A) - I&A is a mechanism to identify a user and authenticate their identity in order to gain access to the system. This facility includes the programs that perform the identification (login ID) and authentication (password verification) of users and the programs that manipulate the information used by the I&A programs.

- TRUSTED PATH (TP) - TP is a mechanism through which a terminal user can gain the attention of a trusted system process, requiring support for identification and authentication. TP provides a trusted communication path, exclusively initiated by a user, between the system and the user. This mechanism ensures that the password is being requested by login and not by a malicious program that masquerades as a system program to gain sensitive information.

The user gains the attention and access to the trusted system via a terminal using the Secure Attention Key (SAK). The user must enter the SAK, a character or asynchronous line condition, such as a break or line drop to invoke the trusted path. The SAK is defined by a system administrator as a site configurable option.

- TRUSTED FACILITY MANAGEMENT (TFM) - TFM is a mechanism to support a variety of trusted user classes, including auditors, administrators, and operators. Separate operator, administrator, and security operator functions must be defined and implemented to support administrative least privilege. During normal operations, these roles will replace the current single administrative role, superuser.

Effects of Enhanced Security

The addition of the Enhanced Security features result in changes that affect previous UNIX System behavior. When the system is running with Enhanced Security, every command and data file must have appropriate discretionary and mandatory access control information. Additionally, those programs that require privilege, must have process privilege information associated with them.

Access Algorithm Changes

Any access to files/IPC objects will be constrained by the addition of the enhanced security features. These new features and the effect they will have when an access request is made are described below.

- ACLs enable DAC to be a finer grained control mechanism. In addition to specifying permissions for the owner, the owning group, and all others, permissions may be specified for particular users and particular groups. Thus, programs that look at the permission bits to determine access permissions may not receive all of the relevant access permissions. When access is attempted through a system call, the kernel will mediate the access based on the ACL entries.
- With the introduction of MAC into the system, whenever an attempt is made to access an object, there will be additions to the checks currently made for the access check.
    - If the requested access is for reading or execution, then the MAC level of the process must dominate (meaning equal to or greater than) the MAC level of the object, or the process must have appropriate privilege.

– If the requested access is for writing , then the MAC level of the process must be equal to the MAC level of the object, or the process must have appropriate privilege.

Attempts to directly access a device file may no longer succeed. If the device file is not in the *public* state, then it must first be allocated before being used.

How Users Acquire Privileges

All users, including administrators, log in as unprivileged users; i.e. the initial user process has no privilege associated with it. Some users designated as administrators or operators that do need to execute commands that require one or more process privileges, do so through the **tfadmin** command (See TFADMIN COMMand section below for more details).

The addition of a least privilege mechanism separates the privileges formally bestowed upon the super-user (uid 0). Access formally granted to processes with process-ID 0 may now be denied access, since a process-ID of 0 will no longer possess privilege.

Programs that check to determine if they are executing with a uid of zero, assuming that they are privileged will not function properly. These programs should be changed to use the required set of discrete privileges for them to successfully complete the task.

Several distinct levels of authorization are created through the proper assignment of process privileges according to the least privilege principle and the separation of duties that is accomplished through the TFM database. The least privilege principle requires that each subject in a system be granted the most restrictive sets of privileges needed for the performance of authorized tasks. These mechanisms ensure that privileged processes run only with the privilege(s) required for the actions they are authorized to perform, and that unprivileged processes cannot perform privileged actions.

Least Privilege Mechanism

The **Least Privilege** concept defines that a process only acquires the minimum amount of privilege it requires to execute the operation and only holds that privilege for the duration of the operation. Additionally, the requested privilege must be associated with both the process and the executable file to be successfully enabled. The user may acquire privilege in one of two ways; (1) by invoking a process with **fixed** privilege(s) associated with it or (2) by acquiring the privilege(s) via the **tfadmin** command. The way the privileges are set varies between the two and is described briefly below.

Privilege Descriptors

In the Enhanced Security Extension, a process has a **maximum** and **working** set of privileges associated with it. The **maximum** set represents the most privilege the process could ever attain and the **working** set represents the minimum set of privileges required to execute the task. An executable file may have associated with it an **inheritable** or **fixed** set of privileges. An **inheritable** privilege is a privilege which is kept (i.e. left "turned on") only if it already existed in the process. A **fixed** privilege is a privilege which is always given to the process independent of the previous process privileges. The **fixed** and **inheritable** privileges are disjoint, a privilege cannot be present in both sets at the same time.

Fixed Privileges

The concept of **fixed** file privileges is similar to the current concept of **setuid** bits. When a file has a privilege or privileges set as **fixed** those privileges are unioned with the **maximum** privilege set of the invoking process forming the new processes maximum privilege set. In essence these privileges are added (or forced) onto the invoking process. For example if a site determined that all users should be able to execute the *ps* command and not be subject to mandatory or discretionary access control checks the administrator would set the access control override privileges as **fixed** privileges on the command. Any user invoking *ps* would then acquire these privileges for the duration of the execution of the *ps* command. This scheme does not lend itself well to administrative operations such as mounting a file system since there is no restriction on the acquisition of the privilege (aside from normal access checks).

TFADMIN COMMand

The *tfadmin* command and its associated database allow for fine grain control over the acquisition of privilege, typically for administrative operations.

The *tfadmin* database is organized by "roles" subdivided by "tasks". For example the role of "operator" may have the task of user backup associated with it. In this scenario the operator would login to the system and then invoke the *tfadmin* command in the role of "operator" to execute the "backup" task. The *tfadmin* command will add the privilege(s) which are associated with the invoking user for the requested task to the **maximum** privilege set, then exec the task. The enabling of privilege for the task is then determined by the intersection of the process's **maximum** privilege set (acquired via *tfadmin*) and the file's **inheritable** privilege set. The result of this operation is then unioned with the file's **fixed** privilege set resulting in the new processes privilege set.

Multilevel Directories

With Mandatory Access Control (MAC) installed unprivileged users may only create files at the same level as the level of the parent directory (and at their current login level). This creates problems for utilities which require access to "public" directories (i.e. /tmp). To provide the functionality of "public" directories within a Mandatory Access Control environment a new type of directory known as a multilevel directory has been introduced.

In normal use, a multilevel directory has the appearance of a directory whose contents are all at that user's level. To another user at a different level, that the same multilevel directory would appear to contain a different set of files. This is because each user sees an "effective" directory consisting only of objects at their own level. Other directories may be created as multilevel directories at administrative discretion. An unprivileged user cannot create a multilevel directory; that is a privileged operation.

Changes to Existing Commands

Several commands may behave differently when the Enhanced Security feature is supported in a system. These include:

at	**find**	**lp**	**mount**	**userdel**
batch	**fsck**	**lpstat**	**passwd**	**usermod**
cpio	**ipcs**	**ls**	**ps**	**volcopy**
cron	**listusers**	**mkdir**	**useradd**	**whodo**
crontab	**logins**	**mkfs**		

See pages for details of changes.

NEW COMMandS

New commands have been introduced for the feature areas described above. Also, a new enhanced (trusted) cpio command, **tcpio**, and a new command to check for mail at different levels, **mailcheck**, have been introduced.

Summary of LIBRARY ROUTINES

The following routines are supported by the Enhanced Security Extension. All of the routines in this section have been internationalized and may reference environment variables for localization information. [See envvar(BA_ENV)].

acl	**devdealloc**	**lvlequal**	**lvlout**	**mkmld**
aclipc	**devstat**	**lvlfile**	**lvlproc**	**mldmode**
aclsort	**filepriv**	**lvlin**	**lvlvalid**	**procpriv**
devalloc	**lvldom**	**lvlipc**	**lvlvfs**	**procprivl**

Summary of Commands and Utilities

The following commands and utilities are supported by the Enhanced Security Extension. All of the commands and utilities in this section have been internationalized and may reference environment variables for localization information. [See envvar(BA_ENV)].

admalloc	**defsak**	**getacl**	**lvlprt**	**setacl**
adminrole	**devattr**	**getdev**	**mailcheck**	**tcpio**
adminuser	**devstat**	**lvldelete**	**mldmode**	**tfadmin**
chlvl	**filepriv**	**lvlname**	**putdev**	

ORGANIZATION of TECHNICAL INFORMATION

The ''Enhanced Security Library Routines'' chapter provides manual page descriptions of library routines supported by this extension.

The ''Enhanced Security Commands and Utilities'' chapter provides manual page descriptions of commands and utilities supported by this extension.

Enhanced Security Extension Library Routines

The following section contains the manual pages for the ES_LIB routines.

NAME

acl – set a file's Access Control List (ACL)

SYNOPSIS

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

int acl (char *pathp, int cmd, int nentries, struct acl *aclbufp)
```

DESCRIPTION

The **acl** system call is used to manipulate ACLs on file system objects.

pathp points to a pathname naming a file.

nentries specifies how many ACL entries fit into buffer *aclbufp*.

aclbufp a pointer to the **acl** struct which contains the following fields:

```
int    a_type;       /* entry type */
uid_t  a_id;         /* user or group ID */
ushort a_perm;       /* entry permissions */
```

The values for **a_type** are:

USER_OBJ	Permissions for the owner of the object.
USER	Permissions for additional users.
GROUP_OBJ	Permissions for members of the owning group of the object.
GROUP	Permissions for members of additional groups.
CLASS_OBJ	Maximum permissions granted to the file group class.
OTHER_OBJ	Permissions for other users.
DEF_USER_OBJ	Default permissions for the object owner.
DEF_USER	Default permissions for additional users.
DEF_GROUP_OBJ	Default permissions for members of the owning group of the object.
DEF_GROUP	Default permissions for members of additional groups
DEF_CLASS_OBJ	Default maximum permissions granted to the file group class.
DEF_OTHER_OBJ	Default permissions for other users.

cmd The following three values for *cmd* are available:

ACL_SET *nentries* ACL entries, specified in buffer *aclbufp*, are stored in the file's ACL. Any existing ACL on the file is replaced by the new ACL. This value for *cmd* can only be executed by a process that has an effective user ID equal to the owner of the file, or by a process with the appropriate privileges. All directories in the pathname must be searchable. Mandatory write access to the file is required.

ACL_GET Buffer *aclbufp* is filled with the file's ACL entries. Discretionary read access to the file is not required, but all directories in the pathname must be searchable. Mandatory read access to the file is required.

ACL_CNT The number of entries in the file's ACL is returned. Discretionary read access to the file is not required, but all directories in the pathname must be searchable. Mandatory read access to the file is required. *nentries* and *aclbufp* are ignored.

For cmd **ACL_SET**, the **acl** call will succeed if all of the following are true:

- There is exactly one entry each of type USER_OBJ, GROUP_OBJ, CLASS_OBJ, and OTHER_OBJ.
- There is at most one entry each of type DEF_USER_OBJ, DEF_GROUP_OBJ, DEF_CLASS_OBJ, and DEF_OTHER_OBJ.
- Entries of type USER, GROUP, DEF_USER, or DEF_GROUP may not contain duplicate entries. A duplicate entry is one of the same type containing the same numeric ID.
- If an ACL contains no entries of type USER and no entries of type GROUP, then the entries of type GROUP-OBJ and CLASS_OBJ must have the same permissions.
- If an ACL contains no entries of type DEF_USER and no entries of type DEF_GROUP, and an entry of type DEF_GROUP_OBJ is specified, then an entry of type DEF_CLASS_OBJ must also be specified and the two entries must have the same permissions.

RETURN VALUE

Upon successful completion, if *cmd* is **ACL_SET**, a value of **0** is returned. If *cmd* is **ACL_GET** or **ACL_CNT**, the number of ACL entries is returned. Otherwise, a value of **-1** is returned and *errno* is set to indicate the error.

ERRORS

acl() will fail if one or more of the following is true:

EACCES if the caller does not have access to a component of the pathname.

EACCES if the caller does not have mandatory read access to the file for **ACL_GET** and **ACL_CNT**, or mandatory write access to the file for **ACL_SET**.

EINVAL if *cmd* is not **ACL_GET**, **ACL_SET**, or **ACL_CNT**.

EINVAL if *cmd* is **ACL_SET** and *nentries* is less than the number of mandatory ACL entries (4).

EINVAL if *cmd* is **ACL_SET** and the ACL specified in *aclbufp* is not valid [see **aclsort**(ES_LIB)].

EIO if a disk I/O error has occurred while storing or retrieving the ACL.

EPERM if *cmd* is **ACL_SET** and the effective user ID of the caller does not match the owner of the file, and the caller does not have the appropriate privileges to perform the operation.

ENOENT if a component of the path does not exist.

ENOSPC if *cmd* is **ACL_GET** and *nentries* is less than the number of entries in the file's ACL.

ENOSPC if *cmd* is **ACL_SET** and there is insufficient space to store the ACL.

ENOSPC if *cmd* is **ACL_SET** and *nentries* is greater than the tunable parameter *aclmax*.

ENOTDIR if a component of the path specified by *pathp* is not a directory.

ENOTDIR if *cmd* is **ACL_SET** and an attempt is made to set a default ACL on a file type other than a directory.

ENOSYS if **cmd** is **ACL_SET**, the file specified by *pathp* resides on a file system that does not support ACLs, and additional entries were specified in the ACL.

EROFS if *cmd* is **ACL_SET** and the file specified by *pathp* resides on a file system that is mounted read-only.

SEE ALSO

aclipc(ES_LIB), aclsort(ES_LIB), getacl(ES_CMD), setacl(ES_CMD).

LEVEL

Level 1.

NAME

aclipc – get or set an IPC object's ACL, return the number of ACL entries

SYNOPSIS

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

int aclipc(int type, int id, int cmd, int nentries, struct acl *aclbufp)
```

DESCRIPTION

aclipc will get or set an IPC object's ACL, or return the number of ACL entries. To get the ACL, the user must have read access to the object. To set an ACL, the user must be the owner or creator of the object or have appropriate privileges.

nentries specifies how many ACL entries fit into buffer *aclbufp*.

aclbufp a pointer to the **acl** struct which contains the following fields:

```
int    a_type;        /* entry type */
uid_t  a_id;          /* user or group ID */
ushort a_perm;        /* entry permissions */
```

The values for **a_type** are:

USER_OJB Permissions for the owner of the object.

USER Permissions for additional users.

GROUP_OBJ Permissions for members of the owning group of the object.

GROUP Permissions for members of additional groups.

CLASS_OBJ Maximum permissions granted to the file group class.

OTHER_OBJ Permissions for other users.

type must be one of the following:

IPC_SHM *id* must be a valid shared memory identifier returned by **shmget**.

IPC_SEM *id* must be a valid semaphore identifier returned by **semget**.

IPC_MSG *id* must be a valid message queue identifier returned by **msgget**.

cmd must be one of the following:

ACL_GET The ACL information for the IPC object specified by *type* and *id* is copied into the user supplied buffer *aclbufp*. *nentries* specifies the number of ACL entries which will fit into *aclbufp*. The user must have read access to the IPC object.

ACL_SET The ACL for the IPC object specified by *type* and *id* is set to the ACL entries in the user supplied buffer *aclbufp*. *nentries* specifies the number of ACL entries currently in *aclbufp*. The entries in *aclbufp* must be valid and in the proper ACL order. The user must have the appropriate privileges, or be the creator or owner of the object, to alter the IPC object.

ACL_CNT Returns the number of ACL entries for the IPC object specified by *type* and *id*. *nentries* and *aclbufp* are ignored. The user must have read access to the IPC object.

When the ACL for an IPC object is set, the permission mode (in *ipc_perm*) may change. The first three bits of the permission mode are set to the permissions of the object user entry. The middle three bits of the permission mode are set to the **OR**ed value of the permissions for the additional users, object group, and additional group entries. The last three bits of the permission mode are set to the permissions of the other entry.

For *cmd* **ACL_SET** the **aclipc** call will succeed if all of the following are true:

- There is exactly one entry each of type **USER_OBJ**, **GROUP_OBJ**, **CLASS_OBJ**, and **OTHER_OBJ**.
- Entries of type **USER** or **GROUP** may not contain duplicate entries. A duplicate entry is one of the same type containing the same numeric ID.
- If an ACL contains no entries of type **USER** and no entries of type **GROUP**, then the entries of type **GROUP_OBJ** and **CLASS_OBJ** must have the same permissions.

RETURN VALUE

Upon successful completion, the return value is the number of ACL entries for *cmd* **ACL_CNT** and **ACL_GET**, and **0** for *cmd* **ACL_SET**. Otherwise, a value of **-1** is returned and **errno** is set to indicate the error.

ERRORS

aclipc will fail if one or more of the following are true:

EINVAL if *type* is not one of **IPC_SHM**, **IPC_SEM**, or **IPC_MSG**.

EINVAL if *id* is not a valid *type* identifier.

EINVAL if *cmd* is not one of **ACL_GET**, **ACL_SET**, or **ACL_CNT**.

EINVAL if *cmd* is **ACL_SET** and the ACL entries in *aclbufp* are not valid or in proper order.

EPERM if *cmd* is **ACL_SET** and the user does not have the appropriate privileges and is neither the creator nor owner of the IPC object.

EINVAL if *cmd* is **ACL_SET** and the security level of the calling process is not equal to the security level of the IPC object.

EINVAL if *cmd* is **ACL_GET** or **ACL_CNT** and the security level of the calling process is dominated by the security level of the IPC object.

EACCES if *cmd* is **ACL_GET** or **ACL_CNT** and the user does not have discretionary read access to the IPC object.

ENOSPC if *cmd* is **ACL_SET** and there is not enough space to store the ACL.

ENOSPC if *cmd* is **ACL_GET** and the number of ACL entries for the IPC object exceeds *nentries*.

ENOSPC if *cmd* is **ACL_SET and** *nentries* is greater than the tunable parameter **aclmax**.

EINVAL if *cmd* is **ACL_SET** and the number of ACL entries is less than the number of mandatory ACL entries (4).

SEE ALSO

acl(ES_LIB), msgget(KE_OS), semget(KE_OS), shmget(KE_OS).

LEVEL

Level 1.

NAME

aclsort – sort an Access Control List

SYNOPSIS

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

int aclsort(int nentries, int calclass, struct acl *aclbufp);
```

DESCRIPTION

The **aclsort** routine sorts Access Control List (ACL) entries into the correct order to be accepted by the **acl** system call.

aclbufp points to a buffer containing ACL entries; **calclass**, if non-zero, indicates that the CLASS_OBJ permissions should be recalculated; and **nentries** specifies the number of ACL entries in the buffer.

aclsort sorts the contents of the ACL buffer as follows:

1) Entries will be in order USER_OBJ, USER, GROUP_OBJ, GROUP, CLASS_OBJ, OTHER_OBJ, DEF_USER_OBJ, DEF_USER, DEF_GROUP_OBJ, DEF_GROUP, DEF_CLASS_OBJ, and DEF_OTHER_OBJ.
2) Entries of type USER, GROUP, DEF_USER, and DEF_GROUP will be sorted in increasing order by ID.

The **aclsort** call will succeed if all of the following are true:

- There is exactly one entry each of type USER_OBJ, GROUP_OBJ, CLASS_OBJ, and OTHER_OBJ.
- There is at most one entry each of type DEF_USER_OBJ, DEF_GROUP_OBJ, DEF_CLASS_OBJ, and DEF_OTHER_OBJ.
- Entries of type USER, GROUP, DEF_USER, or DEF_GROUP may not contain duplicate entries. A duplicate entry is one of the same type containing the same numeric ID.
- If the **calclass** argument is zero and there are no entries of type USER and GROUP, the permissions of the GROUP_OBJ and CLASS_OBJ entries must be the same.
- If there are no entries of type DEF_USER and DEF_GROUP, and the DEF_GROUP_OBJ entry is specified, then the DEF_CLASS_OBJ entry must also be specified, and the permissions of the DEF_GROUP_OBJ and DEF_CLASS_OBJ entries must be the same.

RETURN VALUE

Upon successful completion, the return value is **0**. If there are duplicate entries, the return value is the position of the first duplicate entry. If there is more than one entry of type USER_OBJ, GROUP_OBJ, CLASS_OBJ, OTHER_OBJ, DEF_USER_OBJ, DEF_GROUP_OBJ, DEF_CLASS_OBJ, or DEF_OTHER_OBJ, they will be treated as duplicate entries, and the return value is the position of the first duplicate entry. For all other errors, the return value is **–1**.

SEE ALSO

acl(ES_LIB).

LEVEL

Level 1.

NAME

devalloc – get and set the security attributes of a device

SYNOPSIS

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

int devalloc(const char *device, int cmd, struct dev_alloca *bufp)
```

DESCRIPTION

The **devalloc()** routine allows privileged processes to get or set the security attributes of *device*, based on the specified *cmd*. The *device* can be either a device alias name defined in the Device Database (DDB) or an absolute pathname to a character or block special file. *bufp* is a pointer to a *struct dev_alloca*, defined in **mac.h**, which defines the following security attributes:

state	the device state,
mode	the device mode,
level	the current device level,
hilevel	the high level of the device level range,
lolevel	the low level of the device level range,
uid	the UID (for checking authorization permission), and
relflag	the device release flag.

If *cmd* is **DEV_GET**, **devalloc()** gets the security attributes for *device* from the Device Database (DDB) and places them into the structure pointed to by *bufp*. In this case, **devalloc()** does not return any values for the release flag, UID or level.

If the *cmd* is **DEV_SET**, **devalloc()** determines whether the *device* is allocatable by comparing the security attributes pointed to by *bufp* to those defined for the device in the DDB. **devalloc()** checks if:

- the device is allocatable with the specified *state*
- the device is allocatable with the specified *mode*
- the specified *hilevel* and *lolevel* range is enclosed by the range stored in the Device Database
- the specified *level* is enclosed by the *hilevel* and *lolevel* range specified
- the release flag passed is either set as **dev_persistent** or **dev_lastclose**
- the UID (when a valid UID is passed) is authorized to allocate the specified device
- the device is not in use (the release flag setting on all the device special files mapped to the device is **dev_system**, and *usecount* is 0)

If all these conditions are met, **devalloc()** issues a **lvlfile** system call to change the level of the device to that specified in *bufp*, clears any access control lists (ACLs) on the device, changes the DAC permissions to give ownership and read/write access to only the user, and issues a **devstat** system call to set the security attributes of the *device*, according to information passed in *bufp*. The DDB is locked during the entire process of allocation.

If *device* is an absolute pathname, **devalloc()** performs these actions on that pathname only. If *device* is a device alias name, **devalloc()** performs these actions on each pathname mapped to that device according to information stored in the Device Database.

If any of the system calls fails on one of the pathnames, **devalloc()** tries to undo all the work on the other pathnames. **devalloc()** will reset the level to the previous level and previous DAC ownership and reset the flag to **dev_system**. If the Enhanced Security Extension is not implemented, **devalloc()** will fail.

RETURN VALUE

Upon successful completion, the system call **devalloc()** returns a value of **0**; otherwise, a value of **–1** is returned and **errno** is set to indicate an error.

ERRORS

Under the following conditions, **devalloc()** fails and sets **errno** to the indicated value. (Refer to the descriptions of the system calls called by this function for other possible **errno** values.)

EACCES if access to the DDB is denied because of MAC or DAC.

EAGAIN if *cmd* is **DEV_SET**, and the DDB is in use and cannot be locked.

EBUSY if *cmd* is **DEV_SET**, and the specified device is in use (not tranquil).

EINVAL if *cmd* is **DEV_SET**, and the specified *hilevel*, *lolevel* or *level* is an invalid level.

EINVAL if *cmd* is **DEV_SET**, and *hilevel* does not dominate *lolevel*.

EINVAL if *level* or the level range specified is not enclosed by the *range* stored in DDB for that device.

EINVAL if *cmd* is **DEV_SET**, and *level* is not enclosed by the specified level.

EINVAL if the specified *state* is not valid for *device*.

EINVAL if the specified *mode* is not valid for *device*.

EINVAL if invalid *state* specified.

EINVAL if invalid *mode* specified.

EINVAL if invalid *cmd* specified.

EINVAL if *cmd* is **DEV_SET**, and the release flag specified is invalid.

EINVAL if *cmd* is **DEV_SET**, and the user ID specified is invalid.

EINVAL if *cmd* is **DEV_SET**, and the security attributes are not defined or are invalid for the specified device.

ENODEV if *device* is not defined in the DDB.

ENOENT if the DDB cannot be found.

EPERM If *cmd* is **DEV_SET** and the specified user ID does not have authorization permission to have *device* allocated.

SEE ALSO

acl(ES_LIB), devdealloc(ES_LIB), devstat(ES_LIB), lvlfile(ES_LIB), lvldom(ES_LIB), chown(BA_OS), chmod(BA_OS).

LEVEL

Level 1.

NAME

devdealloc - deallocates a device and sets its security attributes to system configuration

SYNOPSIS

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

int devdealloc(const char *device)
```

DESCRIPTION

The **devdealloc** routine sets the security attributes of the specified *device* back to "system configuration." The *device* can be either a device alias name defined in the Device Database (DDB) or an absolute pathname to a character or block special file. If the alias is a logical alias, only those device special files mapped to that alias in the DDB are deallocated. If the alias is a secure device alias, then the routine deallocates all device special files mapped to all the logical aliases that define **secdev** equal to the secure device alias.

The system configuration is as follows:

range	**hilevel=lolevel=0**
state	**private** (unless the driver was configured with security flag set to INITPUB, in which case *state* is set to **public**)
mode	**static**
release_flag	**DEV_SYSTEM**

devdealloc sets the device attributes by invoking the **devstat** system call with the *release_flag* set to **DEV_SYSTEM**. If the **devstat** system call fails on one of the pathnames, then **devdealloc** will continue to work on the remaining pathnames and will exit with a negative value. Note that **devdealloc** does not check if the device is in use.

The Device Database is locked during the entire process of deallocation of all device special files.

RETURN VALUE

If successful, **devdealloc** returns a **0**; otherwise, it returns **-1** and sets **errno** to one of the following values. (See devstat(ES_LIB), lvlfile(ES_LIB), chown(BA_OS), and chmod(BA_OS) for other **errno** that may be set if it fails.)

ERRORS

EACCES	if access to the DDB is denied because of MAC or DAC.
EACCES	if Device Database files are not in a consistent state.
EAGAIN	if the DDB is in use and cannot be locked.
ENODEV	if *device* is not defined in the DDB.
ENOENT	if the DDB cannot be found.
EPERM	if the invoking process does not have the appropriate privilege to deallocate a device.

SEE ALSO

devstat(ES_LIB), devalloc(ES_LIB).

LEVEL

Level 1.

NAME

devstat – get or set device security attributes

SYNOPSIS

#include <mac.h>

int devstat(const char ***path*, int** *cmd***, struct devstat ***bufp***);**

DESCRIPTION

The **devstat** system call gets or sets the security attributes of a device represented by *path*. The value in *cmd* determines if the system call sets or gets the security attributes. The **devstat** system call is a privileged operation and requires appropriate privileges.

path points to the pathname of a disk-based block or character special file. *cmd* will define the operation to be performed. The value of *cmd* may be one of the following:

DEV_GET to retrieve security attributes of a device.

DEV_SET to set security attributes of a device.

bufp is a pointer to a **devstat** structure. The security attributes to be set are taken from the structure or are returned in the structure, depending on the operation.

The contents of the structure pointed to by *bufp* include the following members:

```
ushort  dev_state;     /* device state */
                       /* DEV_PRIVATE or DEV_PUBLIC */
ushort  dev_mode;      /* mode of the device */
                       /* DEV_STATIC or DEV_DYNAMIC */
level_t dev_hilevel;   /* maximum level range of the device */
level_t dev_lolevel;   /* minimum level range of the device */
ushort  dev_usecount;  /* 0 if no open connections, */
                       /* 1 otherwise */
ushort  dev_relflag;   /* DEV_PERSISTENT, DEV_LASTCLOSE, or */
                       /* DEV_SYSTEM */
```

dev_state is either **private** or **public**. When a device is in private state, no unprivileged access to the device special file is allowed. All new **open**, **read**, **write**, **ioctl**, **mmap**, **getmsg**, **getpmsg**, **putmsg**, and **putpmsg** calls will fail if the process does not have the appropriate privileges. A process requires appropriate privileges and MAC and DAC access to open a device special file in the **private** state.

The device state is used to indicate if the device is a single-level or a multi-level device. If the device state is **private**, then the device can be either a single-level or a multi-level device. If the device state is **public**, then the device is single-level, because it can be used by an unprivileged process.

dev_mode should always be **DEV_STATIC**. Level change on a static device is allowed only if the device is in private state or if there are no open connections to the device special file. Please refer to **lvlfile**(ES_LIB). Another possible mode, **DEV_DYNAMIC**, is provided solely for sites upgrading from another secure system.

When the **dev_mode** is set to **DEV_DYNAMIC**, the level of the device can change while it is open, MAC access checks are performed for every **read**, **write**, **ioctl**, **putmsg**, and **getmsg** operations.

dev_hilevel and **dev_lolevel** specify the allowed level range that will constrain the **lvlfile** system call. The level of the device special file referenced by *path* must be dominated by **hilevel** and must dominate **lolevel**. **dev_hilevel** and **dev_lolevel** limit the level at which the device can be used.

dev_usecount is set to 1 if there are open connections to the device special file or if there is any mapping active. It is set to 0 otherwise. This field can only be retrieved and cannot be set.

dev_relflag indicates how these security attributes can be released. This flag can take one of three values:

DEV_PERSISTENT Indicates that the security attributes will be set for the device while the system is running or until the next **devstat** operation to set attributes.

DEV_LASTCLOSE Indicates that the security attributes will be released after the last close on a device and will be set to the one defined by the system.

DEV_SYSTEM For each device special file, the system defines the following security attributes: the **dev_lolevel** and **dev_hilevel** are set to 0, **state** is set to **private**, and **mode** is set to **static**.

If *cmd* is **DEV_GET**, the system call returns the security attributes of the device in the buffer pointed to by *bufp*.

If the *cmd* is **DEV_SET**, the device named by *path* has its security attributes set to the values passed in *bufp*. When setting the device with the **DEV_SYSTEM** release flag, all other information passed in *bufp* is ignored.

The calling process must ensure that no device special file that maps to the same device as *path*, as defined by the Device Database, is currently in use. The calling process must also ensure that the parameters for the device, as defined in the Device Database, are correctly applied when this system call is used.

RETURN VALUE

The system call returns zero (0) if successful. Otherwise, it returns -1 and sets **errno** to one of the below values.

ERRORS

EPERM The process does not have the appropriate privileges.

EINVAL The arguments to the system call are invalid.

EINVAL If the *cmd* is **DEV_SET**, **dev_hilevel** does not dominate **dev_lolevel**.

EINVAL If the *cmd* is **DEV_SET**, the range delimited by **dev_hilevel** and **dev_lolevel** does not enclose the level of the device special file.

ENOTDIR	A component of the *path* prefix is not a directory.
ENOENT	A component of the pathname of the named file does not exist.
EACCES	Access to *path* is denied by DAC, MAC or other access restrictions.
ENODEV	*path* indicates a file that is not a disk-based block or character special file.
ENAMETOOLONG	if the length of path exceeds PATH_MAX, or the length of a path component exceeds NAME_MAX while POSIX_NO_TRUNC is in effect

SEE ALSO

lvlfile(ES_LIB).

LEVEL

Level 1.

NAME

filepriv – set, get, or count the privileges associated with a file

SYNOPSIS

```
#include   <priv.h>

int filepriv(const char *path, int cmd, priv_t *privp,
int nentries)
```

DESCRIPTION

The **filepriv** system call is used to set, retrieve, or count the privileges associated with a file. *privp* is defined as a pointer to an array of privilege descriptors each of which contains a privilege set and the identity of the requested privilege. (See the Enhanced Security Extension Introduction for descriptions of terms such as ''privilege set.'')

The *path* argument specifies an executable file. *nentries* is the number of entries contained in *privp*.

The recognized *cmd*s and their functions are described below:

PUTPRV the fixed and inheritable privilege sets associated with the file indicated by *path* are set based on the privilege descriptor(s) contained in *privp*. The fixed and inheritable privilege sets resulting from the privilege descriptor(s) contained in *privp* must be disjoint. Privileges contained in either privilege set that are not in the maximum set of the calling process are ignored. The calling process must have the appropriate privilege in its working set when using **PUTPRV**. If any argument is invalid, none of the file privileges are changed. The new file privileges pointed to by **privp* replace the existing file privileges.

GETPRV the fixed and inheritable privilege sets associated with the file indicated by *path* are returned in *privp* in the form of privilege descriptors. None of the file privileges are changed.

CNTPRV the return value is set to the number of privileges associated with the named file. The *privp* and *nentries* arguments are ignored. None of the file privileges are changed.

RETURN VALUE

A value of –1 is returned and **errno** is set to indicate the error if **filepriv** is unsuccessful. If successful, **filepriv** returns the number of privileges associated with the named file.

ERRORS

filepriv fails if one or more of the following is true:

ENOENT a component of *path* does not exist.

ENOTDIR a component of *path* is not a directory.

EINVAL *cmd* is invalid.

EINVAL the *cmd* is **GETPRV** and *privp* is not large enough to hold the number of privileges associated with the named file.

EINVAL the *cmd* is **PUTPRV** and 1) the file pointed to by *path* is not an executable file, 2) the fixed and inheritable privilege sets are not disjoint, 3) *nentries* is less than 0, or 4) *privp* includes undefined privileges.

EACCES access is prohibited by an access restriction.

EPERM the calling process does not have the appropriate privileges to set file privileges.

EAGAIN insufficient kernel memory to allocate a privilege table entry when setting file privileges.

SEE ALSO

procpriv(ES_LIB), procprivl(ES_LIB).

LEVEL

Level 1.

NAME

lvldom – determine domination relationship of two levels

SYNOPSIS

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

int lvldom(level_t *level1p, level_t *level2p);
```

DESCRIPTION

The **lvldom()** routine compares the level pointed to by *level1p* and *level2p* and determines whether or not *level1p* dominates *level2p*. *level1p* and *level2p* must be pointers to valid levels, as may be obtained from the **lvlin()** routine or the **lvlfile** or **lvlproc** system calls.

RETURN VALUE

If the first level dominates the second, a positive integer is returned; if the first level does not dominate the second, a value of **0** is returned (note that this does not imply that the second level dominates the first); otherwise, a value of **–1** is returned and **errno** is set to indicate an error.

ERRORS

Under the following conditions, **lvldom()** fails and sets **errno** to:

EINVAL if the level referenced by *level1p* or *level2p* is not defined on the system.

SEE ALSO

lvlequal(ES_LIB), lvlin(ES_LIB), lvlproc(ES_LIB), lvlfile(ES_LIB), lvlvalid(ES_LIB).

LEVEL

Level 1.

NAME

lvlequal – determine equality of two levels

SYNOPSIS

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

int lvlequal(level_t *level1p, level_t *level2p);
```

DESCRIPTION

The **lvlequal()** routine compares the levels referenced by *level1p* and *level2p* and determines whether or not they are equal. *level1p* and *level2p* must be pointers to valid levels, as may be obtained from the **lvlin**(ES_LIB) routine or the **lvlfile**(ES_LIB) or **lvlproc**(ES_LIB) system calls.

RETURN VALUE

If the first level equals the second, a positive integer is returned; if the first level does not equal the second, a value of **0** is returned; otherwise, a value of **–1** is returned and **errno** is set to indicate an error.

ERRORS

Under the following conditions, **lvlequal()** fails and sets **errno** to:

EINVAL if the level referenced by *level1p* or *level2p* is not defined on the system.

SEE ALSO

lvldom(ES_LIB), lvlfile(ES_LIB), lvlin(ES_LIB), lvlproc(ES_LIB), lvlvalid(ES_LIB).

LEVEL

Level 1.

NAME

lvlfile – get or set the level of a regular file, directory, named pipe or device special file

SYNOPSIS

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

int lvlfile(char *path, int cmd, level_t *levelp);
```

DESCRIPTION

The **lvlfile** system call will get or set the level of the file represented by *path*, depending on the value in *cmd*.

If *cmd* is **MAC_GET**, the system call returns the level of the named file to the variable referenced by *levelp*. The invoking subject must have MAC read permission on the file.

If *cmd* is **MAC_SET** and the file is a regular file, directory or FIFO, the following conditions apply:

privilege — The invoking subject must be the owner or have appropriate privileges. The effective user ID of the calling process must also be the owner of the file, or the calling process must have appropriate privilege.

access — The invoking subject must have MAC write access and be the owner.

tranquillity — The file must be tranquil, i.e., it should not be open or mapped.

level validity — The level must be a valid level, as may be obtained from the **lvlin** routine or the **lvlfile** or **lvlproc** system calls.

If all these conditions are met, the system call will set the level of the named file to the level referenced by *levelp*.

If *cmd* is **MAC_SET** and the file is a character or block special file, the following conditions apply:

privilege — The invoking subject must have the **P_DEV** privilege if the device state is **private**. If the device state is **public**, then the invoking subject must either have the **P_SETFLEVEL** privilege or, if the new level strictly dominates the existing level, the **P_MACUPGRADE** privilege.

access — The invoking subject must be the owner of the file. If the device state is **public**, the invoking subject must also have MAC write access.

tranquillity — If the device special file is in **public** state and has its security mode set to **static**, then the device special file must be tranquil.

device range — If the device range has been set by a previous call to **devstat**, then the new level must be strictly dominated by the high level of the device, and must dominate the low level of device.

level validity The level must be a valid level, as may be obtained from the **lvlin** routine or the **lvlfile** or **lvlproc** system calls.

If all these conditions are met, the system call will set the level of the named character or block special file to the level referenced by *levelp*.

Note that the **lvlfile** system call must be used to set the level of a regular file, directory or FIFO.

RETURN VALUE

Upon successful completion, the system call returns zero (0). Otherwise, –1 is returned and **errno** is set to indicate the error.

ERRORS

EPERM	The *cmd* is **MAC_SET**, and the invoking subject does not have the appropriate privileges.
EPERM	The *cmd* is **MAC_SET**, and the invoking subject is not the owner of the file referred to by *path* or *fildes*.
ENOTDIR	For **lvlfile**, a component of the path prefix is not a directory.
ENOENT	For **lvlfile**, a component of *path* does not exist.
EACCES	For **lvlfile**, the invoking subject fails the access checks on a component of *path*.
EACCES	The invoking subject does not have MAC access to the file referred to by *path* or *fildes*.
EINVAL	The *cmd* is invalid.
EINVAL	The *cmd* is **MAC_SET**, and *levelp* is not defined on the system.
EBUSY	The *cmd* is **MAC_SET**, and the file is not tranquil (open or mapped).
ERANGE	The *cmd* is **MAC_SET**, and the file is a character or block device special file, and *levelp* is not within device level range.
ENOSYS	The *cmd* is **MAC_SET**, and the file system type does not support labeling.
EROFS	The *path* or *fildes* refers to a file that resides on a read-only file system.
ENAMETOOLONG	if the length of *path* exceeds PATH_MAX, or the length of a *path* component exceeds NAME_MAX while POSIX_NO_TRUNC is in effect

SEE ALSO

devstat(ES_LIB), lvlproc(ES_LIB), lvlin(ES_LIB).

LEVEL

Level 1.

NAME

lvlin – translate a level from text format to internal format

SYNOPSIS

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

int lvlin(char *bufp, level_t *levelp);
```

DESCRIPTION

The **lvlin()** routine translates the null terminated character string referenced by *bufp* to the corresponding internal format of the level and places it in the level referenced by *levelp*. The character string can contain either an alias or a fully qualified level name of the following format:

h_name [:*c_name* [,*c_name*]**...**]

where *h_name* is a hierarchical classification name and *c_name* a non-hierarchical category name.

RETURN VALUE

Upon successful completion, the **lvlin()** routine returns a value of **0** and the resultant level is placed in the variable referenced by *levelp*; otherwise, a value of **–1** is returned and **errno** is set to indicate an error.

ERRORS

Under the following conditions, **lvlin()** fails and sets **errno** to:

EINVAL if any of the text names given are not defined in the Level Translation Database (LTDB).

EINVAL if the resultant level is not defined on the system.

EACCES the calling process does not pass the access checks for the LTDB.

SEE ALSO

lvlout(ES_LIB).

LEVEL

Level 1.

NAME

lvlipc – manipulate an IPC object's level

SYNOPSIS

```
#include <sys/types.h>
#include <mac.h>
#include <sys/ipc.h>

int lvlipc (int type, int id, int cmd, level_t *levelp);
```

DESCRIPTION

The **lvlipc()** system call will manipulate an IPC object's level.

The only *cmd* currently supported is **MAC_GET**. This implies that this system call can only be used to retrieve an IPC object's level.

type must be one of the following:

IPC_SHM *id* must be a valid shared memory identifier returned by **shmget**.

IPC_SEM *id* must be a valid semaphore identifier returned by **semget**.

IPC_MSG *id* must be a valid message queue identifier returned by **msgget**.

The level of the IPC object specified by *type* and *id* is copied into the user supplied buffer *levelp*. Note that the level returned is in internal format of a level, which may be converted to text format via the **lvlout()** routine.

The user must have read access to the IPC object. An unprivileged user has read access to an IPC object only if the user's security level dominates the object's security level, and the user has discretionary read access. A user with the appropriate privilege has access to all objects.

RETURN VALUE

Upon successful completion, the system call **lvlipc()** returns a value of **0**; otherwise, a value of **–1** is returned and **errno** is set to indicate an error.

ERRORS

Under the following conditions, **lvlipc()** fails and sets **errno** to:

EINVAL if *cmd* is **MAC_GET** and the security level of the calling process is strictly dominated by the security level of the IPC object, and the calling process lacks the appropriate privileges.

EINVAL if *type* is not **IPC_SHM**, **IPC_SEM**, or **IPC_MSG**.

EINVAL if *id* is not a valid (or active) *type* identifier.

EINVAL if *cmd* is not **MAC_GET**.

EACCES if the user does not have discretionary read access to the IPC object.

SEE ALSO

lvlout(ES_LIB), msgget(KE_OS), semget(KE_OS), shmget(KE_OS).

LEVEL

Level 1.

NAME

lvlout – translate a level from internal format to text format

SYNOPSIS

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

int lvlout(level_t *levelp, char *bufp, int bufsize, int format);
```

DESCRIPTION

The **lvlout()** routine translates the level referenced by *levelp* to the corresponding alias or fully qualified level (depending on the value in *format*), and places it in the character buffer referenced by *bufp*.

format must be one of the following:

LVL_ALIAS the corresponding alias is placed in *bufp*. If the alias does not exist, the character representation of the decimal value of the level identifier (LID) is returned.

LVL_FULL the corresponding fully qualified level is placed in *bufp*. If the level is valid but inactive (deleted), the character representation of the decimal value of the LID is returned.

If *bufsize* is **0**, the return value is the length that *bufp* must be to hold the resultant string.

RETURN VALUE

Upon successful completion, the following occurs:

bufsize == 0 The size requirement for the resultant null terminated character string is returned.

bufsize != 0 If the level is in the valid-active state, a value of **0** is returned. If the level is in the valid-inactive state, a positive integer is returned.

Otherwise, a value of **–1** is returned and **errno** is set to indicate the error.

ERRORS

Under the following conditions, **lvlout()** fails and sets **errno** to:

EINVAL if the *format* is not valid.

EINVAL if the level referenced by *levelp* does not exist on the system.

ENOSPC if the resultant text string is larger than *bufsize*.

SEE ALSO

lvlin(ES_LIB)

LEVEL

Level 1.

NAME

lvlproc – get or set the level of a process

SYNOPSIS

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

int lvlproc(int cmd, level_t *levelp)
```

DESCRIPTION

The **lvlproc**() system call gets or sets the level of the calling process, depending on the value in *cmd*.

If *cmd* is **MAC_GET**, **lvlproc**() returns the level of the calling process to the variable referenced by *levelp*.

If *cmd* is **MAC_SET**, the calling process must have the appropriate privileges. If it does, **lvlproc**() sets the level of the calling process to the level referenced by *levelp*. The level referenced by *levelp* must be a valid level obtained by a previous **lvlin**(ES_LIB) routine or **lvlfile**(ES_LIB) or **lvlproc**() system call.

RETURN VALUE

Upon successful completion, the system call **lvlproc**() returns a value of **0** and the following action is taken:

MAC_GET The object pointed to by *levelp* contains the level of the calling process.

MAC_SET The level of the calling process is set to the object pointed to by *levelp*.

Otherwise, a value of **–1** is returned and **errno** is set to indicate an error.

ERRORS

Under the following conditions, **lvlproc**() fails and sets **errno** to:

EINVAL if *cmd* is invalid.

EINVAL if *cmd* is **MAC_SET** and the level referenced by *levelp* is not defined on the system.

EPERM if *cmd* is **MAC_SET** and the caller lacked the appropriate privileges.

SEE ALSO

lvlfile(ES_LIB), lvlin(ES_LIB).

LEVEL

Level 1.

NAME

lvlvalid – check the validity of a level

SYNOPSIS

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

int lvlvalid(level_t *levelp);
```

DESCRIPTION

The **lvlvalid()** routine checks the validity of the level referenced by *levelp*.

RETURN VALUE

If the level is valid in the active state, a value of **0** is returned; if the level is valid in the inactive state, a positive integer is returned; otherwise, a value of –1 is returned and **errno** is set to indicate an error.

ERRORS

Under the following conditions, **lvlvalid()** fails and sets **errno** to:

EINVAL if the level referenced by *levelp* does not exist on the system.

EACCES If the calling process does not pass the access checks for the **Level Translation Database (LTDB)**.

LEVEL

Level 1.

NAME

lvlvfs - get or set the level ceiling of a mounted file system

SYNOPSIS

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

int lvlvfs(char *path, int cmd, level_t *hilevelp);
```

DESCRIPTION

Depending on the value of *cmd*, the **lvlvfs** system call gets or sets the level ceiling of a mounted file system in which *path* resides. **lvlvfs** requires the appropriate privileges.

If *cmd* is **MAC_GET**, **lvlvfs** returns the level ceiling of the mounted file system in the variable pointed to by *hilevelp*.

If *cmd* is **MAC_SET**, **lvlvfs** sets the level ceiling of the mounted file system to the value pointed to by *hilevelp*. The level pointed to by *hilevelp* must be a valid level, which may be obtained from the **lvlin** routine.

RETURN VALUE

When successful, **lvlvfs** returns 0. Otherwise, it returns -1 and sets **errno** to one of the following values:

ERRORS

EACCES The invoking subject failed the access checks on a component of *path*.

EINVAL The *cmd* is invalid.

EINVAL The *cmd* is **MAC_SET**, and *hilevelp* is not defined on the system.

ERANGE The *cmd* is **MAC_SET**, and *hilevelp* does not dominate the floor level of the file system.

ENOENT A component of *path* does not exist.

ENOTDIR A component of the *path* prefix is not a directory.

ENOSYS The *cmd* is **MAC_SET**, and the file system does not support labeling.

EPERM The invoking subject does not have the appropriate privileges.

ENAMETOOLONG
if the length of *path* exceeds PATH_MAX, or the length of a *path* component exceeds NAME_MAX while POSIX_NO_TRUNC is in effect

SEE ALSO

devstat(ES_LIB), lvlproc(ES_LIB), lvlfile(ES_LIB), lvlin(ES_LIB).

LEVEL

Level 1.

NAME

mkmld – make a Multilevel Directory

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <mac.h>

int mkmld(const char *path, mode_t mode);
```

DESCRIPTION

mkmld creates a Multilevel Directory (MLD) named by *path*. The calling process must possess the appropriate privileges. The mode of the new directory is initialized from *mode* [see **chmod**(BA_OS) for values of mode]. The protection part of the *mode* argument is modified by the process's file creation mask [see **umask**(BA_OS)].

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to the process's effective group ID, or if the **S_ISGID** bit is set in its parent directory, then the group ID of the directory is inherited from the parent. The **S_ISGID** bit of the new directory is inherited from the parent directory.

If the final component of *path* is a symbolic link, it is not followed.

The newly created directory is empty with the exception of entries for itself (**.**) and its parent directory (**..**).

Upon successful completion, **mkmld** marks for update the **st_atime**, **st_ctime** and **st_mtime** fields of the directory. Also, the **st_ctime** and **st_mtime** fields of the directory that contains the new entry are marked for update.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned, and **errno** is set to indicate the error.

ERRORS

mkmld fails and creates no directory if one or more of the following are true:

EACCES Either a component of the path prefix denies search permission or write permission is denied on the parent directory of the directory to be created.

EEXIST The named file already exists.

EIO An I/O error has occurred while accessing the file system.

ELOOP Too many symbolic links were encountered in translating *path*.

EMLINK The maximum number of links to the parent directory would be exceeded.

ENAMETOOLONG The length of the *path* argument exceeds **PATH_MAX**, or the length of a *path* component exceeds **NAME_MAX** while **_POSIX_NO_TRUNC** is in effect.

ENOENT A component of the path prefix does not exist.

ENOSPC	No free space is available on the device containing the directory.
ENOTDIR	A component of the path prefix is not a directory.
EROFS	The path prefix resides on a read-only file system.
EPERM	The calling process lacks the appropriate privileges.
ENOSYS	The file system on which **path** resides does not support Multilevel Directories.

SEE ALSO

chmod(BA_OS), mkdir(BA_OS), mknod(BA_OS), umask(BA_OS).

LEVEL

Level 1.

NAME

mldmode - Retrieve or set the Multilevel Directory mode of a process

SYNOPSIS

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

int mldmode(int mode)
```

DESCRIPTION

The **mldmode()** system call retrieves or sets the Multilevel Directory mode of the calling process based on the value in *mode*. Acceptable values of *mode* are **MLD_QUERY**, **MLD_REAL** and **MLD_VIRT**. If **MLD_QUERY** is specified, the return value of the call will be **MLD_REAL** or **MLD_VIRT**, indicating the current Multi-Level Directory mode of the process. Specifying **MLD_REAL** puts the process in *real* mode so that MLDs are treated as regular directories. Specifying **MLD_VIRT** puts the process in *virtual* mode so that the process deflects through the MLD to the effective directory at the level of the process.

RETURN VALUE

If **MLD_QUERY** is specified, successful completion is indicated by the return value of **MLD_REAL** or **MLD_VIRT**. If **MLD_REAL** or **MLD_VIRT** is specified, successful completion is indicated by a return value of 0. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

mldmode() fails if the following is true:

EINVAL Arguments to the system call are invalid.

SEE ALSO

mkmld(ES_LIB).

LEVEL

Level 1.

NAME

procpriv – add, remove, set, retrieve, or count privileges associated with the calling process

SYNOPSIS

#include <priv.h>

int procpriv(int *cmd*, **priv_t** ***privp**, **int** *nentries***)**

DESCRIPTION

The **procpriv** system call is used to add, remove, retrieve, count, or put the privileges associated with the calling process. *privp* is a pointer to an array of privilege descriptors, each of which contains the privilege set and identity of the requested privilege. *nentries* is the number of entries contained in *privp*. (See the Enhanced Security Extension Introduction for descriptions of terms such as ''privilege set.'')

The recognized *cmd*s and their functions are described below:

SETPRV the working privilege set for the current process is set based on the privilege descriptor(s) contained in *privp*. All requested privileges not contained in the current maximum privilege set are ignored. All requested working privileges that are in the current maximum set are added to the working privilege set. A request may include adding one or more privileges that are already in the current working privilege set without causing an error. If any argument is invalid, none of the process privileges are changed.

CLRPRV the working and maximum privilege sets for the current process are cleared based on the privilege descriptor(s) contained in *privp*. All requested privileges are removed from their respective sets. If a privilege is removed from the maximum set it's automatically removed from the working set. A request may include removing one or more privileges from the current working (or maximum) privilege set that are not in the current working (or maximum) privilege set without causing an error. If any argument is invalid, none of the process privileges are changed.

PUTPRV the working and maximum privilege sets for the current process are set based on the privilege descriptor(s) contained in *privp*. The working privilege set is adjusted to be a subset of the resulting maximum set. Privileges contained in either privilege set that are not in the maximum set of the calling process are ignored. If any argument is invalid, none of the process privileges are changed.

GETPRV the working and maximum privilege sets for the current process are returned in *privp* in the form of privilege descriptors. None of the process privileges are changed.

CNTPRV returns the number of privileges associated with the current process. The *privp* and *nentries* arguments are ignored. None of the process privileges are changed.

RETURN VALUE

A value of –1 is returned and **errno** is set to indicate the error if **procpriv** is unsuccessful. If successful, **procpriv** returns 0 for **SETPRV**, **CLRPRV**, **PUTPRV** or the number of privileges associated with the current process for **GETPRV**, and **CNTPRV**.

ERRORS

procpriv fails if the following is true:

EINVAL *cmd* or privilege specified is invalid, or *nentries* is less than 0, or *cmd* is **GETPRV** and the process privileges exceeds *nentries*.

EPERM the calling process does not have the appropriate privileges to set file privileges.

SEE ALSO

filepriv(ES_LIB), procprivl(ES_LIB).

LEVEL

Level 1.

NAME

procprivl – add, remove, set, or count privileges associated with the calling process

SYNOPSIS

```
#include <priv.h>

int procprivl(int cmd, priv_t priv1, ...)
```

DESCRIPTION

The **procprivl** system call is used to add, remove, count, or put the privileges associated with the calling process. *priv1* is a list of privilege descriptors, each of which contains the privilege set and identity of the requested privilege. The list is terminated with a **(priv_t)**0 value. (See the Enhanced Security Extension Introduction for descriptions of terms such as ''privilege set.'')

The recognized *cmd*s and their functions are described below:

SETPRV the working privilege set for the current process is set based on the privilege descriptor(s) contained in the list. All requested privileges not contained in the current maximum privilege set are ignored. All requested working privileges that are in the current maximum set are added to the working privilege set. A request may include adding one or more privileges that are already in the current working privilege set without causing an error. If any argument is invalid, none of the process privileges are changed.

CLRPRV the working and maximum privilege sets for the current process are cleared based on the privilege descriptor(s) contained in the list. All requested privileges are removed from their respective sets. If a privilege is removed from the maximum set it's automatically removed from the working set. A request may include removing one or more privileges from the current working (or maximum) privilege set that are not in the current working (or maximum) privilege set without causing an error. If any argument is invalid, none of the process privileges are changed.

PUTPRV the working and maximum privilege sets for the current process are set based on the privilege descriptor(s) contained in the list. The working privilege set is adjusted to be a subset of the resulting maximum set. Privileges contained in either privilege set that are not in the maximum set of the calling process are ignored. If any argument is invalid, none of the process privileges are changed.

CNTPRV returns the number of privileges associated with the current process. The rest of the arguments, if any, are ignored. None of the process privileges are changed.

RETURN VALUE

A value of –1 is returned and **errno** is set to indicate the error if **procprivl** is unsuccessful. If successful, **procpriv** returns 0 for **SETPRV**, **CLRPRV**, **PUTPRV** or the number of privileges associated with the current process for **GETPRV**, and **CNTPRV**.

ERRORS

procprivl fails if the following is true:

EINVAL *cmd* or privilege specified is invalid.

EPERM the calling process does not have the appropriate privileges to set file privileges.

SEE ALSO

filepriv(ES_LIB), procpriv(ES_LIB).

LEVEL

Level 1.

Enhanced Security Extension Commands And Utilities

The following section contains the manual pages for the ES_CMD routines

NAME

admalloc – allocates devices to users based on information stored in the Device Database (DDB).

SYNOPSIS

admalloc [–o] [–m] [–w *level* **–u** *user***,***group***] [–r** *hilevel***-***lolevel***]** *device* **...**

admalloc –s

admalloc –d [–f] [*device* **...]**

DESCRIPTION

admalloc in the first form, allocates the specified device(s) to the user (*user, group*), and sets the specified security attributes (*level, hilevel, lolevel*) on the *device*. If *user* and *group* are not specified the device is allocated to the invoking user (real UID and GID). If *level* is not specified then the invoker's level is used as the working level on the allocated device. If a level range (*hilevel***-***lolevel*) is not specified, the range defined in the DDB for that *device* is used.

In the second form (**–s** option), it allocates only those devices defined in the Device Database, with their *startup* flag set to ''yes'' (enabled). The startup attributes defined in the DDB are used to set-up the DAC permissions on the device special files mapped to each device.

In the third form (**–d** option), it deallocates the specified device(s).

When the device is allocated, by default the *release flag* is set to **DEV_PERSISTENT**, and *mode* is set to the value defined in the DDB for that *device*. The access control list (*acl*) on all device special files (dsfs) allocated will be removed before giving access permission to the device.

When the device is deallocated, the *release flag* is set to **DEV_SYSTEM**. The *acl* will be removed on those devices (dsfs) that have *startup_owner* defined in the DDB.

If *admalloc* cannot allocate or deallocate a device, the appropriate error message is printed and processing resumes at the next device specified in the argument list or the Device Database (–s option).

The options and arguments for this command are:

device is the alias name of a device in the Device Database, or the absolute pathname of a device special file (dsf) defined in the Device Database. If *device* is a dsf, then only that dsf gets allocated/deallocated based on the other options. If *device* is an alias, than all dsfs mapping to that alias (as defined in the DDB), get allocated. If *device* is a *secure device* alias (see **putdev**) then, all dsfs mapped to all aliases that define *secdev* equal to the specified *secure device* get allocated.

Allocation options:

-o used to override the user access restrictions (only), imposed by the *User Authorization List* in the Device Database. This option must be used with care.

-m indicates that the *device* must be allocated in *private* state (for multi-level device usage). If [**-m**] is not specified, then the device is allocated in *public* state.

-w used to set the working level, *level*, of the device (device special file). This option must be used with the **–u** option.

-u specifies the *user* and *group* to which the device is allocated. The specified values become the owner and group of the device special files mapped to the device, and their DAC is set to allow read and write to only the *user*. If *user, group* is not specified, then the invoking user (real UID and GID) are used.

-r specifies the MAC level range, *hilevel-lolevel*, to be used to set the level range on the device (device special file), when it is allocated. A dash character (-) is the range delimiter. If [**-r**] is not specified, then the range defined in the DDB for that device is used. The specified range, must be within the level range defined in DDB for that device. Otherwise the command fails.

Allocation at startup option:

-s all devices (aliases) in the Device Database, that have the *startup* attribute set to *yes* will be allocated, based on information stored for that device in the DDB. The device is allocated with the values of *range, state, and mode* defined in the DDB. The DAC ownership and permissions on devices allocated are also taken from the DDB. If all the startup attributes (*startup_level*, *startup_owner*, *startup_group*, and *startup_other*) are not defined in the DDB, then the command fails.

Deallocation options:

-d used to deallocate the specified *device*. Deallocation will be successful if none of the device special files mapped to a device are open or mapped. If deallocation is successful and the DDB entry for the specified device defines startup level and startup owner attributes, then the level and DAC ownership of the device are reset to those values. However, if the *startup* attributes are not defined, then the DAC permissions and MAC level of the device (dsf) are unchanged. If no device argument is specified, then *admalloc* will attempt to deallocate every device defined in the DDB.

-f implies ''forced release''. When this option is used with [**-d**], then the device is deallocated, even if there are open connections or mapping active to the specified *device*.

RETURN VALUE

For incorrect syntax the command fails and the exit code equals 1. For any error message displayed on partial failure of command (where the command successfully works for some of the devices in argument list), the exit code equals 2. If the Enhanced Security Extension is not implemented, then the exit code is set to 3. Exit code equals 4 for all other error messages.

FILES

/etc/device.tab
/etc/security/ddb/ddb_dsfmap
/etc/security/ddsb/ddb_sec

SEE ALSO

devattr(ES_CMD), devstat(ES_CMD), putdev(ES_CMD).

LEVEL

Level 1.

NAME

adminrole – display, add, change, delete roles in the Trusted Facility Management (TFM) database.

SYNOPSIS

adminrole [–n] [–a [*cmd***:***path***[:***priv* **[:***priv* **...]][,...]]** *role* **...**
adminrole [–a [*cmd***:***path***[:***priv* **[:***priv* **...]][,...]]**
    [–r *cmd* **[:***priv***[:***priv* **...]][,...]]** *role* **...**
adminrole [–d] *role* **...**
adminrole

DESCRIPTION

The **adminrole** command allows administrators to display, add, change, and delete roles in the TFM database. A role contains a list of commands. Each command contains a (possibly empty) list of privileges. The **tfadmin** command will use these privileges to set up its process before it invokes this command for a member of the role. The **adminrole** command has the following options:

-n	For every role in the list, create a new role description.
-a	Add a command to a role, add the role to the database if it does not already exist.
-r	Remove a command from a role or remove privileges from a command within a role.
-d	Delete a role.
No options	List the contents of the specified roles.
No arguments	List the contents of all roles in the database.

The **adminrole** command takes as its arguments the list of roles to which the actions specified by the options applies. The argument to the **-a** or **-r** option is a comma-separated list of command descriptions. For the **-a** option the command description includes the name of the command to be added, the full path at which the command file resides, and the privilege set, represented by a colon separated list of privilege names, for example:

```
mount:/etc/mount:macread:mount
```

The command description for the **-r** option is the same as for the **-a** option except that the full path and the separating colon are not given (for example, **mount:macread:mount**). If users in the specified roles get no privilege when they invoke the command, the privilege description may be omitted. When the **–a** and **–r** options are both specified on the command line, the **–r** options are processed first.

RETURN VALUE

This command exits with a **0** if all requested operations succeeded, **1** if any operation failed.

FILES

/etc/security/tfm/roles/*
/etc/security/tfm/roles/*/cmds/*

USAGE

System administrator.

SEE ALSO

adminuser(ES_CMD), tfadmin(ES_CMD).

LEVEL

Level 1.

NAME

adminuser – display, add, change, delete users in the TFM database.

SYNOPSIS

adminuser [**–n**] [**–o** *role*[**,...**]]
[**–a** *cmd***:***path*[**:***priv* [**:***priv***...**]][**,...**]]
[**–D** *cmd*]
user **...**
adminuser [**–o** *role*[**,...**]]
[**–r** *cmd* [**:***priv*[**:***priv***...**]][**,...**]]
[**–a** *cmd***:***path*[**:***priv* [**:***priv***...**]][**,...**]]
[**–D** *cmd*]
user **...**
adminuser [**–d**] *user***...**
adminuser

DESCRIPTION

The **adminuser** command allows administrators to display, add, change, and delete users in the TFM database. A user definition contains a list of commands. Each command contains a list of privileges. The **tfadmin** command uses these privileges to set up its process before invoking this command for the user. In addition to the command definitions, there is a list of roles available to the user, and a default command specification.

-n	For every user in the list, create a new user description, and, optionally, create a role list or add a command to that user.
-o	Create the specified role list for every user in the list.
-a	Add a list of commands to the definitions of a given list of users.
-r	Remove the list of commands from the list of users. If the user supplies privileges in the command descriptions, then leave the command but remove the specified privileges.
-D	Set the default command for a given list of users.
-d	Delete the given list of users.
No options	Print out the capabilities of the given list of users.
No arguments	Print the capabilities of every user in the database.

The **adminuser** command takes as its arguments the list of users to which the actions specified by the options applies. The list of users is a list of user login names.

The argument to the **-o** option is a comma-separated list of role names. This list will create a new role list for the specified users, replacing any existing role lists.

The argument to the **-a** or **-r** option is a comma-separated list of command descriptions. For the **-a** option the command description includes the name of the command to be added, the full path at which the command file resides, and the privilege vector, represented by a colon-separated list of privilege names (for example, **mount:/etc/mount:macread:mount**). The command description for the **-r** option is the same as for the **-a** option except that the full path and the separating colon are not given (for example, **mount:macread:mount**). If the users get no privileges when they invoke the command, the privilege description may be omitted. When the **–a** and **–r** options are both specified on the command line, the **–r** options are processed first.

The argument to the **-D** option is the name of the command to be run if *user* executes **tfadmin** without specifying a command name.

RETURN VALUE

This command exits with a **0** if all requested operations succeeded, **1** if any operation failed.

FILES

/etc/security/tfm/users/*

/etc/security/tfm/users/*/default

/etc/security/tfm/users/*/roles

/etc/security/tfm/users/*/cmds/*

USAGE

System administrator.

SEE ALSO

adminrole(ES_CMD), tfadmin(ES_CMD).

LEVEL

Level 1.

NAME

chlvl – change the level of a file

SYNOPSIS

chlvl *level file1* **...**

DESCRIPTION

chlvl will change the level of the named file(s). The new *level* must be either a valid alias level, or a valid fully qualified level name of the following format:

h_name[:*c_name*[,*c_name*]] ...]

where *h_name* is a hierarchical classification name, and *c_name* is a non-hierarchical category name. A fully qualified level is valid if the classification and categories comprising the level are named, and the level has been assigned a system level identifier number (LID) using the **lvlname** command. An alias name is valid if the alias has been assigned to a fully qualified level using the **lvlname** command. Valid levels can be viewed using the **lvlname** command. *level* must, furthermore, be within the file system level range.

The named file(s) must be accessible by the user. In addition, except for the root of a mounted file system and for block or character device special files that are set in **dynamic** mode or are in **private** state, none of the specified files may be open and/or mapped. If a directory is listed, it must not be the mount point of a currently mounted filesystem. To change the level of a mount point, unmount the filesystem, call **chlvl** on the mount point, and then remount. For a block or character device special file, the specified *level* must also be within the device level range. The security attributes of a device special file can be viewed using the **devstat** command. If **chlvl** encounters an error for a specific file, an error message is printed and processing resumes with the next file (if any).

ERRORS

One or more of the following error messages may appear on output:

invalid invocation syntax

invalid security level specified

LTDB is inaccessible

file ''*filename*'' is inaccessible

file ''*filename*'' is not 'tranquil' (that is, file is open and/or mapped or root of mounted file system)

security level specified is not within device range

file system for file ''*filename*'' does not support per-file labels

file system for ''*filename*'' is mounted read-only

permission denied for file ''*filename*''

FILES

/etc/security/mac/ltf.cat	category names
/etc/security/mac/ltf.class	classification names
/etc/security/mac/ltf.alias	alias names
/etc/security/mac/lid.internal	fully qualified levels

USAGE

This command is restricted to use by an administrator.

SEE ALSO

devstat(ES_CMD), lvlname(ES_CMD), lvldelete(ES_CMD).

LEVEL

Level 1.

NAME

defsak – define, remove, change, or display secure attention key

SYNOPSIS

usr/sbin/defsak -d *sak* [**-x**] *path* **...**
usr/sbin/defsak -d none *path* **...**
usr/sbin/defsak -r *path* **...**
usr/sbin/defsak [*path* **...**]

DESCRIPTION

The **defsak** administrative command is used to define, remove, change, or display the Secure Attention Key (SAK) for terminals. The SAK is a signal that a user sends to the host computer to establish a secure communications channel, or trusted path, for login. Users cannot log in on terminals that do not have a defined SAK.

If invoked without any options or arguments, **defsak** prints the SAKs for all defined terminal *path*s, in the following format:

path: *sak* [**+drop**]

The *path* is the absolute path name of the terminal device. The *sak* is the SAK defined for the terminal. An optional **+drop** suffix may be displayed; if present, it indicates that the line drop signal is also recognized as a SAK.

If invoked without options but with the absolute path name(s) of one or more terminals as an argument, **defsak** displays information about the SAK(s) for the specified terminal(s).

defsak has the following options:

-d *sak* This option defines the SAK for a terminal or terminals. The SAK may be either a control character or the break or line drop signal. A control character is specified either as an octal number in the range 000 to 015 or 020 to 037 (for example, **001**) or as a character preceded by a caret (for example, **^A**). A line drop or break is specified as the SAK by using the strings **break** or **drop** after the **-d** option. For example, the command **defsak -d drop** specifies that the line drop is the SAK.

-d none This option disables the trusted path processing for the terminal specified by the *path* argument. A warning message is printed, indicating that the terminal is no longer secure. This feature is intended to support communications utilities, such as **uucp**.

-x This option defines the line drop as a SAK, in addition to the SAK defined with the **-d** option. The **-x** option can be used only with the **-d** option.

-r This option removes the SAK for a tty device. If the SAK is removed, the terminal is disabled and cannot be used for logins. This is not the same as defining the SAK as **none**. Defining the SAK as **none** allows someone (or some program) to log in without entering the SAK. Removing the SAK with **-r** disables the terminal completely.

RETURN VALUE

Upon successful completion, **defsak** returns a value of **0**. If the SAK is defined as **none**, the following warning message is printed:

> **SAK disabled for terminal** *path***,**
> **terminal is no longer secure**

Otherwise, a non-zero value is returned and one of the following error messages is printed:

path **not defined in SAK database**
: The *path* argument does not correspond to a known terminal.

invalid SAK specified
: The SAK specified to **-d** is not a control character, line drop, break, or **none**.

SAK database is not accessible
: The **_pmtab** database file is not accessible.

Illegal option
: Incorrect syntax used

FILES

/etc/saf/*pmtag***/_pmtab**

USAGE

Because the system will end a user's login session whenever it sees the SAK as input, the SAK should not be a character that users will normally type. It is preferable to use the line drop signal as the SAK, because this signal is not used as normal user input. Use of the line drop as the SAK is recommended unless tty access to the system is via a modem or access emulates modem signals. In these cases, use a break signal.

All terminals at a site should have the same SAK, if possible. This makes it easier for users to remember the SAK and simplifies system administration.

Using a control character as the SAK is discouraged. A control character should be used only if it is not possible to use the line drop or break signals as the SAK. Using a control character as the SAK has the following problems:

- A control character SAK restricts the setting of terminal characteristics, and it may be difficult to find a character that is not used by application programs and commands.
- Control character SAKs may not work well in environments, such as terminal-based windowing packages, where data messages are wrapped by protocol information. Protocol information may contain the SAK, in which case the user will be logged out immediately, possibly in the middle of a protocol.

If you choose a character SAK, do not use any character in the set of default settings for special characters defined in **termio(BA_DEV)**. Doing so will cause **ioctl** failures when the tty device's **termio** characteristics are being set. Choosing one of the following as a SAK is strongly discouraged:

back space	octal **010**
horizontal tab	octal **011**
new line	octal **012**
vertical tab	octal **013**
new page	octal **014**
carriage return	octal **015**
Control-D	octal **004**
Control-S	octal **023**
Control-Q	octal **021**

Redefinition of the SAK is discouraged because doing so does not have any security benefits and can confuse users.

SEE ALSO

termio(BA_DEV)

LEVEL

Level 1.

NAME

devattr – lists device attributes

SYNOPSIS

devattr [**-v**] *device* [*attribute* . . .]

DESCRIPTION

devattr displays the values for a device's attributes. The display can be presented in two formats. When run without the **-v** option, **devattr** shows only the attribute values. When run with **-v**, **devattr** shows the attributes in the format *attribute*=*value*[,*value* . . .]. When no attributes are given on the command line, all attributes for the specified device are displayed in alphabetical order by attribute name. If attributes are given on the command line, only those are shown and they are displayed in command line order.

The options and arguments for this command are:

-v Specifies verbose format. Attribute values are displayed in an *attribute*=*value* format.

device Defines the device for which attributes should be displayed. It can be the absolute pathname of the device or the device alias. If it is an absolute pathname, then **devattr** gets the device alias name to which the pathname maps and displays all the attributes defined for that alias. If the alias is a secure device alias, then security attributes are also displayed.

attribute Defines which attribute, or attributes, should be shown. The default is to show all attributes for a device. [See **putdev**(ES_CMD) for a complete list of attributes.] If the system supports multilevel security, it is possible to query for information on the secure device alias and security attributes. Otherwise, such queries will fail.

RETURN VALUE

Upon successful completion, **devattr** returns a value of **0**. Otherwise, it returns a non-zero value.

FILES

/etc/device.tab
/etc/security/ddb/ddb_dsfmap
/etc/security/ddb/ddb_sec referenced only if the Enhanced Security Extension is implemented

SEE ALSO

getdev(ES_CMD), putdev(ES_CMD).

LEVEL

Level 1.

NAME

devstat – gets the current security attributes of a device

SYNOPSIS

devstat -Z [*device* . . .]

devstat -z [*device* . . .]

DESCRIPTION

The **devstat** command allows administrators to get the current security attributes of a specified device and, thus, to determine which devices are allocated and in use on the system. The device security attributes are those defined in the kernel, not those stored in the Device Database [see **devattr**(ES_CMD)].

Security information is printed for each of the specified *device* arguments. If no arguments are passed, **devstat** displays information on every device (all device special files) defined in the Device Database.

The *device* argument can take any one of the following four forms: (1) an absolute pathname for a device special file defined in the Device Database, (2) an absolute pathname for a device special file that's not defined in the Device Database, (3) a device alias name, or (4) a secure device alias. If *device* is an absolute pathname listed in the Device Database, **devstat** prints the security attributes of that device special file. If *device* is an absolute pathname for a device special file not defined in the Device Database (but the character or block device special file exists in the system), **devstat** displays the information provided by **devstat**(BA_OS). If *device* is a device alias name, **devstat** prints the security attributes of every device special file mapped to that alias. If *device* is a secure device alias, **devstat** prints the security attributes of every device special file mapped to all aliases for which the **secdev** attribute is equal to that secure device alias.

devstat has the following options:

-Z Print security levels as fully qualified level names.

-z Print security levels as level aliases.

If the level is not defined in the Level Translation Database (LTDB), **devstat** prints a text representation of the binary value of the level identifier (LID).

For each specified device, **devstat** displays a status report in the following form:

device name: *secure_device_alias*
pathname: *device_special_file*
  state: *state*
  mode: *mode*
  high: *level*
  low: *level*
  use count: *use count*
  release flag: *relflag*

When a requested device is a character or block special file that is not defined in the Device Database, the values reported for the **device name** and **pathname** fields are the same.

Under **pathname**, the following fields are listed:

state Either **private** (only privileged processes can access the device) or **public**.

mode Either **static** or **dynamic**.

high, **low** Either fully qualified level names or level aliases for valid level identifiers.

use count Set to **1** if connections are open or mappings are being made to the device special file. Otherwise, set to zero (0).

release flag One of three values: **persistent**, **lastclose**, or **system**. If **persistent**, the security attributes remain in effect until they are explicitly reset. If **lastclose**, the device security attributes remain in effect until the last close and are then reset to the system security attributes. If **system**, the device security attributes are set to the system security attributes.

RETURN VALUE

Whenever **devstat** fails to print the status of a specified device, it displays an appropriate error message and continues processing with the next specified device. Upon completion, **devstat** exits with an exit code of **0** if it was successful, **2** if it was partially successful, and the appropriate code if it fails.

FILES

/etc/device.tab
/etc/security/ddb/ddb_dsfmap
/etc/security/ddb/ddb_sec

SEE ALSO

admalloc(ES_CMD), devattr(ES_CMD), lvlname(ES_CMD), putdev(ES_CMD).

LEVEL

Level 1.

NAME

filepriv – set, delete, or display privilege information associated with a file

SYNOPSIS

filepriv [–f *priv***[,...]] [–i** *priv***[,...]]** *file* **...**
filepriv –d *file* **...**

DESCRIPTION

The **filepriv** command is used to set, delete, or display the privilege information associated with a file.

The following options are available:

–d deletes the privileges associated with the named file.

–f specifies the fixed privileges associated with the named file.

–i specifies the inheritable privileges associated with the named file.

The **filepriv** command must have the appropriate privileges when setting or deleting file privileges; otherwise, permission is denied. The argument *priv* is defined as a process privilege name. The *file* argument must be an absolute pathname of an executable file when setting or deleting file privileges. There must be at least one *file* argument specified otherwise **filepriv** exits with an error.

When setting file privileges all fixed and inheritable privileges on the specified file are removed before those privileges specified by the **–f** and **–i** options are applied. The **filepriv** command exits with an error if the **–f** and **–i** options are specified and the same privilege exists in both.

When no options are specified, **filepriv** displays the privileges associated with the named file(s).

RETURN VALUE

filepriv exits with a return code of 0 upon successful completion.

USAGE

System administrators.

EXAMPLE

The following is an example of the output when **filepriv** is executed with one file:

fixed *priv*, *priv*, ...
inher *priv*, *priv*, ...

If no fixed privileges exist on the file, the **fixed** privilege line is not displayed. If no inheritable privileges exist on the file, the **inher** privilege line is not displayed. The space between the privilege type and privileges is a single horizontal tab character.

If more than one file is specified, then the file name followed by a colon (**:**) and space character is printed before the privileges as follows:

file1: **fixed** *priv*, *priv*, ...
file1: **inher** *priv*, *priv*, ...
file2: **fixed** *priv*, *priv*, ...
file2: **inher** *priv*, *priv*, ...
file3: **fixed** *priv*, *priv*, ...
file3: **inher** *priv*, *priv*, ...

SEE ALSO

filepriv(BA_OS).

LEVEL

Level 1.

NAME

getacl – display discretionary information for a file or files

SYNOPSIS

getacl [–ad] *file* **...**

DESCRIPTION

For each argument that is a regular file, special file, or named pipe, **getacl** displays the owner, group, and the Access Control List (ACL). For each directory argument, **getacl** displays the owner, group, and the ACL and, optionally, the default ACL. Only directories contain default ACLs.

With the **–a** option specified, the filename, owner, group, and the ACL of the file will be displayed. With the **–d** option specified, the filename, owner, group, and the default ACL of the file, if it exists, will be displayed. With no option specified **getacl** behaves as if both [**–a**] and [**–d**] were specified.

This command may be executed on a file system that does not support ACLs. It will report the ACL consisting of only the owning user, owning group, class and other entries, based on the permission bits.

When multiple files are specified on the command line, a blank line will separate the ACL for each file. The format of an ACL is:

 # file: *filename*
 # owner: *uid*
 # group: *gid*
user::*perm*
user:*uid*:*perm*
group::*perm*
group:*gid*:*perm*
class:*perm*
other:*perm*
default:user::*perm*
default:user:*uid*:*perm*
default:group::*perm*
default:group:*gid*:*perm*
default:class:*perm*
default:other:*perm*

The first three lines show the filename, the file owner, and the file owning group. Note that when only the **–d** option is specified, and the file has no default ACL, only these three lines will be displayed.

The **user** entry without a user ID indicates the permissions that will be granted to the owner of the file. One or more additional **user** entries indicate the permissions that will be granted to the specified users. The **group** entry without a group identifier indicates the permissions that will be granted to the owning group of the file. One or more additional **group** entries indicate the permissions that will be granted to the specified groups. The **other** entry indicates the permissions that will be granted to others.

The default entries (**default:user**, **default:group**, **default:class**, and **default:other**) may only exist for directories, and indicate the default **user**, **group**, and **other** entries that will be added to a file created within the directory.

A *uid* is a login name, or a user ID if there is no entry for the *uid* in the system's password file; *gid* is a group name, or a group ID if there is no entry for the *gid* in the system's group file; and *perm* is a three character string composed of the letters representing the separate discretionary access rights: **r** (read), **w** (write), **x** (execute/search), or the placeholder character **–**. The *perm* will be displayed in the following order: **rwx**. If a permission is not granted by an ACL entry, the placeholder character will appear.

The ACL entries will be displayed in the order in which they will be evaluated when an access check is performed. The default ACL entries which may exist on a directory have no effect on access checks.

The file owner permission bits represent the access that the owning user ACL entry has. The file group class permission bits constrain the ACL (represent the most access that any entry in the ACL may have). If a user executes the **chmod** command and changes the file group class permission bits, this may change the permissions that would be granted based on the ACL alone. This behavior is necessary for the save/restore model (all permissions are temporarily removed via **chmod 000** file and then restored) to work correctly. The file other permission bits represent the access that the other ACL entry has. If a user invokes the **chmod** command and changes the file group class permission bits, the access granted by the additional ACL entries may be restricted.

In order to indicate that the file group class permission bits restrict an ACL entry, **getacl** will display, on the same line (after each affected entry) text in the form **#effective:***perm*, where *perm* will show only the permissions actually granted.

The output from **getacl** will be in the correct format for input to the **setacl** command. If the output from **getacl** is redirected to a file, the file may be used as input to **setacl**. In this way, a user may easily assign one file's ACL to another file.

FILES

/etc/passwd for user IDs

/etc/group for group IDs

USAGE

System administrator.

EXAMPLE

Given file **filea**, with an ACL five entries long, the command

```
$ getacl filea
```

could print:

```
# file: filea
# owner: fletcher
# group: us
user::rwx
user:spy:---
user:archer:rw-
group::r--
```

```
class:rw-
other:---
```

After the command **chmod 700 filea** was issued on the same file the command

```
$ getacl filea
```

could print:

```
# file: filea
# owner: fletcher
# group: us
user::rwx
user:spy:---
user:archer:rw-       #effective:---
group::r--      #effective:---
class:---
other:---
```

Given directory **fileb**, with an ACL containing default entries, the command

```
$ getacl -d fileb
```

could print:

```
# file: fileb
# owner: fletcher
# group: us
default:user::rwx
default:user:spy:---
default:group::r--
default:other:---
```

Given directory **fileb**, the command

```
$ getacl fileb
```

would print:

```
# file: fileb
# owner: fletcher
# group: us
user::rwx
user:spy:---
user:archer:rw-
group::r--
other:---
default:user::rwx
default:user:spy:---
default:group::r--
default:other:---
```

SEE ALSO

acl(ES_LIB), aclsort(ES_LIB), chmod(BU_CMD), ls(BU_CMD), setacl(ES_CMD).

LEVEL

Level 1.

NAME

getdev – lists devices defined in the Device Database based on criteria

SYNOPSIS

getdev [**-ae**] [*criteria* . . .] [*device* . . .]

DESCRIPTION

getdev generates a list of devices that match certain criteria. The criteria include a list of attributes (given in expressions) and a list of devices. If no criteria are given, all devices are included in the generated list.

Devices must satisfy at least one of the criteria in the list unless the **-a** option is used. Then only those devices that match all of the criteria in a list will be included in the generated list.

Devices named on the command line and that match the criteria are included in the generated list. However, if the **-e** flag is used, the list of devices named on the command line becomes the set of devices to be excluded from the list.

Criteria Expression Types

There are four possible expression forms which the criteria specified in the *criteria* argument may follow:

attribute=*value* Selects all devices whose attribute *attribute* is defined and is equal to *value*.

attribute!=*value* Selects all devices whose attribute *attribute* is defined and does not equal *value*.

attribute:* Selects all devices which have the attribute *attribute* defined.

attribute!:* Selects all devices which do not have the attribute *attribute* defined.

See **putdev**(ES_CMD) for a complete listing and description of available attributes.

Options and Arguments

The options and arguments for this command are:

-a Specifies that the list of devices that follows on the command line must match all criteria to be included in the list generated by this command. The flag has no effect if no criteria are defined.

-e Specifies that the list of devices which follows on the command line should be excluded from the list generated by this command. The flag has no effect if no devices are defined.

criteria Defines criteria that a device must match to be included in the generated list. Should be given in expressions.

device Defines devices that should be included or excluded (based on the command options) in the generated list. Can be the pathname of the device or the device alias.

RETURN VALUES

Upon successful completion, **getdev** returns a value of **0**. Otherwise, it returns a non-zero value.

FILES

/etc/device.tab
/etc/security/ddb/ddb_dsfmap
/etc/security/ddb/ddb_sec referenced only if the Enhanced Security Extension is implemented

SEE ALSO

devattr(ES_CMD), putdev(ES_CMD).

LEVEL

Level 1.

NAME

lvldelete – delete Mandatory Access Control (MAC) levels

SYNOPSIS

lvldelete [**-r**] **–a** *alias_name***[,***alias_name***...]**
lvldelete [**-r**] **–c** *cat_id***[,***cat_id***...]**
lvldelete [**-r**] **–f** *level_name*
lvldelete [**-r**] **–h** *class_id***[,***class_id***...]**
lvldelete [**-r**] **–l** *lid***[,***lid***...]**

DESCRIPTION

The **lvldelete** command will delete (unname) a level, hierarchical classification, non-hierarchical category, or alias name from the system. This command is restricted to use by an administrator.

The following options are recognized:

–a Delete the alias *alias_name*.

–c Delete the category indicated by *cat_id*. *cat_id* may be the category number or the category name.

–f Delete the level whose fully qualified level name is *level_name* (that is, of the format *h_name*[:*c_name*,*c_name***...**], where *h_name* is a classification name and *c_name* a category name).

–h Delete the classification indicated by *class_id*. *class_id* may be the classification number or the classification name.

–l Delete the level whose numeric level identifier (LID) is *lid*.

–r Override restriction on deletion of reserved identifiers. Reserved identifiers are described in **lvlname**(ES_CMD).

Options that allow multiple entries to be deleted at a time should not contain duplicates. Furthermore, entries to be deleted must have been previously named using the **lvlname** command. If an entry on the input line is in error, an error message is produced, the option-argument containing the entry in error is skipped, and processing is resumed with the next option-argument (if any).

A level is deleted using the **–l** or **–f** option. In addition to unnaming the LID or fully qualified level tuple, **lvldelete** also deletes the alias name assigned to the removed level. Note, however, that deleting an alias name using the **–a** option does not delete the level itself. Once a LID or fully qualified level tuple has been deleted, the LID cannot be re-assigned. The fully qualified level name, however, can be assigned a new LID.

Any identifier may be deleted regardless of its current state. The deletion of an alias, classification, or category is not an atomic operation. The effect of the delete is realized when the level (for classification/category) or alias is validated against the system's level translation database. Furthermore, that deleting a classification or category does not automatically delete levels containing the deleted classification or category. It is the administrator's responsibility to delete identifiers in a quiescent state and to delete all dependent identifiers. It is strongly recommended that this command be used in maintenance mode only.

A level is undefined on the system if it was never assigned, it has been deleted, or its classification or any of its categories has been deleted. See **lvlname**(ES_CMD) for details on the various states of a level.

When a level, alias, classification, or category is deleted, an entry will be added to the history log maintained by **lvlname**. The history log entry will contain the deleted identifier and a time stamp. The history log may be printed using the **–p** option of the **lvlname** command.

FILES

/etc/security/mac/lid.internal	fully qualified levels
/etc/security/mac/ltf.alias	alias names
/etc/security/mac/ltf.cat	category names
/etc/security/mac/ltf.class	classification names
/etc/security/mac/hist.*	history files

EXAMPLE

In the following example, classification number 2 is deleted, as are categories 1 and 3. Any user attempting to login at a level containing classification 2 or categories 1 or 3 will be denied access to the system.

```
lvldelete -h 2 -c 1,3
```

In the next example, the alias name, OBSERVE, is deleted. Any user attempting to login using the alias will be denied access to the system.

```
lvldelete -a OBSERVE
```

SEE ALSO

lvlname(ES_CMD).

LEVEL

Level 1.

NAME

lvlname – assign or display Mandatory Access Control (MAC) levels

SYNOPSIS

lvlname [**–r**] [**–a** *alias_name*::*level_name*]
lvlname [**–r**] [**–l** [*level_identifier*::]*level_name*]
lvlname [**–r**] [**–h** *class_no*:*h_name*[,*class_no*:*h_name*...]]
[**–c** *cat_no*:*c_name*[,*cat_no*:*c_name*...]]
lvlname [**–p**]

DESCRIPTION

The **lvlname** command will assign a level, classification, category, or alias name, or display the system's current level definitions, or history log. This command is restricted to use by an administrator.

The command takes the following options:

–a Assign *alias_name* to the fully qualified level, *level_name*.

–c Assign *c_name* to the non-hierarchical category number, *cat_no*.

–h Assign *h_name* to the hierarchical classification number, *class_no*.

–l Assign *level_identifier* (LID) to the fully qualified level, *level_name*.

–p Print the contents of the history log.

–r Override restriction on assignment of reserved identifiers. The following restrictions apply:

h_name, *c_name*, and *alias_name*
may not contain embedded white spaces (that is, tabs, newline sequences or spaces); may not contain control characters, aside from those required for support of the ISO standard international character set and any supplementary code-sets defined by the SVID; may not contain colons, semicolons, commas, plus or minus symbols; may not begin with a number; no more than 30 characters long; and must be unique across a shared name space (for example, a classification cannot have the same name as a category or an alias). Furthermore, the name may not be **ALL**. The name **ALL** is a short hand definition representing all categories on the system.

level_identifier
must be a unique unsigned integer value within the range [1,**MAXLIDS**]. **MAXLIDS** is defined in the system header file **mac.h**.

Some identifiers have been reserved for system use. Assignment of these identifiers are not allowed unless overridden using the **–r** option. All names beginning with the prefix **USER_** or **SYS_,** level identifiers in the range (1...99), classifications in the range (1...5) and (251...256), and categories in the range (1...31) are reserved.

If the **-h** or **-c** option is given, the specified classification or category number(s) must fall within the range of supported numbers for classifications or categories, respectively. Additionally, the classification or category number must be unnamed. A classification or category number repeated on the input line can be viewed as a classification or category that has previously been named. If a classification or category number is out of range or previously named, an error message is printed, the option-argument containing the entry in error is skipped, and processing is resumed with the next option-argument (if any).

The **-l** option is used to assign a new level; that is, a fully qualified level name is assigned a LID, which is the system's sole means of level identification. The classifications and categories must have been previously named, and *level_name* must be unique (that is, it cannot already be assigned to another LID). When invoked with the **-l** option and without a specified *level_identifier*, **lvlname** will automatically assign a LID to the *level_name* on input. The LID assigned automatically to a level is the LID just after the highest assigned LID on the system so far. The **lvlname** command allows for the system assigned LID to be explicitly overridden on input. This ability allows multiple systems to use the same LIDs for the same level names. It is recommended that this option be used with discretion, since "gaps" in the LID sequence may occur. For example, if the next automatically assigned LID is 1027 and the user overrides the system assigned LID with 2052, the next automatically assigned LID will be 2053. The numbers 1027 through 2051, are in essence skipped by the system during automatic LID assignment although they could be manually assigned.

When *level_identifier* is specified, the LID must be in the *invalid* state. A LID is in the *invalid* state if the LID has never been assigned to a level. When a LID is assigned to a level, the LID's state becomes *valid-active*. A LID in the *valid-active* state is valid for both **login** and mandatory access control (MAC) checks. When a LID is deleted, the LID transitions to the *valid-inactive* state. A LID in the *valid-inactive* state is valid for MAC checks but is no longer valid for **login**.

A level is undefined on the system if it has never been assigned, it has been deleted, or its classification or any of its categories has been deleted.

The **-a** option is used to assign an alias name to a level. The *level_name* must have previously been assigned a LID to have an alias assigned. In addition, *level_name* may not already have an alias assigned.

All successful assignments through **lvlname** add an entry in the history log. When invoked with the **-p** option, **lvlname** prints the history log in the following format and order:

> Level Identifiers (LIDs):
> *operation level_identifier*`::`*level_name a_date*
>
> Classifications:
> *operation class_no*`:`*h_name a_date*
>
> Categories:
> *operation cat_no*`:`*c_name a_date*

Alias Names:
operation alias_name`::`*level_name a_date*

where *operation* is **ADD** or **DEL** (delete) indicating the nature of the history log entry, and *a_date* is the date (in setlocale format) the operation took place. Note that the **lvldelete** command is used for the **DEL** operation.

The LIDs, classifications, and categories are listed in ascending order, with the LID or classification/category number used as the sort key. Multiple entries are sorted by ascending date within the number. Alias names are printed in ascending alphabetical order.

When **lvlname** is invoked without options, all the system's current level definitions are printed in the following format and order:

Levels:
level_identifier`::`[*alias_name*`::`]*level_name* [*****]

Classifications:
class_no`:`*h_name*

Categories:
cat_no`:`*c_name*

Levels are listed in ascending order by LID number; classifications are listed in ascending order by classification number; and categories are listed in ascending order by category number. Both *valid-active* and *valid-inactive* levels are displayed; *valid-inactive* levels have a ***** appended to the fully qualified level name. Unnamed classifications and categories do not appear on output.

FILES

/etc/security/mac/hist.*	history log
/etc/security/mac/lid.internal	fully qualified levels
/etc/security/mac/ltf.alias	alias names
/etc/security/mac/ltf.cat	category names
/etc/security/mac/ltf.class	classification names

USAGE

Administrator.

EXAMPLE

Suppose the following state initially:

```
$ lvlname

Levels:
100::Analias::Not_so_secret:group_43
101::Top_secret*

Classifications:
1:Not_so_secret
15:Top_secret
```

Categories:
1:syseng
43:group_43

and the history log is empty. Then, the operations:

```
$ lvlname –h 5:A_bit_secret -c 3:Nato,50:ProjectX
$ lvlname –l 125::Top_secret:Nato,ProjectX
$ lvlname –a NoJoke::Top_secret:Nato,ProjectX
```

will produce the following history log:

$ **lvlname –p**

Level Identifiers (LIDs):
ADD 125::Top_secret:Nato,ProjectX Jan 10 12:01:52 EST 1989

Classifications:
ADD 5:A_bit_secret Jan 10 12:01:10 EST 1989

Categories:
ADD 3:Nato Jan 10 12:01:10 EST 1989
ADD 50:ProjectX Jan 10 12:01:10 EST 1989

Alias Names:
ADD Nojoke::Top_secret:Nato,ProjectX Jan 10 12:02:10 EST 1989

and the following state:

$ **lvlname**

Levels:
100::Analias::Not_so_secret:group_43
101::Top_secret*
125::Nojoke::Top_secret:Nato,ProjectX

Classifications:
1:Not_so_secret
5:A_bit_secret
15:Top_secret

Categories:
1:syseng
3:Nato
43:group_43
50:ProjectX

SEE ALSO

lvldelete(ES_CMD).

LEVEL

Level 1.

NAME

lvlprt – print system's current level definitions

SYNOPSIS

lvlprt [**–s**]

DESCRIPTION

The **lvlprt** command prints the system's current level identification, including fully qualified level names, alias names, classifications, and categories. Only *valid-active* levels, named classifications and categories are displayed. The **–s** option suppresses the printing of classifications and categories.

The format and order of the output are as follows:

Levels:
[*alias_name*::]*level_name*

Classifications:
class_no:*h_name*

Categories:
cat_no:*c_name*

Levels are listed in ascending alphabetical order using *level_name* as the key. When defined, alias names are printed before the fully qualified level names. Classifications are listed in ascending order by classification number, and categories are listed in ascending order by category number.

USAGE

General.

EXAMPLE

$ **lvlprt**

```
Levels:
Zalias::Not_so_secret:group_43
Top_secret

Classifications:
1:Not_so_secret
15:Top_secret

Categories:
1:syseng
43:group_43
```

SEE ALSO

lvlname(ES_CMD).

LEVEL

Level 1.

NAME

mailcheck – check for mail at all security levels

SYNOPSIS

mailcheck [**–Z**]

DESCRIPTION

mailcheck checks for the existence of mail. Whenever it finds some, it prints a message on standard output:

You have mail

If the Enhanced Security Extension is implemented, **mailcheck** loops through the dominated security levels looking for mail. For example, if you were logged in at *TopSecret* , you might see the message:

You have mail at level: Top Secret
You have mail at level: Unclassified

However, if you were logged in at *Unclassified* , you would only see the message:

You have mail at level: Unclassified

If there is no mail, it prints on standard error

No mail

By default, when the Enhanced Security Extension is implemented, **mailcheck** prints the level alias of the fully qualified levels dominated by the level at which the user is currently logged in. The **–Z** option forces **mailcheck** to print the fully qualified level instead of the alias. The **–Z** option is valid only when the Enhanced Security Extension is installed.

mailcheck is commonly used in a person's **$HOME/.profile** as follows:

mailcheck 2>/dev/null

This prints a message when there is mail, and is otherwise silent.

RETURN VALUE

0 mail exists at some level

1 no mail at any checked level

2 some error occurred

FILES

/var/mail mail directory

SEE ALSO

mail(BU_CMD).

LEVEL

Level 1.

NAME

mldmode – change MLD mode or execute a command in a given MLD mode

SYNOPSIS

mldmode
mldmode –r [*string*]
mldmode –v [*string*]

DESCRIPTION

With no options, **mldmode** reports the current Multi-Level Directory (MLD) mode (virtual or real). That is, it reports the MLD mode of the invoking process.

–r [*string*] If **–r** alone is specified, the MLD mode of the interactive shell is changed to real mode.

If a *string* specifying a command line follows the **–r**, that command line alone is executed in real mode. (The actual directory structure of any MLDs encountered will not be hidden from the command.)

–v [*string*] If **–v** alone is specified, the MLD mode of the interactive shell is changed to virtual mode.

If a *string* specifying a command line follows the **–v**, that command line alone is executed in virtual mode. (If an MLD is encountered, the command will see only the corresponding effective directory at the level of the invoking process.)

RETURN VALUE

Upon successful completion, the **mldmode** command returns a value of **0**; otherwise, a diagnostic is printed and a non-zero value is returned.

EXAMPLE

To print the actual directory structure of any directory tree containing an MLD:

```
mldmode -r find . -print
```

SEE ALSO

mkdir(BU_CMD), sh(BU_CMD), mkmld(ES_LIB).

LEVEL

Level 1.

NAME

putdev – creates and updates the device database

SYNOPSIS

putdev –a *alias* [*secdev*=*value*] [*attribute*=*value* ...]
putdev –m *device attribute*=*value* [*attribute*=*value* ...]
putdev –d *device* [*attribute* ...]
putdev –p *device attribute*=*value*[,*value* ...]
putdev –r *device attribute*=*value*[,*value* ...]

DESCRIPTION

The **putdev** command is used to add a new *device* to the Device Database (DDB), modify an existing device's *attributes*, or remove a *device* entry from the DDB. It also allows appending new *values* to *attributes* that take value-lists (separated by commas), and removal of specific *values* from value-lists.

The options for the **putdev** command are:

-a Adds a *device* to the DDB using the specified *attributes*. The *device* must be referenced by its *alias*.

-m Modifies a *device* entry in the DDB. If a specified attribute does not exist in the device entry, **putdev** adds the specified *attribute* to the entry. It also modifies *attributes* that already have a value with the *value* specified.

-d Removes a *device* entry from the DDB, when executed without the *attributes* argument. If the *attribute* argument is specified, the *attribute* and its value are deleted from the device entry.

-p Appends the list of values to the *attribute* value-list of the *device*. If the *value* item is multiply defined in the input value-list or already defined in the DDB, the command fails and prints an error message.

-r Removes the list of values from the *attribute* value-list, of the *device*.

alias must be unique throughout the DDB. *alias* is limited to 64 characters (**DDB_MAXALIAS**) and should contain only alphanumeric characters and any of the following special characters: **.** (period), _ (underscore), **$** (dollar sign), and **-** (hyphen).

secdev designates the alias of the secure device that defines all the security attributes. If *secdev* is not specified during creation (**-a** option) or is deleted (**-d** option), the current *alias* is used as the default value of *secdev*. The validation rules for *secdev* are the same as those for *alias*.

device designates the absolute pathname or alias name of the device whose attribute is to be added, modified, or removed. If *device* is a pathname, then the attributes of the alias to which it maps are updated.

attribute designates a device attribute to be added, modified, or deleted. This prevents an accidental modification or deletion of a device's alias from the DDB.

value designates the value to be assigned to a device's attribute. If any of the values are invalid, then the command fails and prints an error message.

Attributes:

The following list shows all of the attributes that can be defined for a device:

alias A unique name by which a device is known. No two devices in the database may share the same alias name. The name is limited in length to 64 characters (**DDB_MAXALIAS**) and should contain only alphanumeric characters and also the following special characters underscore (_), dollar sign ($), hyphen (-), and period (.).

bdevice The absolute pathname to the block special device node associated with the device, if any, with maximum length of **PATH_MAX**.

bdevlist It contains a list of additional pathnames of block device special files mapping to the same logical or secure device. Each item in the list is separated by a comma, and each must be an absolute pathname of the device special file, with a maximum length of **PATH_MAX**. Since this attribute takes a list of values, options **-p and -r** can be used for this attribute.

capacity The capacity of the device or of the typical volume, if removable.

cdevice The absolute pathname to the character special device node associated with the device, if any, with maximum length of **PATH_MAX**.

cdevlist It contains a list of additional pathnames of character device special files mapping to the same logical or secure device. Each item in the list is separated by a comma, and each must be an absolute pathname of the device special file, with a maximum length of **PATH_MAX**. Since, this attribute takes a list of values, options **-p and -r** can be used for this attribute.

cyl Used by the command specified in the **mkfscmd** attribute.

desc A description of any instance of a volume associated with this device (such as floppy diskette).

dpartlist The list of disk partitions associated with this device. Used only if **type=disk**. The list should contain device aliases, each of which must have **type=dpart**.

dparttype The type of disk partition represented by this device. Used only if **type=dpart**. It should be either **fs** (for filesystem) or **dp** (for data partition).

erasecmd The command string that, when executed, erases the device.

fmtcmd The command string that, when executed, formats the device.

fsname The filesystem name on the file system administered on this partition, as supplied to the **labelit** command. This attribute is specified only if **type=dpart** and **dparttype=fs**.

gap Used by the command specified in the **mkfscmd** attribute.

mkfscmd The command string that, when executed, places a file system on a previously formatted device.

mountpt The default mount point to use for the device. Used only if the device is mountable. For disk partitions where **type=dpart** and **dparttype=fs**, this attribute should specify the location where the partition is normally mounted.

nblocks The number of blocks in the filesystem administered on this partition. Used only if **type=dpart** and **dparttype=fs**.

ninodes The number of inodes in the filesystem administered on this partition. Used only if **type=dpart** and **dparttype=fs**.

norewind The name of the character special device node that allows access to the serial device without rewinding when the device is closed.

pathname Defines the pathname to an i-node describing the device (used for non-block or character device pathnames, such as directories).

type A token that represents inherent qualities of the device. Standard types include: **9-track, ctape, disk, directory, diskette, dpart, and qtape**.

volname The volume name on the filesystem administered on this partition, as supplied to the **labelit** command. Used only if **type=dpart** and **dparttype=fs**.

volume A text string used to describe any instance of a volume associated with this device. This attribute should not be defined for devices which are not removable.

Security Attributes:

The following list of security attributes could be defined for a device alias, if the Enhanced Security Extension is implemented.

secdev the alias name of the physical device or *secure device*, and is unique throughout the Device Database(DDB). This alias name is limited to 64 characters (**DDB_MAXALIAS**), and should contain only alphanumeric characters and the special characters "_", "$", "-" or ".". For a secure device alias this attribute's value is the same as the device's alias. For a logical device alias, this attribute's value is different from the device alias. By default, **secdev** is defined to be equal to the device's alias.

range the sensitivity Mandatory Access Control (MAC) level range of the device. It should by a *hilevel-lolevel* pair, where *hilevel* and *lolevel* are both MAC level names or fully qualified levels. The "-" character is the delimiter between *hilevel* and *lolevel*. These levels are stored in the DDB as LIDs, converted to ASCII characters. The LIDs are validated against the *Label Translation Database*, and *hilevel* is checked to verify that it dominates *lolevel*, before they are saved in the DDB. This attribute must be defined.

state determines whether the device is to be used as a private or public device. It can take any one of **private**, **public**, or **pub_priv**. If it is set to **pub_priv**, then the device can either be used as private or public device. If the **startup** attribute is enabled, then the device is allocated as **private**, when the **state** is set to either **private** or **pub_priv**. This attribute must be defined.

mode determines the mode of the device. This attribute can either be *static* or *dynamic*. This attribute must be defined.

startup is a flag (**y[es]/n[o]**) that indicates whether the device is allocated during startup or not. This attribute is optional, and startup default value is **no**.

startup_level defines the MAC level at which the device should be set at startup. This can be specified as a level name or fully qualified level. However, the value is saved in the DDB as an ASCII LID value. This attribute is optional.

startup_owner defines the owner of the device. The value of **startup_owner** can be specified as the UID or user name followed by the access permissions. The value must be specified in the format *uid*>*rwx*. If any of the read, write, or execute access is denied, that field must contain a "-". The ">" character serves as delimiter between the UID or user name and the access permissions. The uid or user name must be defined on the system (in **/etc/passwd**), at the time this attribute is defined. This attribute is optional but must be defined if attribute **startup** is set to **yes**.

startup_group defines the group to which the device belongs. The value of **startup_group** can be specified as the GID or group name followed by the access permissions. The value must be specified in the format *gid*>*rwx*. If any of the read, write or execute access is denied, that field must contain a "-". The ">" character serves as delimiter between the GID or group name and the access permissions. The gid or group name must be defined on the system (in **/etc/group**), at the time this attribute is defined. This attribute is optional but must be defined if attribute **startup** is set to **yes**.

startup_other defines the access permissions for **other**. The value of **startup_other** must be specified in the format >*rwx*. If any of the read, write or execute access is denied, then that field must contain a "-". This attribute is optional but must be defined if attribute **startup** is set to **yes**.

ual_enable this attribute serves as a flag that enables or disables depending on its value the user authorization list defined in the **users** and **other** attributes. This attribute can either be values: **y[es], n[o]**. If **yes**", then the user authorization list is checked when authorizing an user to use this device. If **no**, then no users are authorized to use this device. This attribute is optional, and value assumed as **no** if **ual_enable** is not defined.

users is the user authorization list that defines the allocation permissions for **users**. Each item is a UID-authorization or username-authorization pair separated by a ">" character. The items in the list are separated by commas. The attribute's value must be specified in the format *uid1*>**n**,*uid2*>**n**,*uid3*>**y**. Each UID or username must be unique in a device entry, and all UIDs or usernames must be defined in **/etc/passwd**, when this attribute is defined. Since, this attribute takes a list of values, options **-p and -r** can be used. This attribute is optional.

other is the other authorization that defines the authorization permissions for **other**. This attribute contains only one item and it can take either **>y[es] or >n[o]**. This attribute is optional, and its value is assumed as **no**, if **other** is not defined.

The following rules and guidelines should be followed when using the **putdev** command.

- The **alias** names of devices must be valid (see description under **Attributes**) and unique throughout the DDB; and will fail if nonunique.
- The pathnames to device special files in attributes *cdevice*, *bdevice*, *cdevlist*, and *bdevlist* must be absolute pathnames. They cannot be repeated within an entry or occur in multiple entries. The **putdev** command checks the uniqueness of pathnames and will fail if nonunique.
- Security attributes can be defined for *device*, or *alias* only if the system is configured for multilevel security; otherwise, the command fails.
- The MAC level values for the security level range (*hilevel-lolevel*) must be valid security level aliases or fully qualified level names defined in the Level Translation Database (LTDB); otherwise, the command fails. If *hilevel* does not dominate *lolevel*, the command fails.

Special handling of the *secdev* attribute:

- The *secdev* attribute is used to define the essential security attributes of a device. This attribute's name must be valid (see description under **Security Attributes**) and unique throughout the DDB; otherwise the command fails.
- By default, when adding a new device alias into the Device Database, if the *secdev* attribute is not defined at the command line, the new device entry is assigned a *secdev* equal to its *alias*.
- The essential security attributes are *range*, *state*, and *mode*.
- The alias that defines security attributes of a device is called a secure device alias. One can define other non-security attributes for this alias, if needed. For all secure devices, by default, *secdev* must have same value as *alias*.
- When adding (using **-a**) or modifying (using **-m**) a device entry and specifying a *secdev* attribute not equal to the *alias* being added or modified, **putdev** performs the following checks in the order specified below:

  1. If the essential security attributes are being defined for *alias*, the command fails and displays an error message. An entry defining the essential security attributes must have the *secdev* attribute be equal to its *alias*.

  2. If the essential security attributes are not being defined for *alias*, and if the specified *secdev* does not exist in the Device Database, a warning message is displayed.

3. If the essential security attributes are not being defined for *alias*, and the specified *secdev* exists in the Device Database but does not define the essential security attributes, the command fails and displays an error message.

4. If the essential security attributes are not being defined for *alias*, and the specified *secdev* exists in the Device Database and defines the essential security attributes, then the command is successful.

- It is recommended that the secure alias be created before any logical aliases are created that map to the same secure alias. Similarly, it is recommended not to remove a secure device alias if any logical alias are currently mapped to that secure alias.
- Additional aliases that share the security attributes defined for a secure device can be created by specifying their *secdev* to have the same value as the *alias* of the secure device. If *secdev* is not specified, and the essential security attributes are also not specified, then a logical device entry is created that does not have security attributes.

Special handling of the essential security attributes:

- The essential security attributes, *mode*, *state*, and *range* must be created (using **-a** or **-m**) and deleted (using **-d**) together. Otherwise, the command fails and issues an error message.
- The essential security attributes of a secure alias can be modified (**-m**) separately after they are defined.
- If the essential security attributes are being deleted from a device entry whose *alias* is a *secdev* attribute for at least another entry in the Device Database, then the command fails and displays an error message.

RETURN VALUE

Upon successful completion, **putdev** returns **0**; otherwise, a diagnostic is printed and a non-zero value is returned.

EXAMPLE

The following example shows you how to create one secure device (**tapedrive1**) and two device aliases (**slowtape**, **fasttape**) that map to the secure device. (In the following example, the input is split onto two lines; you should enter the commands as one line.)

```
putdev -a tapedrive1 range=SYS_PRIVATE-SYS_PUBLIC state=public \
     mode=static startup=n ual_enable=y users="100>n,101>n" other=">y"
```

```
putdev -a slowtape secdev=tapedrive1 cdevice=/dev/tape800
```

```
putdev -a fasttape secdev=tapedrive1 cdevice=/dev/tape1600
```

The preceding command sequence creates one secure device alias (**tapedrive1**) with the specified security attributes for the tape drive, and two logical device aliases (**slowtape** and **fasttape**) with the specified non-security attributes in the DDB.

However, one could create one entry per device with all security attributes specified on the command line:

```
putdev -a tape1 range=SYS_PRIVATE_SYS_PUBLIC state=public mode=static \
     startup=n ual_enable=y users="100>n,101>n" other=">y" \
     cdevlist=/dev/tape800,/dev/tape1600 desc=tape device
```

The DDB can be queried for any alias, or attribute value using the **devattr** and **getdev** commands.

FILES

/etc/device.tab
/etc/security/ddb/ddb_dsfmap
/etc/security/ddb/ddb_sec exists only if the Enhanced Security Extension is implemented

SEE ALSO

admalloc(ES_CMD), devattr(ES_CMD), devstat(ES_CMD), getdev(ES_CMD), devstat(ES_LIB).

LEVEL

Level 1.

NAME

setacl – modify the Access Control List (ACL) for a file or files

SYNOPSIS

setacl [**–r**] **–s** *acl_entries file*...
setacl [**–r**] [**–m** *acl_entries*] [**–d** *acl_entries*] *file*...
setacl [**–r**] **–f** *acl_file file*...

DESCRIPTION

For each *file* specified, **setacl** will either replace its entire ACL, including the default ACL on a directory, or it will add, modify, or delete one or more ACL entries, including default entries on directories.

The **–s** option will set the ACL to the entries specified on the command line. The **–f** option will set the ACL to the entries contained within the file *acl_file*. The **–d** option will delete one or more specified entries from the file's ACL. The **–m** option will add or modify one or more specified ACL entries.

At least one of the options **–s**, **–m**, **–d**, or **–f** must be specified. If **–s** or **–f** are specified, other options are invalid. The **–m** and **–d** options may be combined.

For the **–m** and **–s** options, *acl_entries* are one or more comma-separated ACL entries selected from the following list. For the **–f** option, *acl_file* must contain ACL entries, one to a line, selected from the same list. Default entries may only be specified for directories. "Typewritten" font indicates that characters must be typed as specified, brackets denote optional characters, and italicized characters are to be specified by the user.

u[**ser**]::*operm* | *perm*
u[**ser**]:*uid*:*operm* | *perm*
g[**roup**]::*operm* | *perm*
g[**roup**]:*gid*:*operm* | *perm*
c[**lass**]:*operm* | *perm*
o[**ther**]:*operm* | *perm*
d[**efault**]:**u**[**ser**]::*operm* | *perm*
d[**efault**]:**u**[**ser**]:*uid*:*operm* | *perm*
d[**efault**]:**g**[**roup**]::*operm* | *perm*
d[**efault**]:**g**[**roup**]:*gid*:*operm* | *perm*
d[**efault**]:**c**[**lass**]:*operm* | *perm*
d[**efault**]:**o**[**ther**]:*operm* | *perm*

For the **–d** option, *acl_entries* are one or more comma-separated ACL entries without permissions, selected from the following list. Note that the entries for file owner, owning group, and others may not be deleted.

u[**ser**]:*uid*
g[**roup**]:*gid*
d[**efault**]:**u**[**ser**]:
d[**efault**]:**u**[**ser**]:*uid*
d[**efault**]:**g**[**roup**]:
d[**efault**]:**g**[**roup**]:*gid*
d[**efault**]:**c**[**lass**]:
d[**efault**]:**o**[**ther**]:

In the above lists, the user specifies the following:

perm is a permissions string composed of the characters **r** (read), **w** (write), and **x** (execute), each of which may appear at most one time, in any order. The character **–** may be specified as a placeholder.

operm is the octal representation of the above permissions, with **7** representing all permissions, or **rwx**, and **0** representing no permissions.

uid is a login name or user ID.

gid is a group name or group ID.

The options have the following meanings:

–r Recalculate the group class entry so as to ensure that permissions granted in the additional ACL entries will actually be granted. If the **–r** option is specified, the value specified in the **class** entry is ignored.

–s Set a file's ACL. All old ACL entries are removed, and replaced with the newly specified ACL. There must be exactly one **user** entry specified for the owner of the file, exactly one **group** entry specified for the owning group of the file, exactly one **class** entry specified for the file group class, and exactly one **other** entry specified. There may be additional **user** ACL entries and additional **group** ACL entries specified, but there may not be duplicate additional **user** ACL entries with the same *uid*, or duplicate additional **group** ACL entries with the same *gid*. If the file is a directory, default ACL entries may be specified. There may be at most one default **user** entry for the owner of the file, at most one default **group** entry for the owning group of the file, at most one default **class** entry for the file group class, and at most one default **other** entry for other. There may be additional default **user** entries and additional default **group** entries specified, but there may not be duplicate additional default **user** entries with the same *uid*, or duplicate additional default **group** entries with the same *gid*. An entry with no permissions will result in the specified *uid* or *gid* being denied access to the file. The entries need not be in order. They will be sorted by the command before being applied to the file.

–m Add one or more new ACL entries to the file, and/or change one or more existing ACL entries on the file. If an entry already exists for a specified *uid* or *gid* the specified permissions will replace the current permissions. If an entry does not exist for the specified *uid* or *gid*, an entry will be created.

–d Delete one or more existing ACL entries from the file. The entries for the file owner, the owning group, and others may not be deleted from the ACL. Note that deleting an entry does not necessarily have the same effect as removing all permissions from the entry. Specifically, deleting an entry for a specific user would cause that user's permissions to be determined by the **other** entry (or the owning **group** entry, if the user is in that group).

–f Set a file's ACL with the ACL entries contained in the file named *acl_file*. The same constraints on specified entries hold as with the **–s** option. The entries are not required to be in any specific order in the file specified as **acl_file**. The character "#" in *acl_file* may be used to indicate a comment. All characters, starting with the "#", until the end of the line, will be ignored.

Note that if the *acl_file* has been created as the output of the **getacl** command, any effective permissions, which will have been written with a preceding "#", will also be ignored.

When the **setacl** command is used, it may result in changes to the file permission bits. When the **user** ACL entry for the file owner is changed, the file owner permission bits will be modified. When the **other** ACL entry is changed, the file other permission bits will be modified. When additional **user** ACL entries and/or any **group** ACL entries are set or modified, the file group class permission bits will be modified to reflect the maximum permissions allowed by the additional **user** entries and all the **group** entries.

If an ACL does not contain additional **user** and additional **group** entries, the permissions in the **group** entry for the object owning group and the **class** entry must be the same. Therefore, if the **-d** option is specified and results in no additional user entries and no additional group entries, the **class** entry permissions will be set equal to the permissions of the owning group entry. (Note, this is equivalent to using the **-r** option.)

A directory may contain default ACL entries. If a file is created in a directory which contains default ACL entries, the entries will be added to the newly created file. Note that the default permissions specified for the file owner, file owning group, and others, will be constrained by the **umask** and the mode specified in the file creation call.

If an ACL does not contain additional **default:user** and additional **default:group** entries and a **default:group** entry is specified for the object owning group, a **default:class** entry must also be specified, and the permissions in the **default:group** entry for the object owning group and the permissions for the **default:class** entry must be the same.

This command may be executed on a file system that does not support ACLs, to set the permissions for the three base entries for the file owner, file owning group, and others. Additional entries and default entries will not be allowed in this case.

FILES

/etc/passwd for user IDs

/etc/group for group IDs

EXAMPLE

To add one ACL entry to file **filea**, giving user **archer** read permission only, type:

```
setacl –m user:archer:r–– filea
```

If an entry for user **archer** already exists, this command will set the permissions in that entry to **r––**.

To replace the entire ACL for file **filea**, adding entries for users **archer**, and **fletcher**, allowing read/write access, an entry for the file owner allowing all access, an entry for the file group allowing read access only, and an entry for others disallowing all access, type:

```
setacl –r –s user::rwx,user:archer:rw–,user:fletcher:rw–,\
  group::r––,other:––– filea
```

Note that this command would set the file permission bits to **-rwxrw----**. Even though the file owning group has only read permission, the maximum permissions available to all additional **user** ACL entries and all **group** ACL entries are read and write, since the two additional **user** entries both specify these permissions.

To set the same ACL on file **filea** as in the above example, using the **-f** option, type:

```
setacl -r -f filea.acl filea
```

with file **filea.acl** edited to contain:

```
user::rwx
user:archer:rw-
user:fletcher:rw-
group::r--
other:---
```

Because the **-r** option was specified, no **class** entry was needed. If a **class** entry had been present it would have been ignored.

SEE ALSO

acl(ES_LIB), aclsort(ES_LIB), chmod(BU_CMD), getacl(ES_CMD), ls(BU_CMD).

LEVEL

Level 1.

NAME

tcpio – trusted cpio for copying file archives in and out

SYNOPSIS

tcpio –o [**aLvV**] [**–C** *bufsize*] **–O** *file* [**–M** *message*]

tcpio –i [**bdfkPrsStuvVx**] [**–C** *bufsize*] [**–E** *file*] **–I** *file* [**–M** *message*]
[**–R** *ID*]] [**-N** *level*] [**–T** *file*] [**–X** *low_level,high_level*] [**–n***num*] [*pattern ...*]

DESCRIPTION

The **–i** and **–o** options select the action to be performed. The following list describes each of the actions (which are mutually exclusive).

tcpio –o

(copy out) reads the standard input to obtain a list of path names and copies those files, together with path name and status information, onto the file or device specified with the **-O** option. Output is padded to a 512-byte boundary by default. The data is preceded by MAC and DAC security-related information saved to enable validation of the data when it is read back in.

tcpio –i

(copy in) extracts files from the archive file or device specified by the **-I** option, which is assumed to be the product of a previous **tcpio –o**. Only files with names that match *patterns* are selected. *patterns* are regular expressions given in the filename-generating notation of **sh**(BU_CMD). In *patterns*, meta-characters **?**, *, and **[** . . . **]** match the slash (**/**) character, and backslash (****) is an escape character. A **!** meta-character means *not*. (For example, the **!abc*** pattern would exclude all files that begin with **abc**.) Multiple *patterns* may be specified and if no *patterns* are specified, the default for *patterns* is * (i.e., select all files). When **tcpio** is invoked from the shell, each *pattern* should be quoted; otherwise the pattern may be expanded.

Extracted files are conditionally created based upon the options described below.

Before a file is extracted, the user, group, classification, category and level identifiers (IDs) it references are validated. If any of the identifiers has been deleted from the system, or changed in any way (and is not remapped to a valid identifier), the file will not be extracted.

The permissions of the files will be those of the previous **tcpio –o**. The owner and group of the files will be that of the current user unless the user has appropriate privilege, which causes **tcpio** to retain the owner and group of the files of the previous **tcpio –o**.

NOTE: If **tcpio –i** tries to create a file that already exists and the existing file is the same age or newer, **tcpio** will output a warning message and not replace the file. (The **–u** option can be used to unconditionally overwrite the existing file.)

The meanings of the available options are

–a Reset access times of input files after they have been copied.

–b Reverse the order of the bytes within each word.

–C *bufsize*
Input/output is to be blocked *bufsize* bytes to the record, where *bufsize* is replaced by a positive integer. The default buffer size is 512 bytes when this option is not used. (**–C** is meaningful only with data directed to or from a character special device, e.g., **/dev/rmt/0m**.)

–d Directories are to be created as needed.

–E *file*
Specify an input file (*file*) that contains a list of filenames to be extracted from the archive (one filename per line).

–f Copy in all files except those in *patterns.* (See the paragraph on **tcpio –i** for a description of *patterns*.)

–I *file*
Read the contents of *file* as input. If *file* is a character special device, when the first medium is full replace the medium and type a carriage return to continue to the next medium.

–k Attempt to skip corrupted file headers and I/O errors that may be encountered. If you want to copy files from a medium that is corrupted or out of sequence, this option lets you read only those files with good headers. (For **tcpio** archives that contain other **tcpio** archives, if an error is encountered **tcpio** may terminate prematurely. **tcpio** will find the next good header, which may be one for a smaller archive, and terminate when the smaller archive's trailer is encountered.)

–L Follow symbolic links. The default is not to follow symbolic links. If the **–follow** option is used with **find**, the **–L** option should be used to ensure that the file pointed to by the symbolic link is archived rather than the symbolic link itself.

–M *message*
Define a *message* to use when switching media. When you use the **–O** or **–I** options and specify a character special device, you can use this option to define the message that is printed when you reach the end of the medium. One **%d** can be placed in *message* to print the sequence number of the next medium needed to continue.

–n*num*
Disable the validation of one or more identifiers (type or item). The permissible values of *num* are:

1 - disable the comparison of the original system name to the current system
2 - disable all checks of UIDs
3 - disable all checks of GIDs
4 - disable all checks of LID existence
5 - disable all checks of LID state (LIDs must be valid, but can be in the inactive state)

Any combination of the above is legal. For example, **-n2 -n3** would disable the checks of UIDs and GIDs.

-N *level*
Extract all files and assign them the MAC level *level*. A LID, alias or fully qualified level name may be used to specify *level* [see **lvlname**(ES_CMD)]. It must be defined on the system and within the medium level range.

-O *file*
Direct the output of **tcpio** to *file*. If *file* is a character special device, when the first medium is full replace the medium and type a carriage return to continue to the next medium.

-P Peek at an archive and return the level range. Valid only in combination with the **-i** and **-I** options.

-r Interactively rename files. If the user types a null line, the file is skipped. If the user types a ''.'' the original pathname will be copied.

-R *ID* Reassign ownership and group information for each file to user ID, *ID*, and the group ID corresponding to *ID* in **/etc/passwd** (*ID* must be a valid login ID from **/etc/passwd**).

-s Swap bytes within each half word.

-S Swap halfwords within each word.

-t Print a table of contents of the input. No files are created.

-T *file*
Use the contents of *file* to remap invalid identifiers to valid ones. The structure of *file* (called Tcpio Table Of Content Translation Table or TTOCTT) is as follows (the fields separated by whitespace):

Type Specifier	Saved Numeric/Text ID	New Numeric/Text ID

Where:

Type Specifier is one of:

UI - User ID
GI - Group ID
LI - Level ID

Numeric ID is the decimal representation of an ID.
Text ID is the text name of an ID.
Saved designates IDs saved on the medium. New designates IDs valid on the current system.

-u Copy unconditionally (normally, an older file will not replace a newer file with the same name).

-v Verbose: causes a list of file names to be printed. When used with the **-t** option, the table of contents looks like the output of an **ls -l** command [see **ls**(1)].

–V Special Verbose: print a dot for each file seen. Useful to assure the user that **tcpio** is working without printing out all file names.

–x Quiet mode. Suppresses the printing of all warning messages.

–X *low_level,high_level*
Extract only files with MAC level between *low_level* and *high_level*, inclusive. *high_level* must dominate *low_level*. LIDs, aliases, or fully qualified level names may be used to specify **low_level** and **high_level** [see **lvlname**(ES_CMD)]. Only valid names or aliases may be used; a LID may be used even if it has been deleted.

FILES

/etc/passwd
/etc/group
/etc/security/macl/id.internal
/etc/security/mac/hist.lid.del

USAGE

General.

EXAMPLE

The following examples show some possible ways to use **tcpio**.

When standard input is directed through a pipe to **tcpio –o**, it groups the files so they can be directed to a single file (**../newfile**) specified with the **-O** option. Instead of **ls**(BU_CMD), you could use **find**(BU_CMD), **echo**(BU_CMD), **cat**(BU_CMD), etc. to pipe a list of names to **tcpio**. You could direct the output to a device instead of a file.

```
ls | tcpio –o -O ../ newfile
```

tcpio –i uses the output file of **tcpio –o** (specified with the **-I** option in the example), extracts those files that match the patterns (**memo/a1**, **memo/b**∗), creates directories below the current directory as needed (**–d** option), and places the files in the appropriate directories. If no patterns were given, all files from **newfile** would be placed in the directory.

```
tcpio –idI newfile "memo/a1" "memo/b∗"
```

tcpio –iN Secret extracts all files at level *Secret* while the **–X LessSecret,VerySecret** directive imposes a limit on the level of the files extracted. No file with level lower than **LessSecret** or higher than **VerySecret** will be extracted. The **–I** option specifies the input device, in this example **/dev/tape1**. The **–d** options says to create directories as needed. (It is important to use the **–depth** option of **find**(BU_CMD) to generate path names for **tcpio**. This eliminates problems **tcpio** could have trying to create files under read-only directories.)

```
tcpio –idI /dev/tape1 –N Secret –X LessSecret,VerySecret
```

Note that when you use **tcpio** in conjunction with **find**, if you use the **–L** option with **tcpio** then you must use the **–follow** option with **find** and vice versa. Otherwise there will be undesirable results.

SEE ALSO

ar(BU_CMD), **cat**(BU_CMD), **cpio**(BU_CMD), **echo**(BU_CMD), **find**(BU_CMD), **ls**(BU_CMD), **tar**(AU_CMD), **lvlname**(ES_CMD).

LEVEL

Level 1.

NAME

tfadmin – invoke a command, regulating privilege based on the information in the TFM database.

SYNOPSIS

tfadmin [*role*:] *cmd* [*args*]
tfadmin -t [*role*:] *cmd* [:*priv* [:*priv* . . .]]
tfadmin

DESCRIPTION

The **tfadmin** command invokes a command at the request of a user. If the user is allowed to use privileges with the command, **tfadmin** places the allowed privileges in the working privilege set of the process before invoking the command. **tfadmin** uses the entries for the invoking user in the TFM database to determine the pathname for the command *cmd*; if it does not exist in the user definition, **tfadmin** issues an error and exits with an error code. The **tfadmin** command takes the following options:

-t Test whether the user can invoke the given command with the (optionally) given privileges. Do not execute the command.

No options Execute the specified command for the invoking user taking the definition from the *role* argument (if supplied). If the *role* does not exist in that user's role list, print a message and fail.

No arguments Execute the default command for the invoking user. If no default command is defined, print a message and fail.

The arguments to this command are a pair of names separated by a colon representing an optional role name *role* and a command name (not a pathname) *cmd* followed by the command line arguments to be passed to the command on invocation. In addition, if the **-t** option is used, a privilege vector, consisting of one or more privilege names separated by colons (for example, **macread:mount**) may be appended to the role-command pair, separated from it by a colon (for example, **SSA:mount:macread:mount**). This privilege list is only meaningful when the **-t** option is used because it is used to test whether the given command can be executed by the invoking user with the specified privileges.

RETURN VALUE

If the requested operation succeeds, **tfadmin** executes the command, and, therefore, does not exit. The invoked command exits with whatever value is appropriate. If the **-t** option is used and the requested privileges would have been granted to the user invoking the requested command within the requested role, **tfadmin** exits with a **0**. If the operation fails, or the **-t** option was specified and **tfadmin** would have denied the request, **tfadmin** exits with a **1**.

FILES

/etc/security/tfm/users/∗
/etc/security/tfm/users/∗**/default**
/etc/security/tfm/users/∗**/roles**
/etc/security/tfm/users/∗**/cmds/**∗
/etc/security/tfm/roles/∗
/etc/security/tfm/roles/∗**/cmds/**∗

SEE ALSO

adminrole(ES_CMD), adminuser(ES_CMD).

LEVEL

Level 1.

Remote Services Introduction

The Remote Services Extension provides standard interfaces to support networking applications. Support is provided for Remote Procedure Call (RPC), External Data Representation (XDR), Network Selection, Name to Address Translation, and Distributed File Systems.

The following are prerequisite for support of the Remote Services Extension:

- Base System
- Basic Utilities Extension
- Advanced Utilities Extension
- Administered Systems Extension

Summary of Library Routines

The following library routines are supported by the Services extension. Items marked with a dagger (†) are new to this issue of the SVID.

auth_destroy	**clnt_spcreateerror**	**getpublickey**
authdes_getucred	**clnt_sperrno**	**getsecretkey**
authdes_seccreate	**clnt_sperror**	**host2netname**
authnone_create	**clnt_tli_create**	**key_decryptsession**
authsys_create	**clnt_tp_create**	**key_encryptsession**
authsys_create_default	**clnt_vc_create**	**key_gendes**
clnt_call	**cs_connect**	**key_setsecret**
clnt_control	**cs_perror**	**nc_perror**
clnt_create	**endnetconfig**	**nc_sperror†**
clnt_destroy	**endnetpath**	**netdir_free**
clnt_dg_create	**freenetconfigent**	**netdir_getbyaddr**
clnt_freeres	**get_rpc_createerr**	**netdir_getbyname**
clnt_geterr	**get_t_errno**	**netdir_options**
clnt_pcreateerror	**getnetconfig**	**netdir_perror†**
clnt_perrno	**getnetconfigent**	**netdir_sperror†**
clnt_perror	**getnetname**	**netname2host**
clnt_raw_create	**getnetpath**	**netname2user**

rpc_broadcast
rpc_broadcast_exp†
rpc_call
rpc_reg
rpcb_getaddr
rpcb_getmaps
rpcb_gettime
rpcb_rmtcall
rpcb_set
rpcb_unset
setnetconfig
setnetpath
svc_create
svc_destroy
svc_dg_create
svc_fd_create
svc_freeargs
svc_getargs
svc_getreq_common†
svc_getreq_poll†
svc_getreqset
svc_getrpccaller
svc_raw_create
svc_reg
svc_run
svc_sendreply
svc_tli_create
svc_tp_create
svc_unreg
svc_vc_create
svcerr_auth
svcerr_decode
svcerr_noproc
svcerr_noprog
svcerr_progvers
svcerr_systemerr
svcerr_weakauth
taddr2uaddr
uaddr2taddr
user2netname
xdr_accepted_reply
xdr_array
xdr_authsys_parms
xdr_bool
xdr_bytes
xdr_callhdr
xdr_callmsg
xdr_char
xdr_destroy
xdr_double
xdr_enum
xdr_float
xdr_free
xdr_getpos
xdr_inline
xdr_int
xdr_long
xdr_opaque
xdr_opaque_auth
xdr_pointer
xdr_reference
xdr_rejected_reply
xdr_replymsg
xdr_setpos
xdr_short
xdr_string
xdr_u_char
xdr_u_int†
xdr_u_long
xdr_u_short
xdr_union
xdr_vector
xdr_void
xdr_wrapstring
xdrmem_create
xdrrec_create
xdrrec_endofrecord†
xdrrec_eof
xdrrec_skiprecord†
xdrstdio_create
xprt_register
xprt_unregister

Summary of Commands and Utilities

The following commands and utilities are supported by the Remote Services extension.

chkey	**keyserv**	**rpcinfo**
dfmounts	**newkey**	**share**
dfshares	**rpcbind**	**unshare**
keylogin	**rpcgen**	

Organization of Technical Information

The ''Remote Services Library Routines'' chapter provides manual page descriptions of library routines supported by this extension.

Remote Services Definitions

Generic Distributed File Systems Definitions

Client

A host that has mounted resources from another host (a server).

Host

A computer system.

Mount

Make a resource available in the file hierarchy of a host.

Multihop Access

Multihop access refers to the following remote resource scenario: Suppose host A shares a resource that has mounted within it a resource from host B. If any other host mounts the resource from host A and uses it to access a file on the resource from host B, then that access is termed *multihop* access.

Name Space

The set of names that may be given to the objects in a given class, such as files on a computer system or computer systems on a network.

Resource

A file system object, such as a regular file, a directory, or an entire file system.

Server

A host that has shared local resources with a remote host (a client).

Share

Make a local resource available to remote hosts (clients).

RPC Definitions

Program

A program that implements one or more remote procedures. Remote programs are referenced by program number. See remote procedure.

Procedure

Remote procedures are executed by remote programs on behalf of client processes that make remote procedure calls. A server may support multiple versions of a program. Remote procedures are referenced by program number, version number and procedure number.

Version

All remote programs have a version number, used in conjunction with a program number and procedure number to uniquely identify the remote procedure. See remote procedure.

The Network File System Definitions

Export

Share a local resource with remote systems.

Exporting a resource only involves making the resource available to remote systems. No other host is informed of the availability of the resource. In order to mount the resource, a client must give both the name of the server and the pathname of the resource on the server. Only whole file systems or parts of file systems (regular files or directories) may be exported.

Data Structures

AUTH structure

The **AUTH** structure is used by many of the library routines. It is defined in the other header files included by **<rpc/rpc.h>** file.

The **AUTH** structure contains the following members:

```
struct opaque_auth   ah_cred;      /* Credentials */
struct opaque_auth   ah_verf;      /* Verifier */
union  des_block     ah_key;       /* DES key */
struct auth_ops {
       void   (*ah_nextverf)();   /* nextverf */
       int    (*ah_marshal)();    /* serialize */
       int    (*ah_validate)();   /* validate varifier */
       int    (*ah_refresh)();    /* refresh credentials */
       void   (*ah_destroy)();    /* destroy this structure */
} *ah_ops;
caddr_t ah_private;
```

CLIENT structure

The **CLIENT** structure is used by many of the library routines. It is defined in the other header files included by **<rpc/rpc.h>** file.

The **CLIENT** structure contains the following members:

```
AUTH   *cl_auth;                     /* authenticator */
struct clnt_ops {
   enum clnt_stat (*cl_call)();     /* call remote procedure */
   void           (*cl_abort)();    /* abort a call */
   void           (*cl_geterr)();   /* get specific error code */
   bool_t         (*cl_freeres)();  /* frees results */
   void           (*cl_destroy)();  /* destroy this structure */
   bool_t         (*cl_control)();  /* the ioctl() of rpc */
} *cl_ops;
caddr_t           cl_private;       /* private stuff */
char              *cl_netid;        /* network token */
char              *cl_tp;           /* device name */
```

SVCXPRT structure

The **SVCXPRT** structure is used by many of the library routines. It is defined in the other header files included by **<rpc/rpc.h>** file.

The **SVCXPRT** structure contains the following members:

```
int                xp_fd;            /* associated file descriptor */
struct xp_ops {
    bool_t         (*xp_recv)();     /* receive incoming requests */
    enum xprt_stat (*xp_stat)();     /* get transport status */
    bool_t         (*xp_getargs)();  /* get arguments */
    bool_t         (*xp_reply)();    /* send reply */
    bool_t         (*xp_freeargs)(); /* free mem allocated for args */
    void           (*xp_destroy)();  /* destroy this struct */
} *xp_ops;
char               *xp_tp;       /* transport provider device name */
```

```
char                *xp_netid;    /* network token */
struct netbuf       xp_ltaddr;    /* local transport address */
struct netbuf       xp_rtaddr;    /* remote callers address */
struct opaque_auth  xp_verf;      /* raw response verifier */
caddr_t             xp_p1;        /* private: for use by svc ops */
caddr_t             xp_p2;        /* private: for use by svc ops */
caddr_t             xp_p3;        /* private: for use by svc lib */
```

XDR structure

The **XDR** structure is used by many of the library routines. It is defined in the other header files included by **<rpc/rpc.h>** file.

The **XDR** structure, which is used in all **XDR** routines, contains the following members:

```
enum xdr_op x_op;           /* operation */
struct xdr_ops {
    bool_t  (*x_getlong)(); /* get a long from underlying stream */
    bool_t  (*x_putlong)(); /* put a long to underlying stream */
    bool_t  (*x_getbytes)();/* get some bytes from underlying stream */
    bool_t  (*x_putbytes)();/* put some bytes to underlying stream */
    u_int   (*x_getpostn)();/* returns bytes off from beginning */
    bool_t  (*x_setpostn)();/* reposition the stream */
    long *  (*x_inline)();  /* buf quick ptr to buffered data */
    void    (*x_destroy)(); /* free privates of this xdr_stream */
} *x_ops;
caddr_t     x_public;       /* users' data */
caddr_t     x_private;      /* pointer to private data */
caddr_t     x_base;         /* private used for position info */
int         x_handy;        /* extra private word */
```

opaque_auth structure

The opaque_auth structure is referenced in the **AUTH**, **CLIENT**, **SVCXPRT**, and **XDR** structures.

The opaque_auth structure contains the following members:

```
    enum_t       oa_flavor;   /* flavor of auth */
    caddr_t      oa_base;     /* address of more auth stuff */
    u_int        oa_length;   /* not to exceed 400 bytes */
```

clnt_stat enumeration

The clnt_stat enumeration is referenced in the **AUTH**, **CLIENT**, **SVCXPRT**, and **XDR** structures.

The **clnt_stat** enumeraton contains the following members:

```
RPC_SUCCESS=0,            /* call succeeded */
/*
 * local errors
 */
RPC_CANTENCODEARGS=1,     /* cannot encode arguments */
RPC_CANTDECODERES=2,      /* cannot decode results */
RPC_CANTSEND=3,           /* failure in sending call */
RPC_CANTRECV=4,           /* failure in receiving result */
RPC_TIMEDOUT=5,           /* call timed out */
RPC_INTR=18,              /* call interrupted */
/*
 * remote errors
 */
RPC_VERSMISMATCH=6,       /* rpc versions not compatible */
RPC_AUTHERROR=7,          /* authentication error */
RPC_PROGUNAVAIL=8,        /* program not available */
RPC_PROGVERSMISMATCH=9,   /* program version mismatched */
RPC_PROCUNAVAIL=10,       /* procedure unavailable */
RPC_CANTDECODEARGS=11,    /* decode arguments error */
RPC_SYSTEMERROR=12,       /* generic "other problem" */

/*
 * rpc_call & CLNT creation errors
 */
RPC_UNKNOWNHOST=13,       /* unknown host name */
RPC_UNKNOWNPROTO=17,      /* unknown protocol */
RPC_UNKNOWNADDR=19,       /* Remote address unknown */
RPC_NOBROADCAST=21,       /* Broadcasting not supported */

/*
 * binding errors
 */
RPC_RPCBFAILURE=14,       /* rpcbind failed in its call */
RPC_PROGNOTREGISTERED=15, /* remote program not registered */
RPC_N2AXLATEFAILURE=22,   /* Name to address translation failed */
/*
 * Misc error in the TLI library
 */
RPC_TLIERROR=20,
/*
 * unspecified error
 */
```

```
RPC_FAILED=16
```

Remote Services Languages

EXTERNAL DATA REPRESENTATION (XDR)

XDR is a standard for the description and encoding of data. It is useful for transferring data between different computer architectures, and has been used to communicate data between such diverse machines as the AT&T 3B2, Sun Workstation, VAX, IBM-PC, and Cray. XDR fits into the ISO presentation layer, and is roughly analogous in purpose to X.409, ISO Abstract Syntax Notation. The major difference between these two is that XDR uses implicit typing, while X.409 uses explicit typing.

XDR uses a language to describe data formats. The language can only be used only to describe data; it is not a programming language. This language allows one to describe intricate data formats in a concise manner. The alternative of using graphical representations (itself an informal language) quickly becomes incomprehensible when faced with complexity. The XDR language itself is similar to the C language, just as Courier is similar to Mesa. Network facilities, such as RPC (Remote Procedure Call) and the NFS (Network File System) use XDR to describe the format of their data.

The XDR Language Specification

Notational Conventions

This specification uses an extended Backus-Naur Form notation for describing the XDR language. Here is a brief description of the notation:

1. The characters |, (,), [,], , and * are special.
2. Terminal symbols are strings of any characters surrounded by double quotes.
3. Non-terminal symbols are strings of non-special characters.
4. Alternative items are separated by a vertical bar (|).
5. Optional items are enclosed in brackets.
6. Items are grouped together by enclosing them in parentheses.
7. A * following an item means 0 or more occurrences of that item.

Lexical Notes

1. Comments begin with '/*' and terminate with '*/'.
2. White space serves to separate items and is otherwise ignored.
3. An identifier is a letter followed by an optional sequence of letters, digits or underbar ('_'). The case of identifiers is not ignored.
4. A constant is a sequence of one or more decimal digits, optionally preceded by a minus-sign ('-').

Syntax Information

```
declaration:
        type-specifier identifier
        | type-specifier identifier "[" value "]"
        | type-specifier identifier "<" [ value ] ">"
        | "opaque" identifier "[" value "]"
        | "opaque" identifier "<" [ value ] ">"
        | "string" identifier "<" [ value ] ">"
        | type-specifier "*" identifier
        | "void"

value:
        constant
        | identifier

type-specifier:
          [ "unsigned" ] "int"
        | [ "unsigned" ] "hyper"
        | "float"
        | "double"
        | "bool"
        | enum-type-spec
        | struct-type-spec
        | union-type-spec
        | identifier

enum-type-spec:
        "enum" enum-body

enum-body:
        "{"
        ( identifier "=" value )
        ( "," identifier "=" value )*
```

```
        "}"

struct-type-spec:
        "struct" struct-body

struct-body:
        "{"
        ( declaration ";" )
        ( declaration ";" )*
        "}"

union-type-spec:
        "union" union-body

union-body:
        "switch" "(" declaration ")" "{"
        ( "case" value ":" declaration ";" )
        ( "case" value ":" declaration ";" )*
        [ "default" ":" declaration ";" ]
        "}"

constant-def:
        "const" identifier "=" constant ";"

type-def:
        "typedef" declaration ";"
        | "enum" identifier enum-body ";"
        | "struct" identifier struct-body ";"
        | "union" identifier union-body ";"

definition:
        type-def
        | constant-def

specification:
        definition *
```

Syntax Notes

1. The following are keywords and cannot be used as identifiers: "int", "bool", "char", "case", "const", "default", "double", "enum", "float", "hyper", "opaque", "string", "struct", "switch", "typedef", "union", "unsigned" and "void".

2. Only unsigned constants may be used as size specifications for arrays. If an identifier is used, it must have been declared previously as an unsigned constant in a "const" definition.

3. Constant and type identifiers within the scope of a specification are in the same name space and must be declared uniquely within this scope.

4. Similarly, variable names must be unique within the scope of struct and union declarations. Nested struct and union declarations create new scopes.

5. The discriminant of a union must be of a type that evaluates to an integer. That is, "int", "unsigned int", "bool", an enumerated type or any typedefed type that evaluates to one of these is legal. Also, the case values must be one of the legal values of the discriminant. Finally, a case value may not be specified more than once within the scope of a union declaration.

An Example of an XDR Data Description

Here is a short XDR data description of an object called a "file", which might be used to transfer files from one machine to another.

```
const MAXUSERNAME = 32;     /* max length of a user name */
const MAXFILELEN = 65535;   /* max length of a file      */
const MAXNAMELEN = 255;     /* max length of a file name */

/*
 * Types of files:
 */

enum filekind {
        TEXT = 0,       /* ascii data */
        DATA = 1,       /* raw data   */
        EXEC = 2        /* executable */
};

/*
 * File information, per kind of file:
 */

union filetype switch (filekind kind) {
        case TEXT:
                void;                           /* no extra information */
        case DATA:
                string creator<MAXNAMELEN>;     /* data creator         */
        case EXEC:
                string interpretor<MAXNAMELEN>; /* program interpretor  */
};

/*
 * A complete file:
 */

struct file {
        string filename<MAXNAMELEN>; /* name of file */
        filetype type;               /* info about file */
        string owner<MAXUSERNAME>;   /* owner of file   */
        opaque data<MAXFILELEN>;     /* file data       */
};
```

REMOTE PROCEDURE CALL (RPC)

The RPC Model

The remote procedure call model is similar to the local procedure call model. In the local case, the caller places arguments to a procedure in some well-specified location (such as a result register). It then transfers control to the procedure, and eventually gains back control. At that point, the results of the procedure are extracted from the well-specified location, and the caller continues execution.

The remote procedure call is similar, in that one thread of control logically winds through two processes—one is the caller's process, the other is a server's process. That is, the caller process sends a call message to the server process and waits (blocks) for a reply message. The call message contains the procedure's parameters, among other things. The reply message contains the procedure's results, among other things. Once the reply message is received, the results of the procedure are extracted, and caller's execution is resumed.

On the server side, a process is dormant awaiting the arrival of a call message. When one arrives, the server process extracts the procedure's parameters, computes the results, sends a reply message, and then awaits the next call message.

Note that in this model, only one of the two processes is active at any given time. However, this model is only given as an example. The RPC protocol makes no restrictions on the concurrency model implemented, and others are possible. For example, an implementation may choose to have RPC calls be asynchronous, so that the client may do useful work while waiting for the reply from the server. Another possibility is to have the server create a task to process an incoming request, so that the server can be free to receive other requests.

The RPC Language

Just as there was a need to describe the XDR data-types in a formal language, there is also need to describe the procedures that operate on these XDR data-types in a formal language as well. We use the RPC Language for this purpose. It is an extension to the XDR language.

The RPC Language Specification

The RPC language is identical to the XDR language, except for the added definition of a **program-def** described below.

```
program-def:
        "program" identifier "{"
                version-def
                version-def *
        "}" "=" constant ";"

version-def:
        "version" identifier "{"
                procedure-def
                procedure-def *
        "}" "=" constant ";"

procedure-def:
        type-specifier identifier "(" type-specifier ")"
        "=" constant ";"
```

Syntax Notes

1. The following keywords are added and cannot be used as identifiers: "program" and "version";
2. A version name cannot occur more than once within the scope of a program definition. Nor can a version number occur more than once within the scope of a program definition.
3. A procedure name cannot occur more than once within the scope of a version definition. Nor can a procedure number occur more than once within the scope of version definition.
4. Program identifiers are in the same name space as constant and type identifiers.
5. Only unsigned constants can be assigned to programs, versions and procedures.

Remote Services Environment

Remote Procedure Call (RPC)

Remote Procedure Call (RPC) is a high-level communications paradigm, including functions, that provide a protocol-independent application interface to networking services. Application developers access the functions that provide services at a particular level and need not care about the protocol implementation that is providing those services. These services provide end-to-end data transmission using the services of an underlying network. Applications written using the top most layers of the RPC interface are independent of the underlying transport protocols. By providing media and protocol independence, the interface enables networking applications to have the flexibility to run in various protocol environments. The RPC protocol compiler (`rpcgen`) and the **C**-like RPC language that it uses to specify RPC applications and define network data give application developers a simplified interface to the lower-level RPC mechanism. The RPC system uses External Data Representation (XDR) (a set of library routines) as its data transfer syntax mechanism.

External Data Representation (XDR)

External Data Representation (XDR) interfaces allow a user to describe arbitrary data structures in a machine-independent fashion. Any program running on any machine can use XDR to create portable data by translating local representations to XDR standard representations; similarly, any program running on any machine can read portable data by translating XDR standard representations to local equivalents. By solving data portability problems, the XDR library interface provides networking applications with the flexibility to run in various operating environments. XDR is the backbone of RPC, in the sense that the RPC system uses XDR as its data transfer syntax mechanism.

Network Selection

Network Selection interfaces provide protocol-independent applications with a simple, consistent mechanism for dynamically selecting communication service providers (*e.g.*, transport providers as currently supported by the Transport Level Interface (TLI)) according to users preferences and availability. Typically, this capability is employed by the client portion of an application in its initialization stage. On a machine having only a single network, this makes it possible for the application to use that network without requiring any application-specific action by the administrator or user. On machines having multiple networks, this makes it easy for the application to try each of the alternative networks in turn until it succeeds in establishing communication, and to try them in the order preferred by the user or specified as the local default by the administrator. This component is built around a network configuration database, listing the networks available on that system, and an optional `NETPATH` environment variable, set by a user to contain an ordered list of network identifiers (as defined in the network configuration database). The interface consists of a set of library routines for determining the identifiers of the networks available for use, and certain information relevant for each network.

Network Selection is used in the Name-to-Address Translation facility and in the RPC mechanism.

Name to Address Translation

The Name-to-Address Translation interfaces provide a protocol-independent means for finding the protocol specific addresses for services on a given machine. Given the name of the service and the name of the machine, the communications address(es) can be determined. This facility is typically used by the client portion of an application when it wishes to establish a communication path with a server. It is used by the RPC mechanism, but it can also be used directly by an application in conjunction with TLI. The facility will accommodate the addressing style of any communication service provider, and will function in environments where there are multiple communication service providers per machine, and multiple sources of addresses for each communication service provider. Queries may use the Network Selection facility to determine the communication service provider(s) for which addresses are to be retrieved.

The interface consists of a set of library routines that return one (or, optionally, all) of the addresses that can be found for the specified service on the specified machine. The addresses returned are communication provider’s addresses, in a form appropriate for use with TLI.

Distributed File Systems

The Remote Services Extension provides mechanisms for sharing resources among interconnected systems and utilities for administering these mechanisms. Such mechanisms and utilities comprise a distributed file system. The Remote Services Extension supports the distributed file system: The Network File System (NFS). Using NFS, programs can access files resident on remote systems as though the files were on the local system.

The generic utilities support the administration of different distributed file systems through the use of a flexible command syntax. This syntax includes a `-F` option, for specifying a file system type, and a `-o` option, for passing suboptions to commands that are specific to a file system type. A new distributed file system type can be administered with the generic utilities, provided that commands to support each generic operation are supplied with the new file system type.

The Remote Services Extension provides basic functionality for administering distributed file systems and expands the functionality of some components of the Base System, Basic Utilities Extension and the Administered Systems Extension [see effects(RS_ENV) and errno(RS_ENV)].

Conforming System Characteristics

Systems that support the Remote Services Extension provide an overall Distributed Files Systems environment having the following characteristics:

- network compatibility
- operation across heterogeneous processors
- reliability against a single point of failure

These characteristics ensure portability of source code from single-system environments to a network of systems sharing resources.

Network Compatibility

There are implementation-specific criteria for the underlying network(s) that would support distributed file systems. The NFS requires either the User Datagram Protocol (UDP) or the OSI connectionless transport-level protocol, TP4.

Operation Across Heterogeneous Processors

Some application-level operations may depend on characteristics of the underlying processor. For example, when an application writes a floating-point number into a file, it is typically stored in a format specific to that processor, which may differ in size or byte-ordering from the representation of the same number on a different processor. Similar considerations apply to the representation of more elaborate structured data items, which may also differ across processors in their alignment characteristics. Because the identification and interpretation of such complex data items are solely under the control of the application process and is not known to the operating system, the operating system cannot automatically perform the translations required for the proper interpretation of those data items when they are shared among processors of different types. By agreeing on a standard external data representation format, applications may manipulate arbitrarily complex data items as a pure sequence of bytes, and thus share those data items across dissimilar processors.

For any set of systems that are running NFS, applications on those systems will be able to share regular files and directories without concern for the underlying processor characteristics.

Reliability Against a Single Point of Failure

If one system running NFS ceases operation, then the operation of NFS between pairs of other systems must not be affected, except that access to a resource on a client may not be possible if any component of the pathname on that client resides on the system that ceased operation.

Distributed File Systems

NFS provides a user with access to files from remote systems as though they were on the system that the user has logged into. Remote files are named using the same conventions as for local files, and most operations on remote files work the same as they do on local files. This section presents an overview of the functionality and administrative features of Distributed File Systems.

In a network of systems that support the Remote Services Extension, a system is able to make selected parts of its file tree available to remote systems, by *sharing* them. Correspondingly, each system is able to augment its own file tree by *mounting* the shared files from other systems. The system that shares a resource is called the *server system*, while the system that uses the resource is called a *client system*. The following sections describes the concepts *share*, *unshare*, and *remote mount*.

Share

The right to allow remote access to a file belongs to the administrator of the system where the file resides. To allow remote access, an administrator shares a resource using the `share` command.

NFS allows any directory or file to be shared. Once a directory is shared by NFS, all of the regular files and directories under it are accessible to an authorized system, provided they are in the same file system as the directory shared. Named pipes and special devices on the server are not accessible to the client, however. Any such object in a shared directory is assumed to be on the client system.

Figure 10-1: A System V File Tree

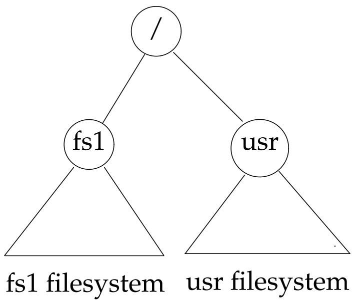

Figure 17-1 shows part of a typical file tree.

To share the file system portion under `/fs1` via NFS, the administrator may type

```
share -F nfs /fs1
```

The above command specifies that other systems will use the name *server*`:/fs1` to refer to the resource when they mount it from the system *server* via NFS.

Unshare

The administrator can *unshare* a resource at any time after it has been shared by using the `unshare` command.

With NFS, unsharing a resource prevents all remote access to the resource by any client, regardless of whether that client has previously mounted the resource. Once the resource has been shared again, a client that has mounted the resource may continue to access the resource without re-mounting it.

Remote Mount

The Distributed File System Extension extends the `mount()` operation to include remote resources [see effects(RS_ENV)]. After a system has shared a resource, another system may remotely mount that resource on a directory in its own file tree.

With NFS, an administrator mounts the remote resource shared above on the local `/fs2` directory by typing

```
mount -F nfs foo:/fs1 /fs2
```

where `foo` is the name of the server on which `/fs1` resides.

Figure 10-2: Remote Mount

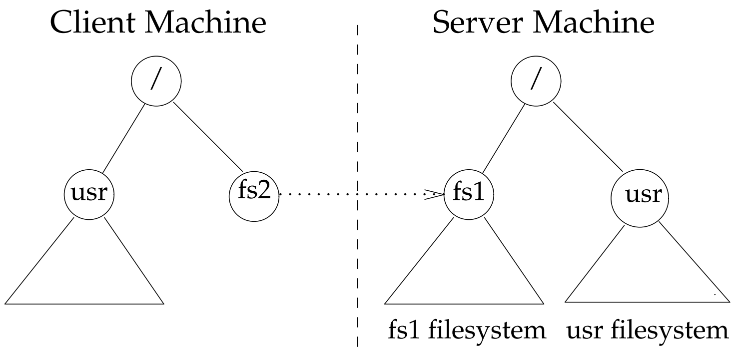

Figure 2 shows the two systems' file systems after the remote mount. When a user on the client machine refers to the subtree under `/fs2`, the file referenced is the one on the server machine subtree under `/fs1`. For example, a user on the client system who uses the file name `/fs2/src/uts` refers to the file `/fs1/src/uts` on the server system.

There is no need for the structures of client and server file trees to match in any way, or for shared resources to be mounted at the same level on the client as they occupy on the server. If the client had done the remote mount onto its `/usr` directory, then its references to files under `/usr` would be to the server subtree under `/fs1`.

A client cannot get to parts of the server file tree that are not under the shared directory. For example, if a user on a client system uses ''cd ..'' to move up from the top directory in a remotely mounted subtree, the user always ends up back in the client file tree.

An NFS client may even be able to access files that are not accessible on the server, since a server can mount another file system over a resource after a client has established its means of access to the file.

The Network File System Administration

The following sections describe the resource naming and security features of NFS.

Resource Naming

Resource names are composed of two parts, the server's name and the pathname of the resource on the server. For example, a client would refer to resource `/usr/smith` from server `foo` as `foo:/usr/smith`.

Security Features

Security in NFS is provided by three mechanisms: client authentication, client authorization, and id mapping.

Client Authentication By default, the client, at each access, provides the server with the client's system name and the requesting user's user id (uid).

To provide greater security, the server machine may share the filesystem as follows:

```
share -F nfs -o secure /usr/private
```

The client must then mount the file system specifying the secure option as follows:

mount -F nfs -o secure *server*:/usr/private /fs2

Client Authorization NFS provides a way for an administrator to share directories selectively through the `share` command. For example, to share `/usr/private` so that only systems *mach1* and *mach2* could mount that directory, the administrator could issue the command

```
share -F nfs -o rw=mach1:mach2 /usr/private
```

Without a list of systems, the `share` command puts no restriction on availability.

An administrator may also choose to share a directory read-only by using the `–o ro` suboption. Here, a remote mount will only succeed if the mount command also includes the `–o ro` suboption.

ID Mapping Within a group of systems sharing resources via NFS, administration is simplified when the `/etc/passwd` and `/etc/group` files are identical or can be made to appear identical across all systems. More elaborate mechanism may add flexibility in particular installations.

Remote Services Environment Routines

The following section contains the manual pages for the RS_ENV routines.

NAME

effects – effects of the Remote Services Extension on other extensions.

DESCRIPTION

Support for the Remote Services extension effects the behavior of some routines belonging to other extensions. The effects are listed below for each routine.

mount(AS_CMD)

For users and applications processes, the effect of a remote mount is the same as a local mount: an additional file system has been mounted into the local file tree. Once a remote resource has been mounted, all operating system service routines will operate on the remote files as they do on local files, with the following exceptions. For The Network File System, only regular files and directories are accessible as remote resources. For Remote File Sharing, it is implementation-specific whether the following operating system service routines will accept a remote file:

acct(KE_OS)	poll(BA_OS)
getmsg(BA_OS)	putmsg(BA_OS)

Errors. If the command

`mount –F` *FSType* `–o` *suboptions options special directory*

is given and any of the additional conditions below hold, then an error message will be sent to standard error. The additional conditions are the following: (1) The distributed file system *FSType* is not available on the local host, (2) the resource is not currently shared, or (3) the client is not authorized to access the resource.

For Remote File Sharing, the following additional conditions will also cause an error message to be sent to standard error: (1) the mount point *directory* is itself shared as a resource, (2) the mount point *directory* is already a mount point, (3) the `–r` option or `–o ro` suboption is not specified and the resource was shared as read-only, or (4) the resource is already mounted.

umount(AS_CMD)

Errors. With Remote File Sharing, additional error conditions can arise on servers when they attempt to unmount local resources that are currently shared or remotely mounted. If (1) the resource has not been unshared or (2) the resource is still currently mounted on a remote system, then an error message will be sent to standard error.

fuser(AS_CMD)

For all distributed file systems, remote resources mounted locally can be specified on the command line by giving the resource name or the mount point directory as an argument.

sar(AS_CMD)

For Remote File Sharing, the options `–S` and `–D` are available with `sar`. If neither of these options is specified on the command line, the output of `sar` will not change. The complete synopsis is:

`sar [–ubdycwaqvmprADS] [–o` *file*`] t [n]`
`sar [–ubdycwaqvmprADS] [–s` *time*`] [–e` *time*`] [–i` *sec*`] [–f` *file*`]`

The –D option is used in combination with either the –u or –c option. If the –D is used and neither –u nor –c is specified, –u is assumed.

The command `sar -u` reports time spent in user mode, in system mode, idle with some process waiting for block I/O, and otherwise idle. If the –D option is also specified, system time is reported for time servicing remote requests and all other system time. The command `sar -c` reports activity data on system calls. If the –D option is also specified, the data are reported for three categories: system calls resulting in outgoing remote activity, system calls resulting from incoming remote activity, and strictly local system calls.

The –S option is used to obtain reports on server processes and request queue status. Every request from a remote host to access your resources is conveyed by a request message that is handled by a *server process*. When there are too many messages for the servers to handle, the messages are placed on a request queue. Messages leave the queue and are processed when servers are available. The data reported by the –S option are the following: average number of server processes on the system (`serv/lo-hi`), percent of time request messages are on the request queue (`request %busy`), average number of request messages waiting for service when the request queue is occupied (`request avg lgth`), percent of time there are idle servers (`server %avail`), and average number of idle servers when idle ones exist (`server avg avail`).

sa1(AS_CMD)

The new –S and –D options described for `sar` are also available for `sa2`; the interfaces to `sa1` and `sadc` are unchanged. The complete synopsis for `sa2` is:

/usr/lib/sa/sa2 [–ubdycwaqvmprADS] [–s *time*]
[–e *time*] [–i *sec*]

FUTURE DIRECTIONS

The four operating system service routines acct(KE_OS), poll(BA_OS), getmsg(BA_OS) and putmsg(BA_OS) will be extended in the future to operate with remote files accessed via Remote File Sharing.

Due to changes in Remote File Sharing architecture, `sar -Dc` will be removed in a future issue of the SVID. `sar` will instead report Remote File Sharing operations with a different option.

LEVEL

Level 1.

The following have moved to Level 2 effective September 30, 1989: sar(AS_CMD) and sa1(AS_CMD).

NAME

errno – Remote Services error codes and condition definitions

SYNOPSIS

```
#include <errno.h>

‡ extern int errno;

errno
```

DESCRIPTION

The numerical value represented by the symbolic name of an error condition is assigned to `errno` for errors that occur when executing a system service routine or general library routine.

To be consistent with the C Standard, the interface definition of `errno` has been change in the SIVD, Fourth Edition. Programs should obtain the value of `errno` by including `<errno.h>`.

The macro `errno` expands to a modifiable *lvalue* that has type `int`, the value of which is set to a positive error number by several library functions. `errno` need not be the identifier of an object, *e.g.*, it might expand to a modifiable *lvalue* resulting from a function call. It is unspecified whether `errno` is a macro or an identifier declared with external linkage. If an `errno` macro definition is suppressed to access an actual object, or if a program defines an identifier with the name `errno`, the behavior is undefined.

Additional error codes defined in the Remote Services extension are listed below.

`ECOMM` Communication error: occurs on any operating system service routine that references a remote resource (through a file descriptor or path name), whenever there is a communications error while trying to send the request for that service routine to the server system.

`EMULTIHOP`
Multihop not allowed: This error may occur on any operating system service routine that has a path name as one of its arguments, and indicates that resolution of that path name involves multihop access to a remote resource, when multihop access is not supported by the underlying implementation. Whether multihop access is supported is implementation-specific, but if it is not supported, then the `EMULTIHOP` error condition must be returned on any attempted multihop access.

`ENOLINK` The link has been severed: occurs on any operating system service routine that references a remote file, when the communications link to the server for that resource has been lost; any file descriptor associated with this remote file should not be used for further I/O.

`EREMOTE` The object is remote: occurs on the `mount()` operating system service routine when the requested mount point resides on a remote resource during a Remote File Sharing mount attempt.

`ESTALE` The file handle is stale: occurs during a Network File System operation, when the file handle used to reference the remote resource is not currently valid or is temporarily disabled.

In addition, some operating system service routines may return the `errno` value of `EINTR` when accessing a remote resource. The following operating system service routines may return this value of `errno` when operating on objects via distributed file systems:

access	chown	dup	link	unlink
chdir	close	exec	mknod	ustat
chmod	creat	fcntl	stat	utime

An application that checks the value of `errno` must include the header file `<errno.h>`.

SEE ALSO

errno(BA_ENV), errno(KE_ENV), mount(BA_OS).

LEVEL

Level 1.

NAME

netconfig — network configuration database

SYNOPSIS

`#include <netconfig.h>`

DESCRIPTION

The network configuration database (or netconfig database) is a system database for information about the networks connected to a system, where "network" is used in the sense of "community of mutually addressable entities." It is implemented via the administrative file `/etc/netconfig`. It contains entries giving information about the networks that are available for use.

For each available network, this file contains an entry; entries are separated by newlines. Each entry contains the following information, in whitespace (blank or tab) -separated fields, in this order (whitespace can be embedded as "*blank*" or "*tab*", and a backslash as "\\"):

network ID (sometimes called a "token")
semantics of protocol (*i.e.*, connectionless or connection oriented)
flags
protocol family
protocol name
network device
comma-separated list of directory lookup libraries

These fields correspond to entries in the `struct netconfig` structure (see below). This structure, and the identifiers described on this manpage, are defined in `<netconfig.h>`.

The *network ID* is a string used to denote a particular network; it consists of non-null, printable ASCII characters, and has length at least 1 (no specified maximum). This namespace is locally-significant, and the local system administrator is the naming authority. All network IDs on a system must be unique (otherwise, the namespace is not well-defined).

The *flags* field records certain attributes of networks. It consists of a string composed of either a "`v`" or a "`–`": "`v`" indicates visible ("default") network, used when `NETPATH` is *unset*; "`–`" indicates that none of these flags apply.

The *protocol family* and *protocol name* fields are provided for those applications that wish to be protocol-specific ("`–`" indicates these fields are unspecified). These fields should not be used by programs that are protocol-independent. The *protocol family* and *protocol name* can be any alphanumeric string of at least 1 character.

The *directory lookup libraries* support a "network directory service" (name-to-address mapping service) for the network, which is implemented by the *name-to-address* (or *netdir*) feature [see netdir(RS_LIB)]. Networks must be provided with such libraries, in order for the netdir feature to work ("`-`" indicates the absence of a directory for a network). This field consists of a comma-separated list of full pathnames to dynamically-linked libraries (a comma can be embedded as "`\,`", and a backslash as "\\").

The *network device* is the full pathname of the device used to connect to the transport provider. Typically, this device will be in the `/dev` directory. This device must be specified.

The `struct netconfig` structure includes the following members, which correspond to the fields in the entries in the netconfig database:

`char * nc_netid` — Network ID, including ASCII NUL terminator.
`unsigned long nc_semantics` — semantics of protocol (*i.e.*, connectionless or connection oriented)
`char * nc_flag` — Flags.
`unsigned long nc_proto` — Protocol name.
`char * nc_protofmly` — Protocol family.
`char * nc_device` — The network device (full pathname).
`unsigned long nc_nlookups` — Number of directory lookup libraries.
`char ** nc_lookups` — The directory lookup libraries themselves (full pathnames).

The `nc_semantics` field contains one of the following values, depending upon whether the transport is connection oriented, connection oriented and supports orderly release, or connectionless:

```
NC_TPI_COTS
NC_TPI_COTS_ORD
NC_TPI_CLTS
```

The `nc_flag` field is a bitfield. The following bits are recognized, corresponding to the "`v`" and "`-`" respectively.

```
NC_VISIBLE
NC_NOFLAG
```

The `nc_protofmly` field takes on values of the protocol family character strings. The `nc_proto` field takes on values of the protocol names. These can be any character string of at least 1 character.

USAGE

The combination of the layer and the mode (circuit or datagram) determines the "semantics" of the network. Typically, an application will specify an API (application programming interface) by pushing appropriate STREAMS modules (such as `timod`, and using user-level library functions (such as the TLI library).

SEE ALSO

getnetconfig(RS_LIB), getnetpath(RS_LIB), netdir(RS_LIB).

FILES

`/etc/netconfig`.

LEVEL

Level 1.

NAME

publickey – public key database

SYNOPSIS

`publickey`

DESCRIPTION

`publickey` is the public key database used in secure RPC. Each entry in the database consists of a network user name (which may either refer to a user or a hostname), followed by the user's public key (in hex notation), a colon, and then the user's secret key encrypted with a password (also in hex notation).

This file is altered either by the user through the `chkey` command [see chkey(RS_CMD)] or by the system administrator through the `newkey` command [see newkey(RS_CMD)].

SEE ALSO

chkey(RS_CMD), newkey(RS_CMD), publickey(RS_LIB).

LEVEL

Level 1.

NAME

rpc – rpc program number data base

SYNOPSIS

`rpc`

DESCRIPTION

The `rpc` program number database contains user readable names that can be used in place of RPC program numbers. Each line has the following information:

name of server for the RPC program
RPC program number
aliases

Items are separated by any number of blanks and/or tab characters. A # indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Below is an example of an RPC database:

```
#
#            rpc
#

rpcbind                     100000    portmap sunrpc portmapper
rusersd                     100002    rusers
nfs                         100003    nfsprog
mountd                      100005    mount showmount
walld                       100008    rwall shutdown
sprayd                      100012    spray
llockmgr      100020
nlockmgr      100021
status                      100024
bootparam     100026
keyserv                     100029    keyserver
```

LEVEL

Level 1.

Remote Services Library Routines

The following section contains the manual pages for the RS_LIB routines.

NAME

cs_connect, cs_perror – application interface to the Connection Server

SYNOPSIS

```
#include <cs.h>
#include <netconfig.h>
#include <netbuf.h>
```

`int cs_connect(char *`*host*`, char *`*service*`, struct csopts *`*cs_opt*`, int *`*error*`);`

`void cs_perror(char *`*string*`, int `*error*`)`

DESCRIPTION

The library routines **cs_connect()**, and **cs_perror()** provide an interface that network applications use to establish an authenticated TLI/XTI connection to a network *service* on *host*. The Connection Server interface shields the client application from details of connection establishment and authentication. Since **cs_connect()** performs authentication on behalf of the client process, authentication is effectively automated. The way in which **cs_connect()** accesses authentication schemes also allows the system administrator to use modular schemes that are interchangeable and can be administered on a per-service basis.

cs_connect() communicates with the Connection Server daemon which establishes a TLI/XTI connection on behalf of the client application and returns a file descriptor associated with the connection. The Connection Server uses the Network Selection mechanism to determine the transport provider needed to connect to the specified service and uses the Name-to-Address Mapping facility to obtain the address of the network service over that transport.

The arguments are defined as follows:

host The name of the server machine that is supplying the service. This name can be any string acceptable to the Name-to-Address Mapping facility.

service The name of the service the application wishes to communicate with.

csopts The **csopts** structure is provided to allow the programmer more flexibility. In most applications the third argument, **cs_opt**, will be NULL. **csopts** is defined in the header file **/usr/include/cs.h** as:

```
struct csopts {
     struct netconfig *nc_p;
     int nd_opt;
     struct netbuf *nb_p;
}
```

Each element of this structure is described below.

struct netconfig *nc_p

To restrict the networks which may be used in making a connection, the user should set the element **nc_p** to point to a **netconfig** structure. A network will be selected which matches with all the elements in the **netconfig** structure that have been filled in by the user (see **netconfig**(RS_ENV)). For example, if the user wants to use only TCP protocol networks then **nc_p->nc_proto** should be set to **tcp** and all other elements should be set to zero or NULL. If the user does not want to restrict network selection, **nc_p** should be set to

(struct netconfig *)NULL

int nd_opt
To bind to a reserved port, set this element to **ND_SET_RESERVEDPORT** (see **netdir**(RS_LIB)).

struct netbuf *nb_p
To bind to a reserved port on a specific address, **nd_opt** should be set as described above and **nb_p** should be set to point to a **netbuf** structure (see **netdir**(RS_LIB)).

error A pointer to an **int**. When an error occurs, **cs_connect()** sets the value of *error*. **cs_perror()** can then be called by the application with *error* as an argument to print a description of the error.

string The string that is to precede error messages.

cs_connect() establishes communication with the Connection Server daemon via a named stream and sends the host name and service name as parameters. **cs_connect()** also sends the value of the NETPATH environment variable, or a NULL value if NETPATH is not set, and the contents of the **csopts** structure. Note that it does *not* send the values of the last two elements of **nc_p**.

The Connection Server daemon uses the Network Selection and Name-to-Address Mapping facilities to attempt to establish an authenticated connection to *host* for *service* over each available transport until a connection is established or connection establishment fails for every tranport.

The Connection Server consults the **/etc/iaf/serve.allow** file for the list of authentication schemes acceptable to the client machine for *service* on *host*.

If an authenticated connection is established, the Connection Server returns a file descriptor associated with the connection. The application can then perform all TLI/XTI operations (**t_snd**, **t_rcv**, etc.) on the file descriptor.

cs_perror() prints an error message on the standard error. The error message is derived from indexing a value referenced by *error*, which was set by **cs_connect**. The message is preceded by *string* and a colon.

RETURN VALUE

On successful completion, **cs_connect()** returns a file descriptor containing a positive integer. On failure **cs_connect()** returns a **–1**.

On failure, **cs_perror()** may report the following errors:

CS_NOERROR	**No error**
CS_SYSERROR	**System Error**
CS_MALLOC	**No Memory**
CS_AUTHNOTACCEPTABLE	**Authentication scheme specified by server is not acceptable**
CS_CONNECTFAILED	**Connection to service failed**
CS_INVOKEFAILED	**Error in invoking authentication scheme**
CS_SCHEMEFAILED	**Authentication scheme unsuccessful**
CS_NOTRANSPORT	**Could not obtain address of service over any transport**

CS_PIPE	Could not create CS pipe
CS_FATTACH	Could not mount remote stream to CS pipe
CS_CONNLD	Could not push CONNLD
CS_FORK	Could not fork CS child request
CS_CHDIR	Could not chdir
CS_SETNETPATH	Host/Service not found over available transport
CS_TOPEN	TLI/XTI failure: t_open failed
CS_TBIND	TLI/XTI failure: t_bind failed
CS_TCONNECT	TLI/XTI failure: t_connect failed
CS_TALLOC	TLI/XTI failure: t_alloc failed
CS_MACFAILED	MAC check failure or Secure Device access denied
CS_DACFAILED	DAC check failure or Secure Device access denied
CS_TIMEDOUT	Connection attempt timed out
CS_NETPRIV	Privileges not correct for requested network options
CS_BADOPTION	Netdir option incorrectly set in csopts struct
CS_DIALERROR	Dial error
CS_STATERROR	Unable to do devalloc() or stat()
CS_NOTFOUND	Service not found in _pmtab

USAGE

Not all values stored in the **csopts** structure are sent to the Connection Server. In particular, the last two elements of **nc_p**, that is, **nc_lookups** and **nc_nlookups**, are not sent. See **netconfig**(RS_ENV).

The Connection Server daemon will log a message to **/var/connserv/log** on startup.

The Connection Server daemon will print debug information to **/var/connserv/debug** if it is invoked with the debug option:

/usr/sbin/cs -d

In order for network applications to use **cs_connect()**, the following network components must be correctly administered:

- The port monitor administrative files.
- Authentication schemes, where used.
- ID Mapping.

EXAMPLE

A typical call to **cs_connect** will be of the form:

```
#include <netconfig.h>
#include <netbuf.h>
#include <cs.h>
      .
      .
      .
int error = 0;
      .
      .
      .
if ((fd = cs_connect("host", "service", (struct csopts *)NULL, &error)) < 0) {
     /* do error handling */
```

```
        cs_perror("application specific string", error);
        exit(1);
    }
    /* continue with normal execution */
         .
         .
         .
```

LEVEL

Level 1.

NAME

getnetconfig, setnetconfig, endnetconfig, getnetconfigent, freenetconfigent – network configuration database

SYNOPSIS

```
#include <netconfig.h>

struct netconfig *getnetconfig(void *handlep);
void *setnetconfig(void);
int endnetconfig(void *handlep);
struct netconfig *getnetconfigent(char *netid);
void freenetconfigent(struct netconfig *netconfigp);
void nc_perror (char *msg);
char *nc_sperror (void);
```

DESCRIPTION

These routines are part of the network selection feature. They are a set of manipulation routines for the local system network configuration (netconfig) database [see netconfig(RS_ENV)].

A call to `setnetconfig()` has the effect of "binding" or "rewinding" (figuratively speaking) the netconfig database. It must be called before the first call to `getnetconfig()` (but not before `getnetconfigent()`), and may be called any other time. It returns a "handle" that is passed to `getnetconfig()` when looping. The handle uniquely identifies each instance of a loop.

`getnetconfig()`, when first called, returns a pointer to the (formatted) first entry in the netconfig database; formatted as a `struct netconfig` thereafter, it subsequently returns a pointer to the successive entries in the database. In this manner, `getnetconfig()` can be used to traverse the netconfig database. It takes the handle returned by `setnetconfig()` as an argument to uniquely identify each instance of the loop.

`endnetconfig()` may be called to "unbind" the netconfig database after it has been bound by `setnetconfig()`, when processing is complete. It takes the handle returned by `setnetconfig()` as an argument.

`getnetconfigent()` returns a pointer to the netconfig database entry corresponding to the network identifier *netid*.

`freenetconfigent()` frees the space allocated by `getnetconfigent()`.

nc_perror prints a message to the standard error indicating why any of the above routines failed. The message is prepended with *string msg* and a colon. A NEWLINE is appended at the end of the message.

nc_sperror is similar to **nc_perror** but instead of sending the message to the standard error indicating why the network selection routines failed, it returns a pointer to the message.

RETURN VALUE

When the database has been exhausted, `getnetconfig()` returns `NULL`. It returns `NULL` and sets `errno` in case of failure (*e.g.*, if `setnetconfig()` was not called previously).

`setnetconfig()` returns a handle to be used in looping. Each call returns a different handle, so loops can be nested.

`endnetconfig()` returns `0` on success, `-1` on failure (*e.g.*, if `setnetconfig()` was not called previously).

`getnetconfigent()` returns `NULL` if *netid* is invalid (does not name an entry in the netconfig database).

`nc_sperror` returns `NULL` if space can not be allocated for the message.

USAGE

These routines do not use static memory areas. All their data areas are dynamically allocated, and must be freed by the user. `endnetconfig()` does this automatically; `freenetconfigent()` frees data allocated by `getnetconfgent()`.

SEE ALSO

getnetpath(RS_LIB), netconfig(RS_ENV).

LEVEL

Level 1.

NAME

getnetpath, setnetpath, endnetpath – manipulate NETPATH

SYNOPSIS

```
#include <netconfig.h>

struct netconfig *getnetpath(void *handlep);
void *setnetpath(void);
int endnetpath(void *handlep);
```

DESCRIPTION

These routines are part of the network selection feature. They are a set of manipulation routines for the system network configuration (netconfig) database [see netconfig(RS_ENV)], as "filtered" by the `NETPATH` environment variable.

A call to `setnetpath()` has the effect of "binding" or "rewinding" (figuratively speaking) `NETPATH`. It must be called before the first call to `getnetpath()`, and may be called any other time. It returns a "handle" used by `getnetpath()`.

`getnetpath()`, when first called, returns a pointer to the (formatted) netconfig database entry corresponding to the first component of `NETPATH` (unless `NETPATH` is *unset* — see below); thereafter, it subsequently returns a pointer to the successive entries of `NETPATH`. In this manner, `getnetpath()` can be used to search the whole of `NETPATH`. It takes as an argument the handle returned by `setnetpath()`.

`getnetpath()` silently ignores invalid components of `NETPATH` (components which do not have a corresponding entry in the netconfig database).

`endnetpath()` may be called to "unbind" `NETPATH` when processing is complete. It takes as an argument the handle returned by `setnetpath()`.

If the `NETPATH` variable is not set (or has been unset), then `getnetpath()`, `setnetpath()` and `endnetpath()` behave as though `NETPATH` were set to the sequence of "default" (visible) networks in the netconfig database (in the order they are listed there). The default networks are those with a "v" in the *flags* field of the netconfig database.

RETURN VALUE

When `NETPATH` has been exhausted, `getnetpath()` returns `NULL`. It returns `NULL` if an error occurs (*e.g.*, if `setnetpath()` was not called previously). `nc_perror`(RS_LIB) can be called to report the error.

`setnetpath()` returns a handle that is to be used by `getnetpath()`.

`endnetpath()` returns `0` on success, `-1` on failure (*e.g.*, if `setnetpath()` was not called previously).

USAGE

These routines do not use static data memory areas. All their data areas are dynamically allocated, and must be freed by the user. `endnetconfig()` does this automatically.

SEE ALSO

getnetconfig(RS_LIB), netconfig(RS_ENV).

LEVEL

Level 1.

NAME

netdir: netdir_free, netdir_getbyname, netdir_getbyaddr, netdir_options, taddr2uaddr, uaddr2taddr, netdir_perror, netdir_sperror – generic transport name-to-address translation

SYNOPSIS

```
#include <netdir.h>
#include <netconfig.h>

int netdir_getbyname(const struct netconfig *netconf,
     const struct nd_hostserv *service, struct nd_addrlist **addrs);

int netdir_getbyaddr(const struct netconfig *netconf,
     struct nd_hostservlist **service, const struct netbuf *netaddr);

void netdir_free(void *ptr, int ident);

int netdir_options(const struct netconfig *netconf, int option,
     int fd, const char *pointer_to_args);

char * taddr2uaddr(const struct netconfig *netconf, const struct netbuf *addr);

struct netbuf *uaddr2taddr(const struct netconfig *netconf, const char *uaddr);

void netdir_perror(char *s);

char * netdir_sperror(void);
```

DESCRIPTION

These routines provide a generic interface for name-to-address mapping that will work with all transport protocols. This interface provides a generic way for programs to convert transport specific addresses into common structures and back again.

The `netdir_getbyname()` routine maps the machine name and service name in the `nd_hostserv` structure to a collection of addresses of the type understood by the transport identified in the netconfig structure *netconf*. This routine returns all addresses that are valid for that transport in the `nd_addrlist` structure.

The `nd_hostserv` structure contains the following members:

```
char          *h_host
char          *h_serv
```

The `nd_addrlist` structure contains the following members:

```
int           n_cnt
struct netbuf *n_addrs
```

n_cnt contains the number of addresses which `netdir_getbyname()` found.

`netdir_getbyname()` accepts some special case host names. These host names are hints to the underlying mapping routines that define the intent of the request. This information is required for some transport provider developers to provide the correct information back to the caller. The host names are defined in `/usr/include/netdir.h`. The currently defined host names are:

`HOST_SELF` This host name represents the address to which local programs will bind their endpoints. This differs from the host name provided by `gethostname()` which represents the address to which *remote* programs will bind their endpoints.

`HOST_ANY` This host name represents any host accessible by this transport provider. This name is provided to allow applications to specify a required service without specifying a particular host name.

`HOST_BROADCAST`
This host name represents the address for all hosts accessible by this transport provider. Network requests to this address will be received by all machines.

All fields of the `nd_hostserv` structure must be initialized.

To find all available transports, repeatedly call the `netdir_getbyname()` routine with each `netconfig` structure returned by the `getnetpath()` call.

The `netdir_getbyaddr()` routine maps addresses to service names. This routine returns a list of host and service pairs that would yield this address. If more than one tuple of host and service name is returned then the first tuple contains the preferred host and service names. The `nd_hostservlist` structure contains the following members:

```
int                 *h_cnt
struct nd_hostserv  *h_hostservs
```

h_cnt contains the number of host service names which `netdir_getbyaddr()` found.

The **`netdir_free`** structure is used to free the structures allocated by the name to address translation routines.

The following types of structures may be specified by the *ident* argument:

ND_ADDR Frees a **`netbuf`** structure.

ND_ADDRLIST
Frees the **`nd_addrlist`** structure such as that allocated by **`netdir_getbyname`**.

ND_HOSTSERV
Frees a **`nd_hostserv`** structure.

ND_HOSTSERVLIST
Frees the **`nd_hostservlist`** structure such as that allocated by **`netdir_getbyaddr`**.

The **`netdir_options`** routine is used to pass options in a transport independent manner to the transport provider specified by *netconfig*. There are seven values for *option*:

ND_SET_BROADCAST
ND_CLEAR_BROADCAST
ND_SET_REUSEADDR
ND_CLEAR_REUSEADDR
ND_SET_RESERVEDPORT
ND_CHECK_RESERVEDPORT

ND_MERGEADDR

The specific actions of each option follow.

ND_SET_BROADCAST Sets the transport provider up to allow broadcast, if the transport supports broadcast. *fd* is a file descriptor into the transport (that is, the result of a **t_open** of **/dev/udp**). *pointer_to_args* is not used. If this completes, broadcast operations may be performed on file descriptor *fd*.

ND_CLEAR_BROADCAST
Turn off permission to send broadcast messages for the transport endpoint.

ND_SET_REUSEADDR Allow the transport provider to bind additional transport endpoints to the same local address to which another endpoint has already been bound.

ND_CLEAR_REUSEADDR
Do not allow the transport provider to bind a transport endpoint to a local address to which another endpoint has already been bound.

ND_SET_RESERVEDPORT
Allows the application to bind to a reserved port, if that concept exists for the transport provider. *fd* is a file descriptor into the transport (it must not be bound to an address). If *pointer_to_args* is **NULL**, *fd* will be bound to a reserved port. If *pointer_to_args* is a pointer to a **netbuf** structure, an attempt will be made to bind to a reserved port on the specified address.

ND_CHECK_RESERVEDPORT
Used to verify that an address corresponds to a reserved port, if that concept exists for the transport provider. *fd* is not used. *pointer_to_args* is a pointer to a **netbuf** structure that contains an address. This option returns **0** only if the address specified in *pointer_to_args* is reserved.

ND_MERGEADDR Used to take a ''local address'' and return a ''real address'' that client machines can connect to. *fd* is not used. *pointer_to_args* is a pointer to a **struct nd_mergearg**, which has the following members:

```
char *s_uaddr; /* server's universal address */
char *c_uaddr; /* client's universal address */
char *m_uaddr; /* merged universal address */
```

RETURN VALUE

The `uaddr2taddr()` and `taddr2uaddr()` routines support translation between universal addresses and TLI/XTI type netbufs. They take and return character string pointers. The `taddr2uaddr()` routine returns a pointer to a string that contains the universal address and returns NULL if the conversion is not possible. This is not a fatal condition as some transports may not suppose a universal address form.

The netdir_perror routine prints an error message on the standard output stating why one of the name-to-address mapping routines failed. The error message is preceded by the string given as an argument.

The netdir_sperror routine returns a string containing an error message stating why one of the name-to-address mapping routines failed.

USAGE

General.

SEE ALSO

getnetconfig(RS_LIB), getnetpath(RS_LIB).

LEVEL

Level 1.

NAME

publickey: getpublickey, getsecretkey – get public or secret key

SYNOPSIS

```
#include <rpc/rpc.h>
#include <rpc/key_prot.h>

getpublickey(const char netname[MAXNETNAMELEN],
     char publickey[HEXKEYBYTES]);

getsecretkey(const char netname[MAXNETNAMELEN],
     char secretkey[HEXKEYBYTES], const char *passwd);
```

DESCRIPTION

`getpublickey()` and `getsecretkey()` get public and secret keys for *netname* from the publickey database. `getsecretkey()` has an extra argument, *passwd*, which is used to decrypt the encrypted secret key stored in the database. Both routines return `1` if they are successful in finding the key, `0` otherwise. The keys are returned as NULL-terminated, hexadecimal strings. If the password supplied to `getsecretkey()` fails to decrypt the secret key, the routine will return 1 but the *secretkey* argument will be a NULL string.

SEE ALSO

publickey(RS_ENV).

LEVEL

Level 1.

NAME

rpc_clnt_auth: auth_destroy, authnone_create, authsys_create, authsys_create_default – library routines for client side remote procedure call authentication

DESCRIPTION

These routines are part of the RPC library which allows C language programs to make procedure calls on other machines across the network, with desired authentication. First, the client calls a procedure to send a data packet to the server. Upon receipt of the packet, the server calls a dispatch routine to perform the requested service, and then sends back a reply.

These routines are normally called after creating the CLIENT handle. The client's authentication information is passed to the server when the RPC call is made.

Routines

The following routines require that the header `rpc.h`. be included [see the *Remote Services Definitions* chapter for the definition of the `AUTH` data structure].

```
#include <rpc/rpc.h>
```

```
void
auth_destroy(AUTH *auth);
```

A function macro that destroys the authentication information associated with *auth*. Destruction usually involves deallocation of private data structures. The use of *auth* is undefined after calling `auth_destroy()`.

```
AUTH *
authnone_create(void);
```

Create and return an RPC authentication handle that passes nonusable authentication information with each remote procedure call. This is the default authentication used by RPC.

```
AUTH *
authsys_create(const char *host, const uid_t uid, const gid_t gid,
     const int len, const gid_t *aup_gids);
```

Create and return an RPC authentication handle that contains authentication information. The parameter *host* is the name of the machine on which the information was created; *uid* is the user's user ID; *gid* is the user's current group ID; *len* and *aup_gids* refer to a counted array of groups to which the user belongs.

```
AUTH *
authsys_create_default(void);
```

Call `authsys_create()` with the appropriate parameters.

SEE ALSO

rpc_clnt_create(RS_LIB), rpc_clnt_calls(RS_LIB).

LEVEL

Level 1.

NAME

rpc_clnt_calls: clnt_call, clnt_freeres, clnt_geterr, clnt_perrno, clnt_perror, clnt_sperrno, clnt_sperror, rpc_broadcast, rpc_broadcast_exp, rpc_call – library routines for client side calls

DESCRIPTION

RPC library routines allow C language programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a data packet to the server. Upon receipt of the packet, the server calls a dispatch routine to perform the requested service, and then sends back a reply.

The `clnt_call()`, `rpc_call()` and `rpc_broadcast()`, `rpc_broadcast_exp()` routines handle the client side of the procedure call. The remaining routines deal with error handling in the case of errors.

Routines

See the *Remote Services Definitions* chapter for the definition of the `CLIENT` data structure.

```
#include <rpc/rpc.h>
```

```
enum clnt_stat
clnt_call(CLIENT *clnt, const u_long procnum, const xdrproc_t inproc,
     caddr_t in, const xdrproc_t outproc, caddr_t out,
     const struct timeval tout);
```

A function macro that calls the remote procedure *procnum* associated with the client handle, *clnt*, which is obtained with an RPC client creation routine such as `clnt_create()` [see rpc_clnt_create(RS_LIB)]. The parameter *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s); *inproc* is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results; *tout* is the time allowed for results to be returned.

If the remote call succeeds, the status is returned in `RPC_SUCCESS`, otherwise an appropriate status is returned [see the *Remote Services Definitions* chapter for possible error numbers].

```
bool_t clnt_freeres(CLIENT *clnt, const xdrproc_t outproc, caddr_t out);
```

A function macro that frees any data allocated by the RPC/XDR system when it decoded the results of an RPC call. The parameter *out* is the address of the results, and *outproc* is the XDR routine describing the results. This routine returns `1` if the results were successfully freed, and `0` otherwise.

```
void
clnt_geterr(const CLIENT *clnt, struct rpc_err *errp);
```

A function macro that copies the error structure out of the client handle to the structure at address *errp*.

`void`
`clnt_perrno(const enum clnt_stat` *stat*);

Print a message to standard error corresponding to the condition indicated by *stat*. A NEWLINE is appended at the end of the message. Normally used after a procedure call fails, for instance `rpc_call()`.

`void`
`clnt_perror(const CLIENT *`*clnt*`, const char *`*s*`);`

Print a message to standard error indicating why an RPC call failed; *clnt* is the handle used to do the call. The message is prepended with string *s* and a colon. A NEWLINE is appended at the end of the message. Normally used after a procedure call fails, for instance `clnt_call()`.

`char *`
`clnt_sperrno(const enum clnt_stat` *stat*`);`

Take the same arguments as `clnt_perrno()`, but instead of sending a message to the standard error indicating why an RPC call failed, return a pointer to a string which contains the message.

`clnt_sperrno()` is normally used instead of `clnt_perrno()` when the program does not have a standard error (as a program running as a server quite likely does not), or if the programmer does not want the message to be output with `printf()` [see printf(BA_LIB)], or if a message format different than that supported by `clnt_perrno()` is to be used. Note: unlike `clnt_sperror()` and `clnt_spcreaterror()` [see rpc_clnt_create(RS_LIB)], `clnt_sperrno()` does not return pointer to static data so the result will not get overwritten on each call.

`char *`
`clnt_sperror(const CLIENT *`*clnt*`, const char *`*s*`);`

Like `clnt_perror()`, except that (like `clnt_sperrno()`) it returns a string instead of printing to standard error. However, `clnt_sperror()` does not append a NEWLINE at the end of the message.

Warning: returns pointer to static data that is overwritten on each call.

```
enum clnt_stat
rpc_broadcast(const u_long prognum, const u_long versnum,
     const u_long procnum, const xdrproc_t inproc, caddr_t in,
     const xdrproc_t outproc, caddr_t out, const resultproc_t eachresult,
     const char *nettype);
```

```
enum clnt_stat
rpc_broadcast_exp(const u_long prognum, const u_long versnum,
     const u_long procnum, const xdrproc_t inproc, caddr_t in,
     const xdrproc_t outproc, caddr_t out,
     const resultproc_t eachresult, int inittime
     int waittime, const char *nettype);
```

These calls are like **rpc_call**, except the call message is broadcast to the connectionless network specified by *nettype*. If *nettype* is **NULL**, it defaults to **netpath**. *rpc_broadcast* simply calls *rpc_broadcast_exp* with particular millisecond values of **inittime** and **waittime**. Each time *rpc_broadcast_exp* receives a response, it calls **eachresult**, whose form is:

```
bool_t
eachresult(const caddr_t out, const struct netbuf *addr,
     struct netconfig *netconf);
```

where *out* is the same as *out* passed to **rpc_broadcast** and **rpc_broadcast_exp** except that the remote procedure's output is decoded in **rpc_broadcast_exp**; *addr* points to the address of the machine that sent the results, and *netconf* is the netconfig structure of the transport on which the remote server responded. If **eachresult** returns **0**, **rpc_broadcast_exp** and therefore **rpc_broadcast** wait for more replies; otherwise they return with appropriate status.

Warning: broadcast file descriptors are limited in size to the maximum transfer size of that transport. For Ethernet, this value is 1500 bytes.

```
enum clnt_stat
rpc_call(const char *host, const u_long prognum,
     const u_long versnum, const u_long procnum,
     const xdrproc_t inproc, const char *in,
     const xdrproc_t outproc, char *out,
     const char *nettype);
```

Call the remote procedure associated with *prognum*, *versnum*, and *procnum* on the machine, *host*. The parameter *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s); *inproc* is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results. *nettype* can be any of the values listed in the *Remote Services Definitions* chapter. If *nettype* is NULL, it defaults to `netpath`. This routine returns `0` if it succeeds, or the value of `enum clnt_stat()` cast to an integer if it fails. Use the `clnt_perrno()` routine to translate failure statuses into messages.

Warning: `rpc_call()` uses the first available transport belonging to the class *nettype*, on which it can create a connection. You do not have control of timeouts or authentication using this routine. There is also no way to destroy the client handle.

SEE ALSO

printf(BA_LIB), rpc_clnt_auth(RS_LIB), rpc_clnt_create(RS_LIB).

LEVEL

Level 1.

NAME

rpc_clnt_create: clnt_control, clnt_create, clnt_destroy, clnt_dg_create, clnt_pcreateerror, clnt_raw_create, clnt_spcreateerror, clnt_tli_create, clnt_tp_create, clnt_vc_create – library routines for dealing with creation and manipulation of CLIENT handles

DESCRIPTION

RPC library routines allow C language programs to make procedure calls on other machines across the network. First a CLIENT handle is created and then the client calls a procedure to send a data packet to the server. Upon receipt of the packet, the server calls a dispatch routine to perform the requested service, and then sends back a reply.

Routines

See the *Remote Services Definitions* chapter for the definition of the `CLIENT` data structure.

```
#include <rpc/rpc.h>
```

```
bool_t
clnt_control(CLIENT *clnt, const u_int req, char *info);
```

A function macro used to change or retrieve various information about a client object. *req* indicates the type of operation, and *info* is a pointer to the information. For both connectionless and connection-oriented transports, the supported values of *req* and their argument types and what they do are:

`CLSET_TIMEOUT`	struct timeval	set total timeout
`CLGET_TIMEOUT`	struct timeval	get total timeout

Note: if you set the timeout using `clnt_control()`, the timeout parameter passed to `clnt_call()` will be ignored in all future calls.

`CLGET_FD`	int	get the associated file descriptor
`CLGET_SVC_ADDR`	struct netbuf	get servers address
`CLSET_FD_CLOSE`	int	close the file descriptor when destroying the client handle [see `clnt_destroy`()]
`CLSET_FD_NCLOSE`	int	do not close the file descriptor when destroying the client handle

The following operations are valid for connectionless transports only:

`CLSET_RETRY_TIMEOUT`	struct timeval	set the retry timeout
`CLGET_RETRY_TIMEOUT`	struct timeval	get the retry timeout

The retry timeout is the time that RPC waits for the server to reply before retransmitting the request.

`clnt_control()` returns `1` on success and `0` on failure.

```
CLIENT *
clnt_create(const char *host, const u_long prognum,
                     const u_long versnum, const char *nettype);
```

Generic client creation routine. *host* identifies the name of the remote host where the server is located. *nettype* indicates the class of transport protocol to use. The transports are tried in left to right order in `NETPATH` variable or in top to down order in the netconfig database.

`clnt_create()` tries all the transports of the *nettype* class available from the `NETPATH` environment variable and the the netconfig database, and chooses the first successful one. Default timeouts are set, but can be modified using `clnt_control()`.

```
void
clnt_destroy(CLIENT *clnt);
```

A function macro that destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including *clnt* itself. Use of *clnt* is undefined after calling `clnt_destroy()`. If the RPC library opened the associated file descriptor, or `CLSET_FD_CLOSE` was set using `clnt_control()`, it will be closed.

```
CLIENT *
clnt_dg_create(const int fd, const struct netbuf *svcaddr,
                     const u_long prognum, const u_long versnum,
                     const u_int sendsz, const u_int recvsz);
```

This routine creates an RPC client for the remote program *prognum* and version *versnum*; the client uses a connectionless transport. The remote program is located at address *svcaddr*. The parameter *fd* is an open and bound file descriptor. This routine will resend the call message in intervals of 15 seconds until a response is received or until the call times out. The total time for the call to time out is specified by `clnt_call()` [see `clnt_call()` in rpc_clnt_calls(RS_LIB)]. This routine returns NULL if it fails. The retry time out and the total time out periods can be changed using `clnt_control()`. The user may set the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of `0` choose suitable defaults.

```
void
clnt_pcreateerror(const char *s);
```

Print a message to standard error indicating why a client RPC handle could not be created. The message is prepended with the string *s* and a colon, and appended with a NEWLINE.

```
CLIENT *
clnt_raw_create(const u_long prognum, const u_long versnum);
```

This routine creates a toy RPC client for the remote program *prognum* and version *versnum*. The transport used to pass messages to the service is a buffer within the process's address space, so the corresponding RPC server should live in the same address space; [see `svc_raw_create()` in rpc_clnt_calls(RS_LIB)]. This allows simulation of RPC and acquisition of RPC overheads, such as round trip times, without any kernel interference. This routine returns NULL if it fails.

```
char *
clnt_spcreateerror(const char *s);
```

Like `clnt_pcreateerror()`, except that it returns a string instead of printing to the standard error. A NEWLINE is not appended to the message in this case.

Warning: returns a pointer to static data that is overwritten on each call.

```
CLIENT *
clnt_tli_create(const int fd, const struct netconfig *netconf,
                     const struct netbuf *svcaddr, u const_long prognum,
                     const u_long versnum, const u_int sendsz,
                     const u_int recvsz);
```

This routine creates an RPC client handle for the remote program *prognum* and version *versnum*. The remote program is located at address *svcaddr*. If *svcaddr* is NULL and it is connection-oriented, it is assumed that the file descriptor is connected. For connectionless transports, if *svcaddr* is NULL, `RPC_UNKNOWNADDR` error is set. *fd* is a file descriptor which may be open, bound and connected. If it is `RPC_ANYFD`, it opens a file descriptor on the transport specified by *netconf*. If *netconf* is NULL, a `RPC_UNKNOWNPROTO` error is set. If *fd* is unbound, then it will attempt to bind the descriptor. The user may specify the size of the buffers with the parameters *sendsz* and *recvsz*; values of `0` choose suitable defaults. Depending upon the type of the transport (connection-oriented or connectionless), `clnt_tli_create()` calls appropriate client creation routines. This routine returns NULL if it fails. The `clnt_pcreaterror()` routine can be used to print the reason for failure. The remote `rpcbind` service [see rpcbind(RS_CMD)] will not be consulted for the address of the remote service.

```
CLIENT *
clnt_tp_create(const char *host, const u_long prognum,
                     const u_long versnum, const struct netconfig *netconf);
```

`clnt_tp_create()` creates a client handle for the network specified by *netconf*. Default options are set, which can be changed using `clnt_control()` calls. The remote `rpcbind` service on the host *host* is consulted for the address of the remote service. This routine returns NULL if it fails. The `clnt_pcreaterror()` routine can be used to print the reason for failure.

```
CLIENT *
clnt_vc_create(const int fd, const struct netbuf *svcaddr,
               const u_long prognum, const u_long versnum,
               const u_int sendsz, const u_int recvsz);
```

This routine creates an RPC client for the remote program *prognum* and version *versnum*; the client uses a connection-oriented transport. The remote program is located at address *svcaddr*. The parameter *fd* is an open and bound file descriptor. The user may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of `0` choose suitable defaults. This routine returns NULL if it fails.

The address *svcaddr* should not be NULL and should point to the actual address of the remote program. `clnt_vc_create()` will not consult the remote `rpcbind` service for this information.

SEE ALSO

rpcbind(RS_CMD), rpc_clnt_auth(RS_LIB), rpc_clnt_calls(RS_LIB).

LEVEL

Level 1.

NAME

rpc_svc_calls: rpc_reg, svc_reg, svc_unreg, xprt_register, xprt_unregister – library routines for registering servers

DESCRIPTION

These routines are a part of the RPC library which allows the RPC servers to register themselves with `rpcbind` [see rpcbind(RS_CMD)], and it associates the given program and version number with the dispatch function.

Routines

See the *Remote Services Definitions* chapter for the definition of the `SVCXPRT` data structure.

```
#include <rpc/rpc.h>
```

```
int
rpc_reg(const u_long prognum, const u_long versnum,
     const u_long procnum, const char *(*procname)(),
     const xdrproc_t inproc, const xdrproc_t outproc,
     const char *nettype);
```

Register program *prognum*, procedure *procname*, and version *versnum* with the RPC service package. If a request arrives for program *prognum*, version *versnum*, and procedure *procnum*, *procname* is called with a pointer to its parameter(s); *procname* should return a pointer to its static result(s); *inproc* is used to decode the parameters while *outproc* is used to encode the results. Procedures are registered on all available transports of the class *nettype*. *nettype* defines a class of transports which can be used for a particular application. The transports are tried in left to right order in `NETPATH` variable or in top to down order in the netconfig database.

If *nettype* is NULL, it defaults to `netpath`. This routine returns `0` if the registration succeeded, `-1` otherwise.

```
int
svc_reg(const SVCXPRT *xprt, const u_long prognum, const u_long versnum,
     const void (*dispatch), const struct netconfig *netconf);
```

Associates *prognum* and *versnum* with the service dispatch procedure, *dispatch*. If *netconf* is NULL, the service is not registered with the `rpcbind` service. If *netconf* is non-zero, then a mapping of the triple [*prognum, versnum, netconf–>nc_netid*] to `xprt->xp_ltaddr` is established with the local `rpcbind` service.

The `svc_reg()` routine returns `1` if it succeeds, and `0` otherwise

```
void
svc_unreg(const u_long prognum, const u_long versnum);
```

Remove all mapping of the double [*prognum, versnum*] to dispatch routines, and of the triple [*prognum, versnum,* *] to network address.

```
void
xprt_register(const SVCXPRT *xprt);
```

After RPC service transport handle *xprt* is created, it is registered with the RPC service package. This routine modifies the global variable `svc_fds`. Service implementors usually do not need this routine.

```
void
xprt_unregister(const SVCXPRT *xprt);
```

Before an RPC service transport handle *xprt* is destroyed, it unregisters itself with the RPC service package. This routine modifies the global variable `svc_fds`. Service implementors usually do not need this routine.

SEE ALSO

rpcbind(RS_CMD), rpcbind(RS_LIB), rpc_svc_err(RS_LIB), rpc_svc_create(RS_LIB), rpc_svc_reg(RS_LIB).

LEVEL

Level 1.

NAME

rpc_svc_create: svc_create, svc_destroy, svc_dg_create, svc_fd_create, svc_raw_create, svc_tli_create, svc_tp_create, svc_vc_create – library routines for dealing with the creation of server handles

DESCRIPTION

These routines are part of the RPC library which allows C language programs to make procedure calls on servers across the network. First, the client calls a procedure to send a data packet to the server. Upon receipt of the packet, the server calls a dispatch routine to perform the requested service, and then sends back a reply.

Routines

See the *Remote Services Definitions* chapter for the definition of the `SVCXPRT` data structure.

```
#include <rpc/rpc.h>
```

```
int
svc_create(const void (*dispatch)(const struct svc_req *,const SVCXPRT *),
     const u_long prognum, const u_long versnum, const char *nettype);
```

`svc_create()` creates server handles for all the transports belonging to the class *nettype*.

nettype defines a class of transports which can be used for a particular application. The transports are tried in left to right order in `NETPATH` variable or in top to down order in the netconfig database.

If *nettype* is NULL, it defaults to `netpath`. `svc_create()` registers itself with the `rpcbind` service [see rpcbind(RS_CMD)]. *dispatch* is called when there is a remote procedure call for the given *prognum* and *versnum*; this requires calling `svc_run()` [see `svc_run()` in rpc_svc_reg(RS_LIB)]. If it succeeds, `svc_create()` returns the number of server handles it could create, otherwise it returns `0` and the error message is logged.

```
void
svc_destroy(SVCXPRT *xprt);
```

A function macro that destroys the RPC service transport handle *xprt*. Destruction usually involves deallocation of private data structures, including *xprt* itself. Use of *xprt* is undefined after calling this routine.

```
SVCXPRT *
svc_dg_create(const int fd, const u_int sendsz, const u_int recvsz);
```

This routine creates a connectionless RPC service handle, and returns a pointer to it. This routine returns NULL if it fails, and an error message is logged. *sendsz* and *recvsz* are parameters used to specify the size of the buffers. If they are `0`, suitable defaults are chosen. The file descriptor *fd* should be open and bound.

Warning: since connectionless-based RPC messages can only hold limited amount of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

```
SVCXPRT *
svc_fd_create(const int fd, const u_int sendsz, const u_int recvsz);
```

This routine creates a service on top of any open and bound descriptor, and returns the handle to it. Typically, this descriptor is a connected file descriptor for a stream protocol. *sendsz* and *recvsz* indicate sizes for the send and receive buffers. If they are `0`, a reasonable default is chosen. This routine returns NULL, if it fails, and an error message is logged.

```
SVCXPRT *
svc_raw_create(void);
```

This routine creates a toy RPC service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding RPC client should live in the same address space; [see `clnt_raw_create()` in rpc_clnt_create()]. This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel interference. This routine returns NULL if it fails, and an error message is logged.

```
SVCXPRT *
svc_tli_create(int fd, const struct netconfig *netconf,
     const struct t_bind *bindaddr, const u_int sendsz,
     const u_int recvsz);
```

This routine creates an RPC server handle, and returns a pointer to it. *fd* is the file descriptor on which the service is listening. If *fd* is `RPC_ANYFD`, it opens a file descriptor on the transport specified by *netconf*. If the file descriptor is unbound, it is bound to the address specified by *bindaddr*, if *bindaddr* is non-NULL, otherwise it is bound to a default address chosen by the transport. In the case where the default address is chosen, the number of outstanding connect requests is set to 8 for connection-oriented transports. The user may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of `0` choose suitable defaults. This routine returns NULL if it fails, and an error message is logged.

```
SVCPRT *
svc_tp_create(const void (*dispatch)(const struct svc_req *,
     const SVCXPRT *), const u_long prognum, const u_long versnum,
     const struct netconfig *netconf);
```

`svc_tp_create()` creates a server handle for the network specified by *netconf*, and registers itself with the `rpcbind` service. *dispatch* is called when there is a remote procedure call for the given *prognum* and *versnum*; this requires calling `svc_run()`. `svc_tp_create()` returns the service handle if it succeeds, otherwise a NULL is returned, and an error message is logged.

```
SVCXPRT *
svc_vc_create(const int fd, const u_int sendsz, const u_int recvsz);
```

This routine creates a connection-oriented RPC service and returns a pointer to it. This routine returns NULL if it fails, and an error message is logged. The users may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of 0 choose suitable defaults. The file descriptor *fd* should be open and bound.

SEE ALSO

rpcbind(RS_CMD), rpc_svc_calls(RS_LIB), rpc_svc_err(RS_LIB), rpc_svc_reg(RS_LIB).

LEVEL

Level 1.

NAME

rpc_svc_err: svcerr_auth, svcerr_decode, svcerr_noproc, svcerr_noprog, svcerr_progvers, svcerr_systemerr, svcerr_weakauth – library routines for server side remote procedure call errors

DESCRIPTION

These routines are part of the RPC library which allows C language programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a data packet to the server. Upon receipt of the packet, the server calls a dispatch routine to perform the requested service, and then sends back a reply.

These routines can be called by the server side dispatch function if there is any error in the transaction with the client.

Routines

See the *Remote Services Definitions* chapter for the definition of the `SVCXPRT` data structure.

`#include <rpc/rpc.h>`

`void`
`svcerr_auth(const SVCXPRT *`*xprt*`, const enum auth_stat `*why*`);`

> Called by a service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

`void`
`svcerr_decode(const SVCXPRT *`*xprt*`);`

> Called by a service dispatch routine that cannot successfully decode the remote parameters [see `svc_getargs()` in rpc_svc_reg(RS_LIB)].

`void`
`svcerr_noproc(const SVCXPRT *`*xprt*`);`

> Called by a service dispatch routine that does not implement the procedure number that the caller requests.

`void`
`svcerr_noprog(const SVCXPRT *`*xprt*`);`

> Called when the desired program is not registered with the RPC package. Service implementors usually do not need this routine.

`void`
`svcerr_progvers(const SVCXPRT *`*xprt*`);`

> Called when the desired version of a program is not registered with the RPC package. Service implementors usually do not need this routine.

`void`
`svcerr_systemerr(const SVCXPRT *`*xprt*`);`

> Called by a service dispatch routine when it detects a system error not covered by any particular protocol. For example, if a service can no longer allocate storage, it may call this routine.

```
void
svcerr_weakauth(const SVCXPRT *xprt);
```

Called by a service dispatch routine that refuses to perform a remote procedure call due to insufficient (but correct) authentication parameters. The routine calls `svcerr_auth(xprt, AUTH_TOOWEAK)`.

SEE ALSO

rpc_svc_calls(RS_LIB), rpc_svc_create(RS_LIB), rpc_svc_reg(RS_LIB).

LEVEL

Level 1.

NAME

rpc_svc_reg: svc_freeargs, svc_getargs, svc_getreqset, svc_getrpccaller, svc_run, svc_sendreply, svc_getreq_common, svc_getreq_poll, svc_getreq_poll_parallel, svc_run_parallel – library routines for RPC servers

DESCRIPTION

These routines are part of the RPC library which allows C language programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a data packet to the server. Upon receipt of the packet, the server calls a dispatch routine to perform the requested service, and then sends back a reply.

These routines are associated with the server side of the RPC mechanism. Some of them are called by the server side dispatch function, while others [such as `svc_run()`] are called when the server is initiated.

Routines

See the *Remote Services Definitions* chapter for the definition of the `SVCXPRT` data structure.

`#include <rpc/rpc.h>`

`bool_t`
`svc_freeargs(const SVCXPRT *`*xprt*`, const xdrproc_t `*inproc*`, char *`*in*`);`

A function macro that frees any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using `svc_getargs()`. This routine returns `1` if the results were successfully freed, and `0` otherwise.

`bool_t`
`svc_getargs(const SVCXPRT *`*xprt*`, const xdrproc_t `*inproc*`, char *`*in*`);`

A function macro that decodes the arguments of an RPC request associated with the RPC service transport handle *xprt*. The parameter *in* is the address where the arguments will be placed; *inproc* is the XDR routine used to decode the arguments. This routine returns `1` if decoding succeeds, and `0` otherwise.

`void`
`svc_getreqset(fd_set *`*rdfds*`);`

This routine is only of interest if a service implementor does not call `svc_run()`, but instead implements custom asynchronous event processing. It is called when `poll()` has determined that an RPC request has arrived on some RPC file descriptors; *rdfds* is the resultant read file descriptor bit mask. The routine returns when all file descriptors associated with the value of *rdfds* have been serviced

```
struct netbuf *
svc_getrpccaller(const SVCXPRT *xprt);
```

The approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle *xprt*.

```
void
svc_run(void);
```

This routine never returns. It waits for RPC requests to arrive, and calls the appropriate service procedure using `svc_getreqset()` when one arrives. This procedure is usually waiting for a `poll()` library call to return.

```
bool_t
svc_sendreply(const SVCXPRT *xprt, const xdrproc_t outproc,
     const caddr_t out);
```

Called by an RPC service's dispatch routine to send the results of a remote procedure call. The parameter *xprt* is the request's associated transport handle; *outproc* is the XDR routine which is used to encode the results; and *out* is the address of the results. This routine returns `1` if it succeeds, `0` otherwise.

```
#include <sys/poll.h>

void
svc_getreq_common(int fd)
```

This routine processes incoming RPC requests on a file descriptor specified by *fd*. All higher level service implementations like **svc_run**, **svc_getreqset**, and **svc_getreq_poll** use this routine to process RPC requests.

This routine authenticates incoming RPC requests on the file descriptor *fd* and calls the appropriate dispatch routine registered with **rpcbind**. If the transport provider is connection-oriented, the succeeding requests, if any, are processed repeatedly. This is called batched Remote Procedure Calls.

Note that this routine is thread-safe. However, a different file descriptor must be specified in each concurrent call to **svc_getreq_common**.

```
#include <sys/poll.h>

void
svc_getreq_poll(struct pollfd *pfdp, int retval);
```

Like **svc_getreqset**, this routine is only of interest if a service implementor does not call **svc_run**, but instead implements custom asynchronous event processing. The **svc_run** routine provided in the RPC library is currently implemented using this routine.

It should be called when **poll** has determined that an RPC request has arrived on some RPC file descriptors; *pfdp* is the poll data used during poll, and *retval* is the number of file descriptors to service, typically the return value from poll. The routine returns when all file descriptors specified by *pfdp* have been serviced.

Note that this routine is not thread-safe. Hence the service implementor must use appropriate synchronization to avoid calls to this routine from multiple threads at the same time.

```
#include <sys/poll.h>
```

```
void
svc_getreq_poll_parallel(struct pollfd *pfdp, int retval);
```

This routine is the thread-safe version of **svc_getreq_poll** and provides exactly the same functionality.

```
svc_run_parallel(int timeout, int minthreads, int maxthreads);
```

This is the multithreaded version of svc_run. This routine waits for RPC requests to arrive, and calls the appropriate service procedure via a call to svc_getreq_poll_parallel. Depending on the rate of incoming RPC requests, this routine will dynamically create or delete threads from the process. Each created thread services an RPC request and then waits for more to arrive.

The *timeout* argument specifies the number of milliseconds to wait for and RPC request to arrive. After waiting for this time, any thread created by **svc_run_parallel** will exit, provided the total number of threads is above *minthreads*. The maximum number of threads created by this routine is always less than *maxthreads*.

Note that this routine provides a performance gain for server processes which service a sustained rate of incoming RPC requests. Also, the service procedure may be called concurrently from many server threads, so it must be thread-safe. Currently, it is only supported for connectionless transports.

This routine returns -1 if either of *minthreads* or *maxthreads* is less than or equal to zero. It also returns -1 if *maxthreads* is is less than or equal to *minthreads*.

It returns zero if there are no file server file descriptors to wait on.

SEE ALSO

poll(BA_OS), rpc_svc_calls(RS_LIB), rpc_svc_create(RS_LIB), rpc_svc_err(RS_LIB).

LEVEL

Level 1.

NAME

rpc_xdr: xdr_accepted_reply, xdr_authsys_parms, xdr_callhdr, xdr_callmsg, xdr_opaque_auth, xdr_rejected_reply, xdr_replymsg – XDR library routines for remote procedure calls

DESCRIPTION

These routines are used for describing the RPC messages in XDR language. They should normally be used by those who do not want to use the RPC package.

Routines

See the *Remote Services Definitions* chapter for the definition of the XDR data structure.

```
#include <rpc/rpc.h>
```

```
bool_t
xdr_accepted_reply(XDR *xdrs, const struct accepted_reply *ar);
```

Used for encoding RPC reply messages. It encodes the status of the RPC call in the XDR language format, and in the case of success, it encodes the call results also.

```
bool_t
xdr_authsys_parms(XDR *xdrs, const struct authsys_parms *aupp);
```

Used for describing operating system credentials. It includes machine-name, uid, gid list, etc.

```
void
xdr_callhdr(XDR *xdrs, const struct rpc_msg *chdr);
```

Used for describing RPC call header messages. It encodes the static part of the call message header in the XDR language format. It includes information such as transaction ID, RPC version number, program and version number.

```
bool_t
xdr_callmsg(XDR *xdrs, const struct rpc_msg *cmsg);
```

Used for describing RPC call messages. This includes all the RPC call information such as transaction ID, RPC version number, program number, version number, authentication information, etc. This is normally used by servers to determine information about the client RPC call.

```
bool_t
xdr_opaque_auth(XDR *xdrs, const struct opaque_auth *ap);
```

Used for describing RPC opaque authentication information messages.

```
bool_t
xdr_rejected_reply(XDR *xdrs, const struct rejected_reply *rr);
```

Used for describing RPC reply messages. It encodes the rejected RPC message in the XDR language format. The message could be rejected either because of version number mis-match or because of authentication errors.

```
bool_t
xdr_replymsg(XDR *xdrs, const struct rpc_msg *rmsg);
```

Used for describing RPC reply messages. It encodes all the RPC reply message in the XDR language format This reply could be either an acceptance, rejection or NULL.

LEVEL

Level 1.

NAME

rpcbind: rpcb_getmaps, rpcb_getaddr, rpcb_gettime, rpcb_rmtcall, rpcb_set, rpcb_unset – library routines for RPC bind service.

DESCRIPTION

These routines allow client C programs to make procedure calls to the RPC binder service. `rpcbind` [see rpcbind(RS_CMD)] maintains a list of mappings between programs and their universal addresses.

Routines

```
#include <rpc/rpc.h>
```

```
struct rpcblist *
rpcb_getmaps(const struct netconfig *netconf, const char *host);
```

A user interface to the `rpcbind` service, which returns a list of the current RPC program-to-address mappings on the host named. It uses the transport specified through *netconf* to contact the remote `rpcbind` service on host *host*. This routine will return NULL, if the remote `rpcbind` could not be contacted. The command `rpcinfo` [see rpcinfo(RS_CMD)] uses this routine.

```
bool_t
rpcb_getaddr(const u_long prognum, const u_long versnum,
     const struct netconfig *netconf, struct netbuf *svcaddr,
     const char *host);
```

A user interface to the `rpcbind` service, which finds the address of the service on *host* that is registered with program number *prognum*, version *versnum*, and speaks the transport protocol associated with *netconf*. The address found is returned in *svcaddr*. *svcaddr* should be preallocated. This routine returns `1` if it succeeds. A return value of `0` means that the mapping does not exist or that the RPC system failed to contact the remote `rpcbind` service. In the latter case, the global variable `rpc_createerr` contains the RPC status.

```
bool_t
rpcb_gettime(const char *host, time_t *timep);
```

This routine returns the time on *host* in *timep*. If *host* is NULL, `rpcb_gettime()` returns the time on its own machine. This routine returns 1 if it succeeds, 0 if it fails. `rpcb_gettime` can be used to synchronize the time between the client and the remote server. This routine is particularly useful for secure RPC.

```
enum clnt_stat
rpcb_rmtcall(const struct netconfig *netconf, const char *host,
     u_long prognum, u_long versnum, u_long procnum,
     xdrproc_t inproc, caddr_t in, xdrproc_t outproc,
        caddr_t out, struct timeval tout,
     struct netbuf  *svcaddr);
```

A user interface to the `rpcbind` service, which instructs `rpcbind` on *host* to make an RPC call on your behalf to a procedure on that host. The parameter **svcaddr* will be modified to the server's address if the procedure succeeds [see `rpc_call()` and `clnt_call()` in rpc_clnt_calls(RS_LIB) for the definitions of other parameters]. This procedure should be used for a ping and nothing else [see `rpc_broadcast()` in rpc_clnt_calls(RS_LIB)]. This routine allows programs to do lookup and call, all in one step.

```
bool_t
rpcb_set(const u_long prognum, const u_long versnum,
     const struct netconfig *netconf, const struct netbuf *svcaddr);
```

A user interface to the `rpcbind` service, which establishes a mapping between the triple [*prognum, versnum, netconf->nc_netid*] and *svcaddr* on the machine's `rpcbind` service. The value of *transport* must correspond to a network token that is defined by the netconfig database. This routine returns `1` if it succeeds, `0` otherwise, and is automatically performed by `svc_reg()` [see `svc_reg()` in rpc_svc_calls(RS_LIB)].

```
bool_t
rpcb_unset(const u_long prognum, const u_long versnum,
     const struct netconfig *netconf);
```

A user interface to the `rpcbind` service, which destroys all mapping between the triple [*prognum, versnum, netconf->nc_netid*] and the address on the machine's `rpcbind` service. If *netconf* is NULL, `rpcb_unset()` destroys all mapping between the triple [*prognum, versnum, **] and the addresses on the machine's `rpcbind` service. rpcb_unset will return `1` if the registered entry was previously unset or was not found. This routine returns `1` if it succeeds, `0` otherwise.

USAGE

General.

SEE ALSO

rpc_clnt_calls(RS_LIB), rpc_svc_calls(RS_LIB), rpcbind(RS_CMD), rpcinfo(RS_CMD).

LEVEL

Level 1.

NAME

secure_rpc: authdes_seccreate, authdes_getucred, getnetname, host2netname, key_decryptsession, key_encryptsession, key_gendes, key_setsecret, netname2host, netname2user, user2netname – library routines for secure remote procedure calls

DESCRIPTION

RPC library routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a data packet to the server. Upon receipt of the packet, the server calls a dispatch routine to perform the requested service, and then sends back a reply.

RPC allows various authentication flavors [see the *Remote Services Introduction* chapter]. The `authdes_getucred()` and `authdes_seccreate()` routines implement the DES authentication flavor. The keyserver daemon `keyserv` [see keyserv(RS_CMD)] must be running for the DES authentication system to work.

Routines

```
#include <rpc/rpc.h>
```

```
int
authdes_getucred(const struct authdes_cred *adc, uid_t *uidp,
     gid_t *gidp, short *gidlenp, int *gidlist);
```

`authdes_getucred()` is the first of the two routines which interface to the RPC secure authentication system known as DES. The second is `authdes_seccreate()`, below. `authdes_getucred()` is used on the server side for converting a DES credential, which is operating system independent, into a UNIX credential. This routine returns `1` if it succeeds, `0` if it fails.

**uidp* is set to the user's numerical ID associated with *adc*. **gidp* is set to the numerical ID of the group to which the user belongs. *gidlist* contains the numerical IDs of the other groups to which the user belongs. **gidlenp* is set to the number of valid group ID entries in *gidlist* [see `netname2user()`, below].

```
AUTH *
authdes_seccreate(const char *name, const unsigned int window,
     struct netbuf *syncaddr, const des_block *ckey);
```

`authdes_seccreate()`, the second of two DES authentication routines, is used on the client side to return an authentication handle that will enable the use of the secure authentication system. The first parameter *name* is the network name, or *netname*, of the owner of the server process. This field usually represents a hostname derived from the utility routine `host2netname()`, but could also represent a user name using `user2netname()`. The second field is window on the validity of the client credential, given in seconds. A small window is more secure than a large one, but choosing too small of a window will increase the frequency of resynchronizations because of clock drift. The third parameter, *syncaddr*, the host's address, is optional. If it is NULL, then the authentication system will assume that the local clock is always in sync with the *syncaddr* clock, and will not attempt resynchronizations. If an address is supplied, however, then the system will use the address for consulting the remote time service whenever resynchronization is required. This parameter is usually the address of the RPC server itself. The final parameter *ckey* is also optional. If it is NULL, then the authentication system will generate a random DES key to be used for the encryption of credentials. If *ckey* is supplied, then it will be used instead.

```
int
getnetname(char name[MAXNETNAMELEN+1]);
```

`getnetname()` installs the unique, operating-system independent netname of the caller in the fixed-length array *name*. Returns `1` if it succeeds, and `0` if it fails.

```
int
host2netname(char name[MAXNETNAMELEN+1], const char *host,
     const char *domain);
```

Convert from a domain-specific hostname *host* to an operating-system independent netname. Return `1` if it succeeds, and `0` if it fails. Inverse of `netname2host()`. If *domain* is NULL, `getnetname()` uses the default domain name of the machine. If *host* is NULL, it defaults to that machine itself.

```
int
key_decryptsession(const char *remotename, des_block *deskey);
```

`key_decryptsession()` is an interface to the keyserver daemon, which is associated with RPC's secure authentication system (DES authentication). User programs rarely need to call it, or its associated routines `key_encryptsession()`, `key_gendes()` and `key_setsecret()`.

`key_decryptsession()` takes a server netname *remotename* and a DES key *deskey*, and decrypts the key by using the the public key of the the server and the secret key associated with the effective UID of the calling process. It is the inverse of `key_encryptsession()`.

```
int
key_encryptsession(const char *remotename, des_block *deskey);
```

`key_encryptsession()` is a keyserver interface routine. It takes a server netname *remotename* and a DES key *deskey*, and encrypts it using the public key of the the server and the secret key associated with the effective UID of the calling process. It is the inverse of `key_decryptsession()`. This routine returns `0` if it succeeds, `-1` if it fails.

```
int
key_gendes(des_block *deskey);
```

`key_gendes()` is a keyserver interface routine. It is used to ask the keyserver for a secure conversation key. Choosing one at random is usually not good enough, because the common ways of choosing random numbers, such as using the current time, are very easy to guess.

`int`
`key_setsecret(const char *`*key);*

`key_setsecret()` is a keyserver interface routine. It is used to set the key for the effective UID of the calling process. this routine returns `0` if it succeeds, `-1` if it fails.

`int`
`netname2host(const char *`*name*`, char *`*host*`, const int` *hostlen*`);`

Convert from an operating-system independent netname *name* to a domain-specific hostname *host*. *hostlen* is the maximum size of *host*. Returns `1` if it succeeds, and `0` if it fails. Inverse of `host2netname()`.

`int`
`netname2user(const char *`*name*`, uid_t *`*uidp*`, gid_t *`*gidp*`,`
`int *`*gidlenp*`, gid_t` *gidlist*`[NGROUPS]);`

Convert from an operating-system independent netname to a domain-specific user ID. Returns `1` if it succeeds, and `0` if it fails. Inverse of `user2netname()`.

**uidp* is set to the user's numerical ID associated with *name*. **gidp* is set to the numerical ID of the group to which the user belongs. *gidlist* contains the numerical IDs of the other groups to which the user belongs. **gidlenp* is set to the number of valid group ID entries in *gidlist*.

`int`
`user2netname(char` *name*`[MAXNETNAMELEN+1], const uid_t` *uid*`,`
`const char *`*domain*`);`

Convert from a domain-specific username to an operating-system independent netname. Returns `1` if it succeeds, and `0` if it fails. Inverse of `netname2user()`.

SEE ALSO

chkey(RS_CMD), keyserv(RS_CMD), newkey(RS_CMD), rpc_clnt_auth(RS_LIB).

LEVEL

Level 1.

NAME

xdr_admin: xdr_getpos, xdr_inline, xdrrec_endofrecord, xdrrec_eof, xdrrec_skiprecord, xdr_setpos – library routines for external data representation

DESCRIPTION

XDR library routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Protocols such as remote procedure calls (RPC) use these routines to describe the format of the data.

These routines deal specifically with the management of the XDR stream.

Routines

See the *Remote Services Definitions* chapter for the definition of the `XDR` data structure.

```
#include <rpc/xdr.h>
```

```
u_int
xdr_getpos(const XDR *xdrs);
```

A macro that invokes the get-position routine associated with the XDR stream, *xdrs*. The routine returns an unsigned integer, which indicates the position of the XDR byte stream. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances need not guarantee this. Therefore, applications written for portability should not depend on this feature.

```
long *
xdr_inline(XDR *xdrs; const int len);
```

A macro that invokes the in-line routine associated with the XDR stream, *xdrs*. The routine returns a pointer to a contiguous piece of the stream's buffer; *len* is the byte length of the desired buffer. Note: pointer is cast to `long *`.

Warning: `xdr_inline()` may return NULL (`0`) if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency, and applications written for portability should not depend on this feature.

```
bool_t
xdrrec_endofrecord(XDR *xdrs; int sendnow);
```

This routine can be invoked only on streams created by `xdrrec_create()`. The data in the output buffer is marked as a completed record, and the output buffer is optionally written out if *sendnow* is non-zero. This routine returns `1` if it succeeds, `0` otherwise.

```
bool_t
xdrrec_eof(XDR *xdrs);
```

This routine can be invoked only on streams created by `xdrrec_create()`. After consuming the rest of the current record in the stream, this routine returns `1` if the stream has no more input, `0` otherwise.

```
bool_t
xdrrec_skiprecord(XDR *xdrs);
```

This routine can be invoked only on streams created by `xdrrec_create()`. It tells the XDR implementation that the rest of the current record in the stream's input buffer should be discarded. This routine returns `1` if it succeeds, `0` otherwise.

```
bool_t
xdr_setpos(XDR *xdrs, const u_int pos);
```

A macro that invokes the set position routine associated with the XDR stream *xdrs*. The parameter *pos* is a position value obtained from `xdr_getpos()`. This routine returns `1` if the XDR stream was repositioned, and `0` otherwise.

Warning: it is difficult to reposition some types of XDR streams, so this routine may fail with one type of stream and succeed with another. Therefore, applications written for portability should not depend on this feature.

SEE ALSO

xdr_complex(RS_LIB), xdr_create(RS_LIB), xdr_simple(RS_LIB).

LEVEL

Level 1.

NAME

xdr_complex: xdr_array, xdr_bytes, xdr_opaque, xdr_pointer, xdr_reference, xdr_string, xdr_union, xdr_vector, xdr_wrapstring – library routines for external data representation

DESCRIPTION

XDR library routines allow C programmers to describe complex data structures in a machine-independent fashion. Protocols such as remote procedure calls (RPC) use these routines to describe the format of the data.

Routines

See the *Remote Services Definitions* chapter for the definition of the `XDR` data structure.

```
#include <rpc/xdr.h>
```

```
bool_t
xdr_array(XDR *xdrs, caddr_t *arrp, u_int *sizep,
     const u_int maxsize, const u_int elsize, const xdrproc_t elproc);
```

`xdr_array()` translates between variable-length arrays and their corresponding external representations. The parameter *arrp* is a pointer to the array, while *sizep* is the address of the element count of the array; this element count cannot exceed *maxsize*. The parameter *elsize* is the `sizeof` each of the array's elements, and *elproc* is an XDR routine that translates between the array elements' C form and their external representation. This routine returns `1` if it succeeds, `0` otherwise.

```
bool_t
xdr_bytes(XDR *xdrs, char **sp, u_int *sizep,
     const u_int maxsize);
```

`xdr_bytes()` translates between counted byte strings and their external representations. The parameter *sp* is the address of the string pointer. The length of the string is located at address *sizep*; strings cannot be longer than *maxsize*. This routine returns `1` if it succeeds, `0` otherwise.

```
bool_t
xdr_opaque(XDR *xdrs, caddr_t cp, const u_int cnt);
```

`xdr_opaque()` translates between fixed size opaque data and its external representation. The parameter *cp* is the address of the opaque object, and *cnt* is its size in bytes. This routine returns `1` if it succeeds, `0` otherwise.

```
bool_t
xdr_pointer(XDR *xdrs, char **objpp, u_int objsize,
     const xdrproc_t xdrobj);
```

Like `xdr_reference()` except that it serializes NULL pointers, whereas `xdr_reference()` does not. Thus, `xdr_pointer()` can represent recursive data structures, such as binary trees or linked lists.

```
bool_t
xdr_reference(XDR *xdrs, caddr_t *pp, u_int size,
     const xdrproc_t proc);
```

`xdr_reference()` provides pointer chasing within structures. The parameter *pp* is the address of the pointer; *size* is the `sizeof` the structure that **pp* points to; and *proc* is an XDR procedure that translates the structure between its C form and its external representation. This routine returns `1` if it succeeds, `0` otherwise.

Warning: this routine does not understand NULL pointers. Use `xdr_pointer()` instead.

```
bool_t
xdr_string(XDR *xdrs, char **sp, const u_int maxsize);
```

`xdr_string()` translates between C strings and their corresponding external representations. Strings cannot be longer than *maxsize*. Note: *sp* is the address of the string's pointer. This routine returns `1` if it succeeds, `0` otherwise.

```
bool_t
xdr_union(XDR *xdrs, enum_t *dscmp, char *unp,
     const struct xdr_discrim *choices,
     const bool_t (*defaultarm)(const XDR *, const char *, const int));
```

`xdr_union()` translates between a discriminated C `union` and its corresponding external representation. It first translates the discriminant of the union located at *dscmp*. This discriminant is always an `enum_t`. Next the union located at *unp* is translated. The parameter *choices* is a pointer to an array of `xdr_discrim()` structures. Each structure contains an ordered pair of [*value, proc*]. If the union's discriminant is equal to the associated *value*, then the *proc* is called to translate the union. The end of the `xdr_discrim()` structure array is denoted by a routine of value NULL. If the discriminant is not found in the *choices* array, then the *defaultarm* procedure is called (if it is not NULL). Returns 1 if it succeeds, 0 otherwise.

```
bool_t
xdr_vector(XDR *xdrs, char *arrp, const u_int size,
     const u_int elsize, const xdrproc_t elproc);
```

`xdr_vector()` translates between fixed-length arrays and their corresponding external representations. The parameter *arrp* is a pointer to the array, while *size* is is the element count of the array. The parameter *elsize* is the `sizeof` each of the array's elements, and *elproc* is an XDR routine that translates between the array elements' C form and their external representation. This routine returns `1` if it succeeds, `0` otherwise.

```
bool_t
xdr_wrapstring(XDR *xdrs, char **sp);
```

A routine that calls `xdr_string(`*xdrs*`,` *sp*`,` *maxuint*`);` where *maxuint* is the maximum value of an unsigned integer.

Many routines, such as `xdr_array()`, `xdr_pointer()` and `xdr_vector()` take a function pointer of type `xdrproc_t`, which takes two arguments. `xdr_string()`, one of the most frequently used routines, requires three arguments, while `xdr_wrapstring()` only requires two. For these routines, `xdr_wrapstring()` is desirable. This routine returns `1` if it succeeds, `0` otherwise.

SEE ALSO

xdr_admin(RS_LIB), xdr_create(RS_LIB), xdr_simple(RS_LIB).

LEVEL

Level 1.

NAME

xdr_create: xdr_destroy, xdrmem_create, xdrrec_create, xdrstdio_create – library routines for external data representation stream creation

DESCRIPTION

XDR library routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Protocols such as remote procedure calls (RPC) use these routines to describe the format of the data.

These routines deal with the creation of XDR streams. XDR streams have to be created before any data can be translated into XDR format.

Routines

See the *Remote Services Definitions* chapter for the definition of the `XDR`, `CLIENT`, and `SVCXPRT` data structures.

```
#include <rpc/xdr.h>
```

```
void
xdr_destroy(XDR *xdrs);
```

A macro that invokes the destroy routine associated with the XDR stream, *xdrs*. Destruction usually involves freeing private data structures associated with the stream. Using *xdrs* after invoking `xdr_destroy()` is undefined.

```
void
xdrmem_create(XDR *xdrs, const caddr_t addr,
      const u_int size, const enum xdr_op op);
```

This routine initializes the XDR stream object pointed to by *xdrs*. The stream's data is written to, or read from, a chunk of memory at location *addr* whose length is no more than *size* bytes long. The *op* determines the direction of the XDR stream (either `XDR_ENCODE`, `XDR_DECODE`, or `XDR_FREE`).

```
void
xdrrec_create(XDR *xdrs, const u_int sendsz,
      const u_int recvsz, const caddr_t handle,
      const int (*readit(const void *, char *, const int),
      const int (*writeit)(const void *, const char *, const int));
```

This routine initializes the XDR stream object pointed to by *xdrs*. The stream's data is written to a buffer of size *sendsz*; a value of `0` indicates the system should use a suitable default. The stream's data is read from a buffer of size *recvsz*; it too can be set to a suitable default by passing a `0` value. When a stream's output buffer is full, *writeit* is called. Similarly, when a stream's input buffer is empty, *readit* is called. The behavior of these two routines is similar to the system calls `read()` and `write()` [see read(BA_OS) and write(BA_OS), respectively], except that *handle* (`CLIENT`, or `SVCXPRT`) is passed to the former routines as the first parameter instead of a file descriptor. Note: the XDR stream's *op* field must be set by the caller.

Warning: this XDR stream implements an intermediate record stream. Therefore there are additional bytes in the stream to provide record boundary information.

```
void
xdrstdio_create(XDR *xdrs, FILE *file, const enum xdr_op op);
```

This routine initializes the XDR stream object pointed to by *xdrs*. The XDR stream data is written to, or read from, the standard I/O stream *file*. The parameter *op* determines the direction of the XDR stream (either `XDR_ENCODE`, `XDR_DECODE`, or `XDR_FREE`).

Warning: the destroy routine associated with such XDR streams calls `fflush()` on the *file* stream, but never `fclose()` [see fclose(BA_OS)].

SEE ALSO

fclose(BA_OS), read(BA_OS), write(BA_OS), xdr_admin(RS_LIB), xdr_complex(RS_LIB), xdr_simple(RS_LIB).

LEVEL

Level 1.

NAME

xdr_simple: xdr_bool, xdr_char, xdr_double, xdr_enum, xdr_float, xdr_free, xdr_int, xdr_long, xdr_short, xdr_u_char, xdr_u_int, xdr_u_long, xdr_u_short, xdr_void – library routines for external data representation

DESCRIPTION

XDR library routines allow C programmers to describe simple data structures in a machine-independent fashion. Protocols such as remote procedure calls (RPC) use these routines to describe the format of the data.

These routines require the creation of XDR streams [see xdr_create(RS_LIB)].

Routines

See the *Remote Services Definitions* chapter for the definition of the `XDR` data structure.

`#include <rpc/xdr.h>`

`bool_t`
`xdr_bool(XDR *`*xdrs*`, bool_t *`*bp*`);`

> `xdr_bool()` translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either `1` or `0`. This routine returns `1` if it succeeds, `0` otherwise.

`bool_t`
`xdr_char(XDR *`*xdrs*`, char *`*cp*`);`

> `xdr_char()` translates between C characters and their external representations. This routine returns `1` if it succeeds, `0` otherwise. Note: encoded characters are not packed, and occupy 4 bytes each. For arrays of characters, it is worthwhile to consider `xdr_bytes()`, `xdr_opaque()` or `xdr_string()` [see `xdr_bytes()`, `xdr_opaque()` and `xdr_string()` in xdr_complex(RS_LIB)].

`bool_t`
`xdr_double(XDR *`*xdrs*`, double *`*dp*`);`

> `xdr_double()` translates between C `double` precision numbers and their external representations. This routine returns `1` if it succeeds, `0` otherwise.

`bool_t`
`xdr_enum(XDR *`*xdrs*`, enum_t *`*ep*`);`

> `xdr_enum()` translates between C `enums` (actually integers) and their external representations. This routine returns `1` if it succeeds, `0` otherwise.

`void`
`xdr_free(xdrproc_t `*proc*`, char *`*objp*`);`

> Generic freeing routine. The first argument is the XDR routine for the object being freed. The second argument is a pointer to the object itself. Note: the pointer passed to this routine is *not* freed, but what it points to *is* freed (recursively).

```
bool_t
xdr_float(XDR *xdrs, float *fp);
```

`xdr_float()` translates between C `float`s and their external representations. This routine returns `1` if it succeeds, `0` otherwise.

```
bool_t
xdr_int(XDR *xdrs, int *ip);
```

`xdr_int()` translates between C integers and their external representations. This routine returns `1` if it succeeds, `0` otherwise.

```
bool_t
xdr_long(XDR *xdrs, long *lp);
```

`xdr_long()` translates between C `long` integers and their external representations. This routine returns `1` if it succeeds, `0` otherwise.

```
bool_t
xdr_short(XDR *xdrs, short *sp);
```

`xdr_short()` translates between C `short` integers and their external representations. This routine returns `1` if it succeeds, `0` otherwise.

```
bool_t
xdr_u_char(XDR *xdrs, char *ucp);
```

`xdr_u_char()` translates between `unsigned` C characters and their external representations. This routine returns `1` if it succeeds, `0` otherwise.

```
bool_t
xdr_u_int(XDR *xdrs, unsigned int *up);
```

`xdr_u_int()` translates between C `unsigned` integers and their external representations. This routine returns `1` if it succeeds, `0` otherwise.

```
bool_t
xdr_u_long(XDR *xdrs, unsigned long *ulp);
```

`xdr_u_long()` translates between C `unsigned long` integers and their external representations. This routine returns `1` if it succeeds, `0` otherwise.

```
bool_t
xdr_u_short(XDR *xdrs, unsigned short *usp);
```

`xdr_u_short()` translates between C `unsigned short` integers and their external representations. This routine returns `1` if it succeeds, `0` otherwise.

```
bool_t
xdr_void(void);
```

This routine always returns `1`. It may be passed to RPC routines that require a function parameter, where nothing is to be done.

SEE ALSO

xdr_admin(RS_LIB), xdr_complex(RS_LIB), xdr_create(RS_LIB).

LEVEL

Level 1.

Remote Services Commands And Utilities

The following section contains the manual pages for the RS_CMD routines.

NAME

chkey – change your encryption key

SYNOPSIS

`chkey`

DESCRIPTION

The `chkey` command prompts the user for a password, and uses it to encrypt a new encryption key for the user to be stored in the publickey database [see publickey(RS_ENV)].

SEE ALSO

keylogin(RS_CMD), keyserv(RS_CMD), newkey(RS_CMD), publickey(RS_ENV).

LEVEL

Level 1.

NAME

dfmounts – display mounted resource information

SYNOPSIS

`dfmounts` [`-F` *fstype*] [`-h`] [`-o` *specific_options*] [*restriction* ...]

DESCRIPTION

The `dfmounts` command shows the resources shared through a distributed file system *fstype* along with a list of clients that have the resource mounted. If no arguments are given, then information is displayed about the clients that have mounted each local resource via any distributed file system type. If just `-F` *fstype* is given, then only information for that *fstype* is displayed. If one or more *restriction*s are given, `dfmounts` shows the resources that satisfy any of the *restriction*s. If the `-F` flag is omitted, and one or more *restriction*s are given, the file system in the first line of `/etc/dfs/fstypes` is used as the default. The *specific_options*, as well as the availability and semantics of *restriction*, are specific to particular distributed file systems.

The output of `dfmounts` consists of an optional header line (suppressed with the `-h` flag) followed by a list of lines containing whitespace separated fields. For each resource, the first four fields are:

resource server pathname clients

where

resource	specifies the resource name that was given to the `mount` command.
server	specifies the system from which the resource was mounted.
pathname	specifies the pathname that was given to the `share` command.
clients	lists the systems, comma-separated, by which the resource was mounted.

A field may be null. Each null field is indicated by a hyphen (-) unless the remainder of the fields on the line are also null, in which case it may be omitted. Any fields containing whitespace are enclosed in quotes.

ERRORS

If a *restriction* name is invalid, an error message will be sent to standard error.

USAGE

Administrator.

SEE ALSO

fumount(RS_CMD), dfshares(RS_CMD), mount(AS_CMD), share(RS_CMD), unshare(RS_CMD)

LEVEL

Level 1.

NAME

dfshares – list available resources from remote systems

SYNOPSIS

`dfshares` [`-F` *fstype*] [`-h`] [`-o` *specific_options*] [*server* ...]

DESCRIPTION

The `dfshares` command provides information about resources available to the host through a distributed file system of type *fstype*. If the command is given with no arguments, information about resources available through each distributed file system shall be displayed. If the command is given with just `-F` *fstype* as the argument, then only information for that *fstype* is displayed. If one or more *server*s are given, then `dfshares` shows information about resources shared by those *server*s. If the `-F` flag is omitted, and one or more *server*s are given, then the file system in the first line of `/etc/dfs/fstypes` is used as the default. The *specific_options* as well as the syntax of *server* are specific to particular distributed file systems.

The output of `dfshares` consists of an optional header line (suppressed with the `-h` flag) followed by a list of lines containing whitespace separated fields. For each resource, the first five fields are:

resource server access transport description

where

resource specifies the resource name that must be given to the `mount` command [see mount(AS_CMD)].

server specifies the system from which the resource is available.

access specifies the access granted the client systems, either `ro` or `rw` (for read-only or read/write, respectively).

transport specifies the transport provider on which the resource is shared.

description describes the resource.

A field may be null. Each null field is indicated by a hyphen (–) unless the remainder of the fields on the line are also null, in which case it may be omitted. Any fields containing whitespace are enclosed in quotes.

ERRORS

If (1) the domain name server cannot be contacted or (2) the argument is a domain name unknown to the domain name server, an error message will be sent to standard error.

USAGE

Administrator, End-User.

SEE ALSO

dfmounts(RS_CMD), mount(AS_CMD), share(RS_CMD), unshare(RS_CMD)

LEVEL

Level 1.

NAME

keylogin – decrypt and store secret key

SYNOPSIS

`keylogin`

DESCRIPTION

The `keylogin` command prompts the user for a password, and uses it to decrypt the user's secret key stored in the publickey database [see publickey(RS_ENV)]. Once decrypted, the user's key is stored by the local key server process `keyserv` [see keyserv(RS_CMD)] to be used by any secure network services, such as NFS.

SEE ALSO

chkey(RS_CMD), keyserv(RS_CMD), newkey(RS_CMD), publickey(RS_ENV).

LEVEL

Level 1.

NAME

keyserv – server for storing public and private keys

SYNOPSIS

`keyserv [ –n ]`

DESCRIPTION

The `keyserv` command is a daemon that is used for storing the private encryption keys of each user logged into the system. These encryption keys are used for accessing secure network services such as secure NFS.

Normally, root's key is read from the rootkey database when the daemon is started. This is useful during power-fail reboots when no one is around to type a password, yet you still want the secure network services to operate normally.

The `–n` option prompts the user for the password to decrypt root's key stored in the publickey database and then store the decrypted key in the rootkey database for future use. Root's key is not read from the rootkey database. This option is useful if the the rootkey database ever gets out of date or corrupted.

SEE ALSO

publickey(RS_ENV).

LEVEL

Level 1.

NAME

newkey – create a new key in the publickey database

SYNOPSIS

`newkey [ –u` *username* `]`

`newkey [ –h` *hostname* `]`

DESCRIPTION

The `newkey` command is normally run by the network administrator on the machine that contains the publickey database, to establish public keys for users and privileged users on the network. These keys are needed when using secure RPC or secure NFS.

`newkey` will prompt for a password for the given *username* and then create a new public/secret key pair for the user in the publickey database, encrypted with the given password.

The following options are available:

`–u` *username* Create a new public/secret key pair for the given *username*. Prompts for a password for the given *username*.

`–h` *hostname* Create a new public/secret key pair for the privileged user at the given *hostname*. Prompts for a root password for the given *hostname*.

SEE ALSO

chkey(RS_CMD), keylogin(RS_CMD), keyserv(RS_CMD), publickey(RS_ENV).

LEVEL

Level 1.

NAME

rpcbind – universal addresses to RPC program number mapper

SYNOPSIS

`rpcbind`

DESCRIPTION

`rpcbind` is a server that converts RPC program numbers into universal addresses. It must be running in order to make RPC calls.

When an RPC service is started, it will tell `rpcbind` at what address it is listening to, and what RPC program numbers it is prepared to serve. When a client wishes to make an RPC call to a given program number, it will first contact `rpcbind` on the server machine to determine the address where RPC packets should be sent.

Normally, standard RPC servers are started by port monitors, so `rpcbind` must be started before port monitors are invoked.

`rpcbind` is restricted to users with appropriate privileges.

USAGE

Administrator.

If `rpcbind` crashes, all RPC servers must be restarted.

SEE ALSO

rpcinfo(RS_CMD).

LEVEL

Level 1.

NAME

rpcgen – an RPC protocol compiler

SYNOPSIS

```
rpcgen infile
rpcgen [–Dname[=value]] [–T] infile
rpcgen –c|–h|–l|–m|–t [–o outfile ] [infile]
rpcgen –s nettype [–o outfile] [infile]
rpcgen –n netid [–o outfile] [infile]
```

DESCRIPTION

`rpcgen` is a tool that generates C code to implement an RPC protocol. The input to `rpcgen` is a language similar to C known as RPC Language (Remote Procedure Call Language) [see the *Remote Services Introduction* chapter for details on the RPC Language].

`rpcgen` is normally used as in the first synopsis where it takes an input file and generates four output files. If the *infile* is named `proto.x`, then `rpcgen` will generate a header file in `proto.h`, XDR routines in `proto_xdr.c`, server-side stubs in `proto_svc.c`, and client-side stubs in `proto_clnt.c`. With the `–T` option, it will also generate the RPC dispatch table in `proto_tbl.i`.

The second synopsis provides special features which allow for the creation of more sophisticated RPC servers. These features include support for user provided `#defines` and RPC dispatch tables, The entries in the RPC dispatch table contain:

- pointers to the service routine corresponding to that procedure,
- a pointer to the input and output arguments
- the size of these routines

A server can use the dispatch table to check authorization and then to execute the service routine; a client library may use it to deal with the details of storage management and XDR data conversion.

The other three synopses shown above are used when one does not want to generate all the output files, but only a particular one. Some examples of their usage is described in the EXAMPLE section below. When `rpcgen` is executed with the `–s` option, it creates servers for that particular class of transports. When executed with the `–n` option, it creates a server for the transport specified by *netid*. If *infile* is not specified, `rpcgen` accepts the standard input.

The C preprocessor, `cc –E` [see cc(SD_CMD)], is run on the input file before it is actually interpreted by `rpcgen`. For each type of output file, `rpcgen` defines a special preprocessor symbol for use by the `rpcgen` programmer:

`RPC_HDR`	defined when compiling into header files
`RPC_XDR`	defined when compiling into XDR routines
`RPC_SVC`	defined when compiling into server-side stubs
`RPC_CLNT`	defined when compiling into client-side stubs
`RPC_TBL`	defined when compiling into RPC dispatch tables

Any line beginning with '`%`' is passed directly into the output file, uninterpreted by `rpcgen`.

For every data type referred to in *infile*, `rpcgen` assumes that there exists a routine with the string `xdr_` prepended to the name of the data type. If this routine does not exist in the RPC/XDR library, it must be provided. Providing an undefined data type allows customization of XDR routines.

The following options are available:

`-c` Compile into XDR routines.

`-D`*name*`[=`*value*`]`
Define a symbol *name*. Equivalent to the `#define` directive in the source. If no *value* is given, *value* is defined as `1`. This option may be specified more than once.

`-h` Compile into `C` data-definitions (a header file). `-T` option can be used in conjunction to produce a header file which supports RPC dispatch tables.

`-l` Compile into client-side stubs.

`-m` Compile into server-side stubs, but do not generate a main routine. This option is useful for doing callback-routines and for users who need to write their own main routine to do initialization.

`-n` *netid*
Compile into server-side stubs for the transport specified by *netid*. There should be an entry for *netid* in the netconfig database. This option may be specified more than once, so as to compile a server that serves multiple transports.

`-o` *outfile*
Specify the name of the output file. If none is specified, standard output is used (`-c`, `-h`, `-l`, `-m`, `-n`, `-s` and `-t` modes only).

`-s` *nettype*
Compile into server-side stubs for all the transports belonging to the class *nettype*. The supported classes are `netpath`, `visible`, `circuit_n`, `circuit_v`, `datagram_n`, `datagram_v`, `tcp`, and `udp` [see the *Remote Services Definitions* chapter for the meanings associated with these classes]. This option may be specified more than once. Note: the transports are chosen at run time and not at compile time.

`-t` Compile into RPC dispatch table.

`-T` Generate the code to support RPC dispatch tables.

The options `-c`, `-h`, `-l`, `-m`, `-n`, `-s` and `-t` are used exclusively to generate a particular type of file, while the options `-D` and `-T` are global and can be used with the other options.

USAGE

General.

The RPC Language does not support nesting of structures. As a work-around, structures can be declared at the top-level, and their name used inside other structures in order to achieve the same effect.

Name clashes can occur when using program definitions, since the apparent scoping does not really apply. Most of these can be avoided by giving unique names for programs, versions, procedures and types.

The server code generated with `–n` option refers to the transport indicated by *netid* and hence is very site specific.

EXAMPLE

The following example:

```
$ rpcgen –T prot.x
```

generates all the five files: `prot.h`, `prot_clnt.c`, `prot_svc.c`, `prot_xdr.c` and `prot_tbl.i`.

The following example sends the C data-definitions (header file) to the standard output.

```
$ rpcgen –h prot.x
```

To send the test version of the `-DTEST`, server side stubs for all the transport belonging to the class `datagram_n` on standard output, use:

```
$ rpcgen –s datagram_n –DTEST prot.x
```

To create the server side stubs for the transport indicated by *netid* `tcp`, use:

```
$ rpcgen –n tcp –o prot_svc.c prot.x
```

SEE ALSO

cc(SD_CMD).

LEVEL

Level 1.

NAME

rpcinfo – report RPC information

SYNOPSIS

`rpcinfo [–T` *transport*`] [`*host*`]`
`rpcinfo [–T` *transport*`]` *host program* `[`*version*`]`
`rpcinfo –a` *serv_addres* `–T` *transport program* `[`*version*`]`
`rpcinfo –b` *program version*
`rpcinfo –d [–T` *transport*`]` *program version*

DESCRIPTION

The `rpcinfo` command makes an RPC call to an RPC server and reports what it finds. `rpcinfo` without any arguments lists all the registered RPC services. This usage, shown by the first synopsis, is the most common. In the second synopsis, `rpcinfo` makes an RPC call to 0 on the specified *host* and reports whether a response was received. See EXAMPLE for other ways to use `rpcinfo`.

The following options are available, and all except `–T` are mutually exclusive.

`–T` *transport* Specify the transport on which the service is required. If this option is not specified, `rpcinfo` uses the transport specified in the NETPATH environment variable, or if that is unset, in the netconfig database. This is a generic option, and can be used in conjunction with any other option, except the `–b` option.

`–a` *serv_address* Use *serv_address* as the universal address for the service on *transport*, ping procedure 0 of the specified *program*, and report whether a response was received. This option requires the `–T` option.

`–b` Make an RPC broadcast to procedure 0 of the specified *program* and *version* and report all hosts that respond. Send the broadcast request on all transports that support broadcasts. If broadcasting is not supported by any transport, an error message is printed.

`–d` Delete registration for the RPC service of the specified *program* and *version*. This option can be exercised only by the super-user. If *transport* is specified, unregister the service on only that transport, otherwise unregister the services on all the transports on which it was registered.

The *program* argument can be either a name or a number.

If a *version* is specified, `rpcinfo` attempts to call that version of the specified *program*. Otherwise, `rpcinfo` attempts to find all the registered version numbers for the specified *program* by calling version 0, which is presumed not to exist; if it does exist, `rpcinfo` attempts to obtain this information by calling an extremely high version number instead, and attempts to call each registered version. Note: the version number is required for `–b` and `–d` options.

EXAMPLE

To show all of the RPC services registered on the local machine use:

```
$ rpcinfo
```

To show all of the RPC services registered on the machine named `klaxon` and on transport `tcp` use:

```
$ rpcinfo –T tcp klaxon
```

To delete the registration for version 1 of the `walld` for all transports use:

```
$ rpcinfo –d walld 1
```

USAGE

General.

SEE ALSO

rpcbind(RS_LIB).

LEVEL

Level 1.

NAME

share – make local resource available for sharing by remote systems

SYNOPSIS

`share` [`-F` *fstype*] [`-o` *specific_options*] [`-d` *description*]
[*pathname* [*resourcename*]]

DESCRIPTION

The `share` command makes a resource available for sharing through a distributed file system of type *fstype*. When invoked without any arguments, `share` displays all resources on the local system that are shared through any distributed file system. When invoked with only a file system type, `share` displays all resources shared on the local system through the given file system. If the `-F` flag is omitted, and *pathname* is given, then the file system in the first line of `/etc/dfs/fstypes` is used as the default. *specific_options* as well as the syntax of *resourcename* are specific to particular distributed file systems.

The `-d` flag may be used to provide a description of the resource being shared.

ERRORS

If (1) the network is not up and running, (2) *pathname* is not a full path name or (3) *pathname* is not on a file system mounted locally, an error message will be sent to the standard error output.

FILES

`/etc/dfs/dfstab`
`/etc/dfs/sharetab`
`/etc/dfs/fstypes`

USAGE

Administrator.

SEE ALSO

unshare(RS_CMD).

LEVEL

Level 1.

NAME

unshare – make local resource unavailable for sharing by remote systems

SYNOPSIS

`unshare` [`-F` *fstype*] [`-o` *specific_options*] *pathname*
`unshare` [`-F` *fstype*] [`-o` *specific_options*] *resourcename*

DESCRIPTION

The `unshare` command makes a resource that was shared with `share` unavailable for sharing through a remote file system of type *fstype*. If the `-F` flag is omitted, the file system in the first line of `/etc/dfs/fstypes` is used as the default. *specific_options* as well as the syntax of *resourcename* are specific to particular distributed file systems.

ERRORS

If *resourcename* is not found in the shared information, an error message will be sent to standard error.

FILES

`/etc/dfs/dfstab`
`/etc/dfs/sharetab`
`/etc/dfs/fstypes`

USAGE

Administrator.

SEE ALSO

share(RS_CMD)

LEVEL

Level 1.

Real Time And Memory Management Introduction

Real Time And Memory Management Overview

The Real Time and Memory Management Extension (RT) consists of facilities to allow application programs to respond in a deterministic and timely manner to external interrupts. This issue of the SVID includes BSD-based timer functionality to provide fine granularity alarms and a `memcntl()` interface to provide application control over memory residence [see memcntl(RT_OS)].

USL is committed to support the standardization of a Real Time interface as defined by POSIX. The IEEE P1003.4 working group is currently pursuing a draft standard for a Real Time interface. Full conformance to this standard will be strongly considered upon formal approval.

The following are prerequisite for support of the Real Time and Memory Management Extension:

- Base System
- Kernel Extension

SUMMARY OF OS SERVICE ROUTINES

The following OS service routines are supported by the Real Time and Memory Management Extension. All of these items are new to this issue of the SVID. Items marked with a star (*) are Level 2, as defined in the *General Introduction* to this volume.

```
getitimer*    memcntl    mlockall    munlockall    settimeofday
gettimeofday* mlock      munlock     setitimer     swapctl
```

ORGANIZATION OF TECHNICAL INFORMATION

The *Real Time and Memory Management OS Service Routines* chapter provides manual page descriptions of library routines supported by this extension.

Real Time And Memory Management Routines

The following section contains the manual pages for the RT_OS routines.

NAME

getitimer, **setitimer** – get/set value of interval timer

SYNOPSIS

#include <sys/time.h>

int getitimer(int *which*, **struct itimerval** **value***);**

int setitimer(int *which*, **struct itimerval** **value*, **struct itimerval** **ovalue***);**

DESCRIPTION

The system provides each process with three interval timers, defined in **sys/time.h**. The **getitimer** call stores the current value of the timer specified by *which* into the structure pointed to by *value*. The **setitimer** call sets the value of the timer specified by *which* to the value specified in the structure pointed to by *value*, and if *ovalue* is not **NULL**, stores the previous value of the timer in the structure pointed to by *ovalue*.

A timer value is defined by the **itimerval** structure for the definition of **timeval**], which includes the following members:

```
struct timeval  it_interval;    /* timer interval */
struct timeval  it_value;       /* current value */
```

If **it_value** is non-zero, it indicates the time to the next timer expiration. If **it_interval** is non-zero, it specifies a value to be used in reloading **it_value** when the timer expires. Setting **it_value** to zero disables a timer, regardless of the value of **it_interval**. Setting **it_interval** to zero disables a timer after its next expiration (assuming **it_value** is non-zero).

Time values smaller than the resolution of the system clock are rounded up to this resolution.

The three timers are:

ITIMER_REAL Decrements in real time. A **SIGALRM** signal is delivered when this timer expires.

ITIMER_VIRTUAL Decrements in process virtual time. It runs only when the process is executing. A **SIGVTALRM** signal is delivered when it expires.

ITIMER_PROF Decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by interpreters in statistically profiling the execution of interpreted programs. Each time the **ITIMER_PROF** timer expires, the **SIGPROF** signal is delivered. Because this signal may interrupt in-progress system calls, programs using this timer must be prepared to restart interrupted system calls.

Return Values

If the calls succeed, a value of **0** is returned. If an error occurs, the value **–1** is returned, and an error code is placed in the global variable **errno**.

Errors

Under the following conditions, the functions **getitimer** and **setitimer** fail and set **errno** to:

EINVAL The specified number of seconds is greater than 100,000,000, the number of microseconds is greater than or equal to 1,000,000, or the *which* parameter is unrecognized.

SEE ALSO

alarm(BA_OS)

LEVEL

Level 1.

NOTICES

The microseconds field should not be equal to or greater than one second.

setitimer is independent of the **alarm** system call.

Do not use **setitimer** with the **sleep** routine. A **sleep** following a **setitimer** wipes out knowledge of the user signal handler.

NAME

gettimeofday, settimeofday – get or set the date and time

SYNOPSIS

```
#include <sys/time.h>

int gettimeofday(struct timeval *tp);

int settimeofday(struct timeval *tp);
```

DESCRIPTION

The system's notion of the current Greenwich time is obtained with the `gettimeofday()` call, and set with the `settimeofday()` call. The current time is expressed in elapsed seconds and microseconds since 00:00 UTC, January 1, 1970 (zero hour). The resolution of the system clock is hardware dependent; the time may be updated continuously, or in ticks .

tp points to a `timeval` structure, which includes the following members:

```
long    tv_sec;   /* seconds since Jan. 1, 1970 */
long    tv_usec;  /* and microseconds */
```

The flag indicating the type of daylight savings time correction should have one of the following values (as defined in `<sys/time.h>`):

`DST_NONE`	daylight savings time not observed.
`DST_USA`	United States DST.
`DST_AUST`	Australian DST.
`DST_WET`	Western European DST.
`DST_MET`	Middle European DST.
`DST_EET`	Eastern European DST.
`DST_CAN`	Canadian DST.
`DST_GB`	Great Britain and Eire DST.
`DST_RUM`	Rumanian DST.
`DST_TUR`	Turkish.
`DST_AUSTALT`	Australian-style DST with shift in 1986.

Also note that the offset of the local time zone from UTC may change over time, as may the rules for daylight saving time correction. The `localtime()` routine [see `localtime()` in ctime(BA_LIB)] obtains this information from a file rather than from `gettimeofday()`. Programs should use `localtime()` to convert dates and times.

Only a process with appropriate privileges may set the time of day.

RETURN VALUE

If the call succeeds, a value of `0` is returned. If an error occurs, a value of `–1` is returned and `errno` is set to indicate the error.

ERRORS

Under the following condition, the functions `gettimeofday()` and `settimeofday()` will fail and set `errno` to:

`EPERM` if a process without appropriate privileges attempts to set the time.

In addition, under the following condition, the function `settimeofday()` will fail and set `errno` to:

`EINVAL` if the *tp* parameter is not in canonical form, *i.e.*, the number of microseconds is greater than zero and less than 1,000,000, and the number of seconds is non-negative.

FILES

`/usr/include/sys/time.h`

USAGE

Administrator.

SEE ALSO

adjtime(BA_OS), ctime(BA_LIB).

FUTURE DIRECTIONS

It is expected that these routines will be replaced by POSIX 1003.4 routines in a future issue of the SVID.

LEVEL

Level 2: June 30, 1989.

NAME

memcntl — memory management control

SYNOPSIS

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

int memcntl(caddr_t addr, size_t len, int cmd, int arg, int attr,
     int mask);
```

DESCRIPTION

The function `memcntl()` allows the calling process to apply a variety of control operations over the address space identified by the mappings established for the address range [*addr*, *addr* + *len*).

addr must be a multiple of the page size as returned by `sysconf()`. The scope of the control operations can be further defined with additional selection criteria (in the form of attributes) according to the bit pattern contained in *attr*.

The following attributes are used as the selection criteria:

Page mapping

`SHARED`	Page is mapped shared
`PRIVATE`	Page is mapped private

Page protection

`PROT_READ`	Page can be read
`PROT_WRITE`	Page can be written
`PROT_EXEC`	Page can be executed

The selection criteria are constructed by OR-ing together the attribute bits and must match exactly.

In addition, the following criteria may be specified:

`PROC_TEXT`	process text
`PROC_DATA`	process data

where `PROC_TEXT` specifies all privately mapped segments with read and execute permission, and `PROC_DATA` specifies all privately mapped segments with write permission.

Selection criteria can be used to describe various abstract memory objects within the address space on which to operate. If an operation shall not be constrained by the selection criteria, *attr* must have the value `0`.

mask is reserved for future use and must always have the value `0`.

The operation to be performed is identified by the argument *cmd*. The symbolic names for the operations are defined in `<sys/mman.h>` as follows:

`MC_LOCK` Lock in memory all pages in the range with attributes *attr*. A given page may be locked multiple times through different mappings; however, within a given mapping, page locks do not nest. Multiple lock operations on the same address in the same process will all be removed with a single unlock operation. A page locked in one process and mapped in another (or visible through a different mapping in the locking process) is locked in memory as long as the locking process does neither an implicit nor explicit unlock operation. If a locked mapping is removed, or a page is deleted through file truncation, an unlock operation is implicitly performed. If a writable `MAP_PRIVATE` page in the address range is changed, the lock will be transferred to the private page.

At present *arg* is unused, but must be `0` to ensure compatibility with potential future enhancements.

`MC_LOCKAS` Lock in memory all pages mapped by the address space with attributes *attr*. At present *addr* and *len* are unused, but must be `NULL` and `0` respectively, to ensure compatibility with potential future enhancements. *arg* is a bit pattern built from the flags:

`MCL_CURRENT`	Lock current mappings
`MCL_FUTURE`	Lock future mappings

The value of *arg* determines whether the pages to be locked are those currently mapped by the address space, those that will be mapped in the future, or both. If `MCL_FUTURE` is specified, then all mappings subsequently added to the address space will be locked, provided sufficient memory is available.

`MC_SYNC` Write to their permanent storage locations all modified pages in the range with attributes *attr*. Optionally, invalidate cache copies. *arg* is a bit pattern built from the flags used to control the behavior of the operation:

`MS_ASYNC`	perform asynchronous writes
`MS_SYNC`	perform synchronous writes
`MS_INVALIDATE`	invalidate mappings

`MS_ASYNC` returns immediately once all write operations are scheduled; with `MS_SYNC` the system call will not return until all write operations are completed.

`MS_INVALIDATE` invalidates all cached copies of data in memory, so that further references to the pages will be obtained by the system from their permanent storage locations. This operation should be used by applications that require a memory object to be in a known state.

`MC_UNLOCK` Unlock all pages in the range with attributes *attr*. At present *arg* is unused, but must be `0` to ensure compatibility with potential future enhancements.

MC_UNLOCKAS Remove address space memory locks, and locks on all pages in the address space with attributes *attr*. At present *addr*, *len*, and *arg* are unused, but must be NULL, 0 and 0 respectively, to ensure compatibility with potential future enhancements.

Locks established with the lock operations are not inherited by a child process after fork(). Attempts to lock more memory than a system-specific limit, will fail.

Due to the potential impact on system resources, all operations, with the exception of MC_SYNC, are restricted to processes with appropriate privileges. The memcntl() function subsumes the operations of plock().

RETURN VALUE

Upon successful completion, the function memcntl() returns a value of 0; otherwise, it returns a value of –1 and sets errno to indicate an error.

ERRORS

Under the following conditions, the function memcntl() fails and sets errno to:

EAGAIN if some or all of the memory identified by the operation could not be locked when MC_LOCK or MC_LOCKAS is specified.

EBUSY if some or all the addresses in the range [*addr, addr + len*) are locked and MC_SYNC with MS_INVALIDATE option is specified.

EINVAL if *addr* is not a multiple of the page size as returned by sysconf().

EINVAL if *addr* and/or *len* do not have the value 0 when MC_LOCKAS or MC_UNLOCKAS is specified.

EINVAL if *arg* is not valid for the function specified.

EINVAL if invalid selection criteria are specified in *attr*.

ENOMEM if some or all the addresses in the range [*addr*, *addr + len*) are invalid for the address space of the process or pages not mapped are specified.

EPERM if the process does not have appropriate privilege to perform the requested operation.

SEE ALSO

sysconf(BA_OS), mlock(RT_OS), mlockall(RT_OS), mmap(KE_OS), mprotect(KE_OS), msync(KE_OS), plock(KE_OS).

LEVEL

Level 1.

NAME

mlock, munlock – lock (or unlock) pages in memory

SYNOPSIS

```
#include &lt;sys/types.h&gt;

int mlock(caddr_t addr, size_t len);

int munlock(caddr_t addr, size_t len);
```

DESCRIPTION

The function `mlock()` uses the mappings established for the address range [*addr*, *addr* + *len*) to identify pages to be locked in memory. The effect of `mlock(`*addr*`,` *len*`)` is equivalent to `memcntl(`*addr*`,` *len*`, MC_LOCK, 0, 0)`.

`munlock()` removes locks established with `mlock()`. The effect of `munlock(`*addr*`,` *len*`)` is equivalent to `memcntl(`*addr*`,` *len*`, MC_UNLOCK, 0, 0)`.

Locks established with `mlock()` are not inherited by a child process after a `fork()`.

RETURN VALUE

Upon successful completion, the functions `mlock()` and `munlock()` return a value of `0`; otherwise, they return a value of `–1` and set `errno` to indicate an error.

ERRORS

See memcntl(RT_OS).

USAGE

Use of `mlock()` and `munlock()` requires that the user have appropriate privileges.

SEE ALSO

fork(BA_OS), memcntl(RT_OS), mmap(KE_OS), mlockall(RT_OS), plock(KE_OS), sysconf(BA_OS).

LEVEL

Level 1.

NAME

mlockall, munlockall – lock or unlock address space

SYNOPSIS

```
#include <sys/mman.h>

int mlockall(int flags);

int munlockall(void);
```

DESCRIPTION

The function `mlockall()` causes all pages mapped by an address space to be locked in memory. The effect of `mlockall(`*flags*`)` is equivalent to:

`memcntl(0, 0, MC_LOCKAS, 0,` *flags*`)`

The value of *flags* determines whether the pages to be locked are those currently mapped by the address space, those that will be mapped in the future, or both. [See memcntl(RT_OS) for the values of *flags*.]

The function `munlockall()` removes address space locks and locks on mappings in the address space. The effect of `munlockall()` is equivalent to:

`memcntl(0, 0, MC_UNLOCKAS, 0, 0)`

Locks established with `mlockall()` are not inherited by a child process after a `fork()`.

RETURN VALUE

Upon successful completion, the functions `mlockall()` and `munlockall()` return a value of `0`; otherwise, they return a value of `-1` and set `errno` to indicate an error.

ERRORS

See memcntl(RT_OS).

USAGE

Use of `mlockall()` and `munlockall()` requires that the process have appropriate privileges.

SEE ALSO

fork(BA_OS), memcntl(RT_OS), mlock(RT_OS), mmap(KE_OS), plock(KE_OS), sysconf(BA_OS).

LEVEL

Level 1.

NAME

swapctl – manage swap space

SYNOPSIS

```
#include <sys/stat.h>
#include <sys/swap.h>

int swapctl(int cmd, void *arg);
```

DESCRIPTION

The function `swapctl()` provides a means for a process to add, delete, and identify resources providing memory for swap space. *cmd* specifies one of the following options contained in `<sys/swap.h>`:

```
SC_ADD              /* add a resource for swapping */
SC_LIST             /* list the resources for swapping */
SC_REMOVE           /* remove a resource for swapping */
SC_GETNSWP          /* return number of swap resources */
```

When `SC_ADD` or `SC_REMOVE` are specified, *arg* is a pointer to a `swapres` structure containing the following members:

```
char   *sr_name;     /* pathname of resource */
off_t  sr_start;     /* offset to start of swap area */
off_t  sr_length;    /* length of swap area */
```

A successful `SC_ADD` adds a reference to the associated file, which guarantees that it will continue to be usable for swap space, even if the file is removed from the directory by `unlink(BA_OS)`. This reference will be removed by the corresponding `SC_REMOVE`.

When `SC_LIST` is specified, *arg* is a pointer to a `swaptable` structure containing the following members:

```
int    swt_n;        /* number of swapents following */
struct swapent swt_ent[];/* array of swt_n swapents */
```

A `swapent` structure contains the following members:

```
char   *ste_path;    /* name of the swap file */
off_t  ste_start;    /* starting block for swapping */
off_t  ste_length;   /* length of swap area */
long   ste_pages;    /* number of pages for swapping */
long   ste_free;     /* number of ste_pages free */
long   ste_flags;    /* see below */
```

`SC_LIST` causes `swapctl()` to returns at most `swt_n` entries. The value of `swapctl()` is the number actually returned.

When `SC_GETNSWP` is specified, `swapctl()` returns as its value the number of swap resources in use. *arg* is ignored for this operation.

The `SC_ADD` and `SC_REMOVE` functions will fail if calling process does not have appropriate privileges.

RETURN VALUE

Upon successful completion, the function `swapctl()` returns a value of `0` for `SC_ADD` or `SC_REMOVE`, the number of `struct swapent` entries actually returned for `SC_LIST`, or the number of swap resources in use for `SC_GETNSWP`. Upon failure, the function `swapctl()` returns a value of `–1` and sets `errno` to indicate an error.

ERRORS

Under the following conditions, the function `swapctl()` fails and sets `errno` to:

`EEXIST`	if the specified resource is already being used for swapping (`SC_ADD`) or else can not be removed (`SC_REMOVE`).
`EINTR`	if interrupted by signal (`SC_REMOVE`).
`EINVAL`	if the specified function value is not valid (that is, none of `SC_ADD`, `SC_LIST`, or `SC_REMOVE`).
`EISDIR`	if the path specified for `SC_ADD` is a directory.
`ELOOP`	if too many symbolic links were encountered in translating the pathname provided to `SC_ADD` or `SC_REMOVE`.
`ENAMETOOLONG`	if the length of a component of the path specified for `SC_ADD` or `SC_REMOVE` exceeds `{NAME_MAX}` characters or the length of the path exceeds `{PATH_MAX}` characters and `{_POSIX_NO_TRUNC}` is in effect.
`ENOENT`	Pathname specified for `SC_ADD` or `SC_REMOVE` does not exist.
`ENOMEM`	An insufficient number of `struct swapent` structures were provided to `SC_LIST`, or there were insufficient system storage resources available during an `SC_ADD` or `SC_REMOVE`.
`ENOTDIR`	Pathname provided to `SC_ADD` or `SC_REMOVE` contained a component in the path prefix that was not a directory.
`EPERM`	The process does not have appropriate privileges.

LEVEL

Level 1.

C Language Specification

C Language Specification Overview

The C Language Specification defines the programming language recognized by a SVID-conforming C compiler [see cc(SD_CMD)]. The SVID-conforming C language is based on the American National Standard for Information Systems–Programming Language C (ANSI C Standard), with extensions that provide additional functionality. This definition does not address C library functions. It assumes that the reader is familiar with the C language and does not attempt to duplicate the information in the ANSI C standard, but rather provides a reference of differences and additions to the language.

UNDEFINED AND IMPLEMENTATION DEFINED BEHAVIOR

The ANSI C standard specifies that the behavior of a conforming compilation system is undefined or implementation defined under certain circumstances. Unless explicitly defined in the SVID, ANSI C undefined or implementation defined behaviors are also undefined or implementation defined for SVID-conforming C language implementations.

Undefined behavior is indicated in the ANSI C standard by the words ''undefined behavior'', by the omission of any explicit definition of behavior, or by a violation of a ''shall'' or ''shall not'' requirement outside of a constraint. ANSI C conforming language behavior, in these circumstances, includes ignoring the situation with unpredictable results, behaving in a documented manner, or terminating with a diagnostic message. A SVID-conforming C language extends this to require that the compilation system behavior not be arbitrary, *e.g.*, dump core. Therefore, some behavior not defined by the ANSI C standard is defined for a SVID-conforming compilation system.

FINAL COPY
June 15, 1995
File: pl_c.txt
svid

OPTIONAL BEHAVIOR

The SVID definition of the C language defines several ANSI C ''implementation-defined'' behaviors, extentions, and optional extentions to the standard. The optional extentions, if provided in a SVID-conforming C language, should conform to the definitions provided. They are

- addition of the keyword `asm`, a non-conforming extention,
- addition of the preprocessing directives `#assert`, `#unassert`, and `#ident`,
- specification of certain `#pragma` directives.

SVID-conforming C language implementations may provide additional behavior not defined in the SVID. For example, a SVID-conforming C language may provide old style ''unsigned-preserving'' integral promotion behavior, in addition to the ANSI C ''value-preserving'' integral promotion behavior.

DIAGNOSTICS

The ANSI C standard requires a conforming implementation to define how a diagnostic is identified. Any message written to standard error is taken to be a diagnostic. The SVID-conforming C language recommends, but does not require, the following formats:

`"`*filename*`", line` *lineno*`:` *msg*

and

`"`*filename*`", line` *lineno*`: warning:` *msg*

where

- *filename* is the name of the file containing the error,
- *lineno* is the number of the line on which the error was found,
- and *msg* is the diagnostic message.

In the case where an error occurs while processing the command line before opening any files, the filename may not be available for use in the diagnostic message. Therefore, the described diagnostic format holds only after successfully opening the input file.

CHARACTER SETS

The default execution character set for a SVID-conforming C language is ASCII and the default direction of printing is left-to-right.

SOURCE FILES AND TOKENIZATION

Identifiers

Identifiers are used to name things like variables, functions, data types and macros.

Identifiers are made up of letters, digits, or underscore (_) characters. The first character must be a letter or an underscore.

A SVID-conforming C language will treat upper and lower case letters as distinct in external identifiers. It will also support internal and external identifiers that are significant to at least the first 100 characters.

FINAL COPY
June 15, 1995
File: pl_c.txt
svid

Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

```
asm       default   for       short     union
auto      do        goto      signed    unsigned
break     double    if        sizeof    void
case      else      int       static    volatile
char      enum      long      struct    while
const     extern    register  switch
continue  float     return    typedef
```

The keyword `asm` is a non-conforming extension to the ANSI C standard, and is optional in a SVID-conforming C language. If it is supported, the `asm` keyword may be used to insert assembly-language code directly into the translator output. The most common implementation allows a statement of the form:

`asm (` *character-string-literal* `);`

PREPROCESSING

The SVID defines several optional extensions to the ANSI C standard in the area of preprocessing.

Preprocessing Directives

Assertions

Assertions are optional in a SVID-conforming C language. If assertions are implemented, a line of the form

`#assert` *predicate* `(`*token-sequence*`)`

associates the list of tokens with the *predicate* in the assertion name space (separate from the space used for macro definitions). The *predicate* must be an identifier token. The assertion lasts until a corresponding `#unassert` directive, if any. In the argument preprocessing tokens, parentheses must balance and commas have no special meaning.

`#assert` *predicate*

asserts that *predicate* exists, but does not associate any token sequence with it.

The compiler provides the following predefined predicate by default:

```
#assert system (unix)
```

A line of the form

`#unassert` *predicate* `(`*token-sequence*`)`

deletes the list of tokens asserted on the *predicate*. A line of the form

`#unassert` *predicate*

deletes all assertions on the *predicate*.

Version Control

The `#ident` directive is optional in a SVID-conforming C language and is used to help administer version control information.

`#ident "`*version*`"`

puts an arbitrary string in the `.comment` section of an executable file. The `.comment` section is not loaded into memory when an executable file is executed.

Pragmas

Preprocessing lines of the form

`#pragma` *token-sequence*

specify implementation-defined actions. These lines must be handled by a conforming ANSI C implementation, but need not have any effect.

The following `#pragma`'s are optional in a SVID-conforming C language. If these are implemented, a line of the form

`#pragma ident "`*version*`"`

is identical in function to:

`#ident "`*version*`"`

A SVID-conforming C compiler ignores all unrecognized pragmas.

Macro Replacement

A SVID-conforming C language shall allow empty token list macro arguments. The resulting token list for such an invocation will contain no tokens for any parameter which was associated with an empty argument.

DECLARATIONS AND DEFINITIONS

Types

As an extension to the ANSI C standard, a SVID-conforming C language implementation may support bit-fields having any integral type. In such an implementation, bit-fields that are declared with the `signed` keyword or with the `unsigned` keyword act like their `int` counterpart with respect to the high-order bit's behaving like a sign bit. Whether bit-fields that are declared ‘‘plain’’ `int` sign-extend is implementation-dependent. (Note this means `enum` bit-fields behave like ‘‘plain’’ `int`.)

Storage Class Specifiers

As an extention, SVID-conforming C languages allow ‘‘multiply-defined external definitions’’: there may be more than one external definition for the identifier of an object, with or without the explicit use of the keyword `extern`. If the definitions disagree, or if more than one is initialized, the behavior is undefined.

Software Development Introduction

Software Development Overview

The Software Development Extension provides facilities for the compilation and maintenance of **C** language software. Principal components are the **C** compiler `cc` and its related utilities, the program development aids `yacc` and `lex`, and the Source Code Control System (SCCS) utilities.

The following are prerequisite for support of the Software Development Extension:

- Base System
- Basic Utilities Extension
- Advanced Utilities Extension
- Kernel Extension

SUMMARY OF LIBRARY ROUTINES

The following library routines are supported in a SVID-compliant Software Development Extension (*exception:* items marked with a sharp (#) are optional and need not be supported). Items marked with a star (*) are level 2, as defined in the *General Introduction* to this volume. Items marked with a dagger (†) are new to this edition of the SVID.

`MARK`#	`getutxent`†	`monitor`#	`setutxent`†
`a64l`	`getutxid`†	`nlist`	`sgetl`*
`endutxent`†	`getutxline`†	`putpwent`*	`sputl`*
`getpass`	`l64a`	`pututxline`†	`utmpxname`†

SUMMARY OF COMMANDS AND UTILITIES

The following library commands and utilities are supported in a SVID-compliant Software Development Extension (*exception:* items marked with a sharp (#) are optional and need not be supported). Items marked with a star (*) are Level 2, as defined in the *General Introduction* to this volume. Items marked with a dagger (†) are new to this issue of the SVID. Items marked with a doube dagger (‡) are internationalized.

`admin‡`	`delta‡`	`lint`	`rmdel`	`tsort`
`as#*`	`dis#*`	`lorder`	`sact`	`unget`
`cc‡`	`env`	`m4`	`size`	`val`
`cflow*`	`gcore`	`make`	`strip*`	`what`
`chroot`	`get‡`	`nm*`	`time`	`xargs`
`cxref*`	`ld`	`prof#*`	`truss`	`yacc`
`debug†`	`lex`	`prs`		

ORGANIZATION OF TECHNICAL INFORMATION

The *Software Development Environment* chapter is a new addition to the SVID, appearing first in SVID 4. This chapter describes the `/proc` subsystem, which provides support for the new, enhanced debugger, `debug`, described in the commands section.

The *Software Development Library Routines* chapter provides manual page descriptions of routine interfaces supported by this extension.

The *Software Development Commands and Utilities* chapter provides manual page descriptions of commands and utilities supported by this extension.

Software Development C Library support requirements are defined at the end of this introduction.

C LIBRARY SUPPORT REQUIREMENTS

The following libraries are required to support the **C** compiler command `cc`.

Standard C Library

The Standard **C** library is automatically searched by `cc` to resolve external references. This library supports all of the interfaces of the Base System, as defined in Volume 1, except for the Math Routines.

Standard C Mathematical Library

This library supports the Base System math routines, as defined in Volume 1. The `cc` option `-lm` is used to search this library.

Lex Library

The `lex` library is required by lex(SD_CMD). The `cc` option `-ll` is used to search this library.

Object File Library

The Object File Library is required for use of `sgetl` and `sputl`, as defined under sputl(SD_LIB). The `cc` option `-lld` is used to search this library.

YACC Library

The `yacc` library facilitates use of yacc(SD_CMD). The `cc` option `-ly` is used to search this library.

Software Development Library Routines

The following section contains the manual pages for the SD_LIB routines.

NAME

a64l, l64a – convert between long integer and base-64 ASCII string

SYNOPSIS

```
#include &lt;stdlib.h&gt;

long a64l(const char *s);

char *l64a(long value);
```

DESCRIPTION

These routines are used to maintain numbers stored in base-64 ASCII characters. This is a notation by which long integers can be represented by up to six characters; each character represents a digit in radix-64 notation.

The characters used to represent ‘digits’ are `.` for 0, `/` for 1, `0` through `9` for 2–11, `A` through `Z` for 12–37, and `a` through `z` for 38–63.

The routine `a64l()` takes a pointer to a null-terminated base-64 representation and returns a corresponding `long` value. If the string pointed to by `s` contains more than six characters, `a64l()` will use the first six.

The routine `l64a()` takes a `long` argument and returns a pointer to the corresponding base-64 representation. If *value* is 0, `l64a()` returns a pointer to a null string.

USAGE

The value returned by `l64a()` may be a pointer into a static buffer, the contents of which would therefore be overwritten by each call.

LEVEL

Level 1.

NAME

getpass – read a password

SYNOPSIS

```
#include <stdlib.h>

char *getpass(const char *prompt);
```

DESCRIPTION

The routine `getpass()` reads up to a newline or an `EOF` from the file `/dev/tty`, after prompting on the standard error output with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most `{PASS_MAX}` characters. If `/dev/tty` cannot be opened, a `NULL` pointer is returned. An interrupt will terminate input and send an interrupt signal to the calling program before returning. `getpass()` restores the terminal state and closes `/dev/tty` before returning.

The function `getpass` marks for update the `st_atime` field of the file `/dev/tty`.

FILES

`/dev/tty`

USAGE

The return value points to static data whose content is overwritten by each call.

SEE ALSO

devtty(BA_DEV)

LEVEL

Level 1.

NAME

getutx: getutxent, getutxid, getutxline, pututxline, setutxent, endutxent, utmpxname, getutmp, getutmpx, updwtmp, updwtmpx – access **utmpx** file entry

SYNOPSIS

```
#include <utmpx.h>

struct utmpx *getutxent (void);

struct utmpx *getutxid (const struct utmpx *id);

struct utmpx *getutxline (const struct utmpx *line);

struct utmpx *pututxline (const struct utmpx *utmpx);

void setutxent (void);

void endutxent (void);

int utmpxname (const char *file);

void getutmp (struct utmpx *utmpx, struct utmp *utmp);

void getutmpx (struct utmp *utmp, struct utmpx *utmpx);

void updwtmp (char *wfile, struct utmp *utmp);

void updwtmpx (char *wfilex, struct utmpx *utmpx);
```

DESCRIPTION

getutxent, **getutxid**, **getutxline**, and **pututxline** each return a pointer to a **utmpx** structure.

getutxent reads in the next entry from a **utmpx**-like file. If the file is not already open, it opens it. If it reaches the end of the file, it fails.

getutxid searches forward from the current point in the **utmpx** file until it finds an entry with a **ut_type** matching *id*->**ut_type** if the type specified is **RUN_LVL, BOOT_TIME, OLD_TIME**, or **NEW_TIME.** If the type specified in *id* is **INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS**, or **DEAD_PROCESS**, then **getutxid** returns a pointer to the first entry whose type is one of these four and whose *ut_id* field matches *id*->**ut_id**. If the end of file is reached without a match, it fails.

getutxline searches forward from the current point in the **utmpx** file until it finds an entry of the type **LOGIN_PROCESS** or **USER_PROCESS** which also has a *ut_line* string matching the *line*->**ut_line** string. If the end of file is reached without a match, it fails.

pututxline writes out the supplied **utmpx** structure into the **utmpx** file. It uses **getutxid** to search forward for the proper place if it finds that it is not already at the proper place. It is expected that normally the user of **pututxline** will have searched for the proper entry using one of the **getutx** routines. If so, **pututxline** will not search. If **pututxline** does not find a matching slot for the new entry, it will add a new entry to the end of the file. It returns a pointer to the **utmpx** structure.

setutxent resets the input stream to the beginning of the file. This should be done before each search for a new entry if it is desired that the entire file be examined.

endutxent closes the currently open file.

utmpxname allows the user to change the name of the file examined, from **/var/adm/utmpx** to any other file. It is most often expected that this other file will be **/var/adm/wtmpx**. If the file does not exist, this will not be apparent until the first attempt to reference the file is made. **utmpxname** does not open the file. It just closes the old file if it is currently open and saves the new file name. The new file name must end with the ‘‘**x**’’ character to allow the name of the corresponding **utmp** file to be easily obtainable (otherwise an error code of **0** is returned).

getutmp copies the information stored in the fields of the **utmpx** structure to the corresponding fields of the **utmp** structure. If the information in any field of **utmpx** does not fit in the corresponding **utmp** field, the data is truncated.

getutmpx copies the information stored in the fields of the **utmp** structure to the corresponding fields of the **utmpx** structure.

updwtmp checks the existence of *wfile* and its parallel file, whose name is obtained by appending an ‘‘**x**’’ to *wfile*. If only one of them exists, the second one is created and initialized to reflect the state of the existing file. *utmp* is written to *wfile* and the corresponding **utmpx** structure is written to the parallel file. If neither file exists nothing will happen.

updwtmpx checks the existence of *wfilex* and its parallel file, whose name is obtained by truncating the final ‘‘**x**’’ from *wfilex*. If only one of them exists, the second one is created and initialized to reflect the state of the existing file. *utmpx* is written to *wfilex*, and the corresponding **utmp** structure is written to the parallel file. If neither file exists nothing will happen.

Files

/var/adm/utmp, **/var/adm/utmpx**
/var/adm/wtmp, **/var/adm/wtmpx**

Errors

A null pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write.

LEVEL

Level 1.

NOTICES

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. On each call to either **getutxid** or **getutxline**, the routine examines the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason, to use **getutxline** to search for multiple occurrences it would be necessary to zero out the static after each success, or **getutxline** would just return the same structure over and over again. There is one exception to the rule about emptying the structure before further reads are done. The implicit read done by **pututxline** (if it finds that it is not already at the correct place in the file) will not hurt the contents of the static structure returned by the **getutxent**,

getutxid, or **getutxline** routines, if the user has just modified those contents and passed the pointer back to **pututxline**.

These routines use buffered standard I/O for input, but **pututxline** uses an unbuffered write to avoid race conditions between processes trying to modify the **utmpx** and **wtmpx** files.

NAME

MARK – profile within a function

SYNOPSIS

```
#define MARK
#include <prof.h>

void MARK(name)
```

DESCRIPTION

The macro `MARK()` will introduce a mark called *name* that will be treated the same as a function entry point. Execution of the mark will add to a counter for that mark, and program-counter time spent will be accounted to the immediately preceding mark or to the function if there are no preceding marks within the active function.

The identifier *name* may be any combination of letters, numbers or underscores. Each *name* in a single compilation must be unique, but may be the same as any ordinary program symbol.

For marks to be effective, the symbol `MARK` must be defined before the header file `<prof.h>` is included. This may be defined by a preprocessor directive as in the synopsis, or by a command line argument, i.e:

```
cc –p –DMARK foo.c
```

If `MARK` is not defined, the `MARK`(*name*) statements may be left in the source files containing them and will be ignored.

EXAMPLE

In this example, marks can be used to determine how much time is spent in each loop. Unless this example is compiled with `MARK` defined on the command line, the marks are ignored.

```
#include <prof.h>

foo()
{
     int i, j;
     .
     .
     .
     MARK(loop0);
     for (i = 0; i < 2000; i++) {
          . . .
     }
     MARK(loop1);
     for (j = 0; j < 2000; j++) {
          . . .
     }
     MARK(loop2);
}
```

SEE ALSO

profil(KE_OS), monitor(SD_LIB), prof(SD_CMD).

LEVEL

Level 1.

Optional. (When used, `MARK()` requires the `profil()` system service routine).

NAME

monitor – prepare execution profile

SYNOPSIS

```
#include <mon.h>

void monitor(int (*lowpc)(), int (*highpc)(),
   WORD *buffer, int bufsize, int nfunc);
```

DESCRIPTION

The routine `monitor()` is an interface to the `profil()` system service routine [see profil(KE_OS)]; *lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* `WORD`s (`WORD` is defined in the `<mon.h>` header file). The `monitor()` routine arranges to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*; *lowpc* may not equal 0 for this use of `monitor()`. At most, *nfunc* call counts can be kept; only calls of functions compiled with the profiling option `-p` of `cc` are recorded.

An executable program created by using the `-p` option with `cc` automatically includes calls for the `monitor()` routine with default parameters; therefore `monitor()` need not be called explicitly except to gain fine control over profiling.

For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern int etext();
. . .
monitor((int (*)())2, etext, buf, bufsize, nfunc);
```

The routine `etext()` lies just above all the program text.

To stop execution monitoring and write the results, use

```
monitor((int (*)())0, (int (*)())0, (WORD *) 0, 0, 0);
```

The `prof()` command [see prof(SD_CMD)] can then be used to examine the results.

The name of the file written by `monitor()` is controlled by the environmental variable `PROFDIR`. If `PROFDIR` is not set, then the file `mon.out` is created in the current directory. If `PROFDIR` is set to the null string, then no profiling is done and no output file is created. Otherwise, the value of `PROFDIR` is used as the name of the directory in which to create the output file. If `PROFDIR` is *dirname*, then the output file is named *dirname*/*pid*`.mon.out`, where *pid* is the process ID of the program. (When `monitor()` is called automatically by using the `-p` option of `cc`, the file created is *dirname*/*pid.progname*, where *progname* is the name of the program.)

FILES

`mon.out`

SEE ALSO

profil(KE_OS), cc(SD_CMD), prof(SD_CMD).

LEVEL

Level 1.

Optional. (When used, `monitor()` requires the `profil()` system service routine.)

NAME

nlist – get entries from name list

SYNOPSIS

```
#include <nlist.h>

int nlist(const char *filename, struct nlist *nl);
```

DESCRIPTION

The routine `nlist()` examines the name list in the executable file whose name is pointed to by *filename*, and selectively extracts a list of values and puts them into the array of nlist structures pointed to by *nl*. Each nlist structure contains at least the following information:

```
char            *n_name;
long            n_value;
unsigned short  n_type;
```

`n_name` points to the symbol name, `n_value` is the value of the symbol, `n_type` the type (or derived type).

nl is terminated with a null name; a null string is placed in the name position of the last nlist structure.

Each symbol name is looked up in the name list of the file. If the name is found, the type and value of the symbol are inserted in the appropriate fields. The type field may be set to 0 unless the file was compiled with the `-g` option of `cc`. If the file was compiled with the `-g` option, the type field may contain information such as whether the symbol is a function or an object, but, in general, may not contain useful information. If the name is not found, both entries are set to 0.

RETURN VALUE

Returns –1 upon error; otherwise returns 0.

All value entries are set to 0 if the file cannot be read or if it does not contain a valid name list.

LEVEL

Level 1.

NAME

putpwent – write password file entry

SYNOPSIS

```
#include <pwd.h>

int putpwent(const struct passwd *p, FILE *f);
```

DESCRIPTION

The routine `putpwent()` is the inverse of `getpwent()`. Given a pointer to a password structure created by `getpwent()` (or `getpwuid()` or `getpwnam()`), `putpwent()` writes a line on the file *f*, which must have the format of `/etc/passwd`.

RETURN VALUE

Returns a non-zero value if an error was detected during its operation, otherwise returns 0.

SEE ALSO

getpwent(BA_LIB).

FUTURE DIRECTIONS

The function `putpwent()` may be replaced in a future issue of the SVID.

LEVEL

Level 2: September 30, 1989

NAME

sputl, **sgetl** – access long integer data in a machine-independent fashion

SYNOPSIS

cc [*flag* **. . .]** *file* **. . . -lld [***library***] . . .**

#include <ldfcn.h>

void sputl (long *value***, char** ∗*buffer***);**

long sgetl (const char ∗*buffer***);**

DESCRIPTION

sputl takes the four bytes of the long integer *value* and places them in memory starting at the address pointed to by *buffer*. The ordering of the bytes is the same across all machines.

sgetl retrieves the four bytes in memory starting at the address pointed to by *buffer* and returns the long integer value in the byte ordering of the host machine.

The combination of **sputl** and **sgetl** provides a machine-independent way of storing long numeric data in a file in binary form without conversion to characters.

LEVEL

Level 2

Software Development Commands And Utilities

The following section contains the manual pages for SD_CMD routines.

NAME

admin – create and administer SCCS files

SYNOPSIS

admin [-i[*name***]] [-b] [-n] [-r***rel***] [-t[***name***]] [-f***flag***[***flag-val***]]**
[-d*flag***[***flag-val***]] [-a***login***] [-e***login***] [-m[***mrlist***]] [-y[***comment***]]**
[-h] [-z] *file* **. . .**

DESCRIPTION

admin is used to create new SCCS files and change parameters of existing ones. Arguments to **admin**, which may appear in any order, consist of keyletter arguments (that begin with **-**) and file names (note that SCCS file names (*file*) must begin with the ASCII characters **s.**).

If *file* does not exist, it is created and its parameters are initialized according to the specified keyletter arguments. Parameters not initialized by a keyletter argument are assigned a default value. If *file* does exist, parameters corresponding to specified keyletter arguments are changed, and other parameters are left unchanged.

If *file* is a directory, **admin** behaves as though each file in the directory were specified as *file*, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored.

If *file* is **-**, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

admin recognizes supplementary code set characters in all files, as well as in file names and in arguments given to the **-i**, **-t**, **-f**, and **-y** options (see below), according to the locale specified in the **LC_CTYPE** environment variable As noted, file names must begin with the ASCII characters **s.**

The keyletter arguments are listed below. Each argument is explained as if only one *file* were to be processed because the effect of each argument applies independently to each *file*.

-i[*name*] The *name* of a file from which the contents for a new SCCS file are to be taken. (If *name* is a binary file, then you must specify the **-b** option.) This contents constitutes the first delta of the file (see **-r** keyletter for delta numbering scheme). If the **-i** keyletter is used, but *name* is omitted, the contents are obtained by reading the standard input until an end-of-file is encountered. If this keyletter is omitted, then the SCCS file is created so that the result of a **get**(SD_CMD) will be an empty file. Only one SCCS file may be created by an **admin** command on which the **i** keyletter is supplied. Using a single **admin** to create two or more SCCS files requires that they be created empty (no **-i** keyletter). Note that the **-i** keyletter implies the **-n** keyletter. Supplementary code set characters may be used in *name* and in the file itself.

-b encode the contents of *name*, specified to the **-i** option. This keyletter must be used if *name* is a binary file; otherwise, a binary file will not be handled properly by SCCS commands.

-n This keyletter indicates that a new SCCS file is to be created (implied by **-i**).

-r *rel* The *rel*ease into which the initial delta is inserted. This keyletter may be used only if the **-i** keyletter is also used. If the **-r** keyletter is not used, the initial delta is inserted into release 1. The level of the initial delta is always 1 (by default initial deltas are named 1.1).

-t[*name*] The *name* of a file from which descriptive text for the SCCS file is to be taken. If the **-t** keyletter is used and **admin** is creating a new SCCS file (the **-n** and/or **-i** keyletters also used), the descriptive text file name must also be supplied. In the case of existing SCCS files: (1) a **-t** keyletter without a file name causes removal of the descriptive text (if any) that is currently in the SCCS file, and (2) a **-t** keyletter with a file name causes text (if any) in *file* to replace the descriptive text (if any) that is currently in the SCCS file. Supplementary code set characters may be used in *name* and in the file itself.

-f *flag* This keyletter specifies a *flag*, and, possibly, a value for the *flag*, to be placed in the SCCS file. Several **-f** keyletters may be supplied on a single **admin** command line. The allowable *flag*s and their values are:

- **b** Allows use of the **-b** keyletter on a **get** command to create branch deltas.
- **c** *ceil* The highest release (that is, ceiling): a number greater than 0 but less than or equal to 9999 that may be retrieved by a **get** command for editing. The default value for an unspecified **c** flag is 9999.
- **f** *floor*
  The lowest release (that is, floor): a number greater than 0 but less than 9999 that may be retrieved by a **get** command for editing. The default value for an unspecified **f** flag is 1.
- **d** *SID* The default delta number (SID) to be used by a **get** command.
- **i**[*str*] Causes the **No id keywords (ge6)** message issued by **get** or **delta** to be treated as a fatal error. In the absence of this flag, the message is only a warning. The message is issued if no SCCS identification keywords [see **get**(SD_CMD)] are found in the text retrieved or stored in the SCCS file. If a value is supplied, the keywords must exactly match the given string. The string must contain a keyword, and no embedded newlines.
- **j** Allows concurrent **get** commands for editing on the same SID of an SCCS file. This flag allows multiple concurrent updates to the same version of the SCCS file.
- **l** *list* A *list* of releases to which deltas can no longer be made (**get -e** against one of these ‘‘locked’’ releases fails). *list* has the following syntax:

  <list> ::= <range> | *<list> , <range>*
  <range> ::= RELEASE NUMBER | **a**

The character **a** in *list* is equivalent to specifying all releases for *file*.

n Causes **delta** to create a null delta in each of those releases (if any) being skipped when a delta is made in a new release (for example, in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). These null deltas serve as anchor points so that branch deltas may later be created from them. The absence of this flag causes skipped releases to be non-existent in the SCCS file, preventing branch deltas from being created from them in the future.

q *text* User-definable text substituted for all occurrences of the **%Q%** keyword in SCCS file text retrieved by **get**. Supplementary code set characters may be used in *text*.

m *mod* *mod*ule name of the SCCS file substituted for all occurrences of the **%M%** keyword in SCCS file text retrieved by **get**. If the **m** flag is not specified, the value assigned is the name of the SCCS file with the leading **s.** removed. Supplementary code set characters may be used in the module name *mod*.

t *type* *type* of module in the SCCS file substituted for all occurrences of **%Y%** keyword in SCCS file contents retrieved by **get**.

v[*pgm*]

Causes **delta** to prompt for Modification Request (MR) numbers as the reason for creating a delta. The optional value specifies the name of an MR number validity checking program [see **delta**(SD_CMD)]. This program will receive as arguments the module name, the value of the type flag (see **t** *type* above), and the *mrlist*. (If this flag is set when creating an SCCS file, the **m** keyletter must also be used even if its value is null).

x Causes **get** to create files with execute permissions.

-d *flag* Causes removal (deletion) of the specified *flag* from an SCCS file. The **-d** keyletter may be specified only when processing existing SCCS files. Several **-d** keyletters may be supplied in a single **admin** command. See the **-f** keyletter for allowable *flag* names.

(**l** *list* used with **-d** indicates a *list* of releases to be unlocked. See the **-f** keyletter for a description of the **l** flag and the syntax of a *list*.)

-a *login* A login name, or numerical UNIX System group ID, to be added to the list of users who may make deltas (changes) to the SCCS file. A group ID is equivalent to specifying all login names common to that group ID. Several **a** keyletters may be used on a single **admin** command line. As many logins or numerical group IDs as desired may be on the list simultaneously. If the list of users is empty, then anyone may add deltas. If login or group ID is preceded by a **!** they are to be denied permission to make deltas.

-e *login* A login name, or numerical group ID, to be erased from the list of users allowed to make deltas (changes) to the SCCS file. Specifying a group ID is equivalent to specifying all **login** names common to that group ID. Several **-e** keyletters may be used on a single **admin** command line.

-m[*mrlist*] The list of Modification Requests (MR) numbers is inserted into the SCCS file as the reason for creating the initial delta in a manner identical to **delta**. The **v** flag must be set and the MR numbers are validated if the **v** flag has a value (the name of an MR number validation program). Diagnostics will occur if the **v** flag is not set or MR validation fails.

-y[*comment*]
The *comment* text is inserted into the SCCS file as a comment for the initial delta in a manner identical to that of **delta**. Omission of the **-y** keyletter results in a default comment line being inserted.

The **-y** keyletter is valid only if the **-i** and/or **-n** keyletters are specified (that is, a new SCCS file is being created). Supplementary code set characters may be used in *comment*.

-h Causes **admin** to check the structure of the SCCS file and to compare a newly computed check-sum (the sum of all the characters in the SCCS file except those in the first line) with the check-sum that is stored in the first line of the SCCS file. Appropriate error diagnostics are produced. This keyletter inhibits writing to the file, nullifying the effect of any other keyletters supplied; therefore, it is only meaningful when processing existing files.

-z The SCCS file check-sum is recomputed and stored in the first line of the SCCS file (see **-h**, above). Note that use of this keyletter on a truly corrupted file may prevent future detection of the corruption.

The last component of all SCCS file names must be of the form **s.***file*. New SCCS files are given mode 444 [see **chmod**(BU_CMD)]. Write permission in the pertinent directory is, of course, required to create a file. All writing done by **admin** is to a temporary file, called **x.***file*, [see **get**(SD_CMD)], created with mode 444 if the **admin** command is creating a new SCCS file, or with the same mode as the SCCS file if it exists. After successful execution of **admin**, the SCCS file is removed (if it exists), and **x.***file* is renamed with the name of the SCCS file. This renaming process ensures that changes are made to the SCCS file only if no errors occurred.

It is recommended that directories containing SCCS files have mode 755 and that SCCS files themselves have mode 444. The mode of the directories allows only the owner to modify SCCS files contained in the directories. The mode of the SCCS files prevents any modification at all except by SCCS commands.

admin also makes use of a transient lock file (called **z.***file*), which is used to prevent simultaneous updates to the SCCS file by different users. See **get**(SD_CMD) for further information.

FILES

x.*file* [see **delta**(SD_CMD)]
z.*file* [see **delta**(SD_CMD)]
bdiff Program to compute differences between the ''gotten'' file and the **g.***file* [see **get**(SD_CMD)].

EXAMPLES

The following example shows how to create an SCCS file, **s.prog.c**, from the contents of a file containing a C language program, **prog.c**.

```
admin -iprog.c s.prog.c
```

An example for a file containing a shell program is similar, except that you should use the **-fx** option , so that **get**(SD_CMD) will create **file.sh** to be executable.

```
admin -ifile.sh -fx s.file.sh
```

You should include some SCCS information at the top of a file. In the above shell example, to include the file name, the SCCS version number, and the date and time of the last delta, include the following line at the beginning of **file.sh**:

```
#Id: %W%  Last Delta: %G% %U%
```

The above line would be translated by a **get**(SD_CMD) command as:

```
#Id: @(#)file.sh        1.8 Last Delta: 4/25/91 17:05:19
```

SEE ALSO

delta(SD_CMD), **ed**(BU_CMD), **get**(SD_CMD), **prs**(SD_CMD)

LEVEL

Level 1.

DIAGNOSTICS

Use the **help** command for explanations.

NOTICES

If it is necessary to patch an SCCS file for any reason, the mode may be changed to 644 by the owner allowing use of a text editor. You must run **admin -h** on the edited file to check for corruption followed by an **admin -z** to generate a proper check-sum. Another **admin -h** is recommended to ensure the SCCS file is valid.

NAME

as – common assembler

SYNOPSIS

`as` [`-o`*objfile*] [`-m`] [`-V`] *file*

DESCRIPTION

The `as` command assembles the named file. The following options may be specified in any order:

`–o` *objfile* Put the output of the assembly in *objfile*. Without this option, the default behavior is to create the output file name by removing the suffix, if there is one, from the input file name and appending a suffix.

`–m` Run the `m4` macro pre-processor on the input to the assembler.

`–V` Write the version number of the assembler being run on the standard error output.

USAGE

General.

The command `cc` is the recommended interface to the assembler. The `as` command may not be present on all implementations of System V.

If the `–m` option (`m4` macro pre-processor invocation) is used, keywords for `m4` [see m4(SD_CMD)] cannot be used as symbols (variables, functions, labels) in the input file since `m4` cannot determine which are assembler symbols and which are real `m4` macros.

SEE ALSO

cc(SD_CMD), ld(SD_CMD), m4(SD_CMD).

FUTURE DIRECTIONS

The `–Y` option is reserved for future use. It will be used to allow the user to specify the directories where the `m4` pre-processor and the file of predefined macros are located.

Users will also be able to specify, by means of the `TMPDIR` environmental variable, the directory in which any temporary files are to be created.

These additions are part of the effort to eliminate hard-coded pathnames from the compilation system.

All functionality provided by the `as` command is accessible through the `cc` command. Compilation using the `cc` command may not necessarily invoke `as` as a separate process.

LEVEL

Level 2: June 30, 1989

Optional.

NAME

cc – **C** compiler

SYNOPSIS

`cc` [*options*] *file* ...

DESCRIPTION

The `cc` command is the interface to the C compilation system. The system conceptually consists of preprocessor, compiler, optimizer, assembler, and link-editor phases. The `cc` command processes the supplied options and then executes the various phases with the appropriate arguments.

The suffix of a filename argument indicates how the file is to be treated. Files whose names end with `.c` are taken to be C source programs, and may be preprocessed, compiled, optimized, assembled, and link-edited. The compilation process may be stopped after the completion of any conceptual phase if the appropriate options are supplied. If the compilation process is allowed to complete the assembly phase, then an object program is produced; the object program for a source file called `xyz.c` is created in a file called `xyz.o.` However, the `.o` file is normally deleted if a single C program is compiled and loaded all at one go.

In the same way, arguments whose names end with `.s` are taken to be assembly source programs, and may be assembled and link-edited. Files with names ending in `.i` are taken to be preprocessed C source programs and may be compiled, optimized, assembled, and link-edited. Files whose names do not end in `.c`, `.s`, or `.i` are handed to the link-editor phase.

By default, if an executable file is produced (*i.e.*, the link-editor phase is allowed to complete), the file is called `a.out.` This default name can be changed with the `–o` option (see below).

The following options are interpreted by `cc`:

`–c` Suppress the link-editor phase of the compilation, and do not remove any produced object files.

–d *c* *c* can be either **y** or **n**. If the system supports it, **–dy** specifies a file suitable for dynamic linking. **–dn** specifies a file suitable for static linking. This option and its argument are passed to **ld**.

`–f` Include floating-point support for systems without an automatically included floating-point implementation. This option is ignored on systems that do not need it.

`–g` Cause the compiler to generate additional information needed for the use of a debugger.

–l *name*
Search the library **lib***name***.a** or if shared objects are supported libname.so. Its placement on the command line is significant as a library is searched at a point in time relative to the placement of other libraries and object files on the command line. This option and its argument are passed to **ld**.

`-o` *outfile*
: Use the name *outfile*, instead of the default `a.out`, for the executable file produced. This is a link-editor option.

`-p` Arrange for the compiler to produce code that counts the number of times each routine is called; also, if link-editing takes place, a profiled version of the standard C library is linked, and `monitor()` [see monitor(SD_LIB)] is automatically called. A `mon.out` file will then be produced on normal termination of the program. An execution profile can then be generated by use of `prof`.

`-q` This option is reserved for specification of implementation specific profiling directives.

`-B` *c* *c* can be either **`dynamic`** or **`static`**. If the system supports dynamic linking, **`-B dynamic`** causes the link editor to look for files named **`lib`***x***`.so`** and then for files named **`lib`***x***`.a`** when given the **`-l`***x* option. **`-B static`** causes the link editor to look only for files named **`lib`***x***`.a`**. This option may be specified multiple times on the command line as a toggle. This option and its argument are passed to **`ld`**.

`-E` Preprocess the named C programs and send the result to the standard output.

`-F` This option is reserved for implementation specific optimization directives.

`-G` Used to direct the link editor to produce a shared object rather than a dynamically linked executable. This option is passed to **`ld`**. It cannot be used with the **`-dn`** option.

`-K` [**`PIC`**]

`-K PIC`
: Causes position-independent code (PIC) to be generated if PIC is supported. Other implementation-defined values may be used with this option.

`-L` *dir* Add *dir* to the list of directories searched for libraries by **`ld`**. This option and its argument are passed to **`ld`**.

`-O` Do compilation phase optimization. This option will not affect `.s` files.

`-P` Preprocess the named C programs and leave the result in corresponding files suffixed `.i`.

`-S` Compile and do not assemble or link-edit the named C files. The assembly language output is left in corresponding files suffixed `.s`.

`-V` Cause each invoked phase to print its version information on the standard error output.

`-C` Cause the preprocessing phase to pass along all comments other than those on preprocessing directive lines.

`-D` *name*`[=`*tokens*`]`
: Associates *name* with the specified *tokens* as if by a `#define` preprocessor directive. If no =*tokens* is specified, the token `1` is supplied.

-I *dir* Alter the search for included files whose names do not begin with / to look in dir prior to the usual directories. The directories for multiple -I options are searched in the order specified.

-U *name*
Causes any definition of *name* to be forgotten, as if by a #undef preprocessing directive. If the same *name* is specified for both -D and -U, *name* is not defined, regardless of the order of the options.

-W *c*,*arg1*[,*arg2* ...]
Hand off the argument(s) *argi* to phase *c* where *c* is one of [p02al] indicating preprocessing, compilation, optimization, assembly, or link-editing phases, respectively. For example, -W a,-m passes -m to the assembler phase.

-Y *items*,*dir*
Specify a new directory, *dir*, for the location of *items*. *items* is any grouping of following characters representing directories containing special files:

I directory searched last for include files
P new search path to locate libraries, *dir* takes the form of $PATH.
S directory containing the start-up object files

or, depending upon the implementation, it may also be one of [p02al].

If the location of a phase [p02al] is specified and the phase does not exist as a separate process, then cc may ignore the -Y option for that phase.

If the location of a phase is being specified, then the new pathname for the phase will be *dir*/*phase*. If more than one -Y option is applied to any one item, then the last occurrence holds.

The cc command passes any unrecognized options to the link-editor phase without any diagnostic [see ld(SD_CMD) for descriptions of ld options].

Other arguments are taken to be C-compatible object programs or libraries of C-compatible routines and are passed directly to the link-editor phase. These programs, together with the results of any compilations specified, are linked (in the order given) to produce an executable program with the name a.out (unless the -o link-editor option is used).

The standard C library is automatically available to the C program. Other libraries must be specified explicitly using the -l option with cc [see ld(SD_CMD) for details].

FILES

file.c input file

file.i preprocessed C source file

file.o object file

file.s assembly language file

a.out
link-edited (executable) output

SEE ALSO

ld(SD_CMD), prof(SD_CMD), sdb(SD_CMD), exit(BA_OS), monitor(SD_LIB), Programming Language Specifications Extension.

USAGE

General.

Because the `cc` command usually creates files in the current directory during the compilation process, it is typically necessary to run the `cc` command in a directory in which a file can be created.

The meaning of the terms shared library and dynamic linking are described in the System V ABI.

FUTURE DIRECTIONS

Users will also be able to specify, by means of the `TMPDIR` environment variable, the directory in which any temporary files are to be created.

This addition is part of the effort to eliminate hard-coded pathnames from the compilation system.

If the *c* phase of the `-W` option does not exist as a separate process, then `cc` may ignore the `-W` option for that phase.

LEVEL

Level 2, June 30, 1989
The following options are dependent upon dynamic linking being supported and therefore are marked as Optional:
-d, -B, -K PIC
The following options are marked Level 2, effective September 30, 1993, and will be removed when the three year waiting period has expired:
-f, -F

NAME

cflow – generate C flowgraph

SYNOPSIS

cflow [-r] [-ix] [-i_] [-d*num***]** *files*

DESCRIPTION

The **cflow** command analyzes a collection of C, **yacc**, **lex**, assembler, and object files and builds a graph charting the external function references. Files suffixed with **.y**, **.l**, and **.c** are processed by **yacc**, **lex**, and the C compiler as appropriate. The results of the preprocessed files, and files suffixed with **.i**, are then run through the first pass of **lint**. Files suffixed with **.s** are assembled. Assembled files, and files suffixed with **.o**, have information extracted from their symbol tables. The results are collected and turned into a graph of external references that is written on the standard output. **cflow** processes supplementary code set characters in literals and constants according to the locale specified in the **LC_CTYPE** environment variable [see **LANG** on **envvar**(BA_ENV)].

Each line of output begins with a reference number, followed by a suitable number of tabs indicating the level, then the name of the global symbol followed by a colon and its definition. Normally only function names that do not begin with an underscore are listed (see the **-i** options below). For information extracted from C source, the definition consists of an abstract type declaration (for example, **char** *), and, delimited by angle brackets, the name of the source file and the line number where the definition was found. Definitions extracted from object files indicate the file name and location counter under which the symbol appeared (for example, *text*). If the compilation system adds a leading underscore to external names, it is removed. Once a definition of a name has been printed, subsequent references to that name contain only the reference number of the line where the definition may be found. For undefined references, only **<>** is printed.

As an example, suppose the following code is in **file.c**:

```
int  i;

main()
{
     f();
     g();
     f();
}

f()
{
     i = h();
}
```

The command

```
cflow –ix file.c
```

produces the output

```
1     main: int(), <file.c 4>
2           f: int(), <file.c 11>
3                 h: <>
4                 i: int, <file.c 1>
5           g: <>
```

When the nesting level becomes too deep, the output of **cflow** can be piped to the **pr** command, using the **-e** option, to compress the tab expansion to something less than every eight spaces.

In addition to the **-D**, **-I**, and **-U** options (which are interpreted just as they are by **cc**), the following options are interpreted by **cflow**:

-r Reverse the ''caller:callee'' relationship producing an inverted listing showing the callers of each function. The listing is also sorted in lexicographical order by callee.

-ix Include external and static data symbols. The default is to include only functions in the flowgraph.

-i_ Include names that begin with an underscore. The default is to exclude these functions (and data if **-ix** is used).

-d*num* The *num* decimal integer indicates the depth at which the flowgraph is cut off. By default this number is very large. Attempts to set the cutoff depth to a nonpositive integer will be ignored.

Errors

Complains about multiple definitions and only believes the first.

SEE ALSO

as(SD_CMD), **cc**(SD_CMD), **lex**(SD_CMD), **lint**(SD_CMD), **nm**(SD_CMD), **yacc**(SD_CMD)

LEVEL

Level 2.

NOTICES

Files produced by **lex** and **yacc** cause the reordering of line number declarations, which can confuse **cflow**. To get proper results, feed **cflow** the **yacc** or **lex** input.

NAME

chroot – change root directory for a command

SYNOPSIS

`/usr/sbin/chroot` *newroot command*

DESCRIPTION

The command `chroot` executes the given *command*, relative to root *newroot*. The meaning of any initial slashes (`/`) in path names is changed for *command* and any of its children to *newroot*. Furthermore, the initial working directory is *newroot*.

This command is restricted to the super-user.

Notice that:

`chroot` *newroot command* `>x`

will create the file `x` relative to the original root, not the new one.

The new root path name is always relative to the current root: even if a `chroot` is currently in effect, the *newroot* argument is relative to the current root of the running process.

SEE ALSO

chdir(BA_OS)

USAGE

General.

The user should exercise caution when referencing special files in the new root file system.

LEVEL

Level 1.

NAME

cxref – generate C program cross-reference

SYNOPSIS

cxref [*options***]** *files*

DESCRIPTION

The **cxref** command analyzes a collection of C files and builds a cross-reference table. **cxref** uses a special version of **cc** to include **#define**'d information in its symbol table. It generates a list of all symbols (auto, static, and global) in each individual file, or, with the **-c** option, in combination. The table includes four fields: NAME, FILE, FUNCTION, and LINE. The line numbers appearing in the LINE field also show reference marks as appropriate. The reference marks include:

assignment	=
declaration	–
definition	*

If no reference marks appear, you can assume a general reference.

cxref processes supplementary code set characters according to the locale specified in the **LC_CTYPE** environment variable [see **LANG** on **envvar**(BA_ENV)].

The **-D**, **-I**, and **-U** options are interpreted as by **cc**. In addition, **cxref** interprets the following options:

-c Combine the source files into a single report. Without the **-c** option, **cxref** generates a separate report for each file on the command line.

-o *file* Direct output to *file*.

-s Operates silently; does not print input file names.

-w *num* Width option that formats output no wider than *num* (decimal) columns. This option will default to 80 if *num* is not specified or is less than 51.

-V Prints version information on the standard error.

-W*name***,** *file***,** *function***,** *line*
Changes the default width of at least one field. The default widths are:

Field	Columns
NAME	15
FILE	13
FUNCTION	15
LINE	20 (4 per table column)

EXAMPLES

```
a.c
1    main()
2    {
3         int i;
4         extern char c;
5
6         i=65;
7         c=(char)i;
8    }
```

Resulting cross-reference table:

```
NAME   FILE         FUNCTION   LINE
c      a.c          ---        4-     7=
i      a.c          main       3*     6=    7
main   a.c          ---        2*
u3b2   predefined   ---        0*
unix   predefined   ---        0*
```

Errors

Error messages usually mean you cannot compile the files.

SEE ALSO

cc(SD_CMD).

USAGE

General.

LEVEL

Level 2: June 30, 1989.

NAME

debug – source-level, interactive, object file debugger

SYNOPSIS

debug [*opts***][[-f none|procs|all][-r][-l** *start_loc***]** *cmd_line***]**

debug [*opts***][-f none|procs|all][-l** *object_file***]** *process_id* **...**

debug [*opts***] -c** *core_file object_file*

*opts***: [-V][-i c|x][-X** *opt***][-d** *defaults***][-s** *path***][-Y***item,dir***]**

DESCRIPTION

debug is a tool that facilitates the finding of errors in user programs by allowing the user to control the execution of a program and examine its state. The user can create a new process from an executable program, take over control of an existing process, or examine the state of a process that terminated abnormally with a **core** dump

To take full advantage of the symbolic capabilities of **debug**, the programs examined and controlled by **debug** should be compiled with the **-g** option to the compiler [see **cc**(SD_CMD)]. If the controlled program has not been compiled with **-g**, the capabilities of **debug** will be limited, but the program can still be controlled and examined.

Some implementations of **debug** provide both a command line interface and an X Windows based graphical user interface. Only the command line interface is described here.

Invocation

debug can be invoked in one of three ways. In the first, the user may specify a *cmd_line*. *cmd_line* consists of one or more executable files, and their associated arguments. The individual commands can be linked by shell-style pipes, and the input and output of the *cmd_line* can be redirected (characters special to the shell must be quoted). **debug** creates a new controlled process for each command specified in *cmd_line,* taking care of any necessary redirections of input and output. The processes are set up to stop at the starting address specified by *start_loc*. If no *start_loc* is supplied, the processes are set up to stop at the symbol **main**, if present, otherwise at the starting address specified by the object file. **debug** then **exec**'s each command, passing each the specified arguments.

If no *cmd_line* is specified, **debug** simply enters interactive mode.

In the second form of invocation, the user specifies one or more existing processes by giving a list of *process_id*s. The debugger attempts to control the specified objects as live processes and, if successful, suspends their execution.

Finally, the user may specify an executable program in one of the object file formats understood by **debug**, along with a *core_file*. **debug** interprets the *core_file* as a record of the process state at the time of the death of the process associated with the *object_file* and lets the user examine the contents of the process stack, registers and data segments.

debug associates the name of each object (program name) with all processes derived from the current invocation of that object. This name may be used in any command that accepts a process list. If the object name matches the name of an already existing debugger-controlled program, the debugger will create a new name for the program. The default program name may be reset using the **rename** command (see below).

Options

The following options are recognized:

- **-c** Associate the **core** image *core_file* with the specified *object_file*.
- **-d** Specify a *defaults* file containing debugger commands. If no *defaults* is given, **debug** will search for a file called **.debugrc** in the user's home directory. If a default command file exists, **debug** executes the commands it contains before it processes any other command line options or user requests.
- **-f** Specify whether **debug** will follow all child processes created by any of the *live_object*s or by any of the programs given in the *cmd_line*, (**procs**, or **all**) or none of the child processes (**none**). See ''Process Control''.
- **-i** The interface mode for the debugger. **-i c** instructs **debug** to use the command line interface. **-i x** instructs **debug** to use the X Window based interface, if supported. If no **-i** option is given, **debug** uses the X Window interface, if the necessary hardware and software is present, otherwise, it uses the command line interface.
- **-l** For the first form of invocation, specify the location at which **debug** will stop the process after it is created. For the second form of invocation, specify an alternate *object_file* from which to load symbolic information when debugging a *process_id*. If no alternate object is specified, **debug** finds the object file from which the process image was created. If **-l** is used, only one *process_id* may be specified. See **create** and **grab** under ''Commands''.
- **-r** Redirect input and output of the created objects to a pseudo-terminal (this does not affect subsequent redirection by the shell or the processes themselves). See ''Redirection of Process I/O''.
- **-s** Specify initial value for the global search path, **%global_path**. The *path* is a colon separated list of directory pathnames. See ''Directory Search Paths''.
- **-V** Print out version information about **debug**.
- **-X** Specify option to be passed to the X Windows initialization routine. This option may be specified multiple times.
- **-Y** Specify a new directory *dir* for the location of *item*. *item* can consist of any of the following:
  - **a** file containing definitions of built-in aliases for **debug**
  - **g** graphical user interface for **debug**

Command Language

debug provides a simple, user-extensible command language, with a syntax similar to **sh**(BU_CMD) in style, using keywords and dash options. Command options may appear in any order. Multiple options may be specified together, as in **symbols -lf** or separately, as in **symbols -l -f**, but multiple occurrences of the same option letter are invalid.

Several commands separated by semi-colons (**;**) may be given on a single line. A backslash (****) at the end of a line indicates that the command is continued on the following line. The output of a command may be redirected to a file or shell pipeline using the **sh** syntax of **>**, **>>** and **|**. (For example, **symbols -g | pg**). As in the shell, **>** and **>>** may appear anywhere within a command, but **|** must appear at the end of a debugger command, since the rest of the line is treated as a shell command that will receive the output of the debugger command. A sequence of debugger commands may be enclosed in curly braces (**{}**), forming a command *block*. The output of such a block may be redirected as a whole. A debugger comment is introduced by a pound sign (**#**). Any characters following a pound sign on a line will be ignored.

Many debugger commands have built-in aliases. These are one or two character names that may be used wherever the full command is used. The user can redefine any of the built-in aliases, or may define his or her own aliases. An alias can consist of any valid debugger command sequence and may take parameters. See **alias** under ''Commands'' for more details.

Built-In Variables

debug maintains a set of special variables that describe the current debugger state and allow the user to customize certain debugger features. These variables all begin with a percent sign (**%**). The processor registers are also considered to be built-in variables and use the same naming convention. The current value of a debugger variable may be seen with the **print** or **symbols** commands. Some built-in variables are read-only. Those that can be modified may be changed using the **set** command.

User-Defined Variables

The user may also define variables in the debugger. The names of these variables consist of a dollar sign followed by a **C**-style identifier (**$***username*). A user-defined variable is defined by assigning it an initial value using the **set** command, and may subsequently be modified. All of the user's environment variables are imported to debugger variables of the same name when **debug** is invoked.

User-defined variables are polymorphic, having either string or numeric values, according to the type of the last value assigned to them. Any variable, string or numeric, may be used where a string value is required, and any string-valued variable which is convertible to an integer via the **strtol**(BA_LIB) function may be used where a numeric value is required.

Process Control

debug provides control over both single and multiprocess applications and over both single and multithreaded processes. For each active process under its control, **debug** detects when the object program and shared library association changes and maintains current knowledge of the associations. In particular, processes may attach or detach shared objects into/from their address spaces using the interfaces **dlopen**(BA_OS), **dlclose**(BA_OS), **dlsym**(BA_OS).

debug provides control of an arbitrary number of threads within a given process. These threads may be bound threads or multiplexed threads (see **thr_create**). The only restriction is that in some implementations, the user may not be able to start (**run** or **step**) a multiplexed thread that is not currently associated with some operating system execution entity. Some implementations refer to this execution entity as LWP.

By default, **debug** detects when a new process is created by one of its controlled threads or processes and includes the new object in its set of controlled objects. The user can release such newly created objects from debugger control by using the **release** command (see below). The default behavior may be overridden by individual **create** or **grab** commands, or may be changed by setting the value of the built-in variable **%follow**. Legal values are:

- **none** Do not control child processes.
- **procs** Follow all child processes.
- **all** Follow all child processes (same as **procs**).

debug assigns a unique identifier to each process and thread under its control. Process identifiers are in the form **p***id* (**p1**, **p2**, **p3**, ...). Thread identifiers are in the form **p***id***.***id* (**p1.1**, **p1.2**, **p2.5**, ...). **debug** maintains a record of the current process in the built-in variable **%proc**. The current thread is maintained in the built-in variable **%thread**. For all debugger commands that accept an optional list of threads and processes, the default action, if no such list is given, is to apply the command to the current thread (or the current process, if it is single-threaded).

Foreground and Background Execution

When the user enters a command that sets a controlled object in motion, **debug**, by default, waits for that object to stop before returning control to the user. If the debugger built-in variable **%wait** is set to **0** or **no**, or **background**, the debugger does not wait for the affected object to stop. The default behavior may be re-asserted by setting **%wait** to **1** or **yes**, or **foreground**. This global behavior may be overridden by each command that sets a process in motion.

Redirection of Process I/O

When the user creates a debugger-controlled object, **debug** does not, by default, attempt to intercept the input or output for the generated processes. Subject process output is unlabeled, and the subject competes with the debugger for the terminal input. If the debugger variable **%redir** is set to **1** or **yes**, the process or thread I/O is redirected to a pseudo-terminal. All output from that process or thread is labeled with an indication of which pseudo-terminal has been written. Subsequent input to the process or thread must be made through the **input** command (see below). The default behavior may be re-asserted by setting **%redir** to **0** or **no**. This global behavior may be overridden by an individual **create** command.

debug does not attempt to redirect the I/O of grabbed processes, or of the child processes of some created subject, since it cannot tell what those processes may have already done to redirect their own I/O. Note, too, that all of the processes and threads that result from a single **create** command read and write from/to the same pseudo-terminal.

Process Lists

A *process list* is a way to specify one or more processes and threads as the target of a command. Many debugger commands take an argument (**-p** *proc_list*) that lists the names of those processes and threads which should be affected by the command. A *program* is the set of all processes and threads created as the result of invoking a single binary executable. It does not include processes created from different executables when a process within a program **exec**s.

The command language represents process lists as comma-separated lists of process names. A *process name* is defined as either:

the keyword **all**, denoting all controlled processes, processes and threads,

a user or debugger-generated *program name*, denoting all processes and threads created from the current invocation of the same executable,

a debugger-generated *thread id*, of the form **p***integer.integer*, denoting a specific controlled thread,

a debugger-generated *process id*, of the form **p***integer*, denoting either all threads that belong to a specific controlled process, or, if the process is not multithreaded, the process itself, denoting a specific controlled object,

the debugger built-in variable **%program**, denoting all active processes and threads derived from the current program,

the debugger built-in variable **%thread**, the current thread,

the debugger built-in variable **%proc**, the current process, or all threads derived from the current process,

a decimal integer, denoting the process which has the given integer as its system *pid* (or all threads derived from that process),

any user-defined variable that has an integer value, interpreted as a system *pid*,

any user-defined variable that has a string value, which can be interpreted as one of the above forms, or as a list of the above forms.

Context Variables

The context for the execution of most debugger commands that describe the state of controlled objects is determined by a subset of the debugger built-in variables. **%program**, **%proc** and **%thread** **%program** and **%proc** determine the object(s) to which a command applies. Setting one affects the others. In addition, there is a set of context variables specific to each thread or process. For each controlled object, the following debugger built-in variables are available:

%db_lang	The source language of the current context.
%frame	The current frame (an integer representing the frame number).
%func	The current function.
%file	The current source file.
%line	The current source line number.
%list_file	The next file to be displayed by the **list** command.
%list_line	The next line to be displayed by the **list** command.
%loc	The current program address.

These variables are reset whenever the thread or process that owns them stops for any reason. **%frame** may be explicitly set by the user to any active frame and changes the value of the other context variables accordingly. **%func** may be explicitly set to any function with a currently active frame and results in setting **%frame** to the most recent instance of that function. **%db_lang**, **%file**, **%line**, and **%loc** are read-only. If no debugging or symbolic information is available for the current function, **%db_lang**, **%func**, **%file**, **%list_file**, **%line**, and **%list_line**, may be null.

Verbosity Levels

When a user process or thread under the debugger's control stops for any reason, single step, breakpoint, signal, and so on, the debugger generates output to the terminal. This output can sometimes be more voluminous than the user would desire. For that reason the amount of user-visible output can be adjusted on a global basis by setting the **%verbose** variable. The legal values are:

quiet No output is generated for debugger events.

source The debugger displays the next source or disassembly line.

events If the process stops for an event (system call, signal or stop event) the debugger announces the type of event and the current location. For all stops, it displays the next source line.

reason (default) This is the same as **events**, except that the debugger announces each single step in addition to all of the events.

all The highest verbosity level. Currently, this is the same as **reason**.

Certain commands allow the user to specify the **quiet** verbosity level, with a **-q** option, overriding the global **%verbose** setting.

Thread State Changes

A thread may undergo several different kinds of state changes during its lifetime: it is created and it exits; it can be suspended or continued; and a multiplexed thread may give up its LWP or be picked up by an LWP. The debugger variable **%thread_change** governs the behavior of the debugger when any of these state changes occur. The valid values are:

ignore The debugger will not print a message announcing the change or stop the thread involved. A newly created or continued thread, or a thread picked up by an LWP will be set running, if possible.

announce The debugger will print a message announcing the state change but will not stop the thread involved. A newly created or continued thread, or a thread picked up by an LWP will be set running, if possible.

stop (default) The debugger will print a message announcing the state change and stop the thread involved, if possible. A continued thread or a thread picked up by an LWP will be stopped (or in the Off LWP state). For thread creation, the thread that created the new thread will be stopped and the new thread will stop when it reaches the function specified in the **thr_create** call.

Directory Search Paths

To associate program addresses with source listings, **debug** must know where to look for the source of the programs being debugged. The built-in variable **%global_path** contains a colon-separated list of directory pathnames. **debug** combines this information with the names of source files it derives from the debugging information in the object file, to search for source code. In addition to the global path, each *program* may have a program-specific path. This path is stored in the built-in variable **%path**. Each program has its own version of this variable. When attempting to find the source for a given program, **debug** searches first the list of directories in that program's **%path** variable, and then the list specified by **%global_path**.

Events

Events in the debugger are triggers in the execution sequence of a process or thread that cause control to pass from the process or thread to the debugger. These triggers are activated at the user's request and consist of changes in the process address space, signals and entry to or exit from system calls. Events may also consist of user-specified actions taken by the debugger when a controlled entity stops for any reason.

Event triggers may apply to a given thread or process or to a set of threads and processes. The event fires if any of the specified objects encounter the trigger. Commands that create events apply, by default, to the current program, rather than the current thread. current process.

With each event, the user may specify an optional debugger command block. This block is executed whenever the event triggers. Events and their associated commands can be deleted, or temporarily deactivated and then reactivated.

For each user-specified event, **debug** assigns a unique identifier in a common name space. This identifier may be used in the commands that delete, enable, disable and list events. The last event identifier assigned is maintained in the special variable **%lastevent**, which is updated automatically by the debugger. When an event triggers, **debug** executes the commands associated with the event, after setting the special variables **%program**, **%proc**, **%thread**, **%file**, **%line**, **%func**, **%frame**, **%loc**, **%db_lang** to indicate the process and location at the context in which the event occurred, and **%thisevent** to the event number of the triggered event. These variables are set only for the execution of the commands associated with the triggering event. They revert to their previous values (or are updated to reflect the new debugger state) when those commands complete.

The default action for each event is to announce the occurrence of the event and display the current source line (or current instruction, if no line number information or source is available).

When a controlled process dies, **debug** remembers the events created for that process. If a new process is created for the same program, all events that applied to the entire program (the default) are re-instantiated for the new process. Events that were created to apply only to a single process within a multiprocess program or to a single thread, are not recreated. Similarly, when a process creates a new child process via **fork**(BA_OS), all events that apply to the entire program from which the parent process is derived are copied in the child process. Events that apply to the parent process only or to a single thread are not copied.

When a new thread is created within a process, all events that applied to the entire process or the entire program are copied in the sibling thread. Events that applied only to the original thread that created the new thread are not copied.

Expressions

Many debugger commands accept programming language *expressions*. Each expression is evaluated using the syntax and semantics of the current language, subject to possible limitations of the debugger on that language. The current language, **%db_lang**, is determined dynamically from the source language of the current file. The debugging information in an object file supplies a language attribute describing the programming language of the source file. If the debugger cannot determine the program's source language, **%db_lang** defaults to C. The user may override the information in the object file by setting the variable **%lang**. If the user sets **%lang** to the null string (**""**), the debugger reverts to using **%db_lang**. Expressions referencing variables defined in files compiled from different languages do not change the current language.

debug accepts expressions containing any combination of program variables or functions, qualified names, built-in debugger variables, and user-defined debugger variables. A qualified name specifies a program identifier that may not be visible in the current context. The syntax is:

> **[[***thread id***]@][[***file***]@][[***function***]@][[***line number***]@]***identifier*
> **[[***thread id***]@][[***frame number***]@]***identifier*
> **[[***process id***]@][[***file***]@][[***function***]@][***line number***]@]***identifier*
> **[[***process id***]@][[***frame number***]@]***identifier*

The qualified name is evaluated left to right, and may be disambiguated by supplying **@**'s as needed.

Expressions beginning with a dash (**-**) or containing character sequences with special meanings to **debug** must be enclosed in parentheses, square brackets or curly braces. The special character sequences are: **>**, **>>**, **|**, **||**, **&&**, **#**, '**,**', **;**, **newline**.

Support for the C Language

When **%db_lang** is set to **C**, **debug** supports evaluation of all legal ANSI C expressions, except those involving macro expansion, or structure, union, or enumeration type declarations. An example of a type declaration in an expression is

```
((struct { int i; char c; } *)p)->c = 'a';
```

Definitions

The syntax and semantics for each command are described below. The following terms are used in the synopses and descriptions:

address A constant, user-defined variable, built-in variable or register name that evaluates to an address (an integer or pointer value).

block A list of commands, enclosed in braces, separated by newlines or semicolons.

call A system call name or number. Case is not significant.

cmd	A simple command or a *block*.
cmd_line	A shell-style command line (possibly including shell scripts, environment variables, pipes, and I/O redirection) which will be interpreted by the shell, but the resulting processes will be controlled by the debugger.
core_file	The relative or complete pathname of a file which was created by the kernel upon abnormal termination of some process.
count	An unsigned decimal integer.
event_command	Any of **onstop**, **stop**, **signal** or **syscall**.
event_num	A small integer, assigned by the debugger when any event is created, that identifies the resulting set of actions.
expr	An expression in the current language. See ''Expressions'', above.
func_name	The name of a function in the current process.
location	A designation of an address in a subject process. It includes line numbers, program symbols, processor registers, and limited expressions involving these components. The syntax is: *address*[±*constant*] **# includes debugger and user variables** [*thread id*@][*filename*@]*func_name*[±*constant*] [*thread id*@][*filename*@]*line_number* [*process id*@][*filename*@]*func_name*[±*constant*] [*process id*@][*filename*@]*line_number*
object_file	The relative or complete pathname of an executable object file.
pattern	Simple regular expressions used to restrict a list of names. **sh**(BU_CMD) syntax is used.
process_id	A system process identifier.
proc_list	See ''Process Lists''.
reg_exp	A simple internationalized regular expression using the syntax accepted by **ed**(BU_CMD).
signal	A signal name or number. A signal name may be specified with or without the **SIG** prefix, and case is not significant.
stop_expr	An expression denoting conditions under which specified processes should be stopped. See **stop**.
. . .	Denotes optional repetition of the preceding element.
xxx|*yyy*	Denotes that either *xxx* or *yyy*, but not both, may appear.

Commands

! *shell-command*

This command passes the entire command line, less the exclamation mark, to the shell (**$SHELL**, if set, or else **/usr/bin/sh**) for execution. Note that any redirection will be interpreted by the shell, not the debugger.

If the shell escape operator is given twice, with no arguments, that is, **!!**, **debug** re-executes the last shell escape specified.

alias [-r] [*name* **[***tokens***]]**

The **alias** command, with no arguments, lists the current aliases and their definitions. If the **-r** option is present, it removes the alias with the given *name* from the list of aliases. If no **-r** option is present, but a *name* is given, the **alias** command displays the definition, if any, for the alias with the given *name* . If any characters, other than spaces, tabs, or comments, follow the *name* argument, the command establishes a new alias for the *name* , consisting of all the characters up to, but not including, the comment or newline.

Alias definitions may contain the special identifiers **$1**, **$2**, . . . Each such special identifier **$***n* in an alias definition is replaced by the *n*th argument in an alias invocation, where the arguments are numbered beginning at 1. Each argument must be preceded by whitespace and is terminated by whitespace, a newline, the comment character (**#**) or the beginning of a block (**{**). The special identifiers **$1**, **$2**, . . . will not be replaced within a quoted string.

If an alias definition contains the special identifier **$#**, it will be replaced during invocation of the alias with the number of arguments actually used during the current alias invocation. If an alias definition contains the special identifier **$***, it will be replaced during invocation of the alias with a list of all arguments passed during the current alias invocation, each separated from the next by a single space.

Aliases may be defined in terms of other aliases, but not recursively. At least 20 levels of nested alias definitions are supported.

If the *name* given is the same as any existing built-in command, a warning will be generated. Aliases take precedence over built-in commands.

break The **break** command causes the debugger to exit from the innermost enclosing **while** loop (see **while**).

cancel [-p *proc_list***] [***signal* **...]**

cancel takes a list of signals, that are specified as in the **kill** command. If **debug** has intercepted any of the listed signals for any of the specified objects, it will ensure that those objects do not see the specified signals when they continue execution. If no signals are specified, **debug** cancels all pending signals for the specified objects.

cd [*pathname***]**

The **cd** command changes the debugger's current working directory to *pathname*. If no *pathname* is given, **cd** uses the directory specified by the environment variable **HOME**.

change *event_num* **[-p** *proc_list***] [-eqvx] [-c** *count***] [***stop_expr***|***call...***|** *signal...***] [***block***]**

The **change** command allows the user to modify various attributes associated with a previously assigned event. *event_num* must come before the optional stop expression, signal or system call specifications and must be the number of an event that is currently defined (although it may be disabled).

The list of threads and processes to which the event is applied may be changed with the **-p** option.

The **-q** option specifies that **debug** will not announce the occurrence of the event. **-v** specifies that the event occurrence will be announced.

The **-e** and **-x** options work as in the **syscall** command, and specify whether the system call will be trapped on entry, exit or both entry and exit.

The **-c** option specifies the number of times the event must occur before it triggers. The **-c** option is valid only for **stop** and **syscall** events.

Alternate expressions, signals or system calls and/or an alternate command block, may be specified.

The resulting event will have the same event number as *event_num* . Note that the **change** command does not allow the type of event: **onstop**, **stop**, **signal** or **syscall**, to be changed. Further note that the command list must be in the form of a *block* (that is, with enclosing braces) to distinguish it from a stop expression, system call or signal name.

continue

The **continue** command causes the debugger to begin execution of the next iteration of the innermost enclosing **while** loop. The debugger continues by re-evaluating the **expr** part of the **while** command (see **while**).

create [-f none|procs|all] [-dr] [-l *start_loc***] [***cmd_line***]**

cmd_line consists of one or more executable files, in any of the object file formats understood by the debugger, and their associated arguments. The individual commands can be linked by shell-style pipes, and the input and output of the *cmd_line* can be redirected. Shell meta-characters need not be quoted. The length of *cmd_line* is limited only by the length of the argument list accepted by **exec** (**ARG_MAX**). See **limits**(BA_ENV).

debug creates a new controlled process for each command specified in *cmd_line* , taking care of any necessary redirections of input and output. The processes are set up to stop at the location specified by *start_loc*. If no *start_loc* is supplied, the processes are set up to start at the symbol **main**, if it exists, otherwise at the starting address specified by the object file. **debug** then **exec**'s each command, passing each the specified arguments.

If no *cmd_line* is specified to **create**, **debug** re-executes the last **create** command issued, (first killing all processes created as a result of the last **create** command, if they still exist) in effect, re-running the last process (or processes) created with the same set of arguments.

If the **-r** option is specified, **debug** redirects the I/O of the resulting subjects to a pseudo-terminal, as described above. If the **-d** option is given no redirection is attempted. If neither **-r** nor **-d** is specified, the default is determined by the value of the debugger variable **%redir**.

debug resets its notion of the current program to the first executable specified on the *cmd_line* . The current process is reset to the process generated from that executable. The current thread is set to the first thread in that process, if the program uses the threads interfaces.

The **-f** option may be used to specify whether the debugger should take control of child processes, and overrides the default behavior of the debugger. The arguments to the **-f** option have the same meanings as do the legal values for the **%follow** built-in variable (see ''Process Control'').

debug associates the name of each object (program name) with all processes derived from the current invocation of that object. This name may be used in any command that accepts a process list. If the command name matches the name of an already existing debugger-controlled program, **debug** creates a new name for the program. The default program name may be reset with the **rename** command (see below).

delete *event_num* **...**
delete -a [-p *proc_list***] [***event_command***]**

delete can be invoked in one of two ways. In the first, the user specifies a list of previously assigned event identifiers. **debug** deletes any associated events, removing the planted breakpoint or canceling the signal or system call trigger.

In the second form, all debugger events for the current thread or process (or all events associated with the optional *proc_list)* are deleted. If an *event_command* (**onstop**, **stop**, **signal** or **syscall**) is given, only events of the type specified are deleted.

dis [-p *proc_list***] [-c** *instr_count***] [***location***]**

The **dis** command with no arguments displays the result of disassembling **%num_lines** instructions. **%num_lines** starts out at 10 and may be reset by the user. If an *instr_count* is given, **dis** displays *instr_count* instructions, instead.

If a *location* is given, **dis** begins disassembling at that address. If no location has been specified, and the context for the specified process or thread has not changed since the last **dis** invocation on that object, **dis** begins with the address following the last instruction displayed for that object. Otherwise, **dis** begins its display with the current location, as specified by the debugger variable **%loc**, which is reset whenever the context for the specified process or thread changes.

If more than one thread or process is specified by the *proc_list* argument, the disassembly request is performed for each thread or process in turn.

disable *event_num* **...**
disable -a [-p *proc_list***] [***event_command***]**

disable can be invoked in one of two ways. In the first, the user specifies a list of previously assigned event identifiers. The debugger marks any associated events as inactive, but does not delete them. The event identifiers are still valid, but the actions specified by the events are not performed.

In the second form, all debugger events for the current thread or process (or all events associated with the optional *proc_list*) are disabled. If an *event_command* is given, only events of the type specified are disabled.

dump [-p *proc_list***] [-c** *byte_count***]** *location*

The **dump** command displays **%num_bytes** bytes of memory, 16 bytes per line, starting at the address specified by the *location* truncated to a multiple of 16, in hexadecimal and ASCII. If a *byte_count* is given, that many bytes of memory are dumped instead. **%num_bytes** starts out at 256 and may be set by the user.

If more than one thread or process is specified by the *proc_list* argument, the dump request is performed for each thread or process in turn.

enable *event_num . . .*
enable -a [-p *proc_list***] [***event_command***]**

enable can be invoked in one of two ways. In the first, the user specifies a list of previously assigned event identifiers. For each, if the associated event is currently disabled, the debugger reactivates it.

In the second form, all disabled debugger events for the current thread or process (or all events associated with the optional *proc_list)* are enabled. If an *event_command* is given, only events of the type specified are enabled.

events [-p *proc_list***] [***event_num* **...]**

The **events** command without any arguments prints the entire list of user-specified events for the current program. For each event, the event identifier and status (active or disabled), event type, list of associated processes, the event trigger (stop expression, system call or signal) and the beginning of the associated command list is printed.

If a *proc_list* is specified, those events associated with the list of threads or processes are printed. If a list of event numbers is given, a more detailed record of the specified events is printed, including the full set of associated commands.

export $*username*

The **export** command makes a user-defined variable and its value available in the debugger's environment. The variable is thereafter passed to all processes created by **debug**. If the value of **$***username* is changed using the **set** command, after it has been exported, it must be explicitly re-exported for the new value to be visible in the environment. **$***username* is exported without the leading **$** sign.

grab [-f none|procs|all] [-l *object_file***]** *process_id* **...**
grab -c *core_file object_file*

The **grab** command can take one of two forms. In the first, the user specifies one or more existing processes by giving a list of *process_id*s. In either case, **debug** attempts to control the specified objects as live processes and, if successful, suspends their execution. **debug** resets its notion of the current program to the executable from which the first process specified was derived. The current process is reset to the first process specified.

debug, by default, loads symbolic information for the process from the object file from which the process was created. The **-l** option specifies an alternate *object_file* from which to load symbolic information. If **-l** is used, only one *process_id* may be specified. This option is useful when debugging long running applications that have no symbol information.

The **-f** option may be used to specify whether the debugger should take control of child processes, and overrides the default behavior of the debugger. The arguments to the **-f** option have the same meanings as do the legal values for the **%follow** built-in variable (see ''Process Control'')

In the second form of **grab**, the user specifies an executable program in one of the object file formats understood by the debugger. **debug** interprets the *core_file* as a kernel-created record of the process state at the time of the death of the process associated with the *object_file* and lets the user examine the contents of the process stack, registers and data segments.

debug associates the name of each object with all processes derived from the current invocation of that object. This name may be used in any command that accepts a process list. If the command name matches the name of an already existing debugger-controlled program, **debug** creates a new name for the program. The default program name may be reset using the **rename** command (see below).

halt [-p *proc_list***]**
: **debug** instructs the specified threads or processes to stop execution and waits for them to stop.

help [*topic***]**
: The **help** command, with no arguments, lists all of the available commands and help topics. If a command name is given, it gives a detailed syntax and usage message for that command. If a ''help topic'' name is given, it lists the help available on that topic. Each debugger command has a help message which describes its syntax, options, and usage, and gives examples of its use. In addition, there are help topics which are not also command names, to explain the syntax for process lists, expressions, command output redirection and ''locations,'' and to list the available languages for expression evaluation.

if (*expr***)** *cmd* **[else** *cmd***]**
: This is the traditional conditional branch statement, similar to that present in C, with the exception that semicolons are not necessary, except to separate multiple commands on a single line.

expr can be any valid expression in the current language (see ''Expressions''). The expression is evaluated, and if it evaluates to ''true'' in the semantics of the current language, the *cmd* associated with the **if** clause is executed. Otherwise, if there is an **else** clause, the **cmd** associated with it is executed.

The **if** construct is more likely to be used in commands associated with events, or in scripts, than to be typed interactively as a top-level command.

input [-p *proc_name***|-r** *pseudo_tty***] [-n]** *string*
: The **input** command is used to send user input to a process whose I/O has been redirected by the debugger to a pseudo-terminal (see ''Redirection of Process I/O''). The first argument may be either the name of a single program or process (as specified in a process list), or the name of a pseudo-terminal, as used by **debug** to label process output. If a *proc_name* is specified, **debug** finds the pseudo-terminal (if any) associated with that

program. If neither a program nor a pseudo-terminal is specified, **debug** attempts to find a pseudo-terminal associated with the current program.

debug sends the input *string* to the specified pseudo-terminal, after appending a new-line. If the **-n** option is given, no new-line is appended.

It is an error if the specified **proc_name** has no associated pseudo-terminal.

jump [-p *proc_list***]** *location*

location may be any debugger expression that resolves to an address in one of the specified threads or processes. For each thread or process specified, if the given object is currently stopped, and if the specified *location* is valid for that process, **debug** adjusts the program counter for that object to that *location.* Subsequent **run** or **step** commands for that object continue execution from the specified *location* . **debug** does not attempt to adjust the thread or process stack if the specified *location* is in a different function.

kill [-p *proc_list***] [***signal***]**

kill sends a single signal to the current thread or process or to the list of threads and processes specified by *proc_list*. Unlike most other debugger commands, if a process identifier is given in the *proc_list*, the signal is sent to the process as a whole, rather than to each thread in the process.

If no *signal* is specified, the default is **SIGKILL**. *signal* may be either a valid signal number or a symbolic name, formed from the manifest constant name listed in **signal**(BA_ENV) with or without the **SIG** prefix. Case is ignored.

list [-p *proclist***] [-c** *count***] [***line*|*func_name*|*reg_exp***]**

The **list** command displays source lines for the specified threads or processes. The default is the current thread or process.

If no *count* argument is given, the **list** command displays **%num_lines** source lines. **%num_lines** starts out at 10 and may be reset by the user. If a *count* is given, **list** displays *count* lines, instead.

The starting place for the listing may be specified in several ways. If a regular expression is given, the current file is searched for the next occurrence of a line which matches the given *reg_exp* , beginning from the line immediately following the current line (preceding, if the *reg_exp* is surrounded by question marks). If a match is found, and no *count* is given, only the line containing the match is listed. If a *count* is given, the line containing the match begins the display. **ed**(BU_CMD) syntax is used for regular expressions.

A function name as an argument causes the **list** command to begin its display at the first line of the named function. The function may be specified as in the location syntax: a name, the debugger built-in variable **%func**, or *filename***@** *func_name*.

A line number may be specified as in the location syntax: a single decimal constant, the debugger built-in variables **%line** or **%list_line**, or *filename***@** *line*.

If no starting location is specified, the **list** command begins the display with **%list_file@%list_line**. **%list_file** is set to the current file (**%file**) and **%list_line** is set to the current line (**%line**) whenever the current context changes. In addition, **%list_line** is set to the last line displayed each time **list** is invoked. Thus, if the current context has not changed and no starting location is specified, **list** begins with the last line displayed in the previous **list** invocation.

logoff

The **logoff** command stops session logging.

logon [*filename***]**

The **logon** command starts debugger session logging. All debugger input and output are sent to *filename* in addition to being echoed at the terminal. Output lines are printed as comments.

If no *filename* is given, the last *filename* used in a **logon** command is assumed, and new debugger commands and output are appended to that file.

map [-p *proc_list***]**

The **map** command prints out a list of all mapped segments for the current process, or for each thread or process specified in *proc_list*. The listing includes the virtual address range and access permissions for all segments, and the pathname, for all segments associated with the **a.out** and associated shared libraries.

Note that since all threads within a process share a common address space, the virtual memory map will be identical for each thread within a process.

onstop [-p *proc_list***] [***cmd***]**

The **onstop** command, by default, applies to all threads or processes derived from the current program. The **onstop** command with no arguments prints out the list of **onstop** events with their associated commands.

cmd is a debugger command block. The commands are executed whenever the specified list of processes stops for any reason.

print [-p *proc_list***] [-v] [-f** *fmt***]** *expr* **[,** *expr***] . . .**

The **print** command displays the results of evaluating the (comma-separated) list of expressions. The expressions are evaluated in the context of the current thread or process, unless other threads or processes are specified in the *proc_list* argument. If more than one thread or process is specified, the expressions are evaluated and printed in the context of each specified object, with the **%proc** and **%thread** debugger variables set to the process and thread with the **%proc** debugger variable set to the process identifiers of the object in which the expressions are being evaluated. All events which would be triggered as a side effect of evaluating an expression (breakpoints in a function, a call to which appears in the expression, for example) are ignored, as if they had been disabled.

The **-f** option allows specification of a list of format expressions to be used when printing values. The *fmt* is a string enclosed in quotation marks ("") and may contain a subset of the format expressions accepted by **printf**(BA_LIB). A format expression may have the following form:

%[*flags***][***width***][.[***precision***]][***conversion***]** *specifier*

The *flags*, *width*, *precision*, and *conversion* fields have the same meanings as in the **printf** routine, with the exception that positional parameters are not accepted. The *specifier* may be one of the following characters:

c unsigned char
d,i signed decimal integer
e,E floating point in style **[-]***d.ddde±dd*
f floating point in style **[-]** *ddd.dddd*
g,G floating point in either of above 2 styles
o unsigned octal integer
p **void *** (generic data pointer; hexadecimal address)
s string
u unsigned decimal integer
x,X unsigned hexadecimal integer
z debugger default style for the expression
% **%**

Any character in the *fmt* that is not part of a format expression is printed as given. The default format for a particular expression is determined by the expression evaluator for the current language. The expression evaluators will attempt to present information formatted in a way that is meaningful for the given language. For example, for C, a pointer to a character would be printed as a character string, a reference to an array variable would print all members of that array and dereferencing a pointer to a structure would print each member of that structure. Each *expr* may be any valid expression in the current language (see ‘‘Expressions’’).

Each expression in the list is converted to its printable representation, a newline is added, and the result displayed. This process is repeated for each object named in the *proc_list* . If a *fmt* is given, no terminating newline is printed unless specified in the *fmt* . The **-v** option specifies verbose mode. The debugger prints the function prototype of any function that was called as a result of evaluating the given expressions. This is particularly useful in evaluating C++ expressions to see how overloaded functions or operators are resolved.

ps [-p *proc_list***]**
The **ps** command prints the debugger-generated identifiers, kernel-generated identifiers, current state, location, if the object is stopped, and object name for all controlled threads and processes, or for only those objects specified by the **-p** option, if present.

pwd The **pwd** command prints the debugger’s current working directory. The current working directory may be changed using the **cd** command.

quit The **quit** command causes the debugger to exit, releasing and running any grabbed processes and killing any processes created by the debugger.

If a user wishes to leave a grabbed process suspended, perhaps to be grabbed at a later time from a different invocation of the debugger, he or she should use the **release** command with the **-s** option before quitting.

regs [-p *proc_list***]**
The **regs** command displays in hexadecimal the contents of the processor registers for the current thread or process. If more than one thread or process is specified by the *proc_list* argument, the register display is performed for each process object in turn.

release [-s] [-p *proc_list***]**
debug removes all planted breakpoints from all threads or processes specified in *proc_list* and relinquishes control over them. Releasing all threads within a given process is equivalent to releasing the entire process. If the **-s** option is specified, the processes are released, but halted. Otherwise, the released objects are allowed to continue execution. The **-s** option is ignored for threads. If the current thread or process is released, **debug** chooses a new object to become current.

Processes released in the halted state may be grabbed by the debugger in a different **debug** session.

release can be used on core images as well as live processes. The debugger deletes the core image and associated object file from the list of objects that can be examined.

rename *prog_name name*
The **rename** command changes the name by which a related group of processes are known. All threads and processes derived from a single invocation of the executable from which *prog_name* was derived, can be referred to by the new *name*. *name* can be used in any command that accepts a *proc_list* and will appear in any debugger output that would have used *prog_name*.

run [-p *proc_list***] [-bfr] [-u** *location***]**
debug starts the current thread or single-threaded process or each object specified by *proc_list*. Execution continues from the program address at which it was suspended when the given object last stopped, or at the address specified in a preceding **jump** command.

The **-f** and **-b** options allow the global behavior set by the **%wait** debugger variable to be overridden. **-f** specifies foreground execution for the threads or processes. **-b** specifies background execution.

The **-r** option causes **debug** to continue execution of the given object until each returns from its current stack frame, that is, until the return address of the current function is reached (or until some other event causes execution to halt).

The **-u** option specifies that **debug** continues execution of the specified objects until the address specified by *location* is reached (or until some other event causes execution to halt).

A multiplexed thread that is not currently running on an LWP cannot be set running.

script [-q] *fname*

The **script** command reads and executes debugger commands from the named file. Commands are echoed before execution, unless the **-q** option is given.

Scripts may nest; the debugger implementation does not place a limit on the number of nested scripts (although external limits, such as the number of open files supported by **stdio**, may apply).

set [-p *proc_list***] [-v]** *expr*
set [-p *proc_list***]** *debug_or_user_var* **[=]** *expr* **[,** *expr***] ...**

The **set** command has two forms. In the first, *expr* may be any valid expression in the current language (see ''Expressions''). While any valid language expression may be given, the typical use of the **set** command is to evaluate assignment expressions. The **-v** option specifies verbose mode. The debugger prints the function prototype of any function that was called as a result of evaluating the given expressions. This is particularly useful in evaluating C++ expressions to see how overloaded functions or operators are resolved.

In the second form of the command, **set** is used to change the value of a debugger built-in variable name or user-defined variable name. Debugger built-in variables may have special semantics associated with them, such as **%path**, which requires a string value having a particular structure, or **%frame**, which denotes a frame number and must be within the range of currently active frame numbers. Setting a built-in variable such as **%frame**, may cause the values of other built-in variables to change as well (for example, **%line** or **%func**). There is also an implied string concatenation operator. Any pair of string-valued expressions which appear separated by commas will be concatenated into a single string-valued expression before the assignment is performed.

The *debug_or_user_var* and *expr* are both evaluated in the context of the current thread or single-threaded process, unless one or more other threads or processes have been specified in the *proc_list* argument. If more than one thread or process is specified, the **set** command is evaluated in the context of each of the specified objects, in turn.

signal [-p *proc_list***] [[-iq]** *signal* **... [***cmd***]]**

The **signal** command, by default, applies to all threads or processes derived from the current program. Signals are different from other debugger events in that the debugger catches all signals by default. That is, when a signal is posted to a thread or process, the debugger stops that object and announces that the signal has been posted. The user can then request that the signal be canceled before the thread or process actually receives it (see **cancel**).

debug can be instructed to ignore a given signal for a particular object (or set of objects) with the **-i** option to the **signal** command. So **signal -i sigusr1** instructs the debugger to let **SIGUSR1** go directly to the current thread or process, while **signal sigusr1** re-establishes the default action for **SIGUSR1** for the current object.

The **signal** command can also be used to create events triggered by the receipt of a signal. If a user associates a command block with a signal or set of signals, the debugger creates an event number for that signal in the same name space as the other event commands. These events may be manipulated using **events**, **delete**, **disable** or **enable**. Multiple events may be assigned for the same signal in any given thread or process. The creation of an event for a signal takes precedence over any instruction to ignore that signal (using **signal -i**).

The **-q** option specifies that **debug** will not announce the occurrence of the signal and applies only to signal events.

The **signal** command with no *signal* arguments prints the current signal disposition for each signal and the current list of user-specified signal events, including the event identifier and current status (active or disabled), list of associated processes, signal name and the beginning of any associated command block.

stack [-p *proc_list***] [-c** *count***] [-f** *frame***] [-a** *address***] [-s** *stack***]**

The **stack** command with no arguments prints the entire call stack for the current thread or process. Frames are numbered from 0 for the bottom of the stack (initial stack frame). Displays begin with the top of the stack, unless the **-f** option is given, in which case they begin with *frame.* The *count* argument restricts the display to at most *count* frames from each stack. If more than one object is specified by the *proc_list* argument, the stack request is performed for each object in turn.

The *address* and *stack* arguments may be used to specify beginning values for the program counter and/or stack pointer, respectively. This can be useful when attempting to print a stack trace for a process that has jumped to an illegal address or whose stack pointer has been corrupted. Both the *address* and *stack* arguments must be hexadecimal numbers.

step [-p *proc_list***] [-bfioq] [-c** *count***]**

The **step** command continues execution of the current thread or single-threaded process or of each object specified by *proc_list*.. The **-i** option specifies stepping at the machine instruction level. The specified objects are instructed to execute a single machine instruction, or *count* instructions, if a *count* is specified.

The default is stepping at the source statement level. **debug** continues execution until the object reaches the next source statement as defined by the compiler-generated debugging information. If a *count* is specified, the debugger repeats the **step** command *count* times, or until execution is interrupted by some other event. An explicit *count* of zero is interpreted to mean ‘‘step forever.’’

The **-o** option specifies stepping over function calls. When the debugger encounters a subroutine call while stepping with the **-o** option, it will set a temporary breakpoint at the return point of the call and run at ‘‘full speed’’ until the temporary breakpoint is reached. Stepping over function calls is available with both the instruction and source level stepping.

The **-f** and **-b** options allow the global behavior set by the **%wait** debugger variable to be overridden. **-f** specifies foreground execution for the threads or processes. **-b** specifies background execution.

The **-q** option specifies quiet stepping: the debugger does not announce the step action nor the new source line.

A multiplexed thread that is not currently running on an LWP cannot be stepped.

stop [-p *proc_list***] [[-q] [-c** *count***]** *stop_expr* **[***cmd***]]**

The **stop** command specifies conditions in the address space of one or more controlled objects that should cause a list of threads or processes to stop. By default, the **stop** command applies to all threads or processes derived from the current program.

A *stop_expr* consists of one or more *stop events* , joined by the special debugger conjunction (**&&**) or disjunction (**||**) operators. These operators are left-associative, and **debug** does not guarantee the order in which their operands are evaluated. A *stop event* can take one of three forms:

location
***** *lvalue*
(*expr***)**

Each type of *stop event* has some action that will cause the event to be noticed by the debugger. When such an action occurs, the entire *stop_expr* is evaluated for ''truth''. If true, the event triggers in the normal way (**debug** informs the user of the event and executes any associated commands).

A *location* is an address in the process's text where **debug** can set a breakpoint. When the thread or process reaches the specified location **debug** notices the event. For location stop events that refer to function names, the expression is true as long as that function is active. For location stop events that apply to a particular address or line number, the expression is true only when the thread or process is at that address or line.

lvalue may be any expression in the current language that would be valid on the left-hand side of an assignment statement in that language. The debugger notices this event when the contents of the location change. The change itself makes this kind of stop event true.

expr can be any valid expression in the current language. The debugger notices the stop event when any of the identifiers involved in the expression changes value. The entire expression is then evaluated in the context of the current language.

*stop event*s are evaluated continuously while the thread or process is executing. The debugger is free to choose whatever means it has available to achieve this effect. This may include hardware support or may involve continuous single stepping of the object.

The optional *count* specifies the number of times the *stop_expr* must evaluate to true before the event triggers. After *count* times, the event triggers each time the *stop_expr* evaluates to true.

The **-q** option specifies that **debug** will not announce the occurrence of the event.

The **stop** command with no *stop_expr* arguments prints the list of user-specified stop expressions including the event identifier and current status (active or disabled).

symbols [-p *proc_list***] [-o** *object***] [-n** *filename***] [-dfgltuv] [***pattern***]**

The **symbols** command with no arguments displays ''local'' symbols; that is, names of variables which are defined within the current function (**%frame**) and are visible from the current location. This is also the behavior of the **-l** option.

The **-g** option displays only the names of global variables which are visible from the current location. This includes only those symbols defined within the current object (executable program or shared library). The **-o** option, in conjunction with **-g**, displays the names of global variables in the named *object*.

The **-f** option displays only the names of variables which are local to the current file (**%file**) and are visible from the current location (**%loc**). If the **-n** option is used, the symbols local to *filename* are displayed instead.

The **-d** option displays the debugger built-in variables. The **-u** option displays the debugger-maintained, user-defined variables.

If a *pattern* is given, the display is further restricted to symbols which match the *pattern*. **sh**(BU_CMD) syntax is used.

If the **-v** option is specified, the value of each symbol is displayed, along with its name. The **-t** option displays the type of the variable.

If more than one thread or process is specified by the *proc_list* argument, the **symbols** request is performed in the context of each object in turn.

syscall [-p *proc_list***] [[-eqx] [-c** *count***]** *call* **... [***cmd***]]**

The **syscall** command, by default, applies to all threads or processes derived from the current program. The **syscall** command with no *call* arguments prints the current list of user-specified system call actions, including the event identifier and current status (active or disabled), list of associated processes, system call name and the beginning of any associated command block.

Each *call* may be given as either a system call entry number, or as the name used in the C language interface to the call. The **-e** option specifies system call entry, and is the default. The **-x** option specifies system call exit. Both may be given on a single invocation of the **syscall** command. For each *call* listed, the debugger arranges for the specified objects to stop on entry to or exit from that *call*, or on both entry and exit. The resulting set of actions is then assigned a unique event identifier.

The optional *count* specifies the number of times the *call* must occur before the event triggers. After *count* times, the event triggers each time the *call* occurs.

The **-q** option specifies that **debug** will not announce the occurrence of the system call.

whatis [-p *proc_list*] *expr*

whatis prints the type of *expr* as evaluated in the current context. *expr* can be any valid expression in the current language.

If no *proc_list* is given, the type of *expr* is evaluated in the context of the current thread or process. Otherwise, it is evaluated for each object specified by the *proc_list*, in turn.

while (*expr***)** *cmd*

This is the traditional conditional loop statement, similar to that present in C, with the exception that semicolons are not necessary, except to separate multiple commands on a single line.

expr can be any valid expression in the current language (see ''Expressions''). The expression is evaluated, and if it evaluates to ''true'' in the semantics of the current language, the *cmd* is executed. The expression is then re-evaluated.

Unlike **if**, the **while** construct is often useful as a top-level command.

Summary of Built-In Variables

%db_lang The current language as determined from the object file (read-only, thread specific).

%file The current file (read-only, thread specific).

%follow Should **debug** follow child processes? Valid values are **none**, **procs**, **all** (global).

%frame The current active stack frame. Affects **%db_lang**, **%func**, **%file**, **%line**, **%list_file**, **%list_line**, **%loc** (thread specific).

%func The current function. Affects **%frame** (thread specific).

%global_path The list of directory pathnames used to search for source files for all processes. Searched after the program specific list **%path** (global).

%lang The current language. Setting **%lang** overrides the language as determined from the object file and maintained in **%db_lang** (global) .

%lastevent The id of the last event created (read-only, global).

%line The current line (read-only, thread specific).

%list_file The name of the file to be displayed by the **list** command. Reset when the current context changes (thread specific).

%list_line The number of the next source line to be displayed by the **list** command. Reset when the current context changes. Set to the last line displayed by any invocation of **list** (thread specific).

%loc	The current location (read-only, thread specific).
%num_bytes	The default number of bytes printed by the **dump** command (global).
%num_lines	The default number of lines printed by the **dis** and **list** commands (global).
%path	The list of directory pathnames used to search for source files for a given program. Searched before the global list **%global_path** (program specific).
%proc	The current process (global).
%program	The current program (global).
%prompt	The string used by **debug** to prompt the user for input; default is **debug>** (global).
%redir	Should process I/O be redirected to a pseudo-terminal for processes created by **debug**? Valid values are **0**, **1**, **no**, **yes** (global).
%result	The result status of any debugger command. **0** indicates success, non-zero failure (read-only, global).
%thisevent	The id of the event whose associated command list is currently being executed (read-only, global).
%thread	The current thread (global).
%thread_change	Control debugger behavior when a thread changes state (global).
%verbose	Level of verbosity for event notification (global). Valid values are **quiet**, **source**, **events**, **reason**, **all**.
%wait	Should threads and processes run in the foreground or background? Valid values are **0**, **1**, **background**, **foreground**, **no**, **yes** (global).
%*register*	The processor registers.

DIAGNOSTICS

If **debug** is invoked with invalid arguments, it prints a diagnostic message and exits with a non-zero exit status. If the command-line processing fails for any other reason, **debug** continues execution, allowing the user to enter requests interactively. **debug** prints diagnostics for any failure in processing user requests. The result status of each command is recorded in the debugger variable **%result**. A value of **0** indicates successful execution; a non-zero value indicates failure.

If **debug** cannot create or execute processes for any of the commands specified in *cmd_line*, it acts as if the entire *cmd_line* request had failed. In particular, any processes that had been created as part of the same *cmd_line* request are killed.

On the other hand, if **debug** cannot gain control of one or more of the *live_object*s specified in the second form of invocation, it continues to attempt to control the other objects specified.

If **debug** is invoked with the **-i x** option and cannot start the X Window based interface, it prints a diagnostic message and exits with a non-zero exit status.

FILES

$HOME/.debugrc defaults file
LIBDIR/**debug_alias** built-in alias definitions
LIBDIR/**debug.ol.ui** graphical interface
LIBDIR usually **/usr/ccs/lib**
/usr/lib/locale/C/MSGFILES/debug.str
default message file
/usr/lib/locale/*locale***/LC_MESSAGES/debug.str**
language-specific message file
/usr/lib/locale/C/MSGFILES/debug.ui.str
X interface default message file
/usr/lib/locale/*locale***/LC_MESSAGES/debug.ui.str**
X interface language-specific message file
/usr/X/lib/locale/C/help/debug/*
help screens
/usr/lib/locale/C/MSGFILES/dbg.help.thr
default help messages
/usr/lib/locale/*locale***/LC_MESSAGES/dbg.help.thr**
language-specific help messages

SEE ALSO

cc(SD_CMD), **dlclose**(BA_OS), **dlopen**(BA_OS), **dlsym**(BA_OS), **ed**(BU_CMD), **exec**(BA_OS), **fork**(BA_OS), **printf**(BA_LIB), **sh**(BU_CMD), **strtol**(BA_LIB), **thr_create**(MT_LIB)

LEVEL

Level 1.

NAME

delta – make a delta (change) to an SCCS file

SYNOPSIS

delta [-r*SID***] [-s] [-n] [-g***list***] [-m[***mrlist***]] [-y[***comment***]]**
[-p] *file* **. . .**

DESCRIPTION

delta is used to introduce changes into the named SCCS *file*; *file* must have been retrieved previously by using **get -e** (called the **g.***file* or generated file). The file name specified must be in the form **s.***file* or be the name of a directory. If a directory is named, **delta** behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of **-** is given, the standard input is read (see the NOTICES section); each line of the standard input is taken to be the name of an SCCS file to be processed.

delta may issue prompts on the standard output depending on certain keyletters specified and flags [see **admin**(SD_CMD)] that may be present in the SCCS file (see **-m** and **-y** keyletters below).

Keyletter arguments apply independently to each named file.

-r *SID* Uniquely identifies which delta is to be made to the SCCS file. The use of this keyletter is necessary only if two or more outstanding **get**s for editing (**get -e**) on the same SCCS file were done by the same person (login name). The SID value specified with the **-r** keyletter can be either the SID specified on the **get** command line or the SID to be made as reported by the **get** command [see **get**(SD_CMD)]. A diagnostic results if the specified SID is ambiguous, or, if necessary and omitted on the command line.

-s Suppresses the issue, on the standard output, of the created delta's SID, as well as the number of lines inserted, deleted and unchanged in the SCCS file.

-n Specifies retention of the edited **g.***file* (normally removed at completion of delta processing).

-g *list* Specify a *list* for the definition of *list*] of deltas that are to be ignored when the file is accessed at the change level (SID) created by this delta.

-m[*mrlist*] If the SCCS file has the **v** flag set [see **admin**(SD_CMD)] then a Modification Request (MR) number must be supplied as the reason for creating the new delta. If **-m** is not used and the standard input is a terminal, the prompt **MRs?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The **MRs?** prompt always precedes the **comments?** prompt (see **-y** keyletter). MRs in a list are separated by blanks and/or tab characters. An unescaped new-line character terminates the MR list. Note that if the **v** flag has a value [see **admin**(SD_CMD)] then a Modification Request (MR) number must be it is taken to be the name of a program (or shell procedure) that will validate the correctness of the MR numbers. If a non-zero exit status is returned from the MR number validation program, **delta** terminates. (It is assumed that the MR numbers were not all valid.)

-y[*comment*] Arbitrary text used to describe the reason for making the delta. A null string is considered a valid *comment*. If **-y** is not specified and the standard input is a terminal, the prompt **comments?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped new-line character terminates the comment text. Supplementary code set characters may be used in *comment*.

-p Causes **delta** to print (on the standard output) the SCCS file differences before and after the delta is applied in a **diff**(BU_CMD) **diff**(1) format.

Files

g.*file* Existed before the execution of **delta**; removed after completion of **delta**.

p.*file* Existed before the execution of **delta**; may exist after completion of **delta**.

q.*file* Created during the execution of **delta**; removed after completion of **delta**.

x.*file* Created during the execution of **delta**; renamed to SCCS file after completion of **delta**.

z.*file* Created during the execution of **delta**; removed during the execution of **delta**.

d.*file* Created during the execution of **delta**; removed after completion of **delta**.

bdiff Program to compute differences between the ''gotten'' file and the **g.***file*.

Errors

Use **help** for explanations.

SEE ALSO

admin (SD_CMD), **get** (SD_CMD), **prs** (SD_CMD), **rmdel** (SD_CMD),

LEVEL

Level 1.

NOTICES

A **get** of many SCCS files, followed by a **delta** of those files, should be avoided when the **get** generates a large amount of data. Instead, multiple **get**/**delta** sequences should be used.

If the standard input (**-**) is specified on the **delta** command line, the **-m** (if necessary) and **-y** keyletters must also be present. Omission of these keyletters causes an error.

Comments are limited to text strings of at most 1024 bytes. Line lengths greater than 1000 bytes cause undefined results.

NAME

dis – object code disassembler

SYNOPSIS

dis [-o] [-V] [-L] [-s] [-F *function***]**
[-l *string***]** *file* **. . .**

DESCRIPTION

The **dis** command produces an assembly language listing of *file*, which may be an object file or an archive of object files. The listing includes assembly statements and an octal or hexadecimal representation of the binary that produced those statements.

The following *options* are interpreted by the disassembler and may be specified in any order.

-F *function* Disassemble only the named function in each object file specified on the command line. The **-F** option may be specified multiple times on the command line.

-L Lookup source labels for subsequent printing. This option works only if the file was compiled with additional debugging information (for example, the **-g** option of **cc**).

-l *string* Disassemble the archive file specified by *string*. For example, you would issue the command **dis -l x –l z** to disassemble **libx.a** and **libz.a**, which are assumed to be in *LIBDIR*.

-o Print numbers in octal. The default is hexadecimal.

-s Perform symbolic disassembly where possible. Symbolic disassembly output will appear on the line following the instruction. Symbol names will be printed using C syntax.

-V Print, on standard error, the version number of the disassembler being executed.

Errors

The self-explanatory diagnostics indicate errors in the command line or problems encountered with the specified files.

SEE ALSO

as(SD_CMD), **cc**(SD_CMD), **ld**(SD_CMD)

LEVEL

Level 2: June 30, 1989. Optional

NAME

env, **printenv** – set environment for command execution

SYNOPSIS

env [-] [*name***=***value***] . . . [***command args***]**

DESCRIPTION

env obtains the current *environment*, modifies it according to its arguments, then executes the *command* with the modified environment. Arguments of the form *name***=***value* are merged into the inherited environment before the command is executed. The **-** flag causes the inherited environment to be ignored completely, so that the command is executed with exactly the environment specified by the arguments. If no command is specified, the resulting environment is printed, one name-value pair per line.

env recognizes supplementary code set characters in *value*, *command*, and *args* according to the locale specified in the **LC_CTYPE** environment variable [see **LANG** on **envvar** (BA_ENV).]

If the Application Compatibility Package is installed, then **printenv** replaces **env**.

SEE ALSO

envvar (BA_ENV), **exec** (BA_OS), **sh** (BU_CMD)

LEVEL

Level 1.

NAME

gcore – get core images of running processes

SYNOPSIS

`gcore` [`-o` *filename*] *process-id* ...

DESCRIPTION

`gcore` creates a core image of each specified process. The name of the core image file for the process whose process `ID` is *process-id* will be `core.`*process-id*.

`–o` *filename*
: Substitute *filename* in place of `core` as the first part of the name of the core image files.

FILES

`core.`*process–id* core image

SEE ALSO

kill(BU_CMD), ptrace(KE_OS)

LEVEL

Level 1.

NAME

get – get a version of an SCCS file

SYNOPSIS

get [-c*cutoff***] [-i***list***] [-r***SID***] [-x***list***] [-l[p]]**
[-b] [-e] [-g] [-k] [-m] [-n] [-p] [-s] [-t] *file* **. . .**

DESCRIPTION

get extracts the contents of each named SCCS *file* based on the values of the keyletter arguments. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS *file*s. The file name specified must be in the form **s.***file* or be the name of a directory. If a directory is named, **get** behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of **-** is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed.

The generated text is normally written into a file called the **g.***file* whose name is derived from the SCCS file name by simply removing the leading ‘‘**s.**’’ (see the Files section below).

Each of the keyletter arguments is explained below as though only one SCCS file is to be processed, but the effects of any keyletter argument apply independently to each named file.

-r*SID* The SCCS identification string (SID) of the version (delta) of an SCCS file to be retrieved. Table 1 below shows, for the most useful cases, what version of an SCCS file is retrieved (as well as the SID of the version to be eventually created by **delta**(1) if the **-e** keyletter is also used), as a function of the SID specified.

-c*cutoff* Cutoff date-time, in the form:

*YY***[***MM***[***DD***[***HH***[***MM***[***SS***]]]]]**

No changes (deltas) to the SCCS file that were created after the specified *cutoff* date-time are included in the generated ASCII text file. Units omitted from the date-time default to their maximum possible values; that is, **-c7502** is equivalent to **-c750228235959**. Any number of non-numeric characters may separate the two-digit pieces of the *cutoff* date-time. This feature allows one to specify a *cutoff* date in the form:

-c"77/2/2 9:22:25".

-i*list* A *list* of deltas to be included (forced to be applied) in the creation of the generated file. The *list* has the following syntax:

<list> **::=** *<range>* | *<list>* **,** *<range>*
<range> **::=** *SID* | *SID* **–** *SID*

SID, the SCCS Identification of a delta, may be in any form shown in the ‘‘SID Specified’’ column of Table 1.

-x*list* A *list* of deltas to be excluded in the creation of the generated file. See the **-i** keyletter for the *list* format.

-e Indicates that the **get** is for the purpose of editing or making a change (delta) to the SCCS file via a subsequent use of **delta**(1). The **-e** keyletter used in a **get** for a particular version (SID) of the SCCS file prevents further **get**s for editing on the same SID until **delta** is executed or the **j** (joint edit) flag is set in the SCCS file [see **admin**(SD_CMD)]. [see **admin**(1)]. Concurrent use of **get -e** for different SIDs is always allowed.

If the **g.***file* generated by **get** with an **-e** keyletter is accidentally ruined in the process of editing it, it may be regenerated by re-executing the **get** command with the **-k** keyletter in place of the **-e** keyletter.

SCCS file protection specified via the ceiling, floor, and authorized user list stored in the SCCS file [see **admin**(SD_CMD)] are enforced when the **-e** keyletter is used.

-b Used with the **-e** keyletter to indicate that the new delta should have an SID in a new branch as shown in Table 1. This keyletter is ignored if the **b** flag is not present in the file or if the retrieved **delta** is not a leaf **delta**. (A leaf **delta** is one that has no successors on the SCCS file tree.) A branch **delta** may always be created from a non-leaf **delta**. Partial SIDs are interpreted as shown in the ''SID Retrieved'' column of Table 1.

-k Suppresses replacement of identification keywords in the retrieved text by their value. The **-k** keyletter is implied by the **-e** keyletter.

-l[**p**] Causes a delta summary to be written into an **l.***file*. If **-lp** is used, then an **l.***file* is not created; the delta summary is written on the standard output instead. See the ''Identification Keywords'' section below for detailed information on the **l.***file*.

-p Causes the text retrieved from the SCCS file to be written on the standard output. No **g.***file* is created. All output that normally goes to the standard output goes to file descriptor 2 instead, unless the **-s** keyletter is used, in which case it disappears.

-s Suppresses all output normally written on the standard output. However, fatal error messages (which always go to file descriptor 2) remain unaffected.

-m Causes each text line retrieved from the SCCS file to be preceded by the SID of the delta that inserted the text line in the SCCS file. The format is: SID, followed by a horizontal tab, followed by the text line.

-n Causes each generated text line to be preceded with the **%M%** identification keyword value The format is: **%M%** value, followed by a horizontal tab, followed by the text line. When both the **-m** and **-n** keyletters are used, the format is: **%M%** value, followed by a horizontal tab, followed by the **-m** keyletter generated format.

-g Suppresses the actual retrieval of text from the SCCS file. It is primarily used to generate an **l.***file*, or to verify the existence of a particular SID.

-t Used to access the most recently created delta in a given release (for example, **-r1**), or release and level (for example, **-r1.2**).

For each file processed, **get** responds (on the standard output) with the SID being accessed and with the number of lines retrieved from the SCCS file.

If the **-e** keyletter is used, the SID of the delta to be made appears after the SID accessed and before the number of lines generated. If there is more than one named file or if a directory or standard input is named, each file name is printed (preceded by a new-line) before it is processed. If the **-i** keyletter is used, included deltas are listed following the notation ''**Included**;'' if the **-x** keyletter is used, excluded deltas are listed following the notation ''**Excluded**.''

TABLE 1. Determination of SCCS Identification String

SID* Specified	**-b** Keyletter Used†	Other Conditions	SID Retrieved	SID of Delta to be Created
none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1).1
R	no	R > mR	mR.mL	R.1***
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
R	–	R < mR and R does not exist	hR.mL**	hR.mL.(mB+1).1
R	–	Trunk succ.# in release > R and R exists	R.mL	R.mL.(mB+1).1
R.L	no	No trunk succ.	R.L	R.(L+1)
R.L	yes	No trunk succ.	R.L	R.L.(mB+1).1
R.L	–	Trunk succ. in release ≥ R	R.L	R.L.(mB+1).1
R.L.B	no	No branch succ.	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch succ.	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch succ.	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch succ.	R.L.B.S	R.L.(mB+1).1
R.L.B.S	–	Branch succ.	R.L.B.S	R.L.(mB+1).1

* ''R,'' ''L,'' ''B,'' and ''S'' are the ''release,'' ''level,'' ''branch,'' and ''sequence'' components of the SID, respectively; ''m'' means ''maximum.'' Thus, for example, ''R.mL'' means ''the maximum level number within release R;'' ''R.L.(mB+1).1'' means ''the first sequence number on the new branch (for example, maximum branch number plus one) of level L within release R.'' Note that if the SID specified is of the form ''R.L'', ''R.L.B'', or ''R.L.B.S'', each of the specified components must exist.

** ''hR'' is the highest existing release that is lower than the specified, nonexistent, release R.

*** This is used to force creation of the first delta in a new release.

\# Successor.

† The **-b** keyletter is effective only if the **b** flag [see **admin**(SD_CMD)] is present in the file. An entry of **-** means ''irrelevant.''

‡ This case applies if the **d** (default SID) flag is not present in the file. If the **d** flag is present in the file, then the SID obtained from the **d** flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

Identification Keywords

Identifying information is inserted into the text retrieved from the SCCS file by replacing identification keywords with their value wherever they occur. The following keywords may be used in the text stored in an SCCS file:

Keyword	Value
%M%	Module name: either the value of the **m** flag in the file [see **admin**(SD_CMD)], or if absent, the name of the SCCS file with the leading **s.** removed.
%I%	SCCS identification (SID) (**%R%.%L%.%B%.%S%**) of the retrieved text.
%R%	Release.
%L%	Level.
%B%	Branch.
%S%	Sequence.
%D%	Current date (*YY/MM/DD*).
%H%	Current date (*MM/DD/YY*).
%T%	Current time (*HH:MM:SS*).
%E%	Date newest applied delta was created (*YY/MM/DD*).
%G%	Date newest applied delta was created (*MM/DD/YY*).
%U%	Time newest applied delta was created (*HH:MM:SS*).
%Y%	Module type: value of the **t** flag in the SCCS file [see **admin**(SD_CMD)].
%F%	SCCS file name.
%P%	Fully qualified SCCS file name.
%Q%	The value of the **q** flag in the file [see **admin**(SD_CMD)].
%C%	Current line number. This keyword is intended for identifying messages output by the program such as ''this should not have happened'' type errors. It is not intended to be used on every line to provide sequence numbers.
%Z%	The four-character string **@(#)** recognizable by the **what** command.
%W%	A shorthand notation for constructing **what** strings for UNIX System program files. **%W% = %Z%%M%**<tab>**%I%**
%A%	Another shorthand notation for constructing **what** strings for non-UNIX System program files: **%A% = %Z%%Y% %M% %I%%Z%**

Several auxiliary files may be created by **get**. These files are known generically as the **g.***file*, **l.***file*, **p.***file*, and **z.***file*. The letter before the dot is called the tag. An auxiliary file name is formed from the SCCS file name: the last component of all SCCS file names must be of the form **s.***module-name*, the auxiliary files are named by replacing the leading **s** with the tag. The **g.***file* is an exception to this scheme: the **g.***file* is named by removing the **s.** prefix. For example, **s.xyz.c**, the auxiliary file names would be **xyz.c**, **l.xyz.c**, **p.xyz.c**, and **z.xyz.c**, respectively.

A **g.***file*, containing the generated text, is created in the current directory. It is owned by the real user, and only the real user need have write permission in the current directory. The permissions of the **g.***file* depend on the permissions of the SCCS file, the options used when **get** was executed, and the **x** flag in the SCCS file [see **admin**(SD_CMD)]. Users who have read permission to the SCCS file have read permission to the **g.***file*, and if the **x** flag has been set in the SCCS file, also have execute permission to the **g.***file*. Invoking **get** with the **-e** option enables write permission on the **g.***file* for the invoker.

The **l.***file* contains a table showing which deltas were applied in generating the retrieved text. The **l.***file* is created in the current directory if the **-l** keyletter is used; its mode is 444 and it is owned by the real user. Only the real user need have write permission in the current directory.

Lines in the **l.***file* have the following format:

a. A blank character if the delta was applied; * otherwise.
b. A blank character if the delta was applied or was not applied and ignored; * if the delta was not applied and was not ignored.
c. A code indicating a ''special'' reason why the delta was or was not applied: ''**I**'' (included), ''**X**'' (excluded), or ''**C**'' (cut off by a **-c** keyletter).
d. Blank.
e. SCCS identification (SID).
f. Tab character.
g. Date and time (in the form *YY/MM/DD HH:MM:SS*) of creation.
h. Blank.
i. Login name of person who created **delta**.

   The comments and MR data follow on subsequent lines, indented one horizontal tab character. A blank line terminates each entry.

The **p.***file* is used to pass information resulting from a **get** with an **-e** keyletter along to **delta**. Its contents are also used to prevent a subsequent execution of **get** with an **-e** keyletter for the same SID until **delta** is executed or the joint edit flag, **j**, [see **admin**(SD_CMD)] is set in the SCCS file. The **p.***file* is created in the directory containing the SCCS file and the effective user must have write permission in that directory. Its mode is 644 and it is owned by the effective user. The format of the **p.***file* is: the gotten SID, followed by a blank, followed by the SID that the new delta will have when it is made, followed by a blank, followed by the login name of the real user, followed by a blank, followed by the date-time the **get** was executed, followed by a blank and the **-i** keyletter argument if it was present, followed by a blank and the **-x** keyletter argument if it was present, followed by a new-line. There can be an arbitrary number of lines in the **p.***file* at any time; no two lines can have the same new delta SID.

The **z.***file* serves as a lock-out mechanism against simultaneous updates. Its contents are the binary (2 bytes) process ID of the command (that is, **get**) that created it. The **z.***file* is created in the directory containing the SCCS file for the duration of **get**. The same protection restrictions as those for the **p.***file* apply for the **z.***file*. The **z.***file* is created with mode 444.

Files

g._file_ created by the execution of **get**.
l._file_ created by **-l** option; contains delta summary
p._file_ [see **delta**(SD_CMD)]
q._file_ [see **delta**(SD_CMD)]
z._file_ [see **delta**(SD_CMD)]
bdiff Program to compute differences between the ''gotten'' file and the **g.**_file_.
/usr/lib/locale/_locale_**/LC_MESSAGES/uxue**
language-specific message file [see **LANG** on **envvar**(BA_ENV)].

Errors

Use **help** for explanations.

SEE ALSO

admin (SD_CMD), **delta** (SD_CMD), **prs** (SD_CMD), **what** (SD_CMD)

LEVEL

Level 1.

NOTICES

If the effective user has write permission (either explicitly or implicitly) in the directory containing the SCCS files, but the real user does not, then only one file may be named when the **-e** keyletter is used.

NAME

ld – link editor for object files

SYNOPSIS

`ld` [*options*] *file* ...

DESCRIPTION

The `ld` command combines several object files into one, performs relocation and resolves external symbols. In the simplest case, the names of several object programs are given, and `ld` combines them, producing an object module that can either be executed or, if the `-r` option is specified, used as input for a subsequent `ld` run. The output of `ld` is left in `a.out` if no errors occurred during the load. This file is by default executable. If any input file is not an object file, `ld` assumes it is a library.

If any argument is an archive library, it is searched at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. The archive library symbol table is searched to resolve external references which can be satisfied by library members. The ordering of archive library members is unimportant, unless there exist multiple library members defining the same external symbol.

The following options are recognized by `ld`:

–a In static mode only, produce an executable object file; give errors for undefined references. This is the default behavior for static mode. **–a** may not be used with the **–r** option.

–d *yn* **ld** uses static linking only when *yn* is **n**; otherwise if supported, when *yn* is **y**, **ld** uses dynamic linking.

`-e` *epsym*
Set the default entry point address for the output file to be that of the symbol *epsym*.

–h *name*
In dynamic mode only and dynamic linking is supported, when building a shared object, record *name* in an implementation defined manner in the object. *name* will be recorded in executables that are linked with this object rather than the object's UNIX System file name. Accordingly, *name* will be used by the dynamic linker as the name of the shared object to search for at run time.

`-l`*x* Search the library **lib***name***.a** or if shared objects are supported libname.so. Its placement on the command line is significant as a library is searched at a point in time relative to the placement of other libraries and object files on the command line.

`-o` *outfile*
Produce an output object file by the name *outfile*. The name of the default object file is `a.out`.

`-r` Retain relocation entries in the output object file. Relocation entries must be saved if the output file is to become an input file in a subsequent `ld` run. The link editor will not complain about unresolved references, and the output file will not be made executable.

`-s` Strip all symbolic debugging information from the output object file.

`-u` *symname*
Enter *symname* as an undefined symbol in the symbol table. This is useful for loading entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

-z defs
Force a fatal error if any undefined symbols remain at the end of the link. This is the default when building an executable. It is also useful if dynamic linking is supported when building a shared object to assure that the object is self-contained, that is, that all its symbolic references are resolved internally.

-z nodefs
Allow undefined symbols. This is the default, if dynamic linking is supported, when building a shared object. It may be used when building an executable in dynamic mode and linking with a shared object that has unresolved references in routines not used by that executable. This option should be used with caution.

-z text
If in dynamic mode and dynamic linking is supported, only, force a fatal error if any relocations against non-writable, allocatable sections remain.

-B *arg* *arg* can be any one of the following: *dynsat*, *symb*

dynstat When dynamic linking is supported, *dynstat* can be either **dynamic** or **static**. These options govern library inclusion. **dynamic** is valid in dynamic mode only. If the system supports dynamic linking, **-B dynamic** causes the link editor to look for files named **lib***x***.so** and then for files named **lib***x***.a** when given the **-l***x* option. **-B static** causes the link editor to look only for files named **lib***x***.a**. These options may be specified any number of times on the command line as toggles: if **-Bstatic** is given, no shared objects will be accepted until **-Bdynamic** is seen. See also the **-l** option.

symb When dynamic linking is supported *symb* may take the form **symbolic**[**=***symbol***,** . . .]
When building a shared object, if a definition for *symbol* exists, bind all references to *symbol* to that definition. If no list of symbols is provided, bind all references to symbols to definitions that are available; **ld** will issue warnings for undefined symbols unless **-z defs** overrides. Normally, references to global symbols within shared objects are not bound until run time, even if definitions are available, so that definitions of the same symbol in an executable or other shared objects can override the object's own definition.

-G If dynamic linking is supported and in dynamic mode only, produce a shared object. Undefined symbols are allowed.

`-L` *dir*
: Change the algorithm of searching for the library *x* to look in *dir* before looking in the default library directories. This option is effective only if it precedes the `-l` option on the command line.

`-V` Output a message giving information about the version of `ld` being used.

-YP, *dirlist*
: Change the default directories used for finding libraries. *dirlist* is a colon-separated path list.

FILES

`a.out`
: output file

USAGE

General.

When the link editor is called through `cc`, a startup routine is linked with the user's program. This routine calls `exit()` after execution of the main program. If the user calls the link editor directly, then the user must ensure that the program always calls `exit()` rather than falling through the end of the entry routine.

The symbols `_etext`, `_edata`, and `_end` are reserved and are defined by the link editor. It is erroneous for a user program to redefine them.

The meaning of the terms shared library and dynamic linking are described in the System V ABI.

SEE ALSO

ar(BU_CMD), cc(SD_CMD), strip(SD_CMD).

LEVEL

Level 1.
The following options are dependent upon dynamic linking being supported and therefore are marked as Optional:
-d, -h, -z, -B dynstat, -B symb, -G

NAME

lex – generate programs for simple lexical analysis of text

SYNOPSIS

`lex` [`-ctvn`] [*file*] ...

DESCRIPTION

The command `lex` generates programs to be used in lexical processing of character input and may be used as an interface to `yacc`.

The input *file*(s), which contain `lex` source code, contain a table of regular expressions each with a corresponding action in the form of a **C** program fragment. Multiple input *file*s are treated as a single file. When `lex` processes *file*(s), this source is translated into a **C** program. Normally `lex` writes the program it generates to the file `lex.yy.c`. If the `-t` option is used, the resulting program is written instead to the standard output. When the program generated by `lex` is compiled and executed, it will read character input from the standard input and partition it into strings that match the given expressions. When an expression is matched, the input string that was matched is left in an external character array `yytext` and the expression's corresponding program fragment, or action, is executed. `lex` also provides a count `yyleng` of the number of characters matched. During pattern matching the set of patterns will be searched for a match in the order in which they appeared in the `lex` source and the single longest possible match will be chosen. Among rules that match the same number of characters, the rule given first will be matched.

The program generated by `lex`, *e.g.*, `lex.yy.c,` should be compiled and loaded with the `lex` library (using the `-ll` option with `cc`).

The option `–c` indicates **C** language actions and is the default, `–t` causes the program generated to be written instead to standard output, `–v` provides a one-line summary of statistics of the finite state machine generated, `–n` will not print out the `–v` summary (as explained under **Definitions**, below).

The general format of `lex` source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions and the user subroutines may be omitted. The first %% is required to mark the beginning of the rules (regular expressions and actions); the second %% is required only if user subroutines follow.

Any line in the source beginning with a blank is assumed to contain only **C** text and is copied to `lex.yy.c`; if it precedes %% it is copied into the external definition area of the `lex.yy.c` file. Anything included between lines containing only `%{ and %}` is copied unchanged to `lex.yy.c` and the delimiter lines are discarded. Anything after the third %% delimiter is copied to `lex.yy.c.`

Definitions

Definitions must appear before the first %% delimiter. Any line in this section not contained between `%{ and %}` lines and beginning in column 1 is assumed to define a `lex` substitution string. The format of these lines is *name substitute*

The *name* must begin with a letter and be followed by at least one blank or tab. The *substitute* will replace the string *name* when it is used in a rule.

Certain table sizes for the resulting finite state machine can be set in the definitions section:

`%p` *n* number of positions is *n*

`%n` *n* number of states is *n*

`%e` *n* number of parse tree nodes is *n*

`%a` *n* number of transitions is *n*

`%k` *n* number of packed character classes is *n*

`%o` *n* size of the output array is *n*

The use of one or more of the above automatically implies the `-v` option, unless the `-n` option is used.

Rules

The rules in `lex` source files are a table in which the left column contains regular expressions and the right column contains actions and program fragments to be executed when the expressions are recognized.

regular-expression `<whitespace>` *action*
regular-expression `<whitespace>` *action*
. . .

Because the *regular-expression* portion of a rule is terminated by the first blank or tab, any blank or tab used within a regular expression must be quoted (its special meaning escaped). That is, it must appear within double quotes, square brackets or must be preceded by a backslash character.

The program fragment that is the action associated with a particular *regular-expression* may extend across several lines if it is enclosed in curly braces:

```
regular-expression whitespace { program statement
                                program statement }
```

Regular Expressions

The `lex` command supports the sets of regular expressions recognized by `ed` and `awk`, and some additional expressions. Some characters have special meanings when used in a *regular-expression* and are called regular expression operators. Below is a table of expressions supported by `lex`.

Regular Expression	Pattern Matched
`c`	the character *c* where *c* is not a special character.
`\c`	the character *c* where *c* is any character.
`"c"`	the character *c* where *c* is any character except `\`.
`^`	the beginning of the line being compared.
`$`	the end of the line being compared.
`.`	any character in the input but newline
`[`*s*`]`	any character in the set *s* where *s* is a sequence of characters and/or a range of characters, *c-c*.
`[^`*s*`]`	any character not in the set *s*, where *s* is defined as above.
r`*`	zero or more successive occurrences of the regular expression *r*.
r`+`	one or more successive occurrences of the regular expression *r*.
r`?`	zero or one occurrence of the regular expression *r*.
`(`*r*`)`	the regular expression *r*. (Grouping)
rx	the occurrence of regular expression *r* followed by the occurrence of regular expression *x*. (Concatenation)
r|*x*	the occurrence of regular expression *r* or the occurrence of regular expression *x*.
<*s*>*r*	the occurrence of regular expression *r* only when the program is in start condition (state) *s*.
r`/`*x*	the occurrence of regular expression *r* only if it is followed by the occurrence of regular expression *x*. (Note: This is *r* in the context of *x* and only *r* is matched.)
`{`*S*`}`	the substitution of *S* from the *Definitions* section.
r`{`*m*`,`*n*`}`	*m* through *n* successive occurrences of the regular expression *r*.

The notation *r*`{`*m*`,`*n*`}` in a rule indicates between *m* and *n* instances of regular expression *r*. It has higher precedence than `|`, but lower than `*`, `?`, `+`, and concatenation.

The character `^` at the beginning of an expression permits a successful match only immediately after a newline, and the character `$` at the end of an expression requires a trailing newline.

The character `/` in an expression indicates trailing context; only the part of the expression up to the slash is returned in `yytext`, but the remainder of the expression must follow in the input stream. An operator character may be used as an ordinary symbol if it is within double quotes, "*c*," preceded by `\`, `\`*c*, or is within square brackets, `[`*c*`]`. Two operators have special meaning when used within square brackets. A `-` denotes a range, `[`*c*`-`*c*`]`, unless it is just after the open bracket or before the closing bracket, `[-`*c*`]` or `[`*c*`-]` in which case it has no special meaning. When used within brackets, `^` has the meaning "complement of" if it immediately follows the open bracket, `[^`*c*`]`, elsewhere between brackets, `[`*c*`^]`, it stands for the ordinary character `^`. The special meaning of the `\` operator can be escaped only by preceding it with another `\`.

Actions

The default action when a string in the input to a `lex.yy.c` program is not matched by any expression is to copy the string to the output. Because the default behavior of a program generated by `lex` is to read the input and copy it to the output, a minimal `lex` source program that has just %% will generate a **C** program that simply copies the input to the output unchanged. A null **C** statement, the statement ';', may be specified as an action in a rule. Any string in the `lex.yy.c` input that matches the pattern portion of such a rule will be effectively ignored or skipped.

Three special actions are available, `|`, `REJECT`, and `ECHO`. The action `|` means that the action for the next rule is the action for this rule. `ECHO` prints the contents of `yytext` on the output. Normally only a single expression is matched by a given string in the input. `REJECT` means "continue to the next expression that matches the current input" and causes whatever rule was second choice after the current rule to be executed for the same input. Thus, it allows multiple rules to be matched and executed for one input string or overlapping input strings. For example, given the expressions *xyz* and *yz* and the input *xyz*, normally only one pattern, *xyz* would match and the next attempted match would start after *z*. If the last action in the *xyz* rule is `REJECT`, both this rule and the *yz* rule would be executed.

The `lex` command provides several routines that can be used in the lex source program: `yymore()`, `yyless(`*n*`)`, `input()`, `output(`*c*`)`, and `unput(`*c*`)`.

The function `yymore()` may be called to indicate that the next input string recognized is to be concatenated onto the end of the current string in `yytext` rather than overwriting it in `yytext`.

`yyless(`*n*`)` returns to the input some of the characters matched by the currently successful expression. The argument *n* indicates the number of initial characters in `yytext` to be retained; the remaining trailing characters in `yytext` are returned to the input.

`input()` returns the next character from the input. `input()` returns a zero on end of file.

`unput(`*c*`)` pushes the character *c* back onto the input stream to be read later by `input()`.

`output(`*c*`)` writes the character *c* on the output.

To perform custom processing when the end of input is reached, a user may supply their own `yywrap()` function. `yywrap()` is called whenever `lex.yy.c` reaches an end-of-file. If `yywrap()` returns a one, `lex.yy.c` continues with the normal wrap-up on end of input. The default `yywrap()` always returns a one. If the user wants `lex.yy.c` to continue processing with another source of input, then a `yywrap()` must be supplied that arranges for the new input and returns a zero. These routines may be redefined by the user.

The external names generated by `lex` all begin with the prefix `yy` or `YY`.

The program generated by `lex` is named `yylex();` if the user does not supply a main routine, the default `main()` routine calls `yylex()`. If the user supplies a `main()` routine, it should call `yylex()`.

FILES

lex.yy.c.

USAGE

General.

EXAMPLE

```
%{
void skipcommnts(void);
%}
D      [0–9]
%%
if     printf("IF statement\n");
[a–z]+ printf("tag, value %s\n",yytext);
0{D}+  printf("octal number %s\n",yytext);
{D}+   printf("decimal number %s\n",yytext);
"++"   printf("unary op\n");
"+"    printf("binary op\n");
"/*"   skipcommnts();
%%
void skipcommnts(void)
{
       for(;;) {
       while (input() != '*');
           ;
       if (input() != '/')
           unput(yytext[yylen - 1]);
       else
           return;
       }
}
```

SEE ALSO

cc(SD_CMD), yacc(SD_CMD).

LEVEL

Level 1.

NAME

lint – a **C** program checker

SYNOPSIS

`lint` [*options*] *file* ...

DESCRIPTION

The command `lint` attempts to detect features of the **C** program files that are likely to be bugs, non-portable, or wasteful. It also checks type usage more strictly than do the compilers. Among the things that are currently detected are unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions that return values in some places and not in others, functions called with varying numbers or types of arguments, and functions whose values are not used or whose values are used but none returned. The *options* are described below.

Arguments whose names end with `.c` are taken to be **C** source files. Arguments whose names end with `.ln` are taken to be the result of an earlier invocation of `lint` with either the `–c` or the `–o` option used. The `.ln` files are analogous to `.o` (object) files that are produced by the `cc` command when given a `.c` file as input.

Files with other suffixes are warned about and ignored.

The command `lint` will take all the `.c`, `.ln,` files, and `llib-l`*x* (specified by `-l`*x*), and process them in their command line order. By default, `lint` appends the standard **C** lint library to the end of the list of files. However, if the `–p` option is used, the portable **C** lint library (`llib-port.ln`) is appended instead. When the `–c` option is not used, the second pass of `lint` checks this list of files for mutual compatibility. When the `–c` option is used, the `.ln` files and the lint libraries are ignored.

Any number of `lint` options may be used, in any order, intermixed with file-name arguments. The following options are used to suppress certain kinds of complaints:

`–a` Suppress complaints about assignments of long values to variables that are not long.

`–b` Suppress complaints about `break` statements that cannot be reached. (Programs produced by `lex` or `yacc` will often result in many such complaints).

`–h` Do not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.

`–u` Suppress complaints about functions and external variables used and not defined, or defined and not used. (This option is suitable for running `lint` on a subset of files of a larger program).

`–v` Suppress complaints about unused arguments in functions.

`–x` Do not report variables referred to by external declarations but never used.

The following options alter `lint`'s behavior:

`-l`*x* Include additional lint library *x* (*e.g.*, `-lm` for the math library).

`-n` Do not check compatibility against either the standard or the portable lint library.

`-p` Attempt to check portability.

`-c` Cause `lint` to produce a `.ln` file for every `.c` file on the command line. These `.ln` files are the product of `lint`'s first pass only, and are not checked for inter-function compatibility.

`-o` *lib* Cause `lint` to create a lint library with the name *lib*. The `-c` option nullifies any use of the `-o` option. The lint library produced is the input that is given to `lint`'s second pass. The `-o` option simply causes this file to be saved in the named lint library. To produce the lint library without extraneous messages, use of the `-x` option is suggested. The `-v` option is useful if the source file(s) for the lint library are just external interfaces. These option settings are also available through the use of lint comments (see below).

The `-D`, `-U`, and `-I` options of `cpp` [see cpp(SD_CMD)] are recognized as separate arguments.

The `-g` and `-O` options of `cc` are also recognized as separate arguments. These options are ignored, but, by recognizing these options, `lint`'s behavior is closer to that of the `cc` command. Other options are warned about and ignored. The preprocessor symbol `lint` is defined to allow certain questionable code to be altered or removed for `lint`. Therefore, the symbol `lint` should be thought of as a reserved word for all code that is planned to be checked by `lint`.

Certain conventional comments in the **C** source will change the behavior of `lint`:

`/*NOTREACHED*/`
: at appropriate points stops comments about unreachable code. This comment is typically placed just after calls to functions like `exit`.

`/*VARARGS`*n*`*/`
: suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be zero.

`/*ARGSUSED*/`
: turns on the `-v` option for the next function.

`/*LINTLIBRARY*/`
: at the beginning of a file shuts off complaints about unused functions and function arguments in this file. This is equivalent to using the `-v` and `-x` options.

The command `lint` produces its first output on a per-source-file basis. Complaints regarding included files are collected and printed after all source files have been processed. Finally, if the `-c` option is not used, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source file name will be printed followed by a question mark.

The behavior of the `-c` and the `-o` options allows for incremental use of `lint` on a set of **C** source files. Generally, `lint` is invoked once for each source file with the `-c` option. Each of these invocations produces a `.ln` file which corresponds to the `.c` file, and prints all messages that are about just that source file. After all the source files have been separately run through `lint,` it is invoked once more (without the `-c` option), listing all the `.ln` files with the needed `-l`*x* options. This will print all the inter-file inconsistencies. This scheme works well with `make`; it allows `make` to be used to `lint` only the source files that have been modified since the last time the set of source files were checked by `lint`.

USAGE

General.

SEE ALSO

cc(SD_CMD), cpp(SD_CMD), make(SD_CMD).

LEVEL

Level 1.

NAME

lorder – find ordering relation for an object library

SYNOPSIS

`lorder` *file* ...

DESCRIPTION

The input is one or more object or library archive *files* [see ar(BU_CMD)]. The standard output is a list of pairs of object file names, meaning that the first file of the pair refers to external identifiers defined in the second. The output may be processed by `tsort` to find an ordering of a library suitable for one-pass access by the link editor `ld`. Note that `ld` is capable of multiple passes over an archive in the portable archive format and does not require that `lorder` be used when building an archive. The usage of the `lorder` command may, however, allow for a slightly more efficient access of the archive during the link edit process.

EXAMPLE

The following example builds a new library from existing `.o` files.

```
ar –cr library `lorder *.o | tsort`
```

SEE ALSO

ar(BU_CMD), ld(SD_CMD), tsort(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

m4 – macro processor

SYNOPSIS

`m4` [*options*] [*file* ...]

DESCRIPTION

The command `m4` is a macro processor intended as a front end for **C** and other languages. Each of the argument files is processed in order; if there are no files, or if a file name is `-`, the standard input is read. The processed text is written on the standard output.

The options and their effects are as follows:

`-s` Enable line sync output for the **C** preprocessor (*i.e.*, `#line` directives).

This option must appear before any file names and before the following options.

`-D`*name* `[=`*val* `]`
Defines *name* to *val* or to null if *val* is absent.

`-U`*name*
undefines *name.*

Macro calls have the form:

name`(`*arg1*`,`*arg2*`, ...,` *argn*`)`

The `(` must immediately follow the *name* of the macro. If the *name* of a defined macro is not followed by a `(`, it is deemed to be a call of that macro with no arguments. Potential macro names consist of alphabetic letters, digits, and underscore, _, where the first character is not a digit.

Leading unquoted blanks, tabs, and newlines are ignored while collecting arguments. Left and right single quotes are used to quote strings. The value of a quoted string is the string stripped of the quotes.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. If fewer arguments are supplied than are in the macro definition, the trailing arguments are taken to be null. Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses which happen to turn up within the value of a nested call are as effective as those in the original input text. After argument collection, the value of the macro is pushed back onto the input stream and rescanned.

The command `m4` makes available the following built-in macros. They may be redefined, but once this is done the original meaning is lost. Their values are null unless otherwise stated.

`define`
The second argument is installed as the value of the macro whose name is the first argument. Each occurrence of `$`*n* in the replacement text, where *n* is a digit, is replaced by the *n*-th argument. Argument 0 is the name of the macro; missing arguments are replaced by the null string; `$#` is replaced by the number of arguments; `$*` is replaced by a list of all the arguments separated by commas; `$@` is like `$*`, but each argument is quoted (with the

current quotes).

`undefine`
removes the definition of the macro named in its argument.

`defn` returns the quoted definition of its argument(s). It is useful for renaming macros, especially built-ins.

`pushdef`
is like `define`, but saves any previous definition.

`popdef`
removes the current definition of its argument(s), exposing the previous one, if any.

`ifdef`
If the first argument is defined, the value is the second argument, otherwise the third. If there is no third argument, the value is null.

`shift`
returns all but its first argument. The other arguments are quoted and pushed back with commas in between. The quoting nullifies the effect of the extra scan that will subsequently be performed.

`changequote`
changes quote symbols to the first and second arguments. The symbols may be up to five characters long. the command `changequote` without arguments restores the original values (*i.e.*, \` ´).

`changecom`
changes left and right comment markers from the default # and newline. With no arguments, the comment mechanism is effectively disabled. With one argument, the left marker becomes the argument and the right marker becomes newline. With two arguments, both markers are affected. Comment markers may be up to five characters long.

`divert`
The command `m4` maintains 10 output streams, numbered 0-9. The final output is the concatenation of the streams in numerical order; initially stream 0 is the current stream. The `divert` macro changes the current output stream to its (digit-string) argument. Output diverted to a stream other than 0 through 9 is discarded.

`undivert`
causes immediate output of text from diversions named as arguments, or all diversions if no argument. Text may be undiverted into another diversion. Undiverting discards the diverted text.

`divnum`
returns the value of the current output stream.

`dnl` reads and discards characters up to and including the next newline.

`ifelse`
has three or more arguments. If the first argument is the same string as the second, then the value is the third argument. If not, and if there are more than four arguments, the process is repeated with arguments 4, 5, 6 and 7.

Otherwise, the value is either the fourth string or, if it is not present, null.

`incr` returns the value of its argument incremented by 1. The value of the argument is calculated by interpreting an initial digit-string as a decimal number.

`decr` returns the value of its argument decremented by 1.

`eval` evaluates its argument as an arithmetic expression, using 32-bit arithmetic. Operators include `+`, `–`, `*`, `/`, `%`, `**`, (exponentiation), bitwise `&`, `|`, `^`, and `~`; relationals; parentheses. Octal and hex numbers may be specified as in **C**. The second argument specifies the radix for the result; the default is 10. The third argument may be used to specify the minimum number of digits in the result.

`len` returns the number of characters in its argument.

`index`
returns the position in its first argument where the second argument begins (zero origin), or –1 if the second argument does not occur.

`substr`
returns a substring of its first argument. The second argument is a zero origin number selecting the first character; the third argument indicates the length of the substring. A missing third argument is taken to be large enough to extend to the end of the first string.

`translit`
transliterates the characters in its first argument from the set given by the second argument to the set given by the third. No abbreviations are permitted.

`include`
returns the contents of the file named in the argument.

`sinclude`
is identical to `include`, except that it says nothing if the file is inaccessible.

`syscmd`
executes the system command given in the first argument. No value is returned.

`sysval`
is the return code from the last call to `syscmd`.

`maketemp`
fills in a string of `XXXXX` in its argument with the current process ID.

`m4exit`
causes immediate exit from `m4`. Argument 1, if given, is the exit code; the default is 0.

`m4wrap`
Argument 1 will be pushed back at final EOF; example: `` m4wrap(`cleanup()´) ``

`errprint`
: prints its argument on the diagnostic output file.

`dumpdef`
: prints current names and definitions, for the named items, or for all if no arguments are given.

`traceon`
: with no arguments, turns on tracing for all macros (including built-ins). Otherwise, turns on tracing for named macros.

`traceoff`
: turns off trace globally and for any macros specified. Macros specifically traced by `traceon` can be untraced only by specific calls to `traceoff`.

USAGE

General.

SEE ALSO

cc(SD_CMD), cpp(SD_CMD).

LEVEL

Level 1.

NAME

make – maintain, update, and regenerate groups of programs

SYNOPSIS

`make` [`-f` *makefile*] [`-p`] [`-i`] [`-k`] [`-s`] [`-r`] [`-n`] [`-e`] [`-t`] [`-q`] [*name* ...]

DESCRIPTION

The options are interpreted as follows:

`–f` *makefile*
: Description file name. The argument *makefile* is assumed to be the name of a description file. A file name of – denotes the standard input.

`–p` Print out the complete set of macro definitions and target descriptions.

`–i` Ignore error codes returned by invoked commands. This mode is entered if the fake target name `.IGNORE` appears in the description file.

`–k` Abandon work on the current entry if it fails, but continue on other branches that do not depend on that entry.

`–s` Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name `.SILENT` appears in the description file.

`–r` Do not use the built-in rules.

`–n` No execute mode. Print commands, but do not execute them. Even lines beginning with an `@` are printed.

`–e` Environmental variables override assignments within makefiles.

`–t` Touch the target files (causing them to be up-to-date) rather than issue the usual commands.

`–q` Question. The `make` command returns a zero or non-zero status code depending on whether the target file is or is not up-to-date.

The following target names may be defined in the *makefile*, and are interpreted as follows:

`.DEFAULT`
: If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name `.DEFAULT` are used if it exists.

`.PRECIOUS`
: Dependents of this target will not be removed when quit or interrupt are hit.

`.SILENT`
: Same effect as the `–s` option.

`.IGNORE`
: Same effect as the `–i` option.

The command `make` executes commands in *makefile* to update one or more target names. The argument *name* is typically a program. If no `–f` option is present, `makefile`, `Makefile`, and the SCCS files `s.makefile` and `s.Makefile` are tried in order. If *makefile* is `–`, the standard input is used. More than one `–f`*makefile* argument pair may appear.

The command make updates a target only if its dependents are newer than the target. All prerequisite files of a target are added recursively to the list of targets. Missing files are deemed to be out-of-date.

The argument *makefile* contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated, non-null list of targets, then a colon, then a (possibly null) list of prerequisite files or dependencies. Text following a semicolon and all following lines that begin with a tab are commands to be executed to update the target. The first line that does not begin with a tab or # begins a new dependency or a macro definition. Commands may be continued across lines with the <backslash><newline> sequence. Everything printed by make (except the initial tab) is passed directly to the command interpreter as is.

The symbols # and newline surround comments.

The following *makefile* says that pgm depends on two files a.o and b.o, and that they in turn depend on their corresponding source files (a.c and b.c) and a common file incl.h:

```
pgm: a.o b.o ; cc a.o b.o –o pgm
a.o: incl.h a.c ; cc –c a.c
b.o: incl.h b.c ; cc –c b.c
```

Command lines are executed one at a time. The first one or two characters in a command can be the following: -, @, -@, or @-. If @ is present, printing of the command is suppressed. If - is present, make ignores an error. A line is printed when it is executed unless the –s option is present, or the entry .SILENT: is in *makefile*, or unless the initial character sequence contains a @. The –n option specifies printing without execution; however, if the command line has the string $(MAKE) in it, the line is always executed (see discussion of the MAKEFLAGS macro under **Environment**, below. The –t (touch) option updates the modified date of a file without executing any commands.

Commands returning non-zero status normally terminate make. If the –i option is present, or the entry .IGNORE: appears in *makefile*, or the initial character sequence of the command contains -, the error is ignored. If the –k option is present, work is abandoned on the current entry, but continues on other branches that do not depend on that entry.

Interrupt and quit cause the target to be deleted unless the target is a dependent of the special name .PRECIOUS.

Environment

The environment is read by make. All variables are assumed to be macro definitions and processed as such. The environmental variables are processed before any makefile and after the internal rules; thus, macro assignments in a makefile override environmental variables. The –e option causes the environment to override the macro assignments in a makefile.

The environmental variable MAKEFLAGS is processed by make as containing any legal input option (except –f and –p) defined for the command line. Further, upon invocation, make ‘‘invents’’ the variable if it is not in the environment, puts the current options into it, and passes it on to invocations of commands. Thus, MAKEFLAGS always contains the current input options. This proves very useful for ''super-makes'' where the *makefile* contains actions that (recursively) invoke `make`. In fact, when the `-n` option is used, a recursive invocation of `make`, where the sequence `$(MAKE)` appears anywhere in the invocation command line, is executed anyway; hence, by judicious use of the `$(MAKE)` string in a *makefile*, one can perform a `make -n` recursively on a whole software system to see what would have been executed. This is because the `-n` is put in `MAKEFLAGS` and passed to further invocations of `make`. This is one way of debugging all of the makefiles for a software project without actually doing anything.

Macros

Entries of the form *string1* `=` *string2* are macro definitions. The macro *string2* is defined as all characters up to a comment character or an unescaped newline. Subsequent appearances of `$(`*string1*`[:`*subst1*`=[`*subst2*`]])` are replaced by *string2*. The parentheses are optional if a single character macro name is used and there is no substitute sequence. The optional `:`*subst1*`=`*subst2* is a substitute sequence. If it is specified, all non-overlapping occurrences of *subst1* in the named macro are replaced by *subst2*. Strings (for the purposes of this type of substitution) are delimited by blanks, tabs, newline characters, and beginnings of lines. An example of the use of the substitute sequence is shown under **Libraries**, below.

Internal Macros

There are five internally maintained macros which are useful for writing rules for building targets.

`$*` The macro `$*` stands for the file name part of the current dependent with the suffix deleted. It is evaluated only for inference rules.

`$@` The `$@` macro stands for the full target name of the current target. It is evaluated only for explicitly named dependencies.

`$<` The `$<` macro is only evaluated for inference rules or the `.DEFAULT` rule. It is the module which is out-of-date with respect to the target (*i.e.*, the ''manufactured'' dependent file name). Thus, in the `.c.o` rule, the `$<` macro would evaluate to the `.c` file. An example for making optimized `.o` files from `.c` files is:

```
.c.o
        cc -c -O $*.c
```

or:

```
.c.o:
        cc -c -O $<
```

`$?` The `$?` macro is evaluated when explicit rules from the makefile are evaluated. It is the list of prerequisites that are out-of-date with respect to the target; essentially, those modules which must be rebuilt.

`$%` The `$%` macro is only evaluated when the target is an archive library member of the form `lib(file.o)`. In this case, `$@` evaluates to `lib` and `$%` evaluates to the library member, *file.o*.

Four of the five macros can have alternative forms. When an upper case `D` or `F` is appended to any of the four macros, the meaning is changed to ''directory part'' for `D` and ''file part'' for `F`. Thus, `$(@D)` refers to the directory part of the string `$@`. If there is no directory part, `./` is generated. The only macro excluded from this alternative form is $?.

Suffixes

Certain names (for instance, those ending with .o) have inferable prerequisites such as .c, .s, *etc*. If no update commands for such a file appear in *makefile*, and if an inferable prerequisite exists, that prerequisite is compiled to make the target. In this case, make has inference rules which allow building files from other files by examining the suffixes and determining an appropriate inference rule to use. Inference rules in the *makefile* override the default rules.

The internal rules for make are compiled into the make program. To print out the rules compiled into the make program, the following command is used:

```
make –fp – 2>/dev/null </dev/null
```

A tilde in the above rules refers to an SCCS file. Thus, the rule .c˜.o would transform an SCCS **C** source file into an object file (.o). Because the s. of the SCCS files is a prefix, it is incompatible with make's suffix point of view. Hence, the tilde is a way of changing any file reference into an SCCS file reference.

A rule with only one suffix (*e.g.*, .c:) is the definition of how to build *x* from *x*.c. In effect, the other suffix is null. This is useful for building targets from only one source file (*e.g.*, command scripts, simple **C** programs).

Additional suffixes are given as the dependency list for .SUFFIXES. Order is significant; the first possible name for which both a file and a rule exist is inferred as a prerequisite.

Here again, the above command for printing the internal rules will display the list of suffixes implemented on the current machine. Multiple suffix lists accumulate; .SUFFIXES: with no dependencies clears the list of suffixes.

Inference Rules

The first example can be done more briefly.

```
pgm: a.o b.o
      cc a.o b.o –o pgm

a.o b.o: incl.h
```

This is because make has a set of internal rules for building files. The user may add rules to this list by simply putting them in the *makefile*.

Certain macros are used by the default inference rules to permit the inclusion of optional matter in any resulting commands. For example, CFLAGS, LFLAGS, and YFLAGS are used for compiler options to cc, lex, and yacc, respectively. Again, the previous method for examining the current rules is recommended.

The inference of prerequisites can be controlled. The rule to create a file with suffix .o from a file with suffix .c is specified as an entry with .c.o: as the target and no dependents. Commands associated with the target define the rule for making a .o file from a .c file. Any target that has no slashes in it and starts with a dot is identified as a rule and not a true target.

Libraries

If a target or dependency name contains parentheses, it is assumed to be an archive library, the string within parentheses referring to a member within the library. Thus lib(file.o) and $(LIB)(file.o) both refer to an archive library which contains *file.o*. (This assumes the LIB macro has been previously defined.) The expression $(LIB)(file1.o file2.o) is not legal. Rules pertaining to archive libraries have the form .*XX*.a where the *XX* is the suffix from which the archive member is to be made. The most common use of the archive interface follows. Here, we assume the source files are all **C** type source:

```
lib:
      lib(file1.o) lib(file2.o) lib(file3.o)
      @echo lib is now up-to-date

.c.a:
      $(CC) –c $(CFLAGS) $<
      ar rv $@ $*.o
      rm –f $*.o
```

In fact, the .c.a rule listed above is built into make and is unnecessary in this example. A more interesting, but more limited example of an archive library maintenance construction follows:

```
lib:
      lib(file1.o) lib(file2.o) lib(file3.o)
      $(CC) –c $(CFLAGS) $(?:.o=.c)
      ar rv lib $?
      rm $?
      @echo lib is now up-to-date

.c.a:;
```

Here the substitution mode of the macro expansions is used. The $? list is defined to be the set of object file names (inside *lib*) whose **C** source files are out-of-date. The substitution mode translates the .o to .c. Note also, the disabling of the .c.a: rule, which would have created each object file, one by one. This particular construct speeds up archive library maintenance considerably. This type of construct becomes very cumbersome if the archive library contains a mix of assembly programs and **C** programs.

FILES

[Mm]akefile and s.[Mm]akefile

USAGE

General.

The characters **= : @** in file names may give trouble.

SEE ALSO

cc(SD_CMD), lex(SD_CMD), sh(BU_CMD), yacc(SD_CMD).

LEVEL

Level 1.

NAME

nm – print name list of common object file

SYNOPSIS

`nm` [*options*] *file* ...

DESCRIPTION

The `nm` command displays the symbol table of each common object file *file*. The argument *file* may be a relocatable or absolute common object file; or it may be an archive of relocatable or absolute common object files. For each symbol, at least the following information is printed:

Name The name of the symbol.

Value Its value expressed as an offset or an address, depending on its storage class.

Size Its size in bytes, if available.

The output of `nm` may be controlled using the following *options*:

`-o` Print the value and size of a symbol in octal instead of decimal.

`-x` Print the value and size of a symbol in hexadecimal instead of decimal.

`-e` Print only external and static symbols.

`-f` Produce full output. Print redundant symbols (`.text`, `.data`, and `.bss`), normally suppressed.

`-u` Print undefined symbols only.

`-V` Print the version of the `nm` command executing on the standard error output.

SEE ALSO

cc(SD_CMD), ld(SD_CMD).

USAGE

General.

FUTURE DIRECTIONS

The options `-e` and `-f` will be removed.

LEVEL

Level 2: June 30, 1989.

NAME

prof – display profile data

SYNOPSIS

prof [-t | c | a | n] [-o | x] [-g | l] [-z]
[-m *mdata***]**

DESCRIPTION

The **prof** command interprets a profile file produced by the **monitor** function. The symbol table in the object file *prog* (**a.out** by default) is read and correlated with a profile file (**mon.out** by default). For each external text symbol the percentage of time spent executing between the address of that symbol and the address of the next is printed, together with the number of times that function was called and the average number of milliseconds per call.

The mutually exclusive options **-t**, **-c**, **-a**, and **-n** determine the type of sorting of the output lines:

-t Sort by decreasing percentage of total time (default).

-c Sort by decreasing number of calls.

-a Sort by increasing symbol address.

-n Sort lexically by symbol name.

The mutually exclusive options **-o** and **-x** specify the printing of the address of each symbol monitored:

-o Print each symbol address (in octal) along with the symbol name.

-x Print each symbol address (in hexadecimal) along with the symbol name.

The mutually exclusive options **-g** and **-l** control the type of symbols to be reported. The **-l** option must be used with care; it applies the time spent in a static function to the preceding (in memory) global function, instead of giving the static function a separate entry in the report. If all static functions are properly located (see example below), this feature can be very useful. If not, the resulting report may be misleading.

Assume that **A** and **B** are global functions and only **A** calls static function **S**. If **S** is located immediately after **A** in the source code (that is, if **S** is properly located), then, with the **-l** option, the amount of time spent in **A** can easily be determined, including the time spent in **S**. If, however, both **A** and **B** call **S**, then, if the **-l** option is used, the report will be misleading; the time spent during **B**'s call to **S** will be attributed to **A**, making it appear as if more time had been spent in **A** than really had. In this case, function **S** cannot be properly located.

-g Include static (non-global) functions.

-l Do not include static (non-global) functions (default).

The following options may be used in any combination:

-z Include all symbols in the profile range, even if associated with zero number of calls and zero time.

-m *mdata*
Use file *mdata* instead of **mon.out** as the input profile file.

A program creates a profile file if it has been link edited with the **-p** option of **cc**. This option to the **cc** command arranges for calls to **monitor** at the beginning and end of execution. It is the call to **monitor** at the end of execution that causes the system to write a profile file. The number of calls to a function is tallied if the **-p** option was used when the file containing the function was compiled.

The name of the file created by a profiled program is controlled by the environmental variable **PROFDIR**. If **PROFDIR** is not set, **mon.out** is produced in the directory current when the program terminates. If **PROFDIR=***string*, *string***/***pid***.***progname* is produced, where *progname* consists of **argv[0]** with any path prefix removed, and *pid* is the process ID of the program. If **PROFDIR** is set, but null, no profiling output are produced.

A single function may be split into subfunctions for profiling by means of the **MARK** macro

FILES

mon.out	default profile file
a.out	default namelist (object) file

USAGE

General.

The times reported in successive identical runs may show variances because of varying cache-hit ratios that result from sharing the cache with other processes. Even if a program seems to be the only one using the machine, hidden background or asynchronous processes may blur the data.

In rare cases, the clock ticks initiating recording of the program counter may "beat" with loops in a program, grossly distorting measurements. Call counts are always recorded precisely, however.

Only programs that call **exit(BA_OS)** are guaranteed to produce a profile file, unless a final call to **monitor**(SD_LIB) is explicitly coded.

SEE ALSO

cc(SD_CMD), **exit**(BA_OS), **profil**(KE_OS), **monitor**(SD_LIB), **mark**(SD_LIB).

LEVEL

Level 2.

NOTICES

The times reported in successive identical runs may show variances because of varying cache-hit ratios that result from sharing the cache with other processes. Even if a program seems to be the only one using the machine, hidden background or asynchronous processes may blur the data. In rare cases, the clock ticks initiating recording of the program counter may ‘‘beat’’ with loops in a program, grossly distorting measurements. Call counts are always recorded precisely, however.

Only programs that call **exit** or return from **main** are guaranteed to produce a profile file, unless a final call to **monitor** is explicitly coded.

The times for static functions are attributed to the preceding external text symbol if the **-g** option is not used. However, the call counts for the preceding function are still correct; that is, the static function call counts are not added to the call counts of the external function.

If more than one of the options **-t**, **-c**, **-a**, and **-n** is specified, the last option specified is used and the user is warned.

NAME

prs – print an SCCS file

SYNOPSIS

`prs` [*options*] *files*

DESCRIPTION

The command `prs` prints, on the standard output, parts or all of an SCCS file in a user supplied format. If a directory is named, `prs` behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with `s.`), and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file or directory to be processed; non-SCCS files and unreadable files are silently ignored.

Arguments to `prs`, which may appear in any order, consist of options and filenames.

All the described options apply independently to each named file.

`-d`[*dataspec*]
: Used to specify the output data specification. The *dataspec* is a string consisting of SCCS file *data keywords* (see **Data Keywords**) interspersed with optional user supplied text.

`-r`*SID*
: Used to specify the SCCS identification string of a delta for which information is desired. If no *SID* is specified, the SID of the most recently created delta is assumed.

`-e` Requests information for all deltas created *earlier* than and including the delta designated via the `–r` keyletter or the date given by the `–c` option.

`-l` Requests information for all deltas created *later* than and including the delta designated via the `–r` keyletter or the date given by the `–c` option.

`-c`[*date-time*]
: The cutoff *date-time* is in the form:

YY[*MM*[*DD*[*HH*[*MM*[*SS*]]]]]

Units omitted from the date-time default to their maximum possible values; for example, `–c7502` is equivalent to `-c750228235959`. Any number of non-numeric characters may separate the various two-digit pieces of the *cutoff* date in the form: `–c77/2/2 9:22:25`.

`-a` Requests printing of information for both removed (*i.e.*, delta type = *R*) deltas [see rmdel(SD_CMD)] and existing (*i.e.*, delta type = *D*) deltas. If the `–a` keyletter is not specified, information is provided for existing deltas only.

Data Keywords

Data keywords specify which parts of an SCCS file are to be retrieved and output. All parts of an SCCS file have an associated data keyword. There is no limit on the number of times a data keyword may appear in a *dataspec.*

The information printed by `prs` consists of: (1) the user supplied text; and (2) appropriate values (extracted from the SCCS file) substituted for the recognized data keywords in the order of appearance in the *dataspec* . The format of a data keyword value is either *Simple* (S), in which keyword substitution is direct, or *Multi-line* (M), in which keyword substitution is followed by a carriage return.

User supplied text is any text other than recognized data keywords. A tab is specified by `\t` and carriage return/newline is specified by `\n`. The default data keywords are:

```
":Dt:\t:DL:\nMRs:\n:MR:COMMENTS:\n:C:"
```

Table 1. SCCS Files Data Keywords

Keyword	*Data Item*	*File Section*	*Value*	*Format*
`:Dt:`	Delta information	Delta table	See * below	S
`:DL:`	Delta line statistics	Delta table	`:Li:/:Ld:/:Lu:`	S
`:Li:`	Lines inserted by Delta	Delta table	*nnnnn*	S
`:Ld:`	Lines deleted by Delta	Delta table	*nnnnn*	S
`:Lu:`	Lines unchanged by Delta	Delta table	*nnnnn*	S
`:DT:`	Delta type	Delta table	*D* or *R*	S
`:I:`	SCCS ID string (SID)	Delta table	`:R:.:L:.:B:.:S:`	S
`:R:`	Release number	Delta table	*nnnn*	S
`:L:`	Level number	Delta table	*nnnn*	S
`:B:`	Branch number	Delta table	*nnnn*	S
`:S:`	Sequence number	Delta table	*nnnn*	S
`:D:`	Date delta was created	Delta table	`:Dy:/:Dm:/:Dd:`	S
`:Dy:`	Year delta was created	Delta table	*nn*	S
`:Dm:`	Month delta was created	Delta table	*nn*	S
`:Dd:`	Day delta was created	Delta table	*nn*	S
`:T:`	Time delta was created	Delta table	`:Th:::Tm:::Ts:`	S
`:Th:`	Hour delta was created	Delta table	*nn*	S
`:Tm:`	Minutes delta was created	Delta table	*nn*	S
`:Ts:`	Seconds delta was created	Delta table	*nn*	S
`:P:`	Programmer who created delta	Delta table	`logname`	S
`:DS:`	Delta sequence number	Delta table	*nnnn*	S
`:DP:`	Predecessor delta seq. no.	Delta table	*nnnn*	S
`:DI:`	Seq. no. of deltas incl., excl., ignored	Delta table	`:Dn:/:Dx:/:Dg:`	S
`:Dn:`	Deltas included (seq no.)	Delta table	`:DS: :DS: . . .`	S
`:Dx:`	Deltas excluded (seq no.)	Delta table	`:DS: :DS: . . .`	S
`:Dg:`	Deltas ignored (seq no.)	Delta table	`:DS: :DS: . . .`	S
`:MR:`	MR numbers for delta	Delta table	text	M

:C:	Comments for delta	Delta table	text	M
:UN:	User names	User names	text	M
:FL:	Flag list	Flags	text	M
:Y:	Module type flag	Flags	text	S
:MF:	MR validation flag	Flags	*yes* or *no*	S
:MP:	MR validation program name	Flags	text	S
:KF:	Keyword error/warning flag	Flags	*yes* or *no*	S
:KV:	Keyword validation string	Flags	text	S
:BF:	Branch flag	Flags	*yes* or *no*	S
:J:	Joint edit flag	Flags	*yes* or *no*	S
:LK:	Locked releases	Flags	:R: . . .	S
:Q:	User defined keyword	Flags	text	S
:M:	Module name	Flags	text	S
:FB:	Floor boundary	Flags	:R:	S
:CB:	Ceiling boundary	Flags	:R:	S
:Ds:	Default SID	Flags	:I:	S
:ND:	Null delta flag	Flags	*yes* or *no*	S
:FD:	File descriptive text	Comments	text	M
:BD:	Body	Body	text	M
:GB:	Gotten body	Body	text	M
:W:	A form of what(SD_CMD) string	N/A	:Z::M:\t:I:	S
:A:	A form of what(SD_CMD) string	N/A	:Z::Y: :M: :I::Z:	S
:Z:	what(SD_CMD) string delimiter	N/A	@(#)	S
:F:	SCCS file name	N/A	text	S
:PN:	SCCS file pathname	N/A	text	S

* :Dt: = :DT: :I: :D: :T: :P: :DS: :DP:

EXAMPLES

```
prs –d"Users and/or user IDs for :F: are:\n:UN:" s.file
```

may produce on the standard output:

```
Users and/or user IDs for s.file are:
xyz
131
abc
```

```
prs –d"Newest delta for pgm :M:: :I: Created :D: By :P:" –r s.file
```

may produce on the standard output:

```
Newest delta for pgm main.c: 3.7 Created 77/12/1 By cas
```

As a *special case:*

```
prs s.file
```

may produce on the standard output:

```
D 1.1 77/12/1 00:00:00 cas 1 000000/00000/00000
MRs:
bl78-12345
bl79-54321
COMMENTS:
this is the comment line for s.file initial delta
```

for each delta table entry of the `D` type. The only keyletter argument allowed to be used with the *special case* is the `–a` keyletter.

SEE ALSO

admin(SD_CMD), delta(SD_CMD), get(SD_CMD), what(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

rmdel – remove a delta from an SCCS file

SYNOPSIS

`rmdel -r`*SID files*

DESCRIPTION

The command `rmdel` removes the delta specified by the *SID* from each named SCCS file. The delta to be removed must be the newest (most recent) delta in its branch in the delta chain of each named SCCS file. In addition, the *SID* specified must not be that of a version being edited for the purpose of making a delta (*i.e.*, if a *p-file* [see get(SD_CMD)] exists for the named SCCS file, the *SID* specified must not appear in any entry of the *p-file*).

If a directory is named, `rmdel` behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of **–** is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed; non-SCCS files and unreadable files are silently ignored.

The restrictions on removal of a delta are that only the user who made it or the owner of the file and directory can remove a delta.

SEE ALSO

delta(SD_CMD), get(SD_CMD), prs(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

sact – print current SCCS file editing activity

SYNOPSIS

`sact` *file* ...

DESCRIPTION

The command `sact` informs the user of any impending deltas to a named SCCS file. This situation occurs when `get –e` has been previously executed without a subsequent execution of `delta`. If a directory is named on the command line, `sact` behaves as though each file in the directory were specified as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of – is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

The output for each named file consists of five fields separated by spaces.

Field 1
: specifies the SID of a delta that currently exists in the SCCS file to which changes will be made to create the new delta.

Field 2
: specifies the SID for the new delta to be created.

Field 3
: contains the logname of the user who will make the delta (*i.e.*, executed a `get` for editing).

Field 4
: contains the date that `get –e` was executed.

Field 5
: contains the time that `get –e` was executed.

SEE ALSO

delta(SD_CMD), get(SD_CMD), unget(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

size – print section sizes of object files

SYNOPSIS

`size` [`-o`] [`-x`] [`-V`] *file* ...

DESCRIPTION

The `size` command produces section size information for each section in the named object files. The sizes of the loaded sections are printed along with the sum of these sizes. If an archive file is input to the `size` command, the information for all archive members is displayed.

Numbers are printed in decimal unless either the `–o` or the `–x` option is used, in which case they are printed in octal or hexadecimal, respectively.

The `–V` flag supplies the version information on the `size` command.

SEE ALSO

cc(SD_CMD), ld(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

strip – strip symbol table, debugging and line number information from an object file.

SYNOPSIS

strip [-Vx] *file* **. . .**

DESCRIPTION

The **strip** command strips the symbol table, debugging information, and line number information from ELF object files; COFF object files can no longer be stripped. Once this stripping process has been done, no symbolic debugging access will be available for that file; therefore, this command is normally run only on production modules that have been debugged and tested.

If **strip** is executed on a common archive file [see **ar**(BU_CMD)] in addition to processing the members, **strip** will remove the archive symbol table. The archive symbol table must be restored by executing the **ar**(BU_CMD) command with the **-s** option before the archive can be linked by the **ld**(SD_CMD) command. **strip** will produce appropriate warning messages when this situation arises.

The amount of information stripped from the ELF object file can be controlled by using any of the following options:

-V Print, on standard error, the version number of **strip**.

-x Do not strip the symbol table; debugging and line number information may be stripped.

strip is used to reduce the file storage overhead taken by the object file.

SEE ALSO

ar(BU_CMD), **as**(SD_CMD), **cc**(SD_CMD), **ld**(SD_CMD)

LEVEL

Level 1.

NOTICES

The symbol table section will not be removed if it is contained within a segment, or the file is either a relocatable or dynamic shared object.

The line number and debugging sections will not be removed if they are contained within a segment, or their associated relocation section is contained within a segment.

NAME

time – time a command

SYNOPSIS

`time` *command*

DESCRIPTION

The *command* is executed; after it is complete, `time` prints the elapsed time during the command, the time spent executing system code, and the time spent in execution of the user code. Times are reported in seconds.

The times are printed on standard error.

USAGE

General.

When `time` is used on a multi-processor system the sum of system and user time could be greater than real time.

LEVEL

Level 1.

NAME

truss – trace system calls and signals

SYNOPSIS

truss [-pfcaei] [-t[!]*syscall*[,*syscall*...]] [-v[!]*syscall*[,*syscall*...]]
[-x[!]*syscall*[,*syscall*...]] [-s[!]*signal*[,*signal*...]] [-m[!]*fault*[,*fault*...]]
[-r[!]*fd*[,*fd*...]] [-w[!]*fd*[,*fd*...]] [-o *outfile*] *command*

DESCRIPTION

truss executes the specified command and produces a trace of the system calls it performs, the signals it receives, and the machine faults it incurs. Each line of the trace output reports either the fault or signal name or the system call name with its arguments and return value(s). System call arguments are displayed symbolically, when possible, using defines from relevant system header files; for any pathname pointer argument, the pointed-to string is displayed. Error returns are reported using the error code names described in errno().

The following options are recognized. For those options which take a list argument, the name all can be used as a shorthand to specify all possible members of the list. If the list begins with a '!', the meaning of the option is negated (*e.g.*, exclude rather than trace). Multiple occurrences of the same option may be specified. For the same name in a list, subsequent options (those to the right) override previous ones (those to the left).

–p
Interpret the arguments to truss as a list of process-ids for existing processes [see ps(BU_CMD)]. rather than as a command to be executed. truss takes control of each process and begins tracing it provided that the userid and groupid of the process match those of the user or that the user is super-user.

–f
Follow all children created by fork() and include their signals, faults, and system calls in the trace output. Normally, only the first-level command or process is traced. When –f is specified, the process-id is included with each line of trace output to indicate which process executed the system call or received the signal.

–c
Count traced system calls, faults, and signals rather than displaying the trace line-by-line. A summary report is produced after the traced command terminates or when truss is interrupted. If –f is also specified, the counts include all traced system calls, faults, and signals for child processes.

–a
Show the argument strings which are passed in each exec(BA_OS) system call.

–e
Show the environment strings which are passed in each exec(BA_OS) system call.

–i
Don't display interruptible sleeping system calls. Certain system calls, such as open() and read() on terminal devices or pipes can sleep for indefinite periods and are interruptible. Normally, truss reports such sleeping system calls if they remain asleep for more than one second. The system call is reported again a

second time when it completes. The `-i` option causes such system calls to be reported only once, when they complete.

`-t` [!]*syscall,...* System calls to trace or exclude. Those system calls specified in the comma-separated list are traced. If the list begins with a '!', the specified system calls are excluded from the trace output. Default is `-tall`.

`-v` [!]*syscall,...* Verbose. Display the contents of any structures passed by address to the specified system calls (if traced). Input values as well as values returned by the operating system are shown. For any field used as both input and output, only the output value is shown. Default is `-v!all`.

`-x` [!]*syscall,...* Display the arguments to the specified system calls (if traced) in raw form, usually hexadecimal, rather than symbolically. Default is `-x!all`.

`-s` [!]*signal,...* Signals to trace or exclude. Those signals specified in the comma-separated list are traced. The trace output reports the receipt of each specified signal, even if the signal is being ignored (not blocked) by the process. (Blocked signals are not received until the process releases them.) Signals may be specified by name or number (see `<sys/signal.h>`). If the list begins with a '!', the specified signals are excluded from the trace output. Default is `-sall`.

`-m` [!]*fault,...* Machine faults to trace or exclude. Those machine faults specified in the comma-separated list are traced. Faults may be specified by name or number (see `<sys/fault.h>`). If the list begins with a '!', the specified faults are excluded from the trace output. Default is `-mall -m!fltpage`.

`-r` [!]*fd,...* Show the full contents of the I/O buffer for each `read()` on any of the specified file descriptors. The output is formatted 32 bytes per line and shows each byte as an ASCII character (preceded by one blank) or as a 2-character **C** language escape sequence for control characters such as horizontal tab (`\t`) and newline (`\n`). If ASCII interpretation is not possible, the byte is shown in 2-character hexadecimal representation. (The first 16 bytes of the I/O buffer for each traced `read()` are shown even in the absence of `-r`.) Default is `-r!all`.

`-w` [!]*fd,...* Show the contents of the I/O buffer for each `write()` on any of the specified file descriptors (see `-r`). Default is `-w!all`.

`-o` *outfile* File to be used for the trace output. By default, the output goes to standard error.

If `truss` is used to initiate and trace a specified command and if the `-o` option is used or if standard error is redirected to a non-terminal file, then `truss` runs with hangup, interrupt, and quit signals ignored. This facilitates tracing of interactive programs which catch interrupt and quit signals from the terminal.

If the trace output remains directed to the terminal, or if existing processes are traced (the -p option), then truss responds to hangup, interrupt, and quit signals by releasing all traced processes and exiting. This allows the user to terminate excessive trace output and to release previously-existing processes. Released processes continue normally, as though they had never been touched.

SEE ALSO

errno(BA_ENV)

LEVEL

Level 1.

NAME

tsort – topological sort

SYNOPSIS

`tsort` [*file*]

DESCRIPTION

`tsort` produces on the standard output a totally ordered list of items consistent with a partial ordering of items mentioned in the input *file*. If no *file* is specified, the standard input is understood.

The input consists of pairs of items (nonempty strings) separated by blanks. Pairs of different items indicate ordering. Pairs of identical items indicate presence, but not ordering.

SEE ALSO

lorder(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

unget – undo a previous get of an SCCS file

SYNOPSIS

`unget` [`-r`*SID*] [`-s`] [`-n`] *files*

DESCRIPTION

`Unget` undoes the effect of a `get -e` done prior to creating the intended new delta. If a directory is named, `unget` behaves as though each file in the directory were specified as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of – is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

Keyletter arguments apply independently to each named file.

`–r`*SID*
: Uniquely identifies which delta is no longer intended. (This would have been specified by `get` as the new delta). The use of this keyletter is necessary only if two or more outstanding `get`s for editing on the same SCCS file were done by the same person (login name). An error is reported if the specified *SID* is ambiguous, or if it is necessary and omitted on the command line.

`–s`
: Suppresses the printout of the intended delta's *SID* on the standard output.

`–n`
: Causes the retention of the file that was obtained by `get`, which would normally be removed from the current directory.

SEE ALSO

delta(SD_CMD), get(SD_CMD), sact(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

val – validate SCCS file

SYNOPSIS

`val –`

`val [-s] [-r`*SID*`] [-m`*name*`] [-y`*type*`]` *file* `...`

DESCRIPTION

The command `val` determines if the specified *file* is an SCCS file meeting the characteristics specified by the options. The arguments may appear in any order.

`val` has a special argument, `–`, which causes reading of the standard input until an end-of-file condition is detected. Each line read is independently processed as if it were a command line argument list.

`val` generates diagnostic messages on the standard output for each command line and file processed, and also returns a single 8-bit code upon exit as described below.

The options are defined as follows. The effects of any option apply independently to each named file on the command line.

`-s` Silences the diagnostic message, normally generated on the standard output, for any error that is detected while processing each named file on a given command line.

`-r`*SID* *SID* (SCCS Identification String) is an SCCS delta number. A check is made to determine if the *SID* is ambiguous (*e.g.*, `-r1` is ambiguous because it physically does not exist but implies 1.1, 1.2, *etc.*, which may exist) or invalid (*e.g.*, `-r1.0` or `-r1.1.0` are invalid because neither case can exist as a valid delta number). If the *SID* is valid and not ambiguous, a check is made to determine if it actually exists.

`-m`*name* *name* is compared with the SCCS `%M%` keyword in *file.*

`-y`*type* *type* is compared with the SCCS `%Y%` keyword in *file.*

The 8-bit code returned by `val` is a disjunction of the possible errors, i. e., it can be interpreted as a bit string where (moving from left to right) set bits are interpreted as follows:

bit 0 = missing file argument;
bit 1 = unknown or duplicate keyletter argument;
bit 2 = corrupted SCCS file;
bit 3 = cannot open file or file not SCCS;
bit 4 = *SID* is invalid or ambiguous;
bit 5 = *SID* does not exist;
bit 6 = `%Y%`, `–y` mismatch;
bit 7 = `%M%`, `–m` mismatch;

Note that `val` can process two or more files on a given command line and in turn can process multiple command lines (when reading the standard input). In these cases an aggregate code is returned, *i.e.* the logical `OR` of the codes generated for each command line and file processed.

SEE ALSO

admin(SD_CMD), delta(SD_CMD), get(SD_CMD), prs(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

what – identify SCCS files

SYNOPSIS

`what [–s]` *files*

DESCRIPTION

The `what` command searches the given files for all occurrences of the pattern that the `get` command substitutes for `%Z% (@(#))` and prints out what follows until the first `"`, `>`, newline, `\`, or null character. For example, if the C language program in file `f.c` contains

`char ident[] = "@(#)` *identification information* `";`

and `f.c` is compiled to yield `f.o` and `a.out`, then the command

```
what f.c f.o a.out
```

will print

`f.c:`
  identification information
`f.o:`
  identification information
`a.out:`
  identification information

`what` is intended to be used in conjunction with the SCCS `get` command, which automatically inserts identifying information, but it can also be used where the information is inserted manually.

There is at least one option:

`-s` Quit after finding the first occurrence of pattern in each file.

ERRORS

Exit status is 0 if any matches are found; otherwise it is 1.

SEE ALSO

get(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

xargs – construct argument list(s) and execute command

SYNOPSIS

`xargs` [*options*] [*command* [*initial_arguments*]]

DESCRIPTION

`xargs` combines the fixed *initial_arguments* with arguments read from standard input to execute the specified *command* one or more times. The number of arguments read for each *command* invocation and the manner in which they are combined are determined by the options specified.

If *command* is omitted, `echo` is used.

Arguments read in from standard input are defined to be contiguous strings of characters delimited by one or more blanks, tabs, or new lines; empty lines are always discarded. Blanks and tabs may be embedded as part of an argument if escaped or quoted. Characters enclosed in quotes (single or double) are taken literally, and the delimiting quotes are removed. Outside of quoted strings a backslash (\) quotes the next character.

Each argument list is constructed starting with the initial_arguments, followed by some number of arguments read from standard input (Exception: see `-i`). Options `-i`, `-l`, and `-n` determine how arguments are selected for each command invocation. When none of these options are coded, the initial_arguments are followed by arguments read continuously from standard input until an internal buffer is full, and then *command* is executed with the accumulated arguments. This process is repeated until all arguments have been read. When there are conflicts (*e.g.*, `-l` *vs.* `-n`), the last option has precedence. The recognized options are:

`-l`*number* `Command` is executed for each non-empty *number* lines of arguments from standard input. The last invocation of *command* will be with fewer lines of arguments if fewer than *number* remain. A line is considered to end with the first newline unless the last character of the line is a blank or a tab; a trailing blank/tab signals continuation through the next non-empty line. If *number* is omitted, 1 is assumed. Option `-x` is forced.

`-i`*replstr* Insert mode: *command* is executed for each line from standard input, taking the entire line as a single argument, inserting it in *initial_arguments* for each occurrence of *replstr*. A maximum of five arguments in *initial_arguments* may each contain one or more instances of *replstr*. Blanks and tabs at the beginning of each line are thrown away. Constructed arguments may not expand to more than `{NAME_MAX}` characters, and option `-x` is also forced. `{}` is assumed for `replstr` if not specified.

`-n`*number* Execute *command* using as many standard input arguments as possible, up to *number* arguments maximum. Fewer arguments will be used if their total size is greater than *size* characters (see `-s` option, below), and for the last invocation if there are fewer than *number* arguments remaining. If option `-x` is also invoked, each *number* argument must fit in the *size* limitation, else `xargs` terminates execution.

`-t` Trace mode: The *command* and each constructed argument list are echoed to standard error just prior to their execution.

`-p` Prompt mode: The user is asked whether to execute *command* each invocation. Trace mode (`-t`) is turned on to print the command instance to be executed, followed by a `?...` prompt. A reply of `y` (optionally followed by anything) will execute the command; anything else, including just a carriage return, skips that particular invocation of *command.*

`-x` Causes `xargs` to terminate if any argument list would be greater than *size* characters; `-x` is forced by the options `-i` and `-l`. When neither of the options `-i`, `-l`, or `-n` are coded, the total length of all arguments must be within the *size* limit.

`-s`*size* The maximum total size of each argument list is set to *size* characters; *size* must be a positive integer less than or equal to 470. If `-s` is not coded, 470 is taken as the default. Note that the character count for *size* includes one extra character for each argument and the count of characters in the command name.

`-e`*eofstr* *eofstr* is taken as the logical end-of-file string. Underscore (_) is assumed for the logical `EOF` string if `-e` is not invoked. The option `-e` with no *eofstr* coded turns off the logical `EOF` string capability (underbar is taken literally). `xargs` reads standard input until either end-of-file or the logical `EOF` string is encountered.

`xargs` will terminate if either it receives a return code of `-1` from, or if it cannot execute, *command.* (Thus *command* should explicitly `exit` with an appropriate value to avoid accidentally returning with `-1`.)

USAGE

General.

Note that `xargs` does not perform parameter substitution. In the following examples, only the command processor performs substitutions.

EXAMPLES

The following will move all files from directory $1 to directory $2, and echo each move command just before doing it:

```
ls $1 | xargs -i -t mv $1/{} $2/{}
```

The following will combine the output of the parenthesized commands onto one line, which is then echoed to the end of file `log`:

```
(logname; date; echo $0 $*) | xargs >> log
```

The user is asked which files in the current directory are to be archived and archives them into `arch` (a.) one at a time, or (b.) many at a time.

a. `ls | xargs -p -l ar -r arch`
b. `ls | xargs -p -l | xargs ar -r arch`

The following will execute with successive pairs of arguments originally typed as command line arguments:

```
echo $* | xargs -n2 diff
```

SEE ALSO

echo(BU_CMD) sh(BU_CMD).

LEVEL

Level 1.

NAME

yacc – a compiler-compiler

SYNOPSIS

`yacc [-vdlt]` *grammar*

DESCRIPTION

The `yacc` command provides a general tool for describing the input to a program. More precisely, `yacc` converts a context-free grammar into a set of tables for a simple automaton which executes an *LR(1)* parsing algorithm. The grammar may be ambiguous; built-in precedence rules are used to break ambiguities.

The output file, `y.tab.c`, must be compiled by the **C** compiler to produce a program `yyparse()`. This program must be loaded with the lexical analyzer function, `yylex()`, as well as `main()` and `yyerror()`, an error handling routine. These routines must be supplied by the user (however, see the description of the `yacc` library below); `lex` is useful for creating lexical analyzers usable by `yacc`.

If the `–v` option is used, the file `y.output` is prepared, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

If the `–d` option is used, the file `y.tab.h` is generated with the `#define` statements that associate the `yacc`-assigned ''token codes'' with the user-declared ''token names''. This allows source files other than `y.tab.c` to access the token codes.

If the `–l` option is used, the code produced in `y.tab.c` does not contain any `#line` constructs. This should only be used after the grammar and the associated actions are fully debugged.

Runtime debugging code is always generated in `y.tab.c` under conditional compilation control. By default, this code is not included when `y.tab.c` is compiled. However, when `yacc`'s `–t` option is used, this debugging code will be compiled by default. Independent of whether the `–t` option was used, the runtime debugging code is under the control of `YYDEBUG`, a pre-processor symbol. If `YYDEBUG` has a non-zero value, then the debugging code is included. If its value is zero, then the code is not included. A program produced without the runtime debugging code will be smaller and slightly faster.

yacc Library

The `yacc` library `liby.a` facilitates the initial use of `yacc` by providing the routines:

```
main()
yyerror(char *s)
```

These routines may be loaded by using the `–ly` option with `cc`. The `main()` routine just calls `yyparse()`. `yyerror()` simply prints the string (error message) `s` when a syntax error is detected.

yacc SPECIFICATIONS

The `yacc` user constructs a specification of the input process; this includes rules describing the input structure, the code that will be invoked when these rules are recognized, and a low-level routine to do the basic input. Then `yacc` generates the (integer valued) function `yyparse()`; it in turn calls `yylex()`, the lexical analyzer, to obtain input tokens.

A structure recognized (and returned) by the lexical analyzer is called a *terminal symbol*, here referred to as a *token* (literal characters must also be passed through the lexical analyzer, and are also considered tokens). A structure recognized by the parser is called a *nonterminal symbol*. *Name* refers to either tokens or nonterminal symbols.

Every specification file consists of three sections: declarations, grammar rules, and programs, separated by double percent marks (%%). The declarations and programs sections may be empty. If the latter is empty, then the preceding %% marks separating it from the rules section may be omitted.

Blanks, tabs, and new lines are ignored, except that they may not appear in names or multi-character reserved symbols. Comments are enclosed in `/*` ... `*/`, and may appear wherever a name is legal.

Names may be of arbitrary length, made up of letters, dot (`.`), underscore (_), and non-initial digits. Upper and lower case letters are distinct. Names beginning with `yy` should be avoided because the `yacc` parser uses such names.

A literal consists of a character enclosed in single quotes. The **C** escape sequences (*e.g.*, `\n`) are recognized.

Declarations

The following declarators may be used in the declarations section:

`%token` Names representing tokens must be declared; this may be done by writing:

`%token` *name1 name2 ...*

in the declarations section. Every name not defined in this section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one grammar rule.

`%start` The start symbol represents the largest, most general structure described by the grammar rules. By default, it is the left-hand side of the first grammar rule; this default may be overridden by declaring:

`%start` *symbol*

`%left`
`%right`
`%nonassoc`

Precedence and associativity rules attached to tokens are declared using these keywords. This is done by a series of lines, each beginning with one of the keywords `%left`, `%right`, or `%nonassoc`, followed by a list of tokens. (If a token is declared using one of these keywords, a declaration by `%token` is not needed.) All tokens on the same line have the same precedence level and associativity; the lines are in order of increasing precedence or binding strength. The keyword `%left` denotes that the operators on that line are left associative, and `%right` denotes that the operators are right associative. The keyword `%nonassoc` denotes operators that may not associate with themselves.

`%prec` Unary operators must, in general, be given a precedence. In cases where a unary and binary operator have the same symbolic representation, but need to be given different precedences, the keyword `%prec` is used to change the order of precedence associated with a particular grammar rule. The keyword `%prec` appears immediately after the body of the grammar rule, before the action or closing semicolon (see **Grammar Rules** below). It is followed by a token name or a literal. It causes the precedence of the grammar rule to become that of the following token name or literal.

`%union` By default, the values returned by actions and the lexical analyzer are integers. Other value types, including structures, are supported: the `yacc` value stack is declared to be a union of the various types of values desired. The `yacc` command keeps track of types, and inserts appropriate union member names so that the resulting parser command is strictly type-checked. The declaration is constructed by including a statement of the form:

```
%union {
        body of union
}
```

Alternatively, the union may be declared in a header file, and a typedef used to define the variable `YYSTYPE` to represent this union. The header file must be included in the declarations section, by using a `#include` construct within `%{` and `%}` (see below). Union members must be associated with the various names. The construction <*name*> is used to indicate a union member name; if this follows one of the keywords `%token` `%left`, `%right`, and `%nonassoc`, the union member name is associated with the tokens listed.

`%type` This keyword is used to associate union member names with nonterminals, in the form:

```
%type <ntype> a b ...
```

Other declarations and definitions can appear in the declarations section, enclosed by the marks `%{` and `%}`. These have global scope within the file, so that they may be used in the rules and programs sections.

Grammar Rules

The rules section consists of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

The symbol `A` represents a nonterminal name, and `BODY` represents a sequence of zero or more names and literals. The colon and the semicolon are `yacc` punctuation. If several successive grammar rules have the same left-hand side, the vertical bar (`|`) can be used to avoid rewriting the left-hand side; in this case, the semicolon must occur only after the last rule. The `BODY` part may be empty to indicate that the nonterminal symbol matches the empty string.

The ASCII null character (`0` or `'\0'`) should not be used in grammar rules.

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. In addition, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary **C** statement, and as such can do input or output, call sub-programs, and alter external variables. An action is one or more statements enclosed by braces `{` and `}`. Certain pseudo-variables can be used in the action. A value can be returned by assigning it to `$$`; the variables `$1`, `$2`, ..., refer to the values returned by the components of the right side of a rule, reading from left to right. By default, the value of a rule is the value of the first element in it. Actions may occur in the middle of a rule as well as at the end. An action may access the values returned by symbols (and actions) to its left: and, in turn, the value it returns may be accessed by actions to its right.

Internal rules to resolve ambiguities are:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the grammar rule that occurs *earlier* in the input sequence.

In addition, the declared precedences and associativities (see **Declarations Section** above) are used to resolve parsing conflicts as follows:

1. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the `%prec` keyword is used, it overrides this default. Some grammar rules may have no precedence and associativity.
2. When a reduce/reduce conflict, or a shift/reduce conflict occurs and either the input symbol or the grammar rule has no precedence and associativity, then the two internal rules given above are used.
3. If a shift/reduce conflict occurs, and both the grammar rule and the input symbol have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociative implies error.

Conflicts resolved by precedence are not counted in the shift/reduce and reduce/reduce conflicts reported by `yacc`.

The token name `error` is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. When an error is encountered, the parser behaves as if the token `error` were the current lookahead token, and it performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

To prevent a series of error messages, the parser, after detecting an error, remains in the error state until three tokens have been successfully read and shifted. If an error is detected while the parser is in the error state, no message is given, and the input token is quietly deleted.

The statement

```
yyerrok;
```

in an action resets the parser to its normal mode; it may be used if it is desired to force the parser to believe that an error has been fully recovered from.

The statement

```
yyclearin;
```

in an action is used to clear the previous lookahead token; it may be used if a user supplied routine is to be used to find the correct place to resume input.

Programs

The programs section may include the definition of the lexical analyzer `yylex()`, or other functions, typically those used in the actions specified in the grammar rules.

The lexical analyzer `yylex()` is an integer valued function which returns the token number, representing the kind of token read. If a value is associated with that token, it should be assigned to the external variable `yylval`. The parser and `yylex()` must agree on these token numbers in order for communications between them to take place. The numbers may be chosen by `yacc`, or chosen by the user. In either case, the `#define` construct of C is used to allow `yylex()` to return these numbers symbolically. If the token numbers are chosen by `yacc`, then literals are given the numerical value of the character in the local character set, and other names are assigned token numbers starting at 257.

A token may be assigned a number by following its first appearance in the declarations section with a nonnegative integer. Names and literals not defined this way retain their default definition. All token numbers must be distinct.

The end of the input is marked by a special token called the endmarker. The endmarker must have token number 0 or negative. These values are not legal for any other token. All lexical analyzers should return 0 or negative as a token number upon reaching the end of their input. If the token up to, but not including, the endmarker forms a structure which matches the start symbol, the parser accepts the input. If the endmarker is seen in any other context, it is an error.

ERRORS

The number of reduce-reduce and shift-reduce conflicts is reported on the standard error output; a more detailed report is found in the `y.output` file. Similarly, if some rules are not reachable from the start symbol, this is also reported.

FILES

```
y.output
y.tab.c
y.tab.h
```

USAGE

General.

SEE ALSO

lex(SD_CMD).

LEVEL

Level 1.

Terminal Interface Introduction

Terminal Interface Overview

The Terminal Interface Extension (TI) consists of the facilities provided by the **curses/terminfo** package to allow application programs to perform terminal handling functions in a way that is independent of the type of the terminal actually in use. The **curses/terminfo** package supports an asynchronous color character terminal interface (on asynchronous character and bitmapped terminals).

The following are prerequisite for support of the Terminal Interface Extension:

- Base System
- Basic Utilities Extension
- Advanced Utilities Extension
- Software Development Extension

SUMMARY OF LIBRARY ROUTINES

The following library routines are supported by a SVID-compliant Terminal Interface Extension. Items marked with a (*) are Level 2, as defined in the *General Introduction* to this volume. Items marked with a (†) are new to this issue of the SVID. Only those pages reflecting technical content changes or which are new to the SVID are contained in this volume.

Curses Routines

addch	**addwstr**	**box**	**curs_set**
addchnstr	**attroff**	**can_change_color**	**def_prog_mode**
addchstr	**attron**	**cbreak**	**def_shell_mode**
addnstr	**attrset**	**clear**	**del_curterm**
addnwstr	**baudrate**	**clearok**	**delay_output**
addstr	**beep**	**clrtobot**	**delch**
addwch	**bkgd**	**clrtoeol**	**deleteln**
addwchnstr	**bkgdset**	**color_content**	**delscreen**
addwchstr	**border**	**copywin**	**delwin**

FINAL COPY
June 15, 1995
File: ti_int.txt
svid

derwin
doupdate
dupwin
echo
echochar
echowchar
endwin
erase
erasechar
filter
flash
flushinp
getbegyx
getch
getmaxyx
getnwstr
getparyx
getstr
getsyx
getwch
getwin
getwstr
getyx
halfdelay
has_colors
has_ic
has_il
idcok
idlok
immedok
inch
inchnstr
inchstr
init_color
init_pair
initscr
innstr
innwstr
insch
insdelln
insertln
insnstr
insnwstr
insstr
instr
inswch
inswstr
intrflush
inwch
inwchnstr
inwchstr
inwstr
is_linetouched
is_wintouched
isendwin
keyname
keypad
killchar
leaveok
longname
meta
move
mvaddch
mvaddchnstr
mvaddchstr
mvaddnstr
mvaddnwstr
mvaddstr
mvaddwch
mvaddwchnstr
mvaddwchstr
mvaddwstr
mvcur
mvdelch
mvderwin
mvgetch
mvgetnwstr
mvgetstr
mvgetwch
mvgetwstr
mvinch
mvinchnstr
mvinchstr
mvinnstr
mvinnwstr
mvinsch
mvinsnstr
mvinsnwstr
mvinsstr
mvinstr
mvinswch
mvinswstr
mvinwch
mvinwchnstr
mvinwchstr
mvinwstr
mvprintw
mvscanw
mvwaddch
mvwaddchnstr
mvwaddchstr
mvwaddnstr
mvwaddnwstr
mvwaddstr
mvwaddwch
mvwaddwchnstr
mvwaddwchstr
mvwaddwstr
mvwdelch
mvwgetch
mvwgetnwstr
mvwgetstr
mvwgetwch
mvwgetwstr
mvwin
mvwinch
mvwinchnstr
mvwinchstr
mvwinnstr
mvwinnwstr
mvwinsch
mvwinsnstr
mvwinsnwstr
mvwinsstr
mvwinstr
mvwinswch
mvwinswstr
mvwinwch
mvwinwchnstr
mvwinwchstr
mvwinwstr
mvwprintw
mvwscanw
napms
newpad
newterm
newwin
nl
nocbreak
nodelay
noecho
nonl
noqiflush
noraw

notimeout	**slk_attrset**	**use_env**	**wgetwstr**
overlay	**slk_clear**	**vidattr**	**whline**
overwrite	**slk_init**	**vidputs**	**winch**
pair_content	**slk_label**	**vwprintw**	**winchnstr**
pechochar	**slk_noutrefresh**	**vwscanw**	**winchstr**
pechowchar	**slk_refresh**	**waddch**	**winnstr**
pnoutrefresh	**slk_restore**	**waddchnstr**	**winnwstr**
prefresh	**slk_set**	**waddchstr**	**winsch**
printw	**slk_touch**	**waddnstr**	**winsdelln**
putp	**srcl**	**waddnwstr**	**winsertln**
putwin	**standend**	**waddstr**	**winsnstr**
qiflush	**standout**	**waddwch**	**winsnwstr**
raw	**start_color**	**waddwchnstr**	**winsstr**
redrawwin	**subpad**	**waddwchstr**	**winstr**
refresh	**subwin**	**waddwstr**	**winswch**
reset_prog_mode	**syncok**	**wattroff**	**winswstr**
reset_shell_mode	**termattrs**	**wattron**	**winwch**
resetty	**termname**	**wattrset**	**winwchnstr**
restartterm	**tgetent***	**wbkgd**	**winwchstr**
ripoffline	**tgetflag***	**wbkgdset**	**winwstr**
savetty	**tgetnum***	**wborder**	**wmove**
scanw	**tgetstr***	**wclear**	**wnoutrefresh**
scr_dump	**tgoto***	**wclrtobot**	**wprintw**
scr_init	**tigetflag**	**wclrtoeol**	**wredrawln**
scr_restore	**tigetnum**	**wcursyncup**	**wrefresh**
scr_set	**tigetstr**	**wdelch**	**wscanw**
scroll	**timeout**	**wdeleteln**	**wscrl**
scrollok	**touchline**	**wechochar**	**wsetscrreg**
set_curterm	**touchwin**	**wechowchar**	**wstandend**
set_term	**tparm**	**werase**	**wstandout**
setscrreg	**tputs**	**wgetch**	**wsyncdown**
setsyx	**typeahead**	**wgetnstr**	**wsyncup**
setterm*	**unctrl**	**wgetnwstr**	**wtimeout**
setupterm	**ungetch**	**wgetstr**	**wtouchln**
slk_attroff	**ungetwch**	**wgetwch**	**wvline**
slk_attron	**untouchwin**		

Forms Routines

current_field	form_init	set_field_buffer
data_ahead	form_opts	set_field_fore
data_behind	form_opts_off	set_field_init
dup_field	form_opts_on	set_field_just
dynamic_field_info	form_page	set_field_opts
field_arg	form_sub	set_field_pad
field_back	form_term	set_field_status
field_buffer	form_userptr	set_field_term
field_count	form_win	set_field_type
field_fore	free_field	set_field_userptr
field_index	free_fieldtype	set_fieldtype_arg
field_info	free_form	set_fieldtype_choice
field_init	link_field	set_form_fields
field_just	link_fieldtype	set_form_init
field_opts	move_field	set_form_opts
field_opts_off	new_field	set_form_page
field_opts_on	new_fieldtype	set_form_sub
field_pad	new_form	set_form_term
field_status	new_page	set_form_userptr
field_term	pos_form_cursor	set_form_win
field_type	post_form	set_max_field
field_userptr	scale_form	set_new_page
form_driver	set_current_field	unpost_form
form_fields	set_field_back	

Menu Routines

current_item	item_opts_off	menu_format
free_item	item_opts_on	menu_grey
free_menu	item_term	menu_init
item_count	item_userptr	menu_items
item_description	item_value	menu_mark
item_index	item_visible	menu_opts
item_init	menu_back	menu_opts_off
item_name	menu_driver	menu_opts_on
item_opts	menu_fore	menu_pad

Menu Routines

menu_pattern	**set_item_init**	**set_menu_mark**
menu_sub	**set_item_opts**	**set_menu_opts**
menu_term	**set_item_term**	**set_menu_pad**
menu_userptr	**set_item_userptr**	**set_menu_pattern**
menu_win	**set_item_value**	**set_menu_sub**
new_item	**set_menu_back**	**set_menu_term**
new_menu	**set_menu_fore**	**set_menu_userptr**
pos_menu_cursor	**set_menu_format**	**set_menu_win**
post_menu	**set_menu_grey**	**set_top_row**
scale_menu	**set_menu_init**	**top_row**
set_current_item	**set_menu_items**	**unpost_menu**

Panel Routines

bottom_panel	**panel_above**	**replace_panel**
del_panel	**panel_below**	**set_panel_userptr**
hide_panel	**panel_hidden**	**show_panel**
move_panel	**panel_userptr**	**top_panel**
new_panel	**panel_window**	**update_panels**

SUMMARY OF COMMANDS AND UTILITIES

The following commands and utilities are supported by a SVID-compliant Terminal Interface Extension.

captoinfo	**clear**	**infocmp**	**tic**	**tput**

ORGANIZATION OF TECHNICAL INFORMATION

The ‘‘Terminal Interface Environment’’ chapter provides manual page descriptions of the terminal capability database used by this extension to support device independent terminal I/O.

The ‘‘Terminal Interface Library Routines’’ chapter provides manual page descriptions of routine interfaces supported by this extension.

Terminal Interface Environment

Terminal Interface Environment Variables

The components of the TI extension use the environment variables described below. [See sh(BU_CMD) for information on the shell environment.]

TERM

The environmental variable `TERM`, by convention, contains a user's current terminal type and may be set by the user.

TERMINFO

The environmental variable `TERMINFO`, if set, contains the place where local terminal descriptions can be found. `TERMINFO` can be set by the user. If it is set, any program using `curses` checks the `TERMINFO` location for the description of a terminal before checking `/usr/lib/terminfo`, the standard location for terminal descriptions. [See curses(TI_LIB) and terminfo(TI_ENV) for further information.]

LINES and COLUMNS

The environmental variables `LINES` and `COLUMNS`, if set, contain the number of lines and the number of columns, respectively, on a terminal screen and can be set by the user. If defined, the values of these variables, `LINES` and `COLUMNS`, override the screen size values given in the `terminfo` description of a terminal. [See curses(TI_LIB) and terminfo(TI_ENV) for further information.]

FINAL COPY
June 15, 1995
File: ti_env.txt
svid

MANUAL PAGES

FINAL COPY
June 15, 1995
File: ti_env.txt
svid

Terminal Interface Environment Routines

The following section contains the manual pages for the TI_ENV routines.

NAME

CURSES – CRT screen handling and optimization package

SYNOPSIS

```
#include <curses.h>
```

DESCRIPTION

CURSES library routines give the user a terminal-independent method of updating character screens with reasonable optimization. A program using these routines must be compiled with the `–lcurses` option of `cc`.

The CURSES package allows: overall screen, window and pad manipulation; output to windows and pads; reading terminal input; control over terminal and CURSES input and output options; environment query routines; color manipulation; use of soft label keys; `terminfo` access; and access to low-level CURSES routines.

To initialize the routines, the routine `initscr()` or `newterm()` must be called before any of the other routines that deal with windows and screens are used. The routine `endwin()` must be called before exiting. To get character-at-a-time input without echoing (most interactive, screen-oriented programs want this), the following sequence should be used:

```
initscr(),cbreak(),noecho();
```

Most programs would additionally use the sequence:

```
nonl(),intrflush(stdscr,FALSE),keypad(stdscr,TRUE);
```

Before a CURSES program is run, the tab stops of the terminal should be set and its initialization strings, if defined, must be output. This can be done by executing the `tput init` command after the shell environment variable `TERM` has been exported. [See terminfo(TI_ENV) for further details.]

The CURSES library permits manipulation of data structures, called *windows*, which can be thought of as two-dimensional arrays of characters. A default window called `stdscr`, which is the size of the terminal screen, is supplied. Others may be created with `newwin()`.

Windows are referred to by variables declared as `WINDOW *`. These data structures are manipulated with routines described on TI_LIB pages (whose names begin ''curs_''). Among the most basic routines are `move()` and `addch()`. More general versions of these routines are included that allow the user to specify a window.

After using routines to manipulate a window, `refresh()` is called, telling CURSES to make the user's CRT screen look like `stdscr`. The characters in a window are actually of type `chtype` (character and attribute data) so that other information about the character may also be stored with each character.

Special windows called *pads* may also be manipulated. These are windows that are not necessarily associated with a viewable part of the screen. See curs_pad(TI_LIB) for more information.

In addition to drawing characters on the screen, video attributes and colors may be included, causing the characters to show up in such modes as underlined, reverse video or color on terminals that support such display enhancements. Line drawing characters may be specified to be output. On input, CURSES is also able to translate arrow and function keys that transmit escape sequences into single values.

The video attributes, line drawing characters and input values use names, defined in `<curses.h>`, such as `A_REVERSE`, `ACS_HLINE` and `KEY_LEFT`.

If the environment variables `LINES` and `COLUMNS` are set, or if the program is executing in a window environment, line and column information in the environment will override information read by `terminfo`. This would affect a program running in an AT&T 630 layer, for example, where the size of a screen is changeable.

If the environment variable `TERMINFO` is defined, any program using CURSES checks for a local terminal definition before checking in the standard place. For example, if `TERM` is set to `att4424`, then the compiled terminal definition is found in

    /usr/share/lib/terminfo/a/att4424.

(The `a` is copied from the first letter of `att4424` to avoid creation of huge directories.) However, if `TERMINFO` is set to `$HOME/myterms`, CURSES first checks

    $HOME/myterms/a/att4424,

and if that fails, it then checks

    /usr/share/lib/terminfo/a/att4424.

This is useful for developing experimental definitions or when write permission in `/usr/share/lib/terminfo` is not available.

The integer variables `LINES` and `COLS` are defined in `<curses.h>` and will be filled in by `initscr()` with the size of the screen. The constants `TRUE` and `FALSE` have the values `1` and `0`, respectively.

The CURSES routines also define the `WINDOW *` variable `curscr` which is used for certain low-level operations like clearing and redrawing a screen containing garbage. `curscr` can be used in only a few routines.

International Functions

The number of bytes and the number of columns to hold a character from the supplementary character set is locale-specific (locale category `LC_CTYPE`) and can be specified in the character class table.

For editing, operating at the character level is entirely appropriate. For screen formatting, arbitrary movement of characters on screen is not desirable.

Overwriting characters (`addch()`, for example) operates on a screen level. Overwriting a character by a character that requires a different number of columns may produce *orphaned columns*. These orphaned columns are filled with background characters.

Inserting characters (`insch()`, for example) operates on a character level (that is, at the character boundaries). The specified character is inserted right before the character, regardless of which column of a character the cursor points to. Before insertion, the cursor position is adjusted to the first column of the character.

As with inserting characters, deleting characters (`delch()`, for example) operates on a character level (that is, at the character boundaries). The character at the cursor is deleted whichever column of the character the cursor points to. Before deletion, the cursor position is adjusted to the first column of the character.

A *multi-column* character cannot be put on the last column of a line. When such attempts are made, the last column is set to the background character. In addition, when such an operation creates orphaned columns, the orphaned columns are filled with background characters.

Overlapping and overwriting a window follows the operation of overwriting characters around its edge. The orphaned columns, if any, are handled as in the character operations.

The cursor is allowed to be placed anywhere in a window. If the insertion or deletion is made when the cursor points to the second or later column position of a character that holds multiple columns, the cursor is adjusted to the first column of the character before the insertion or deletion.

Routine and Argument Names

Many CURSES routines have two or more versions. Routines prefixed with `p` require a pad argument. Routines whose names contain a `w` generally require either a window argument or a wide-character argument. If `w` appears twice in a routine name, the routine usually requires both a window and a wide-character argument. Routines that do not require a pad or window argument generally use `stdscr`.

The routines prefixed with `mv` require an *x* and *y* coordinate to move to before performing the appropriate action. The `mv` routines imply a call to `move()` before the call to the other routine. The coordinate *y* always refers to the row (of the window), and *x* always refers to the column. The upper left-hand corner is always (0,0), not (1,1).

The routines prefixed with `mvw` take both a window argument and *x* and *y* coordinates. The window argument is always specified before the coordinates.

In each case, *win* is the window affected, and *pad* is the pad affected; *win* and *pad* are always pointers to type `WINDOW`.

Option setting routines require a Boolean flag *bf* with the value `TRUE` or `FALSE`; *bf* is always of type `bool`. The variables *ch* and *attrs* are always of type `chtype`. The types `WINDOW`, `SCREEN`, `bool` and `chtype` are defined in `<curses.h>`. The type `TERMINAL` is defined in `<term.h>`. All other arguments are integers.

Routine Name Index

The following table lists each CURSES routine and the name of the manual page on which it is described.

CURSES Routine Name	Manual Page Name
`addch()`	curs_addch(TI_LIB)
`addchnstr()`	curs_addchstr(TI_LIB)
`addchstr()`	curs_addchstr(TI_LIB)
`addnstr()`	curs_addstr(TI_LIB)
`addnwstr()`	curs_addwstr(TI_LIB)
`addstr()`	curs_addstr(TI_LIB)
`addwch()`	curs_addwch(TI_LIB)
`addwchnstr()`	curs_addwchstr(TI_LIB)

CURSES Routine Name	Manual Page Name
addwchstr()	curs_addwchstr(TI_LIB)
addwstr()	curs_addwstr(TI_LIB)
attroff()	curs_attr(TI_LIB)
attron()	curs_attr(TI_LIB)
attrset()	curs_attr(TI_LIB)
baudrate()	curs_termattrs(TI_LIB)
beep()	curs_beep(TI_LIB)
bkgd()	curs_bkgd(TI_LIB)
bkgdset()	curs_bkgd(TI_LIB)
border()	curs_border(TI_LIB)
box()	curs_border(TI_LIB)
can_change_color()	curs_color(TI_LIB)
cbreak()	curs_inopts(TI_LIB)
clear()	curs_clear(TI_LIB)
clearok()	curs_outopts(TI_LIB)
clrtobot()	curs_clear(TI_LIB)
clrtoeol()	curs_clear(TI_LIB)
color_content()	curs_color(TI_LIB)
copywin()	curs_overlay(TI_LIB)
curs_set()	curs_kernel(TI_LIB)
def_prog_mode()	curs_kernel(TI_LIB)
def_shell_mode()	curs_kernel(TI_LIB)
del_curterm()	curs_terminfo(TI_LIB)
delay_output()	curs_util(TI_LIB)
delch()	curs_delch(TI_LIB)
deleteln()	curs_deleteln(TI_LIB)
delscreen()	curs_initscr(TI_LIB)
delwin()	curs_window(TI_LIB)
derwin()	curs_window(TI_LIB)
doupdate()	curs_refresh(TI_LIB)
dupwin()	curs_window(TI_LIB)
echo()	curs_inopts(TI_LIB)
echochar()	curs_addch(TI_LIB)
echowchar()	curs_addwch(TI_LIB)
endwin()	curs_initscr(TI_LIB)
erase()	curs_clear(TI_LIB)
erasechar()	curs_termattrs(TI_LIB)
filter()	curs_util(TI_LIB)
flash()	curs_beep(TI_LIB)
flushinp()	curs_util(TI_LIB)
getbegyx()	curs_getyx(TI_LIB)
getch()	curs_getch(TI_LIB)
getmaxyx()	curs_getyx(TI_LIB)
getnwstr()	curs_getwstr(TI_LIB)
getparyx()	curs_getyx(TI_LIB)

CURSES Routine Name	Manual Page Name
getstr()	curs_getstr(TI_LIB)
getsyx()	curs_kernel(TI_LIB)
getwch()	curs_getwch(TI_LIB)
getwin()	curs_util(TI_LIB)
getwstr()	curs_getwstr(TI_LIB)
getyx()	curs_getyx(TI_LIB)
halfdelay()	curs_inopts(TI_LIB)
has_colors()	curs_color(TI_LIB)
has_ic()	curs_termattrs(TI_LIB)
has_il()	curs_termattrs(TI_LIB)
idcok()	curs_outopts(TI_LIB)
idlok()	curs_outopts(TI_LIB)
immedok()	curs_outopts(TI_LIB)
inch()	curs_inch(TI_LIB)
inchnstr()	curs_inchstr(TI_LIB)
inchstr()	curs_inchstr(TI_LIB)
init_color()	curs_color(TI_LIB)
init_pair()	curs_color(TI_LIB)
initscr()	curs_initscr(TI_LIB)
innstr()	curs_instr(TI_LIB)
innwstr()	curs_inwstr(TI_LIB)
insch()	curs_insch(TI_LIB)
insdelln()	curs_deleteln(TI_LIB)
insertln()	curs_deleteln(TI_LIB)
insnstr()	curs_insstr(TI_LIB)
insnwstr()	curs_inswstr(TI_LIB)
insstr()	curs_insstr(TI_LIB)
instr()	curs_instr(TI_LIB)
inswch()	curs_inswch(TI_LIB)
inswstr()	curs_inswstr(TI_LIB)
intrflush()	curs_inopts(TI_LIB)
inwch()	curs_inwch(TI_LIB)
inwchnstr()	curs_inwchstr(TI_LIB)
inwchstr()	curs_inwchstr(TI_LIB)
inwstr()	curs_inwstr(TI_LIB)
is_linetouched()	curs_touch(TI_LIB)
is_wintouched()	curs_touch(TI_LIB)
isendwin()	curs_initscr(TI_LIB)
keyname()	curs_util(TI_LIB)
keypad()	curs_inopts(TI_LIB)
killchar()	curs_termattrs(TI_LIB)
leaveok()	curs_outopts(TI_LIB)
longname()	curs_termattrs(TI_LIB)
meta()	curs_inopts(TI_LIB)
move()	curs_move(TI_LIB)

CURSES Routine Name	Manual Page Name
mvaddch()	curs_addch(TI_LIB)
mvaddchnstr()	curs_addchstr(TI_LIB)
mvaddchstr()	curs_addchstr(TI_LIB)
mvaddnstr()	curs_addstr(TI_LIB)
mvaddnwstr()	curs_addwstr(TI_LIB)
mvaddstr()	curs_addstr(TI_LIB)
mvaddwch()	curs_addwch(TI_LIB)
mvaddwchnstr()	curs_addwchstr(TI_LIB)
mvaddwchstr()	curs_addwchstr(TI_LIB)
mvaddwstr()	curs_addwstr(TI_LIB)
mvcur()	curs_terminfo(TI_LIB)
mvdelch()	curs_delch(TI_LIB)
mvderwin()	curs_window(TI_LIB)
mvgetch()	curs_getch(TI_LIB)
mvgetnwstr()	curs_getwstr(TI_LIB)
mvgetstr()	curs_getstr(TI_LIB)
mvgetwch()	curs_getwch(TI_LIB)
mvgetwstr()	curs_getwstr(TI_LIB)
mvinch()	curs_inch(TI_LIB)
mvinchnstr()	curs_inchstr(TI_LIB)
mvinchstr()	curs_inchstr(TI_LIB)
mvinnstr()	curs_instr(TI_LIB)
mvinnwstr()	curs_inwstr(TI_LIB)
mvinsch()	curs_insch(TI_LIB)
mvinsnstr()	curs_insstr(TI_LIB)
mvinsnwstr()	curs_inswstr(TI_LIB)
mvinsstr()	curs_insstr(TI_LIB)
mvinstr()	curs_instr(TI_LIB)
mvinswch()	curs_inswch(TI_LIB)
mvinswstr()	curs_inswstr(TI_LIB)
mvinwch()	curs_inwch(TI_LIB)
mvinwchnstr()	curs_inwchstr(TI_LIB)
mvinwchstr()	curs_inwchstr(TI_LIB)
mvinwstr()	curs_inwstr(TI_LIB)
mvprintw()	curs_printw(TI_LIB)
mvscanw()	curs_scanw(TI_LIB)
mvwaddch()	curs_addch(TI_LIB)
mvwaddchnstr()	curs_addchstr(TI_LIB)
mvwaddchstr()	curs_addchstr(TI_LIB)
mvwaddnstr()	curs_addstr(TI_LIB)
mvwaddnwstr()	curs_addwstr(TI_LIB)
mvwaddstr()	curs_addstr(TI_LIB)
mvwaddwch()	curs_addwch(TI_LIB)
mvwaddwchnstr()	curs_addwchstr(TI_LIB)
mvwaddwchstr()	curs_addwchstr(TI_LIB)

CURSES Routine Name	Manual Page Name
mvwaddwstr()	curs_addwstr(TI_LIB)
mvwdelch()	curs_delch(TI_LIB)
mvwgetch()	curs_getch(TI_LIB)
mvwgetnwstr()	curs_getwstr(TI_LIB)
mvwgetstr()	curs_getstr(TI_LIB)
mvwgetwch()	curs_getwch(TI_LIB)
mvwgetwstr()	curs_getwstr(TI_LIB)
mvwin()	curs_window(TI_LIB)
mvwinch()	curs_inch(TI_LIB)
mvwinchnstr()	curs_inchstr(TI_LIB)
mvwinchstr()	curs_inchstr(TI_LIB)
mvwinnstr()	curs_instr(TI_LIB)
mvwinnwstr()	curs_inwstr(TI_LIB)
mvwinsch()	curs_insch(TI_LIB)
mvwinsnstr()	curs_insstr(TI_LIB)
mvwinsnwstr	curs_inswstr(TI_LIB)
mvwinsstr()	curs_insstr(TI_LIB)
mvwinstr()	curs_instr(TI_LIB)
mvwinswch()	curs_inswch(TI_LIB)
mvwinswstr()	curs_inswstr(TI_LIB)
mvwinwch()	curs_inwch(TI_LIB)
mvwinwchnstr()	curs_inwchstr(TI_LIB)
mvwinwchstr()	curs_inwchstr(TI_LIB)
mvwinwstr()	curs_inwstr(TI_LIB)
mvwprintw()	curs_printw(TI_LIB)
mvwscanw()	curs_scanw(TI_LIB)
napms()	curs_kernel(TI_LIB)
newpad()	curs_pad(TI_LIB)
newterm()	curs_initscr(TI_LIB)
newwin()	curs_window(TI_LIB)
nl()	curs_outopts(TI_LIB)
nocbreak()	curs_inopts(TI_LIB)
nodelay()	curs_inopts(TI_LIB)
noecho()	curs_inopts(TI_LIB)
nonl()	curs_outopts(TI_LIB)
noqiflush()	curs_inopts(TI_LIB)
noraw()	curs_inopts(TI_LIB)
notimeout()	curs_inopts(TI_LIB)
overlay()	curs_overlay(TI_LIB)
overwrite()	curs_overlay(TI_LIB)
pair_content()	curs_color(TI_LIB)
pechochar()	curs_pad(TI_LIB)
pechowchar()	curs_pad(TI_LIB)
pnoutrefresh()	curs_pad(TI_LIB)
prefresh()	curs_pad(TI_LIB)

CURSES Routine Name	Manual Page Name
printw()	curs_printw(TI_LIB)
putp()	curs_terminfo(TI_LIB)
putwin()	curs_util(TI_LIB)
qiflush()	curs_inopts(TI_LIB)
raw()	curs_inopts(TI_LIB)
redrawwin()	curs_refresh(TI_LIB)
refresh()	curs_refresh(TI_LIB)
reset_prog_mode()	curs_kernel(TI_LIB)
reset_shell_mode()	curs_kernel(TI_LIB)
resetty()	curs_kernel(TI_LIB)
restartterm()	curs_terminfo(TI_LIB)
ripoffline()	curs_kernel(TI_LIB)
savetty()	curs_kernel(TI_LIB)
scanw()	curs_scanw(TI_LIB)
scr_dump()	curs_scr_dump(TI_LIB)
scr_init()	curs_scr_dump(TI_LIB)
scr_restore()	curs_scr_dump(TI_LIB)
scr_set()	curs_scr_dump(TI_LIB)
scroll()	curs_scroll(TI_LIB)
scrollok()	curs_outopts(TI_LIB)
set_curterm()	curs_terminfo(TI_LIB)
set_term()	curs_initscr(TI_LIB)
setscrreg()	curs_outopts(TI_LIB)
setsyx()	curs_kernel(TI_LIB)
setterm()	curs_terminfo(TI_LIB)
setupterm()	curs_terminfo(TI_LIB)
slk_attroff()	curs_slk(TI_LIB)
slk_attron()	curs_slk(TI_LIB)
slk_attrset()	curs_slk(TI_LIB)
slk_clear()	curs_slk(TI_LIB)
slk_init()	curs_slk(TI_LIB)
slk_label()	curs_slk(TI_LIB)
slk_noutrefresh()	curs_slk(TI_LIB)
slk_refresh()	curs_slk(TI_LIB)
slk_restore()	curs_slk(TI_LIB)
slk_set()	curs_slk(TI_LIB)
slk_touch()	curs_slk(TI_LIB)
srcl()	curs_scroll(TI_LIB)
standend()	curs_attr(TI_LIB)
standout()	curs_attr(TI_LIB)
start_color()	curs_color(TI_LIB)
subpad()	curs_pad(TI_LIB)
subwin()	curs_window(TI_LIB)
syncok()	curs_window(TI_LIB)
termattrs()	curs_termattrs(TI_LIB)

CURSES Routine Name	Manual Page Name
termname()	curs_termattrs(TI_LIB)
tgetent()	curs_termcap(TI_LIB)
tgetflag()	curs_termcap(TI_LIB)
tgetnum()	curs_termcap(TI_LIB)
tgetstr()	curs_termcap(TI_LIB)
tgoto()	curs_termcap(TI_LIB)
tigetflag()	curs_terminfo(TI_LIB)
tigetnum()	curs_terminfo(TI_LIB)
tigetstr()	curs_terminfo(TI_LIB)
timeout()	curs_inopts(TI_LIB)
touchline()	curs_touch(TI_LIB)
touchwin()	curs_touch(TI_LIB)
tparm()	curs_terminfo(TI_LIB)
tputs()	curs_termcap(TI_LIB)
tputs()	curs_terminfo(TI_LIB)
typeahead()	curs_inopts(TI_LIB)
unctrl()	curs_util(TI_LIB)
ungetch()	curs_getch(TI_LIB)
ungetwch()	curs_getwch(TI_LIB)
untouchwin()	curs_touch(TI_LIB)
use_env()	curs_util(TI_LIB)
vidattr()	curs_terminfo(TI_LIB)
vidputs()	curs_terminfo(TI_LIB)
vwprintw()	curs_printw(TI_LIB)
vwscanw()	curs_scanw(TI_LIB)
waddch()	curs_addch(TI_LIB)
waddchnstr()	curs_addchstr(TI_LIB)
waddchstr()	curs_addchstr(TI_LIB)
waddnstr()	curs_addstr(TI_LIB)
waddnwstr()	curs_addwstr(TI_LIB)
waddstr()	curs_addstr(TI_LIB)
waddwch()	curs_addwch(TI_LIB)
waddwchnstr()	curs_addwchstr(TI_LIB)
waddwchstr()	curs_addwchstr(TI_LIB)
waddwstr()	curs_addwstr(TI_LIB)
wattroff()	curs_attr(TI_LIB)
wattron()	curs_attr(TI_LIB)
wattrset()	curs_attr(TI_LIB)
wbkgd()	curs_bkgd(TI_LIB)
wbkgdset()	curs_bkgd(TI_LIB)
wborder()	curs_border(TI_LIB)
wclear()	curs_clear(TI_LIB)
wclrtobot()	curs_clear(TI_LIB)
wclrtoeol()	curs_clear(TI_LIB)
wcursyncup()	curs_window(TI_LIB)

CURSES Routine Name	Manual Page Name
`wdelch()`	curs_delch(TI_LIB)
`wdeleteln()`	curs_deleteln(TI_LIB)
`wechochar()`	curs_addch(TI_LIB)
`wechowchar()`	curs_addwch(TI_LIB)
`werase()`	curs_clear(TI_LIB)
`wgetch()`	curs_getch(TI_LIB)
`wgetnstr()`	curs_getstr(TI_LIB)
`wgetnwstr()`	curs_getwstr(TI_LIB)
`wgetstr()`	curs_getstr(TI_LIB)
`wgetwch()`	curs_getwch(TI_LIB)
`wgetwstr()`	curs_getwstr(TI_LIB)
`whline()`	curs_border(TI_LIB)
`winch()`	curs_inch(TI_LIB)
`winchnstr()`	curs_inchstr(TI_LIB)
`winchstr()`	curs_inchstr(TI_LIB)
`winnstr()`	curs_instr(TI_LIB)
`winnwstr()`	curs_inwstr(TI_LIB)
`winsch()`	curs_insch(TI_LIB)
`winsdelln()`	curs_deleteln(TI_LIB)
`winsertln()`	curs_deleteln(TI_LIB)
`winsnstr()`	curs_insstr(TI_LIB)
`winsnwstr()`	curs_inswstr(TI_LIB)
`winsstr()`	curs_insstr(TI_LIB)
`winstr()`	curs_instr(TI_LIB)
`winswch()`	curs_inswch(TI_LIB)
`winswstr()`	curs_inswstr(TI_LIB)
`winwch()`	curs_inwch(TI_LIB)
`winwchnstr()`	curs_inwchstr(TI_LIB)
`winwchstr()`	curs_inwchstr(TI_LIB)
`winwstr()`	curs_inwstr(TI_LIB)
`wmove()`	curs_move(TI_LIB)
`wnoutrefresh()`	curs_refresh(TI_LIB)
`wprintw()`	curs_printw(TI_LIB)
`wredrawln()`	curs_refresh(TI_LIB)
`wrefresh()`	curs_refresh(TI_LIB)
`wscanw()`	curs_scanw(TI_LIB)
`wscrl()`	curs_scroll(TI_LIB)
`wsetscrreg()`	curs_outopts(TI_LIB)
`wstandend()`	curs_attr(TI_LIB)
`wstandout()`	curs_attr(TI_LIB)
`wsyncdown()`	curs_window(TI_LIB)
`wsyncup()`	curs_window(TI_LIB)
`wtimeout()`	curs_inopts(TI_LIB)
`wtouchln()`	curs_touch(TI_LIB)
`wvline()`	curs_border(TI_LIB)

RETURN VALUE

Routines that return an integer return `ERR` upon failure and an integer value other than `ERR` upon successful completion, unless otherwise noted in the routine descriptions.

All macros return the value of the window version, except `setscrreg()`, `wsetscrreg()`, `getyx()`, `getbegyx()` and `getmaxyx()`. The return values of `setscrreg()`, `wsetscrreg()`, `getyx()`, `getbegyx()` and `getmaxyx()` are undefined (i.e., these should not be used as the right-hand side of assignment statements).

Routines that return pointers return `NULL` on error.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

SEE ALSO

TERMINFO(TI_ENV) and TI_LIB pages whose names begin ''curs_'' for detailed routine descriptions.

LEVEL

Level 1.

NAME

FORMS – character based forms package

SYNOPSIS

#include <form.h>

DESCRIPTION

The `form` library is built using the `curses` library, and any program using FORMS routines must call one of the CURSES initialization routines such as `initscr()`. A program using these routines must be compiled with `–lform` and `–lcurses` on the `cc` command line.

The FORMS package gives the applications programmer a terminal-independent method of creating and customizing forms for user-interaction. The FORMS package includes: field routines, which are used to create and customize fields, link fields and assign field types; fieldtype routines, which are used to create new field types for validating fields; and form routines, which are used to create and customize forms, assign pre/post processing functions, and display and interact with forms.

Current Default Values for Field Attributes

The FORMS package establishes initial current default values for field attributes. During field initialization, each field attribute is assigned the current default value for that attribute. An application can change or retrieve a current default attribute value by calling the appropriate set or retrieve routine with a `NULL` field pointer. If an application changes a current default field attribute value, subsequent fields created using `new_field()` will have the new default attribute value. (The attributes of previously created fields are not changed if a current default attribute value is changed.)

Routine Name Index

The following table lists each FORMS routine and the name of the manual page on which it is described.

FORMS Routine Name	Manual Page Name
`current_field()`	form_page(TI_LIB)
`data_ahead()`	form_data(TI_LIB)
`data_behind()`	form_data(TI_LIB)
`dup_field()`	form_field_new(TI_LIB)
`dynamic_field_info()`	form_field_info(TI_LIB)
`field_arg()`	form_field_validation(TI_LIB)
`field_back()`	form_field_attributes(TI_LIB)
`field_buffer()`	form_field_buffer(TI_LIB)
`field_count()`	form_field(TI_LIB)
`field_fore()`	form_field_attributes(TI_LIB)
`field_index()`	form_page(TI_LIB)
`field_info()`	form_field_info(TI_LIB)
`field_init()`	form_hook(TI_LIB)
`field_just()`	form_field_just(TI_LIB)

FORMS Routine Name	Manual Page Name
field_opts()	form_field_opts(TI_LIB)
field_opts_off()	form_field_opts(TI_LIB)
field_opts_on()	form_field_opts(TI_LIB)
field_pad()	form_field_attributes(TI_LIB)
field_status()	form_field_buffer(TI_LIB)
field_term()	form_hook(TI_LIB)
field_type()	form_field_validation(TI_LIB)
field_userptr()	form_field_userptr(TI_LIB)
form_driver()	form_driver(TI_LIB)
form_fields()	form_field(TI_LIB)
form_init()	form_hook(TI_LIB)
form_opts()	form_opts(TI_LIB)
form_opts_off()	form_opts(TI_LIB)
form_opts_on()	form_opts(TI_LIB)
form_page()	form_page(TI_LIB)
form_sub()	form_win(TI_LIB)
form_term()	form_hook(TI_LIB)
form_userptr()	form_userptr(TI_LIB)
form_win()	form_win(TI_LIB)
free_field()	form_field_new(TI_LIB)
free_fieldtype()	form_fieldtype(TI_LIB)
free_form()	form_new(TI_LIB)
link_field()	form_field_new(TI_LIB)
link_fieldtype()	form_fieldtype(TI_LIB)
move_field()	form_field(TI_LIB)
new_field()	form_field_new(TI_LIB)
new_fieldtype()	form_fieldtype(TI_LIB)
new_form()	form_new(TI_LIB)
new_page()	form_new_page(TI_LIB)
pos_form_cursor()	form_cursor(TI_LIB)
post_form()	form_post(TI_LIB)
scale_form()	form_win(TI_LIB)
set_current_field()	form_page(TI_LIB)
set_field_back()	form_field_attributes(TI_LIB)
set_field_buffer()	form_field_buffer(TI_LIB)
set_field_fore()	form_field_attributes(TI_LIB)
set_field_init()	form_hook(TI_LIB)
set_field_just()	form_field_just(TI_LIB)
set_field_opts()	form_field_opts(TI_LIB)
set_field_pad()	form_field_attributes(TI_LIB)
set_field_status()	form_field_buffer(TI_LIB)
set_field_term()	form_hook(TI_LIB)
set_field_type()	form_field_validation(TI_LIB)
set_field_userptr()	form_field_userptr(TI_LIB)
set_fieldtype_arg()	form_fieldtype(TI_LIB)

FORMS Routine Name	Manual Page Name
`set_fieldtype_choice()`	form_fieldtype(TI_LIB)
`set_form_fields()`	form_field(TI_LIB)
`set_form_init()`	form_hook(TI_LIB)
`set_form_opts()`	form_opts(TI_LIB)
`set_form_page()`	form_page(TI_LIB)
`set_form_sub()`	form_win(TI_LIB)
`set_form_term()`	form_hook(TI_LIB)
`set_form_userptr()`	form_userptr(TI_LIB)
`set_form_win()`	form_win(TI_LIB)
`set_max_field()`	form_field_buffer(TI_LIB)
`set_new_page()`	form_new_page(TI_LIB)
`unpost_form()`	form_post(TI_LIB)

RETURN VALUE

Routines that return a pointer always return `NULL` on error. Routines that return an integer return one of the following:

`E_OK`	–	The function returned successfully.
`E_CONNECTED`	–	The field is already connected to a form.
`E_SYSTEM_ERROR`	–	System error.
`E_BAD_ARGUMENT`	–	An argument is incorrect.
`E_CURRENT`	–	The field is the current field.
`E_POSTED`	–	The form is posted.
`E_NOT_POSTED`	–	The form is not posted.
`E_INVALID_FIELD`	–	The field contents are invalid.
`E_NOT_CONNECTED`	–	The field is not connected to a form.
`E_NO_ROOM`	–	The form does not fit in the subwindow.
`E_BAD_STATE`	–	The routine was called from an initialization or termination function.
`E_REQUEST_DENIED`	–	The form driver request failed.
`E_UNKNOWN_COMMAND`	–	An unknown request was passed to the the form driver.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), and TI_LIB pages whose names begin "form_" for detailed routine descriptions.

LEVEL

Level 1.

NAME

MENUS – character based menus package

SYNOPSIS

```
#include &lt;menu.h&gt;
```

DESCRIPTION

The `menu` library is built using the `curses` library, and any program using MENUS routines must call one of the CURSES initialization routines, such as `initscr()`. A program using these routines must be compiled with `–lmenu` and `–lcurses` on the `cc` command line.

The MENUS package gives the applications programmer a terminal-independent method of creating and customizing menus for user interaction. The MENUS package includes: item routines, which are used to create and customize menu items; and menu routines, which are used to create and customize menus, assign pre- and post-processing routines, and display and interact with menus.

Current Default Values for Item Attributes

The MENUS package establishes initial current default values for item attributes. During item initialization, each item attribute is assigned the current default value for that attribute. An application can change or retrieve a current default attribute value by calling the appropriate set or retrieve routine with a `NULL` item pointer. If an application changes a current default item attribute value, subsequent items created using `new_item()` will have the new default attribute value. (The attributes of previously created items are not changed if a current default attribute value is changed.)

Routine Name Index

The following table lists each MENUS routine and the name of the manual page on which it is described.

MENUS Routine Name	Manual Page Name
`current_item()`	menu_item_current(TI_LIB)
`free_item()`	menu_item_new(TI_LIB)
`free_menu()`	menu_new(TI_LIB)
`item_count()`	menu_items(TI_LIB)
`item_description()`	menu_item_name(TI_LIB)
`item_index()`	menu_item_current(TI_LIB)
`item_init()`	menu_hook(TI_LIB)
`item_name()`	menu_item_name(TI_LIB)
`item_opts()`	menu_item_opts(TI_LIB)
`item_opts_off()`	menu_item_opts(TI_LIB)
`item_opts_on()`	menu_item_opts(TI_LIB)
`item_term()`	menu_hook(TI_LIB)
`item_userptr()`	menu_item_userptr(TI_LIB)
`item_value()`	menu_item_value(TI_LIB)
`item_visible()`	menu_item_visible(TI_LIB)
`menu_back()`	menu_attributes(TI_LIB)

MENUS Routine Name	Manual Page Name
menu_driver()	menu_driver(TI_LIB)
menu_fore()	menu_attributes(TI_LIB)
menu_format()	menu_format(TI_LIB)
menu_grey()	menu_attributes(TI_LIB)
menu_init()	menu_hook(TI_LIB)
menu_items()	menu_items(TI_LIB)
menu_mark()	menu_mark(TI_LIB)
menu_opts()	menu_opts(TI_LIB)
menu_opts_off()	menu_opts(TI_LIB)
menu_opts_on()	menu_opts(TI_LIB)
menu_pad()	menu_attributes(TI_LIB)
menu_pattern()	menu_pattern(TI_LIB)
menu_sub()	menu_win(TI_LIB)
menu_term()	menu_hook(TI_LIB)
menu_userptr()	menu_userptr(TI_LIB)
menu_win()	menu_win(TI_LIB)
new_item()	menu_item_new(TI_LIB)
new_menu()	menu_new(TI_LIB)
pos_menu_cursor()	menu_cursor(TI_LIB)
post_menu()	menu_post(TI_LIB)
scale_menu()	menu_win(TI_LIB)
set_current_item()	menu_item_current(TI_LIB)
set_item_init()	menu_hook(TI_LIB)
set_item_opts()	menu_item_opts(TI_LIB)
set_item_term()	menu_hook(TI_LIB)
set_item_userptr()	menu_item_userptr(TI_LIB)
set_item_value()	menu_item_value(TI_LIB)
set_menu_back()	menu_attributes(TI_LIB)
set_menu_fore()	menu_attributes(TI_LIB)
set_menu_format()	menu_format(TI_LIB)
set_menu_grey()	menu_attributes(TI_LIB)
set_menu_init()	menu_hook(TI_LIB)
set_menu_items()	menu_items(TI_LIB)
set_menu_mark()	menu_mark(TI_LIB)
set_menu_opts()	menu_opts(TI_LIB)
set_menu_pad()	menu_attributes(TI_LIB)
set_menu_pattern()	menu_pattern(TI_LIB)
set_menu_sub()	menu_win(TI_LIB)
set_menu_term()	menu_hook(TI_LIB)
set_menu_userptr()	menu_userptr(TI_LIB)
set_menu_win()	menu_win(TI_LIB)
set_top_row()	menu_item_current(TI_LIB)
top_row()	menu_item_current(TI_LIB)
unpost_menu()	menu_post(TI_LIB)

RETURN VALUE

Routines that return pointers always return `NULL` on error. Routines that return an integer return one of the following:

`E_OK`	– The routine returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An incorrect argument was passed to the routine.
`E_POSTED`	– The menu is already posted.
`E_CONNECTED`	– One or more items are already connected to another menu.
`E_BAD_STATE`	– The routine was called from an initialization or termination function.
`E_NO_ROOM`	– The menu does not fit within its subwindow.
`E_NOT_POSTED`	– The menu has not been posted.
`E_UNKNOWN_COMMAND`	– An unknown request was passed to the menu driver.
`E_NO_MATCH`	– The character failed to match.
`E_NOT_SELECTABLE`	– The item cannot be selected.
`E_NOT_CONNECTED`	– No items are connected to the menu.
`E_REQUEST_DENIED`	– The menu driver could not process the request.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), and TI_LIB pages whose names begin "menu_" for detailed routine descriptions.

LEVEL

Level 1.

NAME

PANELS – character based panels package

SYNOPSIS

```
#include <panel.h>
```

DESCRIPTION

The `panel` library is built using the `curses` library, and any program using PANELS routines must call one of the CURSES initialization routines such as `initscr()`. A program using these routines must be compiled with `–lpanel` and `–lcurses` on the `cc` command line.

The PANELS package gives the applications programmer a way to have depth relationships between CURSES windows; a CURSES window is associated with every panel. The PANELS routines allow CURSES windows to overlap without making visible the overlapped portions of underlying windows. The initial CURSES window, `stdscr`, lies beneath all panels. The set of currently visible panels is the *deck* of panels.

The PANELS package allows the applications programmer to create panels, fetch and set their associated windows, shuffle panels in the deck, and manipulate panels in other ways.

Routine Name Index

The following table lists each PANELS routine and the name of the manual page on which it is described.

PANELS Routine Name	Manual Page Name
`bottom_panel()`	panel_top(TI_LIB)
`del_panel()`	panel_new(TI_LIB)
`hide_panel()`	panel_show(TI_LIB)
`move_panel()`	panel_move(TI_LIB)
`new_panel()`	panel_new(TI_LIB)
`panel_above()`	panel_above(TI_LIB)
`panel_below()`	panel_above(TI_LIB)
`panel_hidden()`	panel_show(TI_LIB)
`panel_userptr()`	panel_userptr(TI_LIB)
`panel_window()`	panel_window(TI_LIB)
`replace_panel()`	panel_window(TI_LIB)
`set_panel_userptr()`	panel_userptr(TI_LIB)
`show_panel()`	panel_show(TI_LIB)
`top_panel()`	panel_top(TI_LIB)
`update_panels()`	panel_update(TI_LIB)

RETURN VALUE

Each PANELS routine that returns a pointer to an object returns `NULL` if an error occurs. Each panel routine that returns an integer, returns `OK` if it executes successfully and `ERR` if it does not.

USAGE

Application Program.

The header file `<panel.h>` automatically includes the header file `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), and TI_LIB pages whose names begin "panel_," for detailed routine descriptions.

LEVEL

Level 1.

NAME

terminfo – terminal capability data base

SYNOPSIS

`/usr/share/lib/terminfo/?/*`

DESCRIPTION

`terminfo` is a database produced by `tic` that describes the capabilities of devices such as terminals and printers. Devices are described in `terminfo` source files by specifying a set of capabilities, by quantifying certain aspects of the device, and by specifying character sequences that effect particular results. This database is often used by screen oriented applications such as `vi` and CURSES programs, as well as by some UNIX system commands such as `ls` and `more`. This usage allows them to work with a variety of devices without changes to the programs.

`terminfo` source files consist of one or more device descriptions. Each description consists of a header (beginning in column 1) and one or more lines that list the features for that particular device. Every line in a `terminfo` source file must end in a comma (`,`). Every line in a `terminfo` source file except the header must be indented with one or more white spaces (either spaces or tabs).

Entries in `terminfo` source files consist of a number of comma-separated fields. White space after each comma is ignored. Embedded commas must be escaped by using a backslash. The following example shows the format of a `terminfo` source file.

> $alias_1$ | $alias_2$ | ... | $alias_n$ | *longname*,
> *<white space>* `am, lines #24,`
> *<white space>* `home=\Eeh,`

The first line, commonly referred to as the header line, must begin in column one and must contain at least two aliases separated by vertical bars. The last field in the header line must be the long name of the device and it may contain any string. Alias names must be unique in the `terminfo` database and they must conform to UNIX system file naming conventions [see tic(TI_CMD)]; they cannot, for example, contain white space or slashes.

Every device must be assigned a name, such as "vt100". Device names (except the long name) should be chosen using the following conventions. The name should not contain hyphens because hyphens are reserved for use when adding suffixes that indicate special modes.

These special modes may be modes that the hardware can be in, or user preferences. To assign a special mode to a particular device, append a suffix consisting of a hyphen and an indicator of the mode to the device name. For example, the `-w` suffix means "wide mode"; when specified, it allows for a width of 132 columns instead of the standard 80 columns. Therefore, if you want to use a vt100 device set to wide mode, name the device "vt100-w." Use the following suffixes where possible.

Suffix	Meaning	Example
-w	Wide mode (more than 80 columns)	5410-w
-am	With auto. margins (usually default)	vt100-am
-nam	Without automatic margins	vt100-nam
-*n*	Number of lines on the screen	2300-40
-na	No arrow keys (leave them in local)	c100-na
-*n*p	Number of pages of memory	c100-4p
-rv	Reverse video	4415-rv

The terminfo reference manual page is organized in two sections: "DEVICE CAPABILITIES" and "PRINTER CAPABILITIES."

PART 1: DEVICE CAPABILITIES

Capabilities in terminfo are of three types: Boolean capabilities (which show that a device has or does not have a particular feature), numeric capabilities (which quantify particular features of a device), and string capabilities (which provide sequences that can be used to perform particular operations on devices).

In the following table, a **Variable** is the name by which a **C** programmer accesses a capability (at the terminfo level). A **Capname** is the short name for a capability specified in the terminfo source file. It is used by a person updating the source file and by the tput command. A **Termcap Code** is a two-letter sequence that corresponds to the termcap capability name. (Note that termcap is no longer supported.)

Capability names have no real length limit, but an informal limit of five characters has been adopted to keep them short. Whenever possible, capability names are chosen to be the same as or similar to those specified by the ANSI X3.64-1979 standard. Semantics are also intended to match those of the ANSI standard.

All string capabilities listed below may have padding specified, with the exception of those used for input. Input capabilities, listed under the **Strings** section in the following tables, have names beginning with key_. The #i symbol in the description field of the following tables refers to the *i*th parameter.

Booleans

Variable	Cap-name	Termcap Code	Description
auto_left_margin	bw	bw	cub1 wraps from column 0 to last column
auto_right_margin	am	am	Terminal has automatic margins
back_color_erase	bce	be	Screen erased with background color
can_change	ccc	cc	Terminal can re-define existing color
ceol_standout_glitch	xhp	xs	Standout not erased by overwriting (hp)
col_addr_glitch	xhpa	YA	Only positive motion for hpa/mhpa caps
cpi_changes_res	cpix	YF	Changing character pitch changes resolution
cr_cancels_micro_mode	crxm	YB	Using cr turns off micro mode

Variable	Cap-name	Termcap Code	Description
eat_newline_glitch	xenl	xn	Newline ignored after 80 columns (Concept)
erase_overstrike	eo	eo	Can erase overstrikes with a blank
generic_type	gn	gn	Generic line type (*e.g.*, dialup, switch)
hard_copy	hc	hc	Hardcopy terminal
hard_cursor	chts	HC	Cursor is hard to see
has_meta_key	km	km	Has a meta key (shift, sets parity bit)
has_print_wheel	daisy	YC	Printer needs operator to change character set
has_status_line	hs	hs	Has extra "status line"
hue_lightness_saturation	hls	hl	Terminal uses only HLS color notation (Tektronix)
insert_null_glitch	in	in	Insert mode distinguishes nulls
lpi_changes_res	lpix	YG	Changing line pitch changes resolution
memory_above	da	da	Display may be retained above the screen
memory_below	db	db	Display may be retained below the screen
move_insert_mode	mir	mi	Safe to move while in insert mode
move_standout_mode	msgr	ms	Safe to move in standout modes
needs_xon_xoff	nxon	nx	Padding won't work, xon/xoff required
no_esc_ctlc	xsb	xb	Beehive (f1=escape, f2=ctrl C)
non_rev_rmcup	nrrmc	NR	smcup does not reverse rmcup
no_pad_char	npc	NP	Pad character doesn't exist
over_strike	os	os	Terminal overstrikes on hard-copy terminal
prtr_silent	mc5i	5i	Printer won't echo on screen
row_addr_glitch	xvpa	YD	Only positive motion for vpa/mvpa caps
semi_auto_right_margin	sam	YE	Printing in last column causes cr
status_line_esc_ok	eslok	es	Escape can be used on the status line
dest_tabs_magic_smso	xt	xt	Destructive tabs, magic smso char (t1061)
tilde_glitch	hz	hz	Hazeltine; can't print tilde (˜)
transparent_underline	ul	ul	Underline character overstrikes
xon_xoff	xon	xo	Terminal uses xon/xoff handshaking

Numbers

Variable	Cap-name	Termcap Code	Description
buffer_capacity	bufsz	Ya	Number of bytes buffered before printing
columns	cols	co	Number of columns in a line
dot_vert_spacing	spinv	Yb	Spacing of pins vertically in pins per inch
dot_horz_spacing	spinh	Yc	Spacing of dots horizontally in dots per inch
init_tabs	it	it	Tabs initially every # spaces
label_height	lh	lh	Number of rows in each label
label_width	lw	lw	Number of columns in each label

Variable	Cap-name	Termcap Code	Description
lines	lines	li	Number of lines on a screen or a page
lines_of_memory	lm	lm	Lines of memory if > lines; 0 means varies
magic_cookie_glitch	xmc	sg	Number of blank characters left by smso or rmso
max_colors	colors	Co	Maximum number of colors on the screen
max_micro_address	maddr	Yd	Maximum value in micro_..._address
max_micro_jump	mjump	Ye	Maximum value in parm_..._micro
max_pairs	pairs	pa	Maximum number of color-pairs on the screen
micro_col_size	mcs	Yf	Character step size when in micro mode
micro_line_size	mls	Yg	Line step size when in micro mode
no_color_video	ncv	NC	Video attributes that can't be used with colors
number_of_pins	npins	Yh	Number of pins in print-head
num_labels	nlab	Nl	Number of labels on screen (start at 1)
output_res_char	orc	Yi	Horizontal resolution in units per character
output_res_line	orl	Yj	Vertical resolution in units per line
output_res_horz_inch	orhi	Yk	Horizontal resolution in units per inch
output_res_vert_inch	orvi	Yl	Vertical resolution in units per inch
padding_baud_rate	pb	pb	Lowest baud rate where padding needed
virtual_terminal	vt	vt	Virtual terminal number (UNIX system)
wide_char_size	widcs	Yn	Character step size when in double wide mode
width_status_line	wsl	ws	Number of columns in status line

Strings

Variable	Cap-name	Termcap Code	Description
acs_chars	acsc	ac	Graphic charset pairs aAbBcC
alt_scancode_esc	scesca	S8	Alternate escape for scancode emulation (default is for vt100)
back_tab	cbt	bt	Back tab
bell	bel	bl	Audible signal (bell)
bit_image_repeat	birep	Zy	Repeat bit-image cell #1 #2 times (use tparm)
bit_image_newline	binel	Zz	Move to next row of the bit image (use tparm)
bit_image_carriage_return	bicr	Yv	Move to beginning of same row (use tparm)
carriage_return	cr	cr	Carriage return
change_char_pitch	cpi	ZA	Change number of characters per inch
change_line_pitch	lpi	ZB	Change number of lines per inch
change_res_horz	chr	ZC	Change horizontal resolution
change_res_vert	cvr	ZD	Change vertical resolution
change_scroll_region	csr	cs	Change to lines #1 through #2 (vt100)
char_padding	rmp	rP	Like ip but when in replace mode

Variable	Cap-name	Termcap Code	Description
char_set_names	csnm	Zy	List of character set names
clear_all_tabs	tbc	ct	Clear all tab stops
clear_margins	mgc	MC	Clear all margins (top, bottom, and sides)
clear_screen	clear	cl	Clear screen and home cursor
clr_bol	el1	cb	Clear to beginning of line, inclusive
clr_eol	el	ce	Clear to end of line
clr_eos	ed	cd	Clear to end of display
code_set_init	csin	ci	Init sequence for multiple codesets
color_names	colornm	Yw	Give name for color #1
column_address	hpa	ch	Horizontal position absolute
command_character	cmdch	CC	Terminal settable cmd character in prototype
cursor_address	cup	cm	Move to row #1 col #2
cursor_down	cud1	do	Down one line
cursor_home	home	ho	Home cursor (if no cup)
cursor_invisible	civis	vi	Make cursor invisible
cursor_left	cub1	le	Move left one space.
cursor_mem_address	mrcup	CM	Memory relative cursor addressing
cursor_normal	cnorm	ve	Make cursor appear normal (undo vs/vi)
cursor_right	cuf1	nd	Non-destructive space (cursor or carriage right)
cursor_to_ll	ll	ll	Last line, first column (if no cup)
cursor_up	cuu1	up	Upline (cursor up)
cursor_visible	cvvis	vs	Make cursor very visible
define_bit_image_region	defbi	Yx	Define rectangular bit-image region (use tparm)
define_char	defc	ZE	Define a character in a character set †
delete_character	dch1	dc	Delete character
delete_line	dl1	dl	Delete line
device_type	devt	dv	Indicate language/codeset support
dis_status_line	dsl	ds	Disable status line
display_pc_char	dispc	S1	Display PC character
down_half_line	hd	hd	Half-line down (forward 1/2 linefeed)
ena_acs	enacs	eA	Enable alternate character set
end_bit_image_region	endbi	Yy	End a bit-image region (use tparm)
enter_alt_charset_mode	smacs	as	Start alternate character set
enter_am_mode	smam	SA	Turn on automatic margins
enter_blink_mode	blink	mb	Turn on blinking
enter_bold_mode	bold	md	Turn on bold (extra bright) mode
enter_ca_mode	smcup	ti	String to begin programs that use cup
enter_delete_mode	smdc	dm	Delete mode (enter)
enter_dim_mode	dim	mh	Turn on half-bright mode

Variable	Cap-name	Termcap Code	Description
enter_doublewide_mode	swidm	ZF	Enable double wide printing
enter_draft_quality	sdrfq	ZG	Set draft quality print
enter_insert_mode	smir	im	Insert mode (enter)
enter_italics_mode	sitm	ZH	Enable italics
enter_leftward_mode	slm	ZI	Enable leftward carriage motion
enter_micro_mode	smicm	ZJ	Enable micro motion capabilities
enter_near_letter_quality	snlq	ZK	Set near-letter quality print
enter_normal_quality	snrmq	ZL	Set normal quality print
enter_pc_charset_mode	smpch	S2	Enter PC character display mode
enter_protected_mode	prot	mp	Turn on protected mode
enter_reverse_mode	rev	mr	Turn on reverse video mode
enter_scancode_mode	smsc	S4	Enter PC scancode mode
enter_secure_mode	invis	mk	Turn on blank mode (characters invisible)
enter_shadow_mode	sshm	ZM	Enable shadow printing
enter_standout_mode	smso	so	Begin standout mode
enter_subscript_mode	ssubm	ZN	Enable subscript printing
enter_superscript_mode	ssupm	ZO	Enable superscript printing
enter_underline_mode	smul	us	Start underscore mode
enter_upward_mode	sum	ZP	Enable upward carriage motion
enter_xon_mode	smxon	SX	Turn on xon/xoff handshaking
erase_chars	ech	ec	Erase #1 characters
exit_alt_charset_mode	rmacs	ae	End alternate character set
exit_am_mode	rmam	RA	Turn off automatic margins
exit_attribute_mode	sgr0	me	Turn off all attributes
exit_ca_mode	rmcup	te	String to end programs that use cup
exit_delete_mode	rmdc	ed	End delete mode
exit_doublewide_mode	rwidm	ZQ	Disable double wide printing
exit_insert_mode	rmir	ei	End insert mode
exit_italics_mode	ritm	ZR	Disable italics
exit_leftward_mode	rlm	ZS	Enable rightward (normal) carriage motion
exit_micro_mode	rmicm	ZT	Disable micro motion capabilities
exit_pc_charset_mode	rmpch	S3	Disable PC character display mode
exit_scancode_mode	rmsc	S5	Disable PC scancode mode
exit_shadow_mode	rshm	ZU	Disable shadow printing
exit_standout_mode	rmso	se	End standout mode
exit_subscript_mode	rsubm	ZV	Disable subscript printing
exit_superscript_mode	rsupm	ZW	Disable superscript printing
exit_underline_mode	rmul	ue	End underscore mode
exit_upward_mode	rum	ZX	Enable downward (normal) carriage motion
exit_xon_mode	rmxon	RX	Turn off xon/xoff handshaking
flash_screen	flash	vb	Visible bell (may not move cursor)

Variable	Cap-name	Termcap Code	Description
form_feed	ff	ff	Hardcopy terminal page eject
from_status_line	fsl	fs	Return from status line
init_1string	is1	i1	Terminal or printer initialization string
init_2string	is2	is	Terminal or printer initialization string
init_3string	is3	i3	Terminal or printer initialization string
init_file	if	if	Name of initialization file
init_prog	iprog	iP	Path name of program for initialization
initialize_color	initc	Ic	Initialize the definition of color
initialize_pair	initp	Ip	Initialize color-pair
insert_character	ich1	ic	Insert character
insert_line	il1	al	Add new blank line
insert_padding	ip	ip	Insert pad after character inserted

The ‘‘key_’’ strings are sent by specific keys. The ‘‘key_’’ descriptions include the macro, defined in curses.h, for the code returned by the CURSES routine getch() when the key is pressed [see curs_getch(TI_LIB)].

key_a1	ka1	K1	KEY_A1, upper left of keypad
key_a3	ka3	K3	KEY_A3, upper right of keypad
key_b2	kb2	K2	KEY_B2, center of keypad
key_backspace	kbs	kb	KEY_BACKSPACE, sent by backspace key
key_beg	kbeg	@1	KEY_BEG, sent by beg(inning) key
key_btab	kcbt	kB	KEY_BTAB, sent by back-tab key
key_c1	kc1	K4	KEY_C1, lower left of keypad
key_c3	kc3	K5	KEY_C3, lower right of keypad
key_cancel	kcan	@2	KEY_CANCEL, sent by cancel key
key_catab	ktbc	ka	KEY_CATAB, sent by clear-all-tabs key
key_clear	kclr	kC	KEY_CLEAR, sent by clear-screen or erase key
key_close	kclo	@3	KEY_CLOSE, sent by close key
key_command	kcmd	@4	KEY_COMMAND, sent by cmd (command) key
key_copy	kcpy	@5	KEY_COPY, sent by copy key
key_create	kcrt	@6	KEY_CREATE, sent by create key
key_ctab	kctab	kt	KEY_CTAB, sent by clear-tab key
key_dc	kdch1	kD	KEY_DC, sent by delete-character key
key_dl	kdl1	kL	KEY_DL, sent by delete-line key
key_down	kcud1	kd	KEY_DOWN, sent by terminal down-arrow key
key_eic	krmir	kM	KEY_EIC, sent by rmir or smir in insert mode
key_end	kend	@7	KEY_END, sent by end key
key_enter	kent	@8	KEY_ENTER, sent by enter/send key
key_eol	kel	kE	KEY_EOL, sent by clear-to-end-of-line

Variable	Cap-name	Termcap Code	Description
			key
key_eos	ked	kS	KEY_EOS, sent by clear-to-end-of-screen key
key_exit	kext	@9	KEY_EXIT, sent by exit key
key_f0	kf0	k0	KEY_F(0), sent by function key f0
key_f1	kf1	k1	KEY_F(1), sent by function key f1
key_f2	kf2	k2	KEY_F(2), sent by function key f2
key_f3	kf3	k3	KEY_F(3), sent by function key f3
key_f4	kf4	k4	KEY_F(4), sent by function key f4
key_f5	kf5	k5	KEY_F(5), sent by function key f5
key_f6	kf6	k6	KEY_F(6), sent by function key f6
key_f7	kf7	k7	KEY_F(7), sent by function key f7
key_f8	kf8	k8	KEY_F(8), sent by function key f8
key_f9	kf9	k9	KEY_F(9), sent by function key f9
key_f10	kf10	k;	KEY_F(10), sent by function key f10
key_f11	kf11	F1	KEY_F(11), sent by function key f11
key_f12	kf12	F2	KEY_F(12), sent by function key f12
key_f13	kf13	F3	KEY_F(13), sent by function key f13
key_f14	kf14	F4	KEY_F(14), sent by function key f14
key_f15	kf15	F5	KEY_F(15), sent by function key f15
key_f16	kf16	F6	KEY_F(16), sent by function key f16
key_f17	kf17	F7	KEY_F(17), sent by function key f17
key_f18	kf18	F8	KEY_F(18), sent by function key f18
key_f19	kf19	F9	KEY_F(19), sent by function key f19
key_f20	kf20	FA	KEY_F(20), sent by function key f20
key_f21	kf21	FB	KEY_F(21), sent by function key f21
key_f22	kf22	FC	KEY_F(22), sent by function key f22
key_f23	kf23	FD	KEY_F(23), sent by function key f23
key_f24	kf24	FE	KEY_F(24), sent by function key f24
key_f25	kf25	FF	KEY_F(25), sent by function key f25
key_f26	kf26	FG	KEY_F(26), sent by function key f26
key_f27	kf27	FH	KEY_F(27), sent by function key f27
key_f28	kf28	FI	KEY_F(28), sent by function key f28
key_f29	kf29	FJ	KEY_F(29), sent by function key f29
key_f30	kf30	FK	KEY_F(30), sent by function key f30
key_f31	kf31	FL	KEY_F(31), sent by function key f31
key_f32	kf32	FM	KEY_F(32), sent by function key f32
key_f33	kf33	FN	KEY_F(13), sent by function key f13
key_f34	kf34	FO	KEY_F(34), sent by function key f34
key_f35	kf35	FP	KEY_F(35), sent by function key f35
key_f36	kf36	FQ	KEY_F(36), sent by function key f36
key_f37	kf37	FR	KEY_F(37), sent by function key f37
key_f38	kf38	FS	KEY_F(38), sent by function key f38
key_f39	kf39	FT	KEY_F(39), sent by function key f39

Variable	Cap-name	Termcap Code	Description
key_f40	kf40	FU	KEY_F(40), sent by function key f40
key_f41	kf41	FV	KEY_F(41), sent by function key f41
key_f42	kf42	FW	KEY_F(42), sent by function key f42
key_f43	kf43	FX	KEY_F(43), sent by function key f43
key_f44	kf44	FY	KEY_F(44), sent by function key f44
key_f45	kf45	FZ	KEY_F(45), sent by function key f45
key_f46	kf46	Fa	KEY_F(46), sent by function key f46
key_f47	kf47	Fb	KEY_F(47), sent by function key f47
key_f48	kf48	Fc	KEY_F(48), sent by function key f48
key_f49	kf49	Fd	KEY_F(49), sent by function key f49
key_f50	kf50	Fe	KEY_F(50), sent by function key f50
key_f51	kf51	Ff	KEY_F(51), sent by function key f51
key_f52	kf52	Fg	KEY_F(52), sent by function key f52
key_f53	kf53	Fh	KEY_F(53), sent by function key f53
key_f54	kf54	Fi	KEY_F(54), sent by function key f54
key_f55	kf55	Fj	KEY_F(55), sent by function key f55
key_f56	kf56	Fk	KEY_F(56), sent by function key f56
key_f57	kf57	Fl	KEY_F(57), sent by function key f57
key_f58	kf58	Fm	KEY_F(58), sent by function key f58
key_f59	kf59	Fn	KEY_F(59), sent by function key f59
key_f60	kf60	Fo	KEY_F(60), sent by function key f60
key_f61	kf61	Fp	KEY_F(61), sent by function key f61
key_f62	kf62	Fq	KEY_F(62), sent by function key f62
key_f63	kf63	Fr	KEY_F(63), sent by function key f63
key_find	kfnd	@0	KEY_FIND, sent by find key
key_help	khlp	%1	KEY_HELP, sent by help key
key_home	khome	kh	KEY_HOME, sent by home key
key_ic	kich1	kI	KEY_IC, sent by ins-char/enter ins-mode key
key_il	kil1	kA	KEY_IL, sent by insert-line key
key_left	kcub1	kl	KEY_LEFT, sent by terminal left-arrow key
key_ll	kll	kH	KEY_LL, sent by home-down key
key_mark	kmrk	%2	KEY_MARK, sent by mark key
key_message	kmsg	%3	KEY_MESSAGE, sent by message key
key_move	kmov	%4	KEY_MOVE, sent by move key
key_next	knxt	%5	KEY_NEXT, sent by next-object key
key_npage	knp	kN	KEY_NPAGE, sent by next-page key
key_open	kopn	%6	KEY_OPEN, sent by open key
key_options	kopt	%7	KEY_OPTIONS, sent by options key
key_ppage	kpp	kP	KEY_PPAGE, sent by previous-page key
key_previous	kprv	%8	KEY_PREVIOUS, sent by previous-object key
key_print	kprt	%9	KEY_PRINT, sent by print or copy key

Variable	Cap-name	Termcap Code	Description
key_redo	krdo	%0	KEY_REDO, sent by redo key
key_reference	kref	&1	KEY_REFERENCE, sent by ref(erence) key
key_refresh	krfr	&2	KEY_REFRESH, sent by refresh key
key_replace	krpl	&3	KEY_REPLACE, sent by replace key
key_restart	krst	&4	KEY_RESTART, sent by restart key
key_resume	kres	&5	KEY_RESUME, sent by resume key
key_right	kcuf1	kr	KEY_RIGHT, sent by terminal right-arrow key
key_save	ksav	&6	KEY_SAVE, sent by save key
key_sbeg	kBEG	&9	KEY_SBEG, sent by shifted beginning key
key_scancel	kCAN	&0	KEY_SCANCEL, sent by shifted cancel key
key_scommand	kCMD	*1	KEY_SCOMMAND, sent by shifted command key
key_scopy	kCPY	*2	KEY_SCOPY, sent by shifted copy key
key_screate	kCRT	*3	KEY_SCREATE, sent by shifted create key
key_sdc	kDC	*4	KEY_SDC, sent by shifted delete-char key
key_sdl	kDL	*5	KEY_SDL, sent by shifted delete-line key
key_select	kslt	*6	KEY_SELECT, sent by select key
key_send	kEND	*7	KEY_SEND, sent by shifted end key
key_seol	kEOL	*8	KEY_SEOL, sent by shifted clear-line key
key_sexit	kEXT	*9	KEY_SEXIT, sent by shifted exit key
key_sf	kind	kF	KEY_SF, sent by scroll-forward/down key
key_sfind	kFND	*0	KEY_SFIND, sent by shifted find key
key_shelp	kHLP	#1	KEY_SHELP, sent by shifted help key
key_shome	kHOM	#2	KEY_SHOME, sent by shifted home key
key_sic	kIC	#3	KEY_SIC, sent by shifted input key
key_sleft	kLFT	#4	KEY_SLEFT, sent by shifted left-arrow key
key_smessage	kMSG	%a	KEY_SMESSAGE, sent by shifted message key
key_smove	kMOV	%b	KEY_SMOVE, sent by shifted move key
key_snext	kNXT	%c	KEY_SNEXT, sent by shifted next key
key_soptions	kOPT	%d	KEY_SOPTIONS, sent by shifted options key
key_sprevious	kPRV	%e	KEY_SPREVIOUS, sent by shifted prev key
key_sprint	kPRT	%f	KEY_SPRINT, sent by shifted print key
key_sr	kri	kR	KEY_SR, sent by scroll-backward/up key
key_sredo	kRDO	%g	KEY_SREDO, sent by shifted redo key
key_sreplace	kRPL	%h	KEY_SREPLACE, sent by shifted replace key
key_sright	kRIT	%i	KEY_SRIGHT, sent by shifted

Variable	Cap-name	Termcap Code	Description
			right-arrow key
key_srsume	kRES	%j	KEY_SRSUME, sent by shifted resume key
key_ssave	kSAV	!1	KEY_SSAVE, sent by shifted save key
key_ssuspend	kSPD	!2	KEY_SSUSPEND, sent by shifted suspend key
key_stab	khts	kT	KEY_STAB, sent by set-tab key
key_sundo	kUND	!3	KEY_SUNDO, sent by shifted undo key
key_suspend	kspd	&7	KEY_SUSPEND, sent by suspend key
key_undo	kund	&8	KEY_UNDO, sent by undo key
key_up	kcuu1	ku	KEY_UP, sent by terminal up-arrow key
keypad_local	rmkx	ke	Out of ‘‘keypad-transmit’’ mode
keypad_xmit	smkx	ks	Put terminal in ‘‘keypad-transmit’’ mode
lab_f0	lf0	l0	Labels on function key f0 if not f0
lab_f1	lf1	l1	Labels on function key f1 if not f1
lab_f2	lf2	l2	Labels on function key f2 if not f2
lab_f3	lf3	l3	Labels on function key f3 if not f3
lab_f4	lf4	l4	Labels on function key f4 if not f4
lab_f5	lf5	l5	Labels on function key f5 if not f5
lab_f6	lf6	l6	Labels on function key f6 if not f6
lab_f7	lf7	l7	Labels on function key f7 if not f7
lab_f8	lf8	l8	Labels on function key f8 if not f8
lab_f9	lf9	l9	Labels on function key f9 if not f9
lab_f10	lf10	la	Labels on function key f10 if not f10
label_off	rmln	LF	Turn off soft labels
label_on	smln	LO	Turn on soft labels
meta_off	rmm	mo	Turn off "meta mode"
meta_on	smm	mm	Turn on "meta mode" (8th bit)
micro_column_address	mhpa	ZY	Like column_address for micro adjustment
micro_down	mcud1	ZZ	Like cursor_down for micro adjustment
micro_left	mcub1	Za	Like cursor_left for micro adjustment
micro_right	mcuf1	Zb	Like cursor_right for micro adjustment
micro_row_address	mvpa	Zc	Like row_address for micro adjustment
micro_up	mcuu1	Zd	Like cursor_up for micro adjustment
newline	nel	nw	Newline (behaves like cr followed by lf)
order_of_pins	porder	Ze	Matches software bits to print-head pins
orig_colors	oc	oc	Set all color(-pair)s to the original ones
orig_pair	op	op	Set default color-pair to the original one
pad_char	pad	pc	Pad character (rather than null)
parm_dch	dch	DC	Delete #1 chars

Variable	Cap-name	Termcap Code	Description
parm_delete_line	dl	DL	Delete #1 lines
parm_down_cursor	cud	DO	Move down #1 lines.
parm_down_micro	mcud	Zf	Like parm_down_cursor for micro adjust.
parm_ich	ich	IC	Insert #1 blank chars
parm_index	indn	SF	Scroll forward #1 lines.
parm_insert_line	il	AL	Add #1 new blank lines
parm_left_cursor	cub	LE	Move cursor left #1 spaces
parm_left_micro	mcub	Zg	Like parm_left_cursor for micro adjust.
parm_right_cursor	cuf	RI	Move right #1 spaces.
parm_right_micro	mcuf	Zh	Like parm_right_cursor for micro adjust.
parm_rindex	rin	SR	Scroll backward #1 lines.
parm_up_cursor	cuu	UP	Move cursor up #1 lines.
parm_up_micro	mcuu	Zi	Like parm_up_cursor for micro adjust.
pc_term_options	pctrm	S6	PC terminal options
pkey_key	pfkey	pk	Prog funct key #1 to type string #2
pkey_local	pfloc	pl	Prog funct key #1 to execute string #2
pkey_plab	pfxl	xl	Prog key #1 to xmit string #2 and show string #3
pkey_xmit	pfx	px	Prog funct key #1 to xmit string #2
plab_norm	pln	pn	Prog label #1 to show string #2
print_screen	mc0	ps	Print contents of the screen
prtr_non	mc5p	pO	Turn on the printer for #1 bytes
prtr_off	mc4	pf	Turn off the printer
prtr_on	mc5	po	Turn on the printer
repeat_char	rep	rp	Repeat char #1 #2 times
req_for_input	rfi	RF	Send next input char (for ptys)
reset_1string	rs1	r1	Reset terminal completely to sane modes
reset_2string	rs2	r2	Reset terminal completely to sane modes
reset_3string	rs3	r3	Reset terminal completely to sane modes
reset_file	rf	rf	Name of file containing reset string
restore_cursor	rc	rc	Restore cursor to position of last sc
row_address	vpa	cv	Vertical position absolute
save_cursor	sc	sc	Save cursor position
scancode_escape	scesc	S7	Escape for scancode emulation
scroll_forward	ind	sf	Scroll text up
scroll_reverse	ri	sr	Scroll text down
select_char_set	scs	Zj	Select character set
set0_des_seq	s0ds	s0	Shift into codeset 0 (EUC set 0, ASCII)
set1_des_seq	s1ds	s1	Shift into codeset 1
set2_des_seq	s2ds	s2	Shift into codeset 2
set3_des_seq	s3ds	s3	Shift into codeset 3
set_a_background	setab	AB	Set background color using ANSI escape

Variable	Cap-name	Termcap Code	Description
set_a_foreground	setaf	AF	Set foreground color using ANSI escape
set_attributes	sgr	sa	Define the video attributes #1-#9
set_background	setb	Sb	Set current background color
set_bottom_margin	smgb	Zk	Set bottom margin at current line
set_bottom_margin_parm	smgbp	Zl	Set bottom margin at line #1 or #2 lines from bottom
set_color_band	setcolor	Yz	Change to ribbon color #1
set_color_pair	scp	sp	Set current color-pair
set_foreground	setf	Sf	Set current foreground color1
set_left_margin	smgl	ML	Set left margin at current line
set_left_margin_parm	smglp	Zm	Set left (right) margin at column #1 (#2)
set_lr_margin	smglr	ML	Sets both left and right margins
set_page_length	slines	YZ	Set page length to #1 lines (use tparm)
set_right_margin	smgr	MR	Set right margin at current column
set_right_margin_parm	smgrp	Zn	Set right margin at column #1
set_tab	hts	st	Set a tab in all rows, current column
set_tb_margin	smgtb	MT	Sets both top and bottom margins
set_top_margin	smgt	Zo	Set top margin at current line
set_top_margin_parm	smgtp	Zp	Set top (bottom) margin at line #1 (#2)
set_window	wind	wi	Current window is lines #1-#2 cols #3-#4
start_bit_image	sbim	Zq	Start printing bit image graphics
start_char_set_def	scsd	Zr	Start definition of a character set
stop_bit_image	rbim	Zs	End printing bit image graphics
stop_char_set_def	rcsd	Zt	End definition of a character set
subscript_characters	subcs	Zu	List of ''subscript-able'' characters
superscript_characters	supcs	Zv	List of ''superscript-able'' characters
tab	ht	ta	Tab to next 8-space hardware tab stop
these_cause_cr	docr	Zw	Printing any of these chars causes `cr`
to_status_line	tsl	ts	Go to status line, col #1
underline_char	uc	uc	Underscore one char and move past it
up_half_line	hu	hu	Half-line up (reverse 1/2 linefeed)
xoff_character	xoffc	XF	X-off character
xon_character	xonc	XN	X-on character
zero_motion	zerom	Zx	No motion for the subsequent character

Sample Entry

The following entry, which describes the AT&T 610 terminal, is among the more complex entries in the `terminfo` file as of this writing.

```
610|610bct|ATT610|att610|AT&T 610;80column;98key keyboard
   am, eslok, hs, mir, msgr, xenl, xon,
   cols#80, it#8, lh#2, lines#24, lw#8, nlab#8, wsl#80,
   acsc=‘‘aaffggjjkkllmmnnooppqqrrssttuuvvwwxxyyzz{{||}}˜˜,
   bel=ˆG, blink=\E[5m, bold=\E[1m, cbt=\E[Z,
   civis=\E[?25l, clear=\E[H\E[J, cnorm=\E[?25h\E[?12l,
   cr=\r, csr=\E[%i%p1%d;%p2%dr, cub=\E[%p1%dD, cub1=\b,
   cud=\E[%p1%dB, cud1=\E[B, cuf=\E[%p1%dC, cuf1=\E[C,
```

```
   cup=\E[%i%p1%d;%p2%dH, cuu=\E[%p1%dA, cuu1=\E[A,
   cvvis=\E[?12;25h, dch=\E[%p1%dP, dch1=\E[P, dim=\E[2m,
   dl=\E[%p1%dM, dl1=\E[M, ed=\E[J, el=\E[K, el1=\E[1K,
   flash=\E[?5h$<200>\E[?5l, fsl=\E8, home=\E[H, ht=\t,
   ich=\E[%p1%d@, il=\E[%p1%dL, il1=\E[L, ind=\ED, .ind=\ED$<9>,
   invis=\E[8m,
   is1=\E[8;0 | \E[?3;4;5;13;15l\E[13;20l\E[?7h\E[12h\E(B\E)0,
   is2=\E[0m^O, is3=\E(B\E)0, kLFT=\E[\s@, kRIT=\E[\sA,
   kbs=^H, kcbt=\E[Z, kclr=\E[2J, kcub1=\E[D, kcud1=\E[B,
   kcuf1=\E[C, kcuu1=\E[A, kf1=\EOc, kf10=\ENp,
   kf11=\ENq, kf12=\ENr, kf13=\ENs, kf14=\ENt, kf2=\EOd,
   kf3=\EOe, kf4=\EOf, kf5=\EOg, kf6=\EOh, kf7=\EOi,
   kf8=\EOj, kf9=\ENo, khome=\E[H, kind=\E[S, kri=\E[T,
   ll=\E[24H, mc4=\E[?4i, mc5=\E[?5i, nel=\EE,
   pfxl=\E[%p1%d;%p2%l%02dq%?%p1%{9}%<%t\s\s\sF%p1%1d\s\s\s\s\s
\s\s\s\s\s\s%;%p2%s,
   pln=\E[%p1%d;0;0;0q%p2%:-16.16s, rc=\E8, rev=\E[7m,
   ri=\EM, rmacs=^O, rmir=\E[4l, rmln=\E[2p, rmso=\E[m,
   rmul=\E[m, rs2=\Ec\E[?3l, sc=\E7,
   sgr=\E[0%?%p6%t;1%;%?%p5%t;2%;%?%p2%t;4%;%?%p4%t;5%;
%?%p3%p1% | %t;7%;%?%p7%t;8%;m%?%p9%t^N%e^O%;,
   sgr0=\E[m^O, smacs=^N, smir=\E[4h, smln=\E[p,
   smso=\E[7m, smul=\E[4m, tsl=\E7\E[25;%i%p1%dx,
```

Types of Capabilities in the Sample Entry

The sample entry shows the formats for the three types of `terminfo` capabilities listed: Boolean, numeric, and string. All capabilities specified in the `terminfo` source file must be followed by commas, including the last capability in the source file. In `terminfo` source files, capabilities are referenced by their capability names (as shown in the previous tables).

Boolean capabilities are specified simply by their comma separated cap names.

Numeric capabilities are followed by the character '#' and then a positive integer value. Thus, in the sample, `cols` (which shows the number of columns available on a device) is assigned the value `80` for the AT&T 610. (Values for numeric capabilities may be specified in decimal, octal, or hexadecimal, using normal C programming language conventions.)

Finally, string-valued capabilities such as `el` (clear to end of line sequence) are listed by a two- to five-character capname, an '=', and a string ended by the next occurrence of a comma. A delay in milliseconds may appear anywhere in such a capability, preceded by `$` and enclosed in angle brackets, as in `el=\EK$<3>`. Padding characters are supplied by `tput`. The delay can be any of the following: a number, a number followed by an asterisk, such as `5*`, a number followed by a slash, such as `5/`, or a number followed by both, such as `5*/`. A '`*`' shows that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. (In the case of insert characters, the factor is still the number of lines affected. This is always 1 unless the device has `in` and the software uses it.) When a '`*`' is specified, it is sometimes useful to give a delay of the form `3.5` to specify a delay per unit to tenths of milliseconds. (Only one decimal place is allowed.)

A '/' indicates that the padding is mandatory. If a device has `xon` defined, the padding information is advisory and will only be used for cost estimates or when the device is in raw mode. Mandatory padding will be transmitted regardless of the setting of `xon`. If padding (whether advisory or mandatory) is specified for `bel` or `flash`, however, it will always be used, regardless of whether `xon` is specified.

`terminfo` offers notation for encoding special characters. Both `\E` and `\e` map to an ESCAPE character, *ˆx* maps to a control *x* for any appropriate *x*, and the sequences `\n, \l, \r, \t, \b, \f`, and `\s` give a newline, linefeed, return, tab, backspace, formfeed, and space, respectively. Other escapes include: `\ˆ` for caret (ˆ); `\\` for backslash (\); `\,` for comma (,); `\:` for colon (:); and `\0` for null. (`\0` will actually produce `\200`, which does not terminate a string but behaves as a null character on most devices, providing CS7 is specified. [See stty(AU_CMD).] Finally, characters may be given as three octal digits after a backslash (*e.g.*, \123).

Sometimes individual capabilities must be commented out. To do this, put a period before the capability name. For example, see the second `ind` in the example above. Note that capabilities are defined in a left-to-right order and, therefore, a prior definition will override a later definition.

Preparing Descriptions

The most effective way to prepare a device description is by imitating the description of a similar device in `terminfo` and building up a description gradually, using partial descriptions with `vi` to check that they are correct. Be aware that a very unusual device may expose deficiencies in the ability of the `terminfo` file to describe it or the inability of `vi` to work with that device. To test a new device description, set the environment variable `TERMINFO` to the pathname of a directory containing the compiled description you are working on and programs will look there rather than in `/usr/share/lib/terminfo`. To get the padding for insert-line correct (if the device manufacturer did not document it) a severe test is to comment out `xon`, edit a large file at 9600 baud with `vi`, delete 16 or so lines from the middle of the screen, and then press the `u` key several times quickly. If the display is corrupted, more padding is usually needed. A similar test can be used for insert-character.

Section 1-1: Basic Capabilities

The number of columns on each line for the device is given by the `cols` numeric capability. If the device has a screen, then the number of lines on the screen is given by the `lines` capability. If the device wraps around to the beginning of the next line when it reaches the right margin, then it should have the `am` capability. If the terminal can clear its screen, leaving the cursor in the home position, then this is given by the `clear` string capability. If the terminal overstrikes (rather than clearing a position when a character is struck over) then it should have the `os` capability. If the device is a printing terminal, with no soft copy unit, specify both `hc` and `os`. If there is a way to move the cursor to the left edge of the current row, specify this as `cr`. (Normally this will be carriage return, control M.) If there is a way to produce an audible signal (such as a bell or a beep), specify it as `bel`. If, like most devices, the device uses the xon-xoff flow-control protocol, specify `xon`.

If there is a way to move the cursor one position to the left (such as backspace), that capability should be given as `cub1`. Similarly, sequences to move to the right, up, and down should be given as `cuf1`, `cuu1`, and `cud1`, respectively. These local cursor motions must not alter the text they pass over; for example, you would not normally use ''`cuf1=\s`'' because the space would erase the character moved over.

A very important point here is that the local cursor motions encoded in `terminfo` are undefined at the left and top edges of a screen terminal. Programs should never attempt to backspace around the left edge, unless `bw` is specified, and should never attempt to go up locally off the top. To scroll text up, a program goes to the bottom left corner of the screen and sends the `ind` (index) string.

To scroll text down, a program goes to the top left corner of the screen and sends the `ri` (reverse index) string. The strings `ind` and `ri` are undefined when not on their respective corners of the screen.

Parameterized versions of the scrolling sequences are `indn` and `rin`. These versions have the same semantics as `ind` and `ri`, except that they take one parameter and scroll the number of lines specified by that parameter. They are also undefined except at the appropriate edge of the screen.

The `am` capability tells whether the cursor sticks at the right edge of the screen when text is output, but this does not necessarily apply to a `cuf1` from the last column. Backward motion from the left edge of the screen is possible only when `bw` is specified. In this case, `cub1` will move to the right edge of the previous row. If `bw` is not given, the effect is undefined. This is useful for drawing a box around the edge of the screen, for example. If the device has switch selectable automatic margins, `am` should be specified in the `terminfo` source file. In this case, initialization strings should turn on this option, if possible. If the device has a command that moves to the first column of the next line, that command can be given as `nel` (newline). It does not matter if the command clears the remainder of the current line, so if the device has no `cr` and `lf` it may still be possible to craft a working `nel` out of one or both of them.

These capabilities suffice to describe hardcopy and screen terminals. Thus the AT&T 5320 hardcopy terminal is described as follows:

```
5320|att5320|AT&T 5320 hardcopy terminal,
    am, hc, os,
    cols#132,
    bel=^G, cr=\r, cub1=\b, cnd1=\n,
    dch1=\E[P, dl1=\E[M,
    ind=\n,
```

while the Lear Siegler ADM–3 is described as

```
adm3 | lsi adm3,
  am, bel=^G, clear=^Z, cols#80, cr=^M, cub1=^H,
  cud1=^J, ind=^J, lines#24,
```

Section 1-2: Parameterized Strings

Cursor addressing and other strings requiring parameters are described by a parameterized string capability, with `printf()`-like escapes (%*x*) in it. For example, to address the cursor, the `cup` capability is given, using two parameters: the row and column to address to. (Rows and columns are numbered from zero and refer to the physical screen visible to the user, not to any unseen memory.) If the terminal has memory relative cursor addressing, that can be indicated by `mrcup`.

The parameter mechanism uses a stack and special `%` codes to manipulate the stack in the manner of Reverse Polish Notation (postfix). Typically a sequence will push one of the parameters onto the stack and then print it in some format. Often more complex operations are necessary. Operations are in postfix form with the operands in the usual order. That is, to subtract 5 from the first parameter, one would use `%p1%{5}%-`.

The `%` encodings have the following meanings:

`%%` outputs '%'

`%[[:]`*flags*][*width*[.*precision*]][`doxXs`]
as in `printf()`, flags are `[-+#]` and space

`%c` print pop() gives %c

`%p[1-9]`
push *i*th parm

`%P[a-z]`
set dynamic variable [a-z] to pop()

`%g[a-z]`
get dynamic variable [a-z] and push it

`%P[A-Z]`
set static variable [a-z] to pop()

`%g[A-Z]`
get static variable [a-z] and push it

`%'`*c*`'` push char constant *c*

`%{`*nn*`}`
push decimal constant *nn*

`%l` push strlen(pop())

`%+ %- %* %/ %m`
arithmetic (`%m` is mod): push(pop $integer_2$ op pop $integer_1$) where $integer_1$ represents the top of the stack

`%& %| %^`
bit operations: push(pop $integer_2$ op pop $integer_1$)

`%= %> %<`
logical operations: push(pop $integer_2$ op pop $integer_1$)

`%A %O`
logical operations: and, or

`%! %~`
unary operations: push(op pop())

`%i` (for ANSI terminals) add 1 to first parm, if one parm present, or first two parms, if more than one parm present

`%?` *expr* `%t` *thenpart* `%e` *elsepart* `%;`

if-then-else, `%e` *elsepart* is optional; else-if's are possible ala Algol 68: `%?` c_1 `%t` b_1 `%e` c_2 `%t` b_2 `%e` c_3 `%t` b_3 `%e` c_4 `%t` b_4 `%e` b_5`%;`
c_i are conditions, b_i are bodies.

If the ''-'' flag is used with ''`%[doxXs]`'', then a colon (`:`) must be placed between the ''`%`'' and the ''`-`'' to differentiate the flag from the binary ''`%-`'' operator, *e.g.* ''`%:-16.16s`''.

Consider the Hewlett-Packard 2645, which, to get to row 3 and column 12, needs to be sent `\E&a12c03Y` padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are zero-padded as two digits. Thus its `cup` capability is:

```
cup=\E&a%p2%2.2dc%p1%2.2dY$<6>
```

The Micro-Term ACT-IV needs the current row and column sent preceded by a `^T`, with the row and column simply encoded in binary, ''`cup=^T%p1%c%p2%c`''. Devices that use ''`%c`'' need to be able to backspace the cursor (`cub1`), and to move the cursor up one line on the screen (`cuu1`). This is necessary because it is not always safe to transmit `\n`, `^D`, and `\r`, as the system may change or discard them. (The library routines dealing with `terminfo` set tty modes so that tabs are never expanded, so `\t` is safe to send. This turns out to be essential for the Ann Arbor 4080.)

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus ''`cup=\E=%p1%'\s'%+%c%p2%'\s'%+%c`''. After sending ''`\E=`'', this pushes the first parameter, pushes the ASCII value for a space (32), adds them (pushing the sum on the stack in place of the two previous values), and outputs that value as a character. Then the same is done for the second parameter. More complex arithmetic is possible using the stack.

Section 1-3: Cursor Motions

If the terminal has a fast way to home the cursor (to very upper left corner of screen) then this can be given as `home`; similarly a fast way of getting to the lower left-hand corner can be given as `ll`; this may involve going up with `cuu1` from the home position, but a program should never do this itself (unless `ll` does) because it can make no assumption about the effect of moving up from the home position. Note that the home position is the same as addressing to (0,0): to the top left corner of the screen, not of memory. (Thus, the `\EH` sequence on Hewlett-Packard terminals cannot be used for `home` without losing some of the other features on the terminal.)

If the device has row or column absolute-cursor addressing, these can be given as single parameter capabilities `hpa` (horizontal position absolute) and `vpa` (vertical position absolute). Sometimes these are shorter than the more general two-parameter sequence (as with the Hewlett-Packard 2645) and can be used in preference to `cup`. If there are parameterized local motions (*e.g.*, move *n* spaces to the right) these can be given as `cud`, `cub`, `cuf`, and `cuu` with a single parameter indicating how many spaces to move. These are primarily useful if the device does not have `cup`, such as the Tektronix 4025.

If the device needs to be in a special mode when running a program that uses these capabilities, the codes to enter and exit this mode can be given as `smcup` and `rmcup`. This arises, for example, from terminals, such as the Concept, with more than one page of memory. If the device has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the device for cursor addressing to work properly. This is also used for the Tektronix 4025, where `smcup` sets the command character to be the one used by `terminfo`. If the `smcup` sequence will not restore the screen after an `rmcup` sequence is output (to the state prior to outputting `rmcup`), specify `nrrmc`.

Section 1-4: Area Clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as `el`. If the terminal can clear from the beginning of the line to the current position inclusive, leaving the cursor where it is, this should be given as `el1`. If the terminal can clear from the current position to the end of the display, then this should be given as `ed`. `ed` is only defined from the first column of a line. (Thus, it can be simulated by a request to delete a large number of lines, if a true `ed` is not available.)

Section 1-5: Insert/Delete Line

If the terminal can open a new blank line before the line where the cursor is, this should be given as `il1`; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as `dl1`; this is done only from the first position on the line to be deleted. Versions of `il1` and `dl1` which take a single parameter and insert or delete that many lines can be given as `il` and `dl`.

If the terminal has a settable destructive scrolling region (like the VT100) the command to set this can be described with the `csr` capability, which takes two parameters: the top and bottom lines of the scrolling region. The cursor position is, alas, undefined after using this command. It is possible to get the effect of insert or delete line using this command — the `sc` and `rc` (save and restore cursor) commands are also useful. Inserting lines at the top or bottom of the screen can also be done using `ri` or `ind` on many terminals without a true insert/delete line, and is often faster even on terminals with those features.

To determine whether a terminal has destructive scrolling regions or non-destructive scrolling regions, create a scrolling region in the middle of the screen, place data on the bottom line of the scrolling region, move the cursor to the top line of the scrolling region, and do a reverse index (`ri`) followed by a delete line (`dl1`) or index (`ind`). If the data that was originally on the bottom line of the scrolling region was restored into the scrolling region by the `dl1` or `ind`, then the terminal has non-destructive scrolling regions. Otherwise, it has destructive scrolling regions. Do not specify `csr` if the terminal has non-destructive scrolling regions, unless `ind`, `ri`, `indn`, `rin`, `dl`, and `dl1` all simulate destructive scrolling.

If the terminal has the ability to define a window as part of memory, which all commands affect, it should be given as the parameterized string `wind`. The four parameters are the starting and ending lines in memory and the starting and ending columns in memory, in that order.

If the terminal can retain display memory above, then the `da` capability should be given; if display memory can be retained below, then `db` should be given. These indicate that deleting a line or scrolling a full screen may bring non-blank lines up from below or that scrolling back with `ri` may bring down non-blank lines.

Section 1-6: Insert/Delete Character

There are two basic kinds of intelligent terminals with respect to insert/delete character operations which can be described using `terminfo.` The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can determine the kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type ''`abc def`'' using local cursor motions (not spaces) between the `abc` and the `def`. Then position the cursor before the `abc` and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the `abc` shifts over to the `def` which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability `in`, which stands for ''insert null.'' While these are two logically separate attributes (one line versus multiline insert mode, and special treatment of untyped spaces) we have seen no terminals whose insert mode cannot be described with the single attribute.

`terminfo` can describe both terminals that have an insert mode and terminals which send a simple sequence to open a blank position on the current line. Give as `smir` the sequence to get into insert mode. Give as `rmir` the sequence to leave insert mode. Now give as `ich1` any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give `ich1`; terminals that send a sequence to open a screen position should give it here. (If your terminal has both, insert mode is usually preferable to `ich1`. Do not give both unless the terminal actually requires both to be used in combination.) If post-insert padding is needed, give this as a number of milliseconds padding in `ip` (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in `ip`. If your terminal needs both to be placed into an 'insert mode' and a special code to precede each inserted character, then both `smir`/rmir and `ich1` can be given, and both will be used. The `ich` capability, with one parameter, *n*, will insert *n* blanks.

If padding is necessary between characters typed while not in insert mode, give this as a number of milliseconds padding in `rmp`.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (*e.g.*, if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability `mir` to speed up inserting in this case. Omitting `mir` will affect only speed. Some terminals (notably Datamedia's) must not have `mir` because of the way their insert mode works.

Finally, you can specify dch1 to delete a single character, dch with one parameter, *n*, to delete *n* characters, and delete mode by giving smdc and rmdc to enter and exit delete mode (any mode the terminal needs to be placed in for dch1 to work).

A command to erase *n* characters (equivalent to outputting *n* blanks without moving the cursor) can be given as ech with one parameter.

Section 1-7: Highlighting, Underlining, and Visible Bells

Your device may have one or more kinds of display attributes that allow you to highlight selected characters when they appear on the screen. The following display modes (shown with the names by which they are set) may be available: a blinking screen (blink), bold or extra-bright characters (bold), dim or half-bright characters (dim), blanking or invisible text (invis), protected text (prot), a reverse-video screen (rev), and an alternate character set (smacs to enter this mode and rmacs to exit it). (If a command is necessary before you can enter alternate character set mode, give the sequence in enacs or "enable alternate-character-set" mode.) Turning on any of these modes singly may or may not turn off other modes.

sgr0 should be used to turn off all video enhancement capabilities. It should always be specified because it represents the only way to turn off some capabilities, such as dim or blink.

You should choose one display method as *standout mode* [see CURSES(TI_LIB)] and use it to highlight error messages and other kinds of text to which you want to draw attention. Choose a form of display that provides strong contrast but that is easy on the eyes. (We recommend reverse-video plus half-bright or reverse-video alone.) The sequences to enter and exit standout mode are given as smso and rmso, respectively. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then xmc should be given to tell how many spaces are left.

Sequences to begin underlining and end underlining can be specified as smul and rmul , respectively. If the device has a sequence to underline the current character and to move the cursor one space to the right (such as the Micro-Term MIME), this sequence can be specified as uc.

Terminals with the ''magic cookie'' glitch (xmc) deposit special ''cookies'' when they receive mode-setting sequences, which affect the display algorithm rather than having extra bits for each character. Some terminals, such as the Hewlett-Packard 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline, unless the msgr capability, asserting that it is safe to move in standout mode, is present.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement), then this can be given as flash; it must not move the cursor. A good flash can be done by changing the screen into reverse video, pad for 200 ms, then return the screen to normal video.

If the cursor needs to be made more visible than normal when it is not on the bottom line (to make, for example, a non-blinking underline into an easier to find block or blinking underline) give this sequence as `cvvis`. The boolean `chts` should also be given. If there is a way to make the cursor completely invisible, give that as `civis`. The capability `cnorm` should be given which undoes the effects of either of these modes.

If your terminal generates underlined characters by using the underline character (with no special sequences needed) even though it does not otherwise overstrike characters, then you should specify the capability `ul`. For devices on which a character overstriking another leaves both characters on the screen, specify the capability `os`. If overstrikes are erasable with a blank, then this should be indicated by specifying `eo`.

If there is a sequence to set arbitrary combinations of modes, this should be given as `sgr` (set attributes), taking nine parameters. Each parameter is either `0` or non-zero, as the corresponding attribute is on or off. The nine parameters are, in order: standout, underline, reverse, blink, dim, bold, blank, protect, alternate character set. Not all modes need to be supported by `sgr`; only those for which corresponding separate attribute commands exist should be supported. For example, let's assume that the terminal in question needs the following escape sequences to turn on various modes.

tparm parameter	attribute	escape sequence
	none	`\E[0m`
`p1`	standout	`\E[0;4;7m`
`p2`	underline	`\E[0;3m`
`p3`	reverse	`\E[0;4m`
`p4`	blink	`\E[0;5m`
`p5`	dim	`\E[0;7m`
`p6`	bold	`\E[0;3;4m`
`p7`	invis	`\E[0;8m`
`p8`	protect	not available
`p9`	altcharset	`^O` (off) `^N` (on)

Note that each escape sequence requires a `0` to turn off other modes before turning on its own mode. Also note that, as suggested above, *standout* is set up to be the combination of *reverse* and *dim*. Also, because this terminal has no *bold* mode, *bold* is set up as the combination of *reverse* and *underline*. In addition, to allow combinations, such as *underline+blink*, the sequence to use would be `\E[0;3;5m`. The terminal doesn't have *protect* mode, either, but that cannot be simulated in any way, so `p8` is ignored. The *altcharset* mode is different in that it is either `^O` or `^N`, depending on whether it is off or on. If all modes were to be turned on, the sequence would be `\E[0;3;4;5;7;8m^N`.

Now look at when different sequences are output. For example, `;3` is output when either `p2` or `p6` is true, that is, if either *underline* or *bold* modes are turned on. Writing out the above sequences, along with their dependencies, gives the following:

sequence	when to output	terminfo translation
`\E[0`	always	`\E[0`
`;3`	if `p2` or `p6`	`%?%p2%p6%|%t;3%;`
`;4`	if `p1` or `p3` or `p6`	`%?%p1%p3%|%p6%|%t;4%;`
`;5`	if `p4`	`%?%p4%t;5%;`
`;7`	if `p1` or `p5`	`%?%p1%p5%|%t;7%;`
`;8`	if `p7`	`%?%p7%t;8%;`
`m`	always	`m`
`^N` or `^O`	if `p9` `^N`, else `^O`	`%?%p9%t^N%e^O%;`

Putting this all together into the `sgr` sequence gives:

```
sgr=\E[0%?%p2%p6%|%t;3%;%?%p1%p3%|%p6%
        |%t;4%;%?%p5%t;5%;%?%p1%p5%
        |%t;7%;%?%p7%t;8%;m%?%p9%t^N%e^O%;,
```

Remember that `sgr` and `sgr0` must always be specified.

Section 1-8: Keypad

If the device has a keypad that transmits sequences when the keys are pressed, this information can also be specified. Note that it is not possible to handle devices where the keypad only works in local (this applies, for example, to the unshifted Hewlett-Packard 2621 keys). If the keypad can be set to transmit or not transmit, specify these sequences as `smkx` and `rmkx`. Otherwise the keypad is assumed to always transmit.

The sequences sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as `kcub1, kcuf1, kcuu1, kcud1,` and `khome`, respectively. If there are function keys such as f0, f1, ..., f63, the sequences they send can be specified as `kf0, kf1, ..., kf63`. If the first 11 keys have labels other than the default f0 through f10, the labels can be given as `lf0, lf1, ..., lf10`. The codes transmitted by certain other special keys can be given: `kll` (home down), `kbs` (backspace), `ktbc` (clear all tabs), `kctab` (clear the tab stop in this column), `kclr` (clear screen or erase key), `kdch1` (delete character), `kdl1` (delete line), `krmir` (exit insert mode), `kel` (clear to end of line), `ked` (clear to end of screen), `kich1` (insert character or enter insert mode), `kil1` (insert line), `knp` (next page), `kpp` (previous page), `kind` (scroll forward/down), `kri` (scroll backward/up), `khts` (set a tab stop in this column). In addition, if the keypad has a 3 by 3 array of keys including the four arrow keys, the other five keys can be given as `ka1`, `ka3`, `kb2`, `kc1`, and `kc3`. These keys are useful when the effects of a 3 by 3 directional pad are needed. Further keys are defined above in the capabilities list.

Strings to program function keys can be specified as `pfkey`, `pfloc`, and `pfx`. A string to program screen labels should be specified as `pln`. Each of these strings takes two parameters: a function key identifier and a string to program it with. `pfkey` causes pressing the given key to be the same as the user typing the given string; `pfloc` causes the string to be executed by the terminal in local mode; and `pfx` causes the string to be transmitted to the computer. The capabilities `nlab`, `lw` and `lh` define the number of programmable screen labels and their width and height. If there are commands to turn the labels on and off, give them in `smln` and `rmln`. `smln` is normally output after one or more `pln` sequences to make sure that the change becomes visible.

Section 1-9: Tabs and Initialization

If the device has hardware tabs, the command to advance to the next tab stop can be given as `ht` (usually control I). A ''backtab'' command that moves leftward to the next tab stop can be given as `cbt`. By convention, if tty modes show that tabs are being expanded by the computer rather than being sent to the device, programs should not use `ht` or `cbt` (even if they are present) because the user may not have the tab stops properly set. If the device has hardware tabs that are initially set every *n* spaces when the device is powered up, the numeric parameter `it` is given, showing the number of spaces the tabs are set to. This is normally used by `tput init` [see tput(TI_CMD)] to determine whether to set the mode for hardware tab expansion and whether to set the tab stops. If the device has tab stops that can be saved in nonvolatile memory, the `terminfo` description can assume that they are properly set. If there are commands to set and clear tab stops, they can be given as `tbc` (clear all tab stops) and `hts` (set a tab stop in the current column of every row).

Other capabilities include: `is1`, `is2`, and `is3`, initialization strings for the device; `iprog`, the path name of a program to be run to initialize the device; and `if`, the name of a file containing long initialization strings. These strings are expected to set the device into modes consistent with the rest of the `terminfo` description. They must be sent to the device each time the user logs in and be output in the following order: run the program `iprog`; output `is1`; output `is2`; set the margins using `mgc`, `smgl` and `smgr`; set the tabs using `tbc` and `hts`; print the file `if`; and finally output `is3`. This is usually done using the `init` option of `tput`.

Most initialization is done with `is2`. Special device modes can be set up without duplicating strings by putting the common sequences in `is2` and special cases in `is1` and `is3`. Sequences that do a reset from a totally unknown state can be given as `rs1`, `rs2`, `rf`, and `rs3`, analogous to `is1`, `is2`, `is3`, and `if`. (The method using files, `if` and `rf`, is used for a few terminals, from `/usr/share/lib/tabset/*`; however, the recommended method is to use the initialization and reset strings.) These strings are output by `tput` reset, which is used when the terminal gets into a wedged state. Commands are normally placed in `rs1`, `rs2`, `rs3`, and `rf` only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set a terminal into 80-column mode would normally be part of `is2`, but on some terminals it causes an annoying glitch on the screen and is not normally needed because the terminal is usually already in 80-column mode.

If a more complex sequence is needed to set the tabs than can be described by using `tbc` and `hts`, the sequence can be placed in `is2` or `if`.

Any margin can be cleared with `mgc`. (For instructions on how to specify commands to set and clear margins, see "Margins" below under "PRINTER CAPABILITIES.")

Section 1-10: Delays

Certain capabilities control padding in the `tty` driver. These are primarily needed by hard-copy terminals, and are used by `tput init` to set tty modes appropriately. Delays embedded in the capabilities `cr`, `ind`, `cub1`, `ff`, and `tab` can be used to set the appropriate delay bits to be set in the tty driver. If `pb` (padding baud rate) is given, these values can be ignored at baud rates below the value of `pb`.

Section 1-11: Status Lines

If the terminal has an extra ''status line'' that is not normally used by software, this fact can be indicated. If the status line is viewed as an extra line below the bottom line, into which one can cursor address normally (such as the Heathkit h19's 25th line, or the 24th line of a VT100 which is set to a 23-line scrolling region), the capability `hs` should be given. Special strings that go to a given column of the status line and return from the status line can be given as `tsl` and `fsl`. (`fsl` must leave the cursor position in the same place it was before `tsl`. If necessary, the `sc` and `rc` strings can be included in `tsl` and `fsl` to get this effect.) The capability `tsl` takes one parameter, which is the column number of the status line the cursor is to be moved to.

If escape sequences and other special commands, such as tab, work while in the status line, the flag `eslok` can be given. A string which turns off the status line (or otherwise erases its contents) should be given as `dsl`. If the terminal has commands to save and restore the position of the cursor, give them as `sc` and `rc`. The status line is normally assumed to be the same width as the rest of the screen, *e.g.*, `cols`. If the status line is a different width (possibly because the terminal does not allow an entire line to be loaded) the width, in columns, can be indicated with the numeric parameter `wsl`.

Section 1-12: Line Graphics

If the device has a line drawing alternate character set, the mapping of glyph to character would be given in `acsc`. The definition of this string is based on the alternate character set used in the DEC VT100 terminal, extended slightly with some characters from the AT&T 4410v1 terminal.

glyph name	vt100+ character
arrow pointing right	+
arrow pointing left	,
arrow pointing down	.
solid square block	0
lantern symbol	I
arrow pointing up	-
diamond	'
checker board (stipple)	a
degree symbol	f
plus/minus	g
board of squares	h
lower right corner	j
upper right corner	k
upper left corner	l
lower left corner	m
plus	n

scan line 1	`o`
horizontal line	`q`
scan line 9	`s`
left tee (├)	`t`
right tee (┤)	`u`
bottom tee (┴)	`v`
top tee (┬)	`w`
vertical line	`x`
bullet	`~`

The best way to describe a new device's line graphics set is to add a third column to the above table with the characters for the new device that produce the appropriate glyph when the device is in the alternate character set mode. For example,

glyph name	vt100+ char	new tty char
upper left corner	`l`	`R`
lower left corner	`m`	`F`
upper right corner	`k`	`T`
lower right corner	`j`	`G`
horizontal line	`q`	`,`
vertical line	`x`	`.`

Now write down the characters left to right, as in ''`acsc=lRmFkTjGq\,x.`''.

In addition, `terminfo` allows you to define multiple character sets. See Section 2-5 for details.

Section 1-13: Color Manipulation

Let us define two methods of color manipulation: the Tektronix method and the HP method. The Tektronix method uses a set of N predefined colors (usually 8) from which a user can select "current" foreground and background colors. Thus a terminal can support up to N colors mixed into N*N color-pairs to be displayed on the screen at the same time. When using an HP method the user cannot define the foreground independently of the background, or vice-versa. Instead, the user must define an entire color-pair at once. Up to M color-pairs, made from 2*M different colors, can be defined this way. Most existing color terminals belong to one of these two classes of terminals.

The numeric variables `colors` and `pairs` define the number of colors and color-pairs that can be displayed on the screen at the same time. If a terminal can change the definition of a color (for example, the Tektronix 4100 and 4200 series terminals), this should be specified with `ccc` (can change color). To change the definition of a color (Tektronix 4200 method), use `initc` (initialize color). It requires four arguments: color number (ranging from 0 to `colors`–1) and three RGB (red, green, and blue) values or three HLS colors (Hue, Lightness, Saturation). Ranges of RGB and HLS values are terminal dependent.

Tektronix 4100 series terminals only use HLS color notation. For such terminals (or dual-mode terminals to be operated in HLS mode) one must define a boolean variable `hls`; that would instruct the CURSES `init_color()` routine to convert its RGB arguments to HLS before sending them to the terminal. The last three arguments to the initc string would then be HLS values.

If a terminal can change the definitions of colors, but uses a color notation different from RGB and HLS, a mapping to either RGB or HLS must be developed.

To set current foreground or background to a given color, use setaf (set ANSI foreground) and setab (set ANSI background). They require one parameter: the number of the color. To initialize a color-pair (HP method), use initp (initialize pair). It requires seven parameters: the number of a color-pair (range=0 to pairs–1), and six RGB values: three for the foreground followed by three for the background. (Each of these groups of three should be in the order RGB.) When initc or initp are used, RGB or HLS arguments should be in the order "red, green, blue" or "hue, lightness, saturation"), respectively. To make a color-pair current, use scp (set color-pair). It takes one parameter, the number of a color-pair.

Some terminals (for example, most color terminal emulators for PCs) erase areas of the screen with current background color. In such cases, bce (background color erase) should be defined. The variable op (original pair) contains a sequence for setting the foreground and the background colors to what they were at the terminal start-up time. Similarly, oc (original colors) contains a control sequence for setting all colors (for the Tektronix method) or color-pairs (for the HP method) to the values they had at the terminal start-up time.

Some color terminals substitute color for video attributes. Such video attributes should not be combined with colors. Information about these video attributes should be packed into the ncv (no color video) variable. There is a one-to-one correspondence between the nine least significant bits of that variable and the video attributes. The following table depicts this correspondence.

Attribute	Bit Position	Decimal Value
A_STANDOUT	0	1
A_UNDERLINE	1	2
A_REVERSE	2	4
A_BLINK	3	8
A_DIM	4	16
A_BOLD	5	32
A_INVIS	6	64
A_PROTECT	7	128
A_ALTCHARSET	8	256

When a particular video attribute should not be used with colors, the corresponding ncv bit should be set to 1; otherwise it should be set to zero. To determine the information to pack into the ncv variable, you must add together the decimal values corresponding to those attributes that cannot coexist with colors. For example, if the terminal uses colors to simulate reverse video (bit number 2 and decimal value 4) and bold (bit number 5 and decimal value 32), the resulting value for ncv will be 36 (4 + 32).

Section 1-14: Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as pad. Only the first character of the pad string is used. If the terminal does not have a pad character, specify npc.

If the terminal can move up or down half a line, this can be indicated with `hu` (half-line up) and `hd` (half-line down). This is primarily useful for superscripts and subscripts on hardcopy terminals. If a hardcopy terminal can eject to the next page (form feed), give this as `ff` (usually control L).

If there is a command to repeat a given character a given number of times (to save time transmitting a large number of identical characters) this can be indicated with the parameterized string `rep`. The first parameter is the character to be repeated and the second is the number of times to repeat it. Thus, `tparm(repeat_char, 'x', 10)` is the same as `xxxxxxxxxx.`

If the terminal has a settable command character, such as the Tektronix 4025, this can be indicated with `cmdch`. A prototype command character is chosen which is used in all capabilities. This character is given in the `cmdch` capability to identify it. The following convention is supported on some UNIX systems: If the environment variable `CC` exists, all occurrences of the prototype character are replaced with the character in `CC`.

Terminal descriptions that do not represent a specific kind of known terminal, such as *switch*, *dialup*, *patch*, and *network*, should include the `gn` (generic) capability so that programs can complain that they do not know how to talk to the terminal. (This capability does not apply to *virtual* terminal descriptions for which the escape sequences are known.) If the terminal is one of those supported by the UNIX system virtual terminal protocol, the terminal number can be given as `vt`. A line-turn-around sequence to be transmitted before doing reads should be specified in `rfi`.

If the device uses xon/xoff handshaking for flow control, give `xon`. Padding information should still be included so that routines can make better decisions about costs, but actual pad characters will not be transmitted. Sequences to turn on and off xon/xoff handshaking may be given in `smxon` and `rmxon`. If the characters used for handshaking are not `^S` and `^Q`, they may be specified with `xonc` and `xoffc`.

If the terminal has a ''meta key'' which acts as a shift key, setting the 8th bit of any character transmitted, this fact can be indicated with `km`. Otherwise, software will assume that the 8th bit is parity and it will usually be cleared. If strings exist to turn this ''meta mode'' on and off, they can be given as `smm` and `rmm`.

If the terminal has more lines of memory than will fit on the screen at once, the number of lines of memory can be indicated with `lm`. A value of `lm`#0 indicates that the number of lines is not fixed, but that there is still more memory than fits on the screen.

Media copy strings which control an auxiliary printer connected to the terminal can be given as `mc0`: print the contents of the screen, `mc4`: turn off the printer, and `mc5`: turn on the printer. When the printer is on, all text sent to the terminal will be sent to the printer. A variation, `mc5p`, takes one parameter, and leaves the printer on for as many characters as the value of the parameter, then turns the printer off. The parameter should not exceed 255. If the text is not displayed on the terminal screen when the printer is on, specify `mc5i` (silent printer). All text, including `mc4`, is transparently passed to the printer while an `mc5p` is in effect.

Section 1-15: Special Cases

The working model used by `terminfo` fits most terminals reasonably well. However, some terminals do not completely match that model, requiring special support by `terminfo`. These are not meant to be construed as deficiencies in the terminals; they are just differences between the working model and the actual hardware. They may be unusual devices or, for some reason, do not have all the features of the `terminfo` model implemented.

Terminals that cannot display tilde (˜) characters, such as certain Hazeltine terminals, should indicate `hz`.

Terminals that ignore a linefeed immediately after an `am` wrap, such as the Concept 100, should indicate `xenl`. Those terminals whose cursor remains on the right-most column until another character has been received, rather than wrapping immediately upon receiving the right-most character, such as the VT100, should also indicate `xenl`.

If `el` is required to get rid of standout (instead of writing normal text on top of it), `xhp` should be given.

Those Teleray terminals whose tabs turn all characters moved over to blanks, should indicate `xt` (destructive tabs). This capability is also taken to mean that it is not possible to position the cursor on top of a ‘‘magic cookie.’’ Therefore, to erase standout mode, it is necessary, instead, to use delete and insert line.

Those Beehive Superbee terminals which do not transmit the escape or control–C characters, should specify `xsb`, indicating that the f1 key is to be used for escape and the f2 key for control C.

Section 1-16: Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability `use` can be given with the name of the similar terminal. The capabilities given before `use` override those in the terminal type invoked by `use`. A capability can be canceled by placing *xx*`@` to the left of the capability definition, where *xx* is the capability. For example, the entry

```
att4424-2|Teletype 4424 in display function group ii,
    rev@, sgr@, smul@, use=att4424,
```

defines an AT&T 4424 terminal that does not have the `rev`, `sgr`, and `smul` capabilities, and hence cannot do highlighting. This is useful for different modes for a terminal, or for different user preferences. More than one `use` capability may be given.

PART 2: PRINTER CAPABILITIES

The `terminfo` database allows you to define capabilities of printers as well as terminals. To find out what capabilities are available for printers as well as for terminals, see the two lists under "DEVICE CAPABILITIES" that list capabilities by variable and by capability name.

Section 2-1: Rounding Values

Because parameterized string capabilities work only with integer values, we recommend that `terminfo` designers create strings that expect numeric values that have been rounded. Application designers should note this and should always round values to the nearest integer before using them with a parameterized string capability.

Section 2-2: Printer Resolution

A printer's resolution is defined to be the smallest spacing of characters it can achieve. In general printers have independent resolution horizontally and vertically. Thus the vertical resolution of a printer can be determined by measuring the smallest achievable distance between consecutive printing baselines, while the horizontal resolution can be determined by measuring the smallest achievable distance between the left-most edges of consecutive printed, identical, characters.

All printers are assumed to be capable of printing with a uniform horizontal and vertical resolution. The view of printing that `terminfo` currently presents is one of printing inside a uniform matrix: All characters are printed at fixed positions relative to each ''cell'' in the matrix; furthermore, each cell has the same size given by the smallest horizontal and vertical step sizes dictated by the resolution. (The cell size can be changed as will be seen later.)

Many printers are capable of ''proportional printing,'' where the horizontal spacing depends on the size of the character last printed. `terminfo` does not make use of this capability, although it does provide enough capability definitions to allow an application to simulate proportional printing.

A printer must not only be able to print characters as close together as the horizontal and vertical resolutions suggest, but also of ''moving'' to a position an integral multiple of the smallest distance away from a previous position. Thus printed characters can be spaced apart a distance that is an integral multiple of the smallest distance, up to the length or width of a single page.

Some printers can have different resolutions depending on different ''modes.'' In ''normal mode,'' the existing `terminfo` capabilities are assumed to work on columns and lines, just like a video terminal. Thus the old `lines` capability would give the length of a page in lines, and the `cols` capability would give the width of a page in columns. In ''micro mode,'' many `terminfo` capabilities work on increments of lines and columns. With some printers the micro mode may be concomitant with normal mode, so that all the capabilities work at the same time.

Section 2-3: Specifying Printer Resolution'

The printing resolution of a printer is given in several ways. Each specifies the resolution as the number of smallest steps per distance:

Specification of Printer Resolution

Characteristic	Number of Smallest Steps
`orhi`	Steps per inch horizontally
`orvi`	Steps per inch vertically
`orc`	Steps per column
`orl`	Steps per line

When printing in normal mode, each character printed causes movement to the next column, except in special cases described later; the distance moved is the same as the per-column resolution. Some printers cause an automatic movement to the next line when a character is printed in the rightmost position; the distance moved vertically is the same as the per-line resolution. When printing in micro mode, these distances can be different, and may be zero for some printers.

Specification of Printer Resolution Automatic Motion after Printing	
Normal Mode:	
`orc`	Steps moved horizontally
`orl`	Steps moved vertically
Micro Mode:	
`mcs`	Steps moved horizontally
`mls`	Steps moved vertically

Some printers are capable of printing wide characters. The distance moved when a wide character is printed in normal mode may be different from when a regular width character is printed. The distance moved when a wide character is printed in micro mode may also be different from when a regular character is printed in micro mode, but the differences are assumed to be related: If the distance moved for a regular character is the same whether in normal mode or micro mode (`mcs=orc`), then the distance moved for a wide character is also the same whether in normal mode or micro mode. This doesn't mean the normal character distance is necessarily the same as the wide character distance, just that the distances don't change with a change in normal to micro mode. However, if the distance moved for a regular character is different in micro mode from the distance moved in normal mode (`mcs<orc`), the micro mode distance is assumed to be the same for a wide character printed in micro mode, as the table below shows.

Specification of Printer Resolution Automatic Motion after Printing Wide Character	
Normal Mode or Micro Mode (`mcs` = `orc`):	
`widcs`	Steps moved horizontally
Micro Mode (`mcs` < `orc`):	
`mcs`	Steps moved horizontally

There may be control sequences to change the number of columns per inch (the character pitch) and to change the number of lines per inch (the line pitch). If these are used, the resolution of the printer changes, but the type of change depends on the printer:

Specification of Printer Resolution Changing the Character/Line Pitches	
`cpi`	Change character pitch
`cpix`	If set, `cpi` changes `orhi`, otherwise changes `orc`
`lpi`	Change line pitch
`lpix`	If set, `lpi` changes `orvi`, otherwise changes `orl`
`chr`	Change steps per column
`cvr`	Change steps per line

The `cpi` and `lpi` string capabilities are each used with a single argument, the pitch in columns (or characters) and lines per inch, respectively. The `chr` and `cvr` string capabilities are each used with a single argument, the number of steps per column and line, respectively.

Using any of the control sequences in these strings will imply a change in some of the values of orc, orhi, orl, and orvi. Also, the distance moved when a wide character is printed, widcs, changes in relation to orc. The distance moved when a character is printed in micro mode, mcs, changes similarly, with one exception: if the distance is 0 or 1, then no change is assumed (see items marked with † in the following table).

Programs that use cpi, lpi, chr, or cvr should recalculate the printer resolution (and should recalculate other values see "Effect of Changing Printing Resolution" under "Dot-Mapped Graphics").

Specification of Printer Resolution
Effects of Changing the Character/Line Pitches

Before	*After*
Using cpi with cpix clear:	
orhi′	**orhi**
orc′	$\mathbf{orc} = \dfrac{\mathbf{orhi}}{V_{cpi}}$
Using cpi with cpix set:	
orhi′	$\mathbf{orhi} = \mathbf{orc} \cdot V_{cpi}$
orc′	**orc**
Using lpi with lpix clear:	
orvi′	**orvi**
orl′	$\mathbf{orl} = \dfrac{\mathbf{orvi}}{V_{lpi}}$
Using lpi with lpix set:	
orvi′	$\mathbf{orvi} = \mathbf{orl} \cdot V_{lpi}$
orl′	**orl**
Using chr:	
orhi′	**orhi**
orc′	V_{chr}
Using cvr:	
orvi′	**orvi**
orl′	V_{cvr}
Using cpi or chr:	
widcs′	$\mathbf{widcs} = \mathbf{widcs}' \dfrac{\mathbf{orc}}{\mathbf{orc}'}$
mcs′	$\mathbf{mcs} = \mathbf{mcs}' \dfrac{\mathbf{orc}}{\mathbf{orc}'}$

V_{cpi}, V_{lpi}, V_{chr}, and V_{cvr} are the arguments used with cpi, lpi, chr, and cvr, respectively. The prime marks (′) indicate the old values.

Section 2-4: Capabilities that Cause Movement

In the following descriptions, ‘‘movement’’ refers to the motion of the ‘‘current position.’’ With video terminals this would be the cursor; with some printers this is the carriage position. Other printers have different equivalents. In general, the current position is where a character would be displayed if printed.

`terminfo` has string capabilities for control sequences that cause movement a number of full columns or lines. It also has equivalent string capabilities for control sequences that cause movement a number of smallest steps.

String Capabilities for Motion	
`mcub1`	Move 1 step left
`mcuf1`	Move 1 step right
`mcuu1`	Move 1 step up
`mcud1`	Move 1 step down
`mcub`	Move *N* steps left
`mcuf`	Move *N* steps right
`mcuu`	Move *N* steps up
`mcud`	Move *N* steps down
`mhpa`	Move *N* steps from the left
`mvpa`	Move *N* steps from the top

The latter six strings are each used with a single argument, *N*.

Sometimes the motion is limited to less than the width or length of a page. Also, some printers don't accept absolute motion to the left of the current position. `terminfo` has capabilities for specifying these limits.

Limits to Motion	
`mjump`	Limit on use of `mcub1`, `mcuf1`, `mcuu1`, `mcud1`
`maddr`	Limit on use of `mhpa`, `mvpa`
`xhpa`	If set, `hpa` and `mhpa` can't move left
`xvpa`	If set, `vpa` and `mvpa` can't move up

If a printer needs to be in a ''micro mode'' for the motion capabilities described above to work, there are string capabilities defined to contain the control sequence to enter and exit this mode. A boolean is available for those printers where using a carriage return causes an automatic return to normal mode.

Entering/Exiting Micro Mode	
`smicm`	Enter micro mode
`rmicm`	Exit micro mode
`crxm`	Using `cr` exits micro mode

The movement made when a character is printed in the rightmost position varies among printers. Some make no movement, some move to the beginning of the next line, others move to the beginning of the same line. `terminfo` has boolean capabilities for describing all three cases.

What Happens After Character Printed in Rightmost Position	
`sam`	Automatic move to beginning of same line

Some printers can be put in a mode where the normal direction of motion is reversed. This mode can be especially useful when there are no capabilities for leftward or upward motion, because those capabilities can be built from the motion reversal capability and the rightward or downward motion capabilities. It is best to leave it up to an application to build the leftward or upward capabilities, though, and not enter them in the `terminfo` database. This allows several reverse motions to be strung together without intervening wasted steps that leave and reenter reverse mode.

Entering/Exiting Reverse Modes	
`slm`	Reverse sense of horizontal motions
`rlm`	Restore sense of horizontal motions
`sum`	Reverse sense of vertical motions
`rum`	Restore sense of vertical motions
While sense of horizontal motions reversed:	
`mcub1`	Move 1 step right
`mcuf1`	Move 1 step left
`mcub`	Move *N* steps right
`mcuf`	Move *N* steps left
`cub1`	Move 1 column right
`cuf1`	Move 1 column left
`cub`	Move *N* columns right
`cuf`	Move *N* columns left
While sense of vertical motions reversed:	
`mcuu1`	Move 1 step down
`mcud1`	Move 1 step up
`mcuu`	Move *N* steps down
`mcud`	Move *N* steps up
`cuu1`	Move 1 line down
`cud1`	Move 1 line up
`cuu`	Move *N* lines down
`cud`	Move *N* lines up

The reverse motion modes should not affect the `mvpa` and `mhpa` absolute motion capabilities. The reverse vertical motion mode should, however, also reverse the action of the line ''wrapping'' that occurs when a character is printed in the right-most position. Thus printers that have the standard `terminfo` capability `am` defined should experience motion to the beginning of the previous line when a character is printed in the right-most position under reverse vertical motion mode.

The action when any other motion capabilities are used in reverse motion modes is not defined; thus, programs must exit reverse motion modes before using other motion capabilities.

Two miscellaneous capabilities complete the list of new motion capabilities. One of these is needed for printers that move the current position to the beginning of a line when certain control characters, such as ''line-feed'' or ''form-feed,'' are used. The other is used for the capability of suspending the motion that normally occurs after printing a character.

Miscellaneous Motion Strings	
docr	List of control characters causing cr
zerom	Prevent auto motion after printing next single character

Margins

terminfo provides two strings for setting margins on terminals: one for the left and one for the right margin. Printers, however, have two additional margins, for the top and bottom margins of each page. Furthermore, some printers require not using motion strings to move the current position to a margin and then fixing the margin there, but require the specification of where a margin should be regardless of the current position. Therefore terminfo offers six additional strings for defining margins with printers.

Setting Margins	
smgl	Set left margin at current column
smgr	Set right margin at current column
smgb	Set bottom margin at current line
smgt	Set top margin at current line
smgbp	Set bottom margin at line *N*
smglp	Set left margin at column *N*
smgrp	Set right margin at column *N*
smgtp	Set top margin at line *N*

The last four strings are used with one or more arguments that give the position of the margin or margins to set. If both of smglp and smgrp are set, each is used with a single argument, *N*, that gives the column number of the left and right margin, respectively. If both of smgtp and smgbp are set, each is used to set the top and bottom margin, respectively: smgtp is used with a single argument, *N*, the line number of the top margin; however, smgbp is used with two arguments, *N* and *M*, that give the line number of the bottom margin, the first counting from the top of the page and the second counting from the bottom. This accommodates the two styles of specifying the bottom margin in different manufacturers' printers. When coding a terminfo entry for a printer that has a settable bottom margin, only the first or second parameter should be used, depending on the printer. When writing an application that uses smgbp to set the bottom margin, both arguments must be given.

If only one of smglp and smgrp is set, then it is used with two arguments, the column number of the left and right margins, in that order. Likewise, if only one of smgtp and smgbp is set, then it is used with two arguments that give the top and bottom margins, in that order, counting from the top of the page. Thus when coding a terminfo entry for a printer that requires setting both left and right or top and bottom margins simultaneously, only one of smglp and smgrp or smgtp and smgbp should be defined; the other should be left blank. When writing an application that uses these string capabilities, the pairs should be first checked to see if each in the pair is set or only one is set, and should then be used accordingly.

In counting lines or columns, line zero is the top line and column zero is the left-most column. A zero value for the second argument with `smgbp` means the bottom line of the page.

All margins can be cleared with `mgc`.

Shadows, Italics, Wide Characters, Superscripts, Subscripts

Five new sets of strings are used to describe the capabilities printers have of enhancing printed text.

Enhanced Printing	
`sshm`	Enter shadow-printing mode
`rshm`	Exit shadow-printing mode
`sitm`	Enter italicizing mode
`ritm`	Exit italicizing mode
`swidm`	Enter wide character mode
`rwidm`	Exit wide character mode
`ssupm`	Enter superscript mode
`rsupm`	Exit superscript mode
`supcs`	List of characters available as superscripts
`ssubm`	Enter subscript mode
`rsubm`	Exit subscript mode
`subcs`	List of characters available as subscripts

If a printer requires the `sshm` control sequence before every character to be shadow-printed, the `rshm` string is left blank. Thus programs that find a control sequence in `sshm` but none in `rshm` should use the `sshm` control sequence before every character to be shadow-printed; otherwise, the `sshm` control sequence should be used once before the set of characters to be shadow-printed, followed by `rshm`. The same is also true of each of the `sitm/ritm`, `swidm/rwidm`, `ssupm/rsupm`, and `ssubm/ rsubm` pairs.

Note that `terminfo` also has a capability for printing emboldened text (`bold`). While shadow printing and emboldened printing are similar in that they ‘‘darken’’ the text, many printers produce these two types of print in slightly different ways. Generally, emboldened printing is done by overstriking the same character one or more times. Shadow printing likewise usually involves overstriking, but with a slight movement up and/or to the side so that the character is ‘‘fatter.’’

It is assumed that enhanced printing modes are independent modes, so that it would be possible, for instance, to shadow print italicized subscripts.

As mentioned earlier, the amount of motion automatically made after printing a wide character should be given in `widcs`.

If only a subset of the printable ASCII characters can be printed as superscripts or subscripts, they should be listed in `supcs` or `subcs` strings, respectively. If the `ssupm` or `ssubm` strings contain control sequences, but the corresponding `supcs` or `subcs` strings are empty, it is assumed that all printable ASCII characters are available as superscripts or subscripts.

Automatic motion made after printing a superscript or subscript is assumed to be the same as for regular characters. Thus, for example, printing any of the following three examples will result in equivalent motion:

`Bi` B_i B^i

Note that the existing `msgr` boolean capability describes whether motion control sequences can be used while in ''standout mode.'' This capability is extended to cover the enhanced printing modes added here. `msgr` should be set for those printers that accept any motion control sequences without affecting shadow, italicized, widened, superscript, or subscript printing. Conversely, if `msgr` is not set, a program should end these modes before attempting any motion.

Section 2-5: Alternate Character Sets

In addition to allowing you to define line graphics (described in Section 1-12), `terminfo` lets you define alternate character sets. The following capabilities cover printers and terminals with multiple selectable or definable character sets.

Alternate Character Sets

`scs`	Select character set *N*
`scsd`	Start definition of character set *N*, *M* characters
`defc`	Define character *A*, *B* dots wide, descender *D*
`rcsd`	End definition of character set *N*
`csnm`	List of character set names
`daisy`	Printer has manually changed print-wheels

The `scs`, `rcsd`, and `csnm` strings are used with a single argument, *N*, a number from 0 to 63 that identifies the character set. The `scsd` string is also used with the argument *N* and another, *M*, that gives the number of characters in the set. The `defc` string is used with three arguments: *A* gives the ASCII code representation for the character, *B* gives the width of the character in dots, and *D* is zero or one depending on whether the character is a ''descender'' or not. The `defc` string is also followed by a string of ''image-data'' bytes that describe how the character looks (see below).

Character set 0 is the default character set present after the printer has been initialized. Not every printer has 64 character sets, of course; using `scs` with an argument that doesn't select an available character set should cause a null result from `tparm`.

If a character set has to be defined before it can be used, the `scsd` control sequence is to be used before defining the character set, and the `rcsd` is to be used after. They should also cause a null result from `tparm` when used with an argument *N* that doesn't apply. If a character set still has to be selected after being defined, the `scs` control sequence should follow the `rcsd` control sequence. By examining the results of using each of the `scs`, `scsd`, and `rcsd` strings with a character set number in a call to `tparm`, a program can determine which of the three are needed.

Between use of the scsd and rcsd strings, the defc string should be used to define each character. To print any character on printers covered by terminfo, the ASCII code is sent to the printer. This is true for characters in an alternate set as well as ''normal'' characters. Thus the definition of a character includes the ASCII code that represents it. In addition, the width of the character in dots is given, along with an indication of whether the character should descend below the print line (such as the lower case letter ''g'' in most character sets). The width of the character in dots also indicates the number of image-data bytes that will follow the defc string. These image-data bytes indicate where in a dot-matrix pattern ink should be applied to ''draw'' the character; the number of these bytes and their form are defined below under ''Dot-Mapped Graphics.''

It's easiest for the creator of terminfo entries to refer to each character set by number; however, these numbers will be meaningless to the application developer. The csnm string alleviates this problem by providing names for each number.

When used with a character set number in a call to tparm, the csnm string will produce the equivalent name. These names should be used as a reference only. No naming convention is implied, although anyone who creates a terminfo entry for a printer should use names consistent with the names found in user documents for the printer. Application developers should allow a user to specify a character set by number (leaving it up to the user to examine the csnm string to determine the correct number), or by name, where the application examines the csnm string to determine the corresponding character set number.

These capabilities are likely to be used only with dot-matrix printers. If they are not available, the strings should not be defined. For printers that have manually changed print-wheels or font cartridges, the boolean daisy is set.

Section 2-6: Dot-Matrix Graphics

Dot-matrix printers typically have the capability of reproducing ''raster-graphics'' images. Three new numeric capabilities and three new string capabilities can help a program draw raster-graphics images independent of the type of dot-matrix printer or the number of pins or dots the printer can handle at one time.

Dot-Matrix Graphics

npins	Number of pins, *N*, in print-head
spinv	Spacing of pins vertically in pins per inch
spinh	Spacing of dots horizontally in dots per inch
porder	Matches software bits to print-head pins
sbim	Start printing bit image graphics, *B* bits wide
rbim	End printing bit image graphics

The sbim sring is used with a single argument, *B*, the width of the image in dots.

The model of dot-matrix or raster-graphics that terminfo presents is similar to the technique used for most dot-matrix printers: each pass of the printer's print-head is assumed to produce a dot-matrix that is *N* dots high and *B* dots wide. This is typically a wide, squat, rectangle of dots. The height of this rectangle in dots will vary from one printer to the next; this is given in the npins numeric capability. The size of the rectangle in fractions of an inch will also vary; it can be deduced from the spinv and spinh numeric capabilities. With these three values an application can divide a complete raster-graphics image into several horizontal strips, perhaps interpolating to account for different dot spacing vertically and horizontally.

The `sbim` and `rbim` strings are used to start and end a dot-matrix image, respectively. The `sbim` string is used with a single argument that gives the width of the dot-matrix in dots. A sequence of ''image-data bytes'' are sent to the printer after the `sbim` string and before the `rbim` string. The number of bytes is a integral multiple of the width of the dot-matrix; the multiple and the form of each byte is determined by the `porder` string as described below.

The `porder` string is a comma separated list of pin numbers optionally followed by an numerical offset. The offset, if given, is separated from the list with a semicolon. The position of each pin number in the list corresponds to a bit in an 8-bit data byte. The pins are numbered consecutively from 1 to `npins`, with 1 being the top pin. Note that the term ''pin'' is used loosely here; ''ink-jet'' dot-matrix printers don't have pins, but can be considered to have an equivalent method of applying a single dot of ink to paper. The bit positions in `porder` are in groups of 8, with the first position in each group the most significant bit and the last position the least significant bit. An application produces 8-bit bytes in the order of the groups in `porder`.

An application computes the ''image-data bytes'' from the internal image, mapping vertical dot positions in each print-head pass into 8-bit bytes, using a 1 bit where ink should be applied and 0 where no ink should be applied. This can be reversed (0 bit for ink, 1 bit for no ink) by giving a negative pin number. If a position is skipped in `porder`, a 0 bit is used. If a position has a lower case 'x' instead of a pin number, a 1 bit is used in the skipped position. For consistency, a lower case 'o' can be used to represent a 0 filled, skipped bit. There must be a multiple of 8 bit positions used or skipped in `porder`; if not, 0 bits are used to fill the last byte in the least significant bits. The offset, if given, is added to each data byte; the offset can be negative.

Some examples may help clarify the use of the `porder` string. The AT&T 470, AT&T 475 and C.Itoh 8510 printers provide eight pins for graphics. The pins are identified top to bottom by the 8 bits in a byte, from least significant to most. The `porder` strings for these printers would be `8,7,6,5,4,3,2,1`. The AT&T 478 and AT&T 479 printers also provide eight pins for graphics. However, the pins are identified in the reverse order. The `porder` strings for these printers would be `1,2,3,4,5,6,7,8`. The AT&T 5310, AT&T 5320, DEC LA100, and DEC LN03 printers provide six pins for graphics. The pins are identified top to bottom by the decimal values 1, 2, 4, 8, 16 and 32. These correspond to the low six bits in an 8-bit byte, although the decimal values are further offset by the value 63. The `porder` string for these printers would be `,,6,5,4,3,2,1;63`, or alternately `o,o,6,5,4,3,2,1;63`.

Section 2-7: Effect of Changing Printing Resolution

If the control sequences to change the character pitch or the line pitch are used, the pin or dot spacing may change:

Dot-Matrix Graphics
Changing the Character/Line Pitches

`cpi`	Change character pitch
`cpix`	If set, `cpi` changes `spinh`
`lpi`	Change line pitch
`lpix`	If set, `lpi` changes `spinv`

Programs that use `cpi` or `lpi` should recalculate the dot spacing:

Dot-Matrix Graphics
Effects of Changing the Character/Line Pitches

Before	*After*
Using `cpi` with `cpix` clear:	
spinh′	**spinh**
Using `cpi` with `cpix` set:	
spinh′	$\mathbf{spinh} = \mathbf{spinh'} \cdot \frac{\mathbf{orhi}}{\mathbf{orhi'}}$
Using `lpi` with `lpix` clear:	
spinv′	**spinv**
Using `lpi` with `lpix` set:	
spinv′	$\mathbf{spinv} = \mathbf{spinv'} \cdot \frac{\mathbf{orhi}}{\mathbf{orhi'}}$
Using `chr`:	
spinh′	**spinh**
Using `cvr`:	
spinv′	**spinv**

orhi′ and **orhi** are the values of the horizontal resolution in steps per inch, before using `cpi` and after using `cpi`, respectively. Likewise, **orvi′** and **orvi** are the values of the vertical resolution in steps per inch, before using `lpi` and after using `lpi`, respectively. Thus, the changes in the dots per inch for dot-matrix graphics follow the changes in steps per inch for printer resolution.

Section 2-8: Print Quality

Many dot-matrix printers can alter the dot spacing of printed text to produce near ‘‘letter quality’’ printing or ‘‘draft quality’’ printing. Usually it is important to be able to choose one or the other because the rate of printing generally falls off as the quality improves. There are three new strings used to describe these capabilities.

Print Quality	
snlq	Set near-letter quality print
snrmq	Set normal quality print
sdrfq	Set draft quality print

The capabilities are listed in decreasing levels of quality. If a printer doesn't have all three levels, one or two of the strings should be left blank as appropriate.

Section 2-9: Printing Rate and Buffer Size

Because there is no standard protocol that can be used to keep a program synchronized with a printer, and because modern printers can buffer data before printing it, a program generally cannot determine at any time what has been printed. Two new numeric capabilities can help a program estimate what has been printed.

Print Rate/Buffer Size	
cps	Nominal print rate in characters per second
bufsz	Buffer capacity in characters

`cps` is the nominal or average rate at which the printer prints characters; if this value is not given, the rate should be estimated at one-tenth the prevailing baud rate. `bufsz` is the maximum number of subsequent characters buffered before the guaranteed printing of an earlier character, assuming proper flow control has been used. If this value is not given it is assumed that the printer does not buffer characters, but prints them as they are received.

As an example, if a printer has a 1000-character buffer, then sending the letter ''a'' followed by 1000 additional characters is guaranteed to cause the letter ''a'' to print. If the same printer prints at the rate of 100 characters per second, then it should take 10 seconds to print all the characters in the buffer, less if the buffer is not full. By keeping track of the characters sent to a printer, and knowing the print rate and buffer size, a program can synchronize itself with the printer.

Note that most printer manufacturers advertise the maximum print rate, not the nominal print rate. A good way to get a value to put in for `cps` is to generate a few pages of text, count the number of printable characters, and then see how long it takes to print the text.

Applications that use these values should recognize the variability in the print rate. Straight text, in short lines, with no embedded control sequences will probably print at close to the advertised print rate and probably faster than the rate in `cps`. Graphics data with a lot of control sequences, or very long lines of text, will print at well below the advertised rate and below the rate in `cps`. If the application is using `cps` to decide how long it should take a printer to print a block of text, the application should pad the estimate. If the application is using `cps` to decide how much text has already been printed, it should shrink the estimate. The application will thus err in favor of the user, who wants, above all, to see all the output in its correct place.

FILES

`/usr/share/lib/terminfo/?/*` compiled terminal description database

`/usr/share/lib/.COREterm/?/*`	subset of compiled terminal description database
`/usr/share/lib/tabset/*`	tab settings for some terminals, in a format appropriate to be output to the terminal (escape sequences that set margins and tabs)

SEE ALSO

CURSES(TI_LIB), ls(BU_CMD), pg(BU_CMD), printf(BA_LIB), stty(AU_CMD), tic(TI_CMD), tput(TI_CMD), tty(AU_CMD), vi(AU_CMD).

USAGE

Administrator and Application Program.

The most effective way to prepare a terminal description is by imitating the description of a similar terminal in `terminfo` and to build up a description gradually, using partial descriptions with a screen oriented editor, such as `vi`, to check that they are correct. To easily test a new terminal description the environment variable `TERMINFO` can be set to the pathname of a directory containing the compiled description, and programs will look there rather than in `/usr/share/lib/terminfo`.

LEVEL

Level 1.

Terminal Interface Library Routines

The following section contains the manual pages for the TI_LIB routines.

NAME

curs_addch: addch, waddch, mvaddch, mvwaddch, echochar, wechochar – add a character (with attributes) to a CURSES window and advance cursor

SYNOPSIS

```
#include <curses.h>

int addch(chtype ch);

int waddch(WINDOW *win, chtype ch);

int mvaddch(int y, int x, chtype ch);

int mvwaddch(WINDOW *win, int y, int x, chtype ch);

int echochar(chtype ch);

int wechochar(WINDOW *win, chtype ch);
```

DESCRIPTION

With the `addch()`, `waddch()`, `mvaddch()` and `mvwaddch()` routines, the character *ch* is put into the window at the current cursor position of the window and the position of the window cursor is advanced. Its function is similar to that of `putchar()`. At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if `scrollok()` is enabled, the scrolling region is scrolled up one line.

If *ch* is a tab, newline, or backspace, the cursor is moved appropriately within the window. A newline also does a `clrtoeol()` before moving. Tabs are considered to be at every eighth column. If *ch* is another control character, it is drawn in the ^*X* notation. Calling `winch()` after adding a control character does not return the control character, but instead returns the representation of the control character.

Video attributes can be combined with a character by OR-ing them into the parameter. This results in these attributes also being set. (The intent here is that text, including attributes, can be copied from one place to another using `inch()` and `addch()`.) [see `standout()`, predefined video attribute constants, on the curs_attr(TI_LIB) page].

The `echochar()` and `wechochar()` routines are functionally equivalent to a call to `addch()` followed by a call to `refresh()`, or a call to `waddch()` followed by a call to `wrefresh()`. The knowledge that only a single character is being output is taken into consideration and, for non-control characters, a considerable performance gain might be seen by using these routines instead of their equivalents.

Line Graphics

The following variables may be used to add line drawing characters to the screen with routines of the `addch()` family. When variables are defined for the terminal, the A_ALTCHARSET bit is turned on [see curs_attr(TI_LIB)]. Otherwise, the default character listed below is stored in the variable. The names chosen are consistent with the VT100 nomenclature.

Name	*Default*	*Glyph Description*
ACS_ULCORNER	+	upper left-hand corner
ACS_LLCORNER	+	lower left-hand corner
ACS_URCORNER	+	upper right-hand corner
ACS_LRCORNER	+	lower right-hand corner
ACS_RTEE	+	right tee (┤)
ACS_LTEE	+	left tee (├)
ACS_BTEE	+	bottom tee (┴)
ACS_TTEE	+	top tee (┬)
ACS_HLINE	–	horizontal line
ACS_VLINE	|	vertical line
ACS_PLUS	+	plus
ACS_S1	–	scan line 1
ACS_S9	_	scan line 9
ACS_DIAMOND	+	diamond
ACS_CKBOARD	:	checker board (stipple)
ACS_DEGREE	'	degree symbol
ACS_PLMINUS	#	plus/minus
ACS_BULLET	o	bullet
ACS_LARROW	<	arrow pointing left
ACS_RARROW	>	arrow pointing right
ACS_DARROW	v	arrow pointing down
ACS_UARROW	^	arrow pointing up
ACS_BOARD	#	board of squares
ACS_LANTERN	#	lantern symbol
ACS_BLOCK	#	solid square block

RETURN VALUE

All routines return the integer `ERR` upon failure and an integer value other than `ERR` upon successful completion, unless otherwise noted in the preceding routine descriptions.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that `addch()`, `mvaddch()`, `mvwaddch()`, and `echochar()` may be macros.

SEE ALSO

CURSES(TI_ENV), curs_attr(TI_LIB), curs_clear(TI_LIB), curs_inch(TI_LIB), curs_outopts(TI_LIB), curs_refresh(TI_LIB) putc(BA_LIB).

LEVEL

Level 1.

NAME

curs_addchstr: addchstr, addchnstr, waddchstr, waddchnstr, mvaddchstr, mvaddchnstr, mvwaddchstr, mvwaddchnstr – add string of characters (and attributes) to a CURSES window

SYNOPSIS

```
#include <curses.h>

int addchstr(chtype *chstr);

int addchnstr(chtype *chstr, int n);

int waddchstr(WINDOW *win, chtype *chstr);

int waddchnstr(WINDOW *win, chtype *chstr, int n);

int mvaddchstr(int y, int x, chtype *chstr);

int mvaddchnstr(int y, int x, chtype *chstr, int n);

int mvwaddchstr(WINDOW *win, int y, int x, chtype *chstr);

int mvwaddchnstr(WINDOW *win, int y, int x,
     chtype *chstr, int n);
```

DESCRIPTION

All of these routines copy *chstr* directly into the window image structure starting at the current cursor position. The four routines with *n* as the last argument copy at most *n* elements, but no more than will fit on the line. If `n=-1` then the whole string is copied, to the maximum number that fit on the line.

The position of the window cursor is **NOT** advanced. These routines works faster than `waddnstr()` because they merely copy *chstr* into the window image structure. On the other hand, care must be taken when using these functions because they don't perform any kind of checking (such as for the newline character), they don't advance the current cursor position, and they truncate the string, rather then wrapping it around to the new line.

RETURN VALUE

All routines return the integer `ERR` upon failure and an integer value other than `ERR` upon successful completion, unless otherwise noted in the preceding routine descriptions.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that all routines except `waddchnstr()` may be macros.

SEE ALSO

CURSES(TI_ENV).

LEVEL

Level 1.

NAME

curs_addstr: addstr, addnstr, waddstr, waddnstr, mvaddstr, mvaddnstr, mvwaddstr, mvwaddnstr – add a string of characters to a CURSES window and advance cursor

SYNOPSIS

```
#include <curses.h>

int addstr(char *str);

int addnstr(char *str, int n);

int waddstr(WINDOW *win, char *str);

int waddnstr(WINDOW *win, char *str, int n);

int mvaddstr(int y, int x, char *str);

int mvaddnstr(int y, int x, char *str, int n);

int mvwaddstr(WINDOW *win, int y, int x, char *str);

int mvwaddnstr(WINDOW *win, int y, int x, char *str,
     int n);
```

DESCRIPTION

All of these routines write all the characters of the null terminated character string *str* on the given window. It is similar to calling `waddch()` once for each character in the string. The four routines with *n* as the last argument write at most *n* characters. If *n* is negative, then the entire string will be added.

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that all of these routines except `waddstr()` and `waddnstr()` may be macros.

SEE ALSO

CURSES(TI_ENV), curs_addch(TI_LIB).

LEVEL

Level 1.

NAME

curs_addwch: addwch, waddwch, mvaddwch, mvwaddwch, echowchar, wechowchar – add a wchar_t character (with attributes) to a CURSES window and advance cursor

SYNOPSIS

```
#include <curses.h>

int addwch(chtype wch);
int waddwch(WINDOW *win, chtype wch);
int mvaddwch(int y, int x, chtype wch);
int mvwaddwch(WINDOW *win, int y, int x, chtype wch);
int echowchar(chtype wch);
int wechowchar(WINDOW *win, chtype wch);
```

DESCRIPTION

The **addwch()**, **waddwch()**, **mvaddwch()** and **mvwaddwch()** routines put the character *wch*, holding a **wchar_t** character, into the window at the current cursor position of the window and advance the position of the window cursor. At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if **scrollok()** is enabled, the scrolling region is scrolled up one line.

If *wch* is a tab, newline, or backspace, the cursor is moved appropriately within the window. A newline also does a **clrtoeol** before moving. Tabs are considered to be at every eighth column. If *wch* is another control character, it is drawn in the ^*X* notation. Calling **winwch()** after adding a control character does not return the control character, but instead returns the representation of the control character.

Video attributes can be combined with a **wchar_t** character by OR-ing them into the parameter. This results in these attributes also being set. (The intent here is that text, including attributes, can be copied from one place to another using **inwch()** and **addwch()**.) [See **standout()**, predefined video attribute constants, on the curs_attr(TI_LIB) page].

The **echowchar()** and **wechowchar()** routines are functionally equivalent to a call to **addwch()** followed by a call to **refresh()**, or a call to **waddwch()** followed by a call to **wrefresh()**. The knowledge that only a single character is being output is taken into consideration and, for non-control characters, a considerable performance gain might be seen by using these routines instead of their equivalents.

Line Graphics

The following variables may be used to add line drawing characters to the screen with routines of the **addwch()** family. When variables are defined for the terminal, the **A_ALTCHARSET** bit is turned on [see curs_attr(TI_LIB)]. Otherwise, the default character listed below is stored in the variable. The names chosen are consistent with the VT100 nomenclature.

Name	*Default*	*Glyph Description*
ACS_ULCORNER	+	upper left-hand corner
ACS_LLCORNER	+	lower left-hand corner
ACS_URCORNER	+	upper right-hand corner
ACS_LRCORNER	+	lower right-hand corner
ACS_RTEE	+	right tee (┤)
ACS_LTEE	+	left tee (├)
ACS_BTEE	+	bottom tee (┴)
ACS_TTEE	+	top tee (┬)
ACS_HLINE	–	horizontal line
ACS_VLINE	|	vertical line
ACS_PLUS	+	plus
ACS_S1	–	scan line 1
ACS_S9	_	scan line 9
ACS_DIAMOND	+	diamond
ACS_CKBOARD	:	checker board (stipple)
ACS_DEGREE	'	degree symbol
ACS_PLMINUS	#	plus/minus
ACS_BULLET	o	bullet
ACS_LARROW	<	arrow pointing left
ACS_RARROW	>	arrow pointing right
ACS_DARROW	v	arrow pointing down
ACS_UARROW	^	arrow pointing up
ACS_BOARD	#	board of squares
ACS_LANTERN	#	lantern symbol
ACS_BLOCK	#	solid square block

RETURN VALUE

All routines return the integer **ERR** upon failure and an integer value other than **ERR** upon successful completion, unless otherwise noted in the preceding routine descriptions.

USAGE

Application Program.

The header file **<curses.h>** automatically includes the header files **<stdio.h>** and **<unctrl.h>**.

Note that **addwch()**, **mvaddwch()**, **mvwaddwch()**, and **echowchar()** may be macros.

SEE ALSO

CURSES(TI_ENV), curs_attr(TI_LIB), curs_clear(TI_LIB), curs_inwch(TI_LIB), curs_outopts(TI_LIB), curs_refresh(TI_LIB).

LEVEL

Level 1.

NAME

curs_addwchstr: addwchstr, addwchnstr, waddwchstr, waddwchnstr, mvaddwchstr, mvaddwchnstr, mvwaddwchstr, mvwaddwchnstr – add string of **wchar_t** characters (and attributes) to a CURSES window

SYNOPSIS

#include <curses.h>

int addwchstr(chtype ***_wchstr_);**

int addwchnstr(chtype ***_wchstr_, int** _n_**);**

int waddwchstr(WINDOW ***_win_, chtype ***_wchstr_**);**

int waddwchnstr(WINDOW ***_win_, chtype ***_wchstr_**, int** _n_**);**

int mvaddwchstr(int _y_**, int** _x_**, chtype ***_wchstr_**);**

int mvaddwchnstr(int _y_**, int** _x_**, chtype ***_wchstr_**, int** _n_**);**

int mvwaddwchstr(WINDOW ***_win_, int** _y_**, int** _x_**, chtype ***_wchstr_**);**

int mvwaddwchnstr(WINDOW ***_win_, int** _y_**, int** _x_**, chtype ***_wchstr_**, int** _n_**);**

DESCRIPTION

All of these routines copy _wchstr_, which points to a string of **wchar_t** characters, directly into the window image structure starting at the current cursor position. The four routines with _n_ as the last argument copy at most _n_ elements, but no more than will fit on the line. If _n_=**-1**, then the whole string is copied, to the maximum number that fit on the line.

The position of the window cursor is not advanced. These routines work faster than **waddnwstr()** because they merely copy _wchstr_ into the window image structure. On the other hand, care must be taken when using these functions because they don't perform any kind of checking (such as for the newline character), they don't advance the current cursor position, and they truncate the string, rather than wrapping it around to the new line.

RETURN VALUE

All routines return the integer **ERR** upon failure and an integer value other than **ERR** upon successful completion, unless otherwise noted in the preceding routine descriptions.

USAGE

Application Program.

The header file **<curses.h>** automatically includes the header files **<stdio.h>** and **<unctrl.h>**.

Note that all routines except **waddwchnstr()** may be macros.

SEE ALSO

CURSES(TI_ENV).

LEVEL

Level 1.

NAME

curs_addwstr: addwstr, addnwstr, waddwstr, waddnwstr, mvaddwstr, mvaddnwstr, mvwaddwstr, mvwaddnwstr – add a string of **wchar_t** characters to a CURSES window and advance cursor

SYNOPSIS

#include <curses.h>

int addwstr(wchar_t **wstr*);

int addnwstr(wchar_t **wstr*, **int** *n*);

int waddwstr(WINDOW **win*, **wchar_t** **wstr*);

int waddnwstr(WINDOW **win*, **wchar_t** **wstr*, **int** *n*);

int mvaddwstr(*y*, **int** *x*, **wchar_t** **wstr*);

int mvaddnwstr(*y*, **int** *x*, **wchar_t** **wstr*, **int** *n*);

int mvwaddwstr(WINDOW **win*, **int** *y*, **int** *x*, **wchar_t** **wstr*);

int mvwaddnwstr(WINDOW **win*, **int** *y*, **int** *x*, **wchar_t** **wstr*, **int** *n*);

DESCRIPTION

All of these routines write all the characters of the null-terminated **wchar_t** character string *str* on the given window. The effect is similar to calling **waddwch()** once for each **wchar_t** character in the string. The four routines with *n* as the last argument write at most *n* **wchar_t** characters. If *n* is negative, then the entire string will be added.

RETURN VALUE

All routines return the integer **ERR** upon failure and an integer value other than **ERR** upon successful completion.

USAGE

Application Program.

The header file **<curses.h>** automatically includes the header files **<stdio.h>** and **<unctrl.h>**.

Note that all of these routines except **waddwstr()** and **waddnwstr()** may be macros.

SEE ALSO

CURSES(TI_ENV), curs_addwch(TI_LIB).

LEVEL

Level 1.

NAME

curs_attr: attroff, wattroff, attron, wattron, attrset, wattrset, standend, wstandend, standout, wstandout – CURSES character and window attribute control routines

SYNOPSIS

```
#include <curses.h>

int attroff(chtype attrs);
int wattroff(WINDOW *win, chtype attrs);
int attron(chtype attrs);
int wattron(WINDOW *win, chtype attrs);
int attrset(chtype attrs);
int wattrset(WINDOW *win, chtype attrs);
int standend(void);
int wstandend(WINDOW *win);
int standout(void);
int wstandout(WINDOW *win);
```

DESCRIPTION

All of these routines manipulate the current attributes of the named window. The current attributes of a window are applied to all characters that are written into the window with `waddch()`, `waddstr()` and `wprintw()`. Attributes are a property of the character, and move with the character through any scrolling and insert/delete line/character operations. To the extent possible on the particular terminal, they are displayed as the graphic rendition of characters put on the screen.

The routine `attrset()` sets the current attributes of the given window to *attrs*. The routine `attroff()` turns off the named attributes without turning any other attributes on or off. The routine `attron()` turns on the named attributes without affecting any others. The routine `standout()` is the same as `attron(A_STANDOUT)`. The routine `standend()` is the same as `attrset(0)`, that is, it turns off all attributes.

Attributes

The following video attributes, defined in `<curses.h>`, can be passed to the routines `attron()`, `attroff()`, and `attrset()`, or OR-ed with the characters passed to `addch()`.

`A_STANDOUT`	Best highlighting mode of the terminal.
`A_UNDERLINE`	Underlining
`A_REVERSE`	Reverse video
`A_BLINK`	Blinking
`A_DIM`	Half bright
`A_BOLD`	Extra bright or bold
`A_ALTCHARSET`	Alternate character set
`A_CHARTEXT`	Bit-mask to extract a character
`COLOR_PAIR(`*n*`)`	Color-pair number *n*

The following macro is the reverse of `COLOR_PAIR(`*n*`)`:

`PAIR_NUMBER(`*attrs*`)`	Returns the pair number associated with the `COLOR_PAIR(`*n*`)` attribute.

RETURN VALUE

These routines always return `1`.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that `attroff()`, `wattroff()`, `attron()`, `wattron()`, `attrset()`, `wattrset()`, `standend()` and `standout()` may be macros.

SEE ALSO

CURSES(TI_ENV), curs_addch(TI_LIB), curs_addstr(TI_LIB), curs_printw(TI_LIB).

LEVEL

Level 1.

NAME

curs_beep: beep, flash – CURSES bell and screen flash routines

SYNOPSIS

```
#include <curses.h>

int beep(void);

int flash(void);
```

DESCRIPTION

The `beep()` and `flash()` routines are used to signal the terminal user. The routine `beep()` sounds the audible alarm on the terminal, if possible; if that is not possible, it flashes the screen (visible bell), if that is possible. The routine `flash()` flashes the screen, and if that is not possible, sounds the audible signal. If neither signal is possible, nothing happens. Nearly all terminals have an audible signal (bell or beep), but only some can flash the screen.

RETURN VALUE

These routines always return `OK`.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

SEE ALSO

CURSES(TI_ENV).

LEVEL

Level 1.

NAME

curs_bkgd: bkgdset, wbkgdset, bkgd, wbkgd – CURSES window background manipulation routines

SYNOPSIS

```
#include <curses.h>

void bkgdset(chtype ch);

void wbkgdset(WINDOW *win, chtype ch);

int bkgd(chtype ch);

int wbkgd(WINDOW *win, chtype ch);
```

DESCRIPTION

The `bkgdset()` and `wbkgdset()` routines manipulate the background of the named window. Background is a `chtype` consisting of any combination of attributes and a character. The attribute part of the background is combined (ORed) with all non-blank characters that are written into the window with `waddch()`. Both the character and attribute parts of the background are combined with the blank characters. The background becomes a property of the character and moves with the character through any scrolling and insert/delete line/character operations. To the extent possible on a particular terminal, the attribute part of the background is displayed as the graphic rendition of the character put on the screen.

The `bkgd()` and `wbkgd()` routines combine the new background with every position in the window. Background is any combination of attributes and a character. Only the attribute part is used to set the background of non-blank characters, while both character and attributes are used for blank positions. To the extent possible on a particular terminal, the attribute part of the background is displayed as the graphic rendition of the character put on the screen.

RETURN VALUE

`bkgd()` and `wbkgd()` return the integer `OK`, or a non-negative integer, if `immedok()` is set.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that `bkgdset()` and `bkgd()` may be macros.

SEE ALSO

CURSES(TI_LIB), curs_addch(TI_LIB), curs_outopts(TI_LIB).

LEVEL

Level 1

NAME

curs_border: border, wborder, box, whline, wvline – create CURSES borders, horizontal and vertical lines

SYNOPSIS

```
#include <curses.h>

int border(chtype ls, chtype rs, chtype ts, chtype bs,
    chtype tl, chtype tr, chtype bl, chtype br);

int wborder(WINDOW *win, chtype ls, chtype rs,
    chtype ts, chtype bs, chtype tl, chtype tr,
    chtype bl, chtype br);

int box(WINDOW *win, chtype verch, chtype horch);

int hline(chtype ch, int n);

int whline(WINDOW *win, chtype ch, int n);

int vline(chtype ch, int n);

int wvline(WINDOW *win, chtype ch, int n);
```

DESCRIPTION

With the `border()`, `wborder()` and `box()` routines, a border is drawn around the edges of the window. The argument *ls* is a character and attributes used for the left side of the border, *rs* - right side, *ts* - top side, *bs* - bottom side, *tl* - top left-hand corner, *tr* - top right-hand corner, *bl* - bottom left-hand corner, and *br* - bottom right-hand corner. If any of these arguments is zero, then the following default values (defined in `<curses.h>`) are used instead: `ACS_VLINE`, `ACS_VLINE`, `ACS_HLINE`, `ACS_HLINE`, `ACS_ULCORNER`, `ACS_URCORNER`, `ACS_BLCORNER`, `ACS_BRCORNER`.

`box(`*win*`, `*verch*`, `*horch*`)` is a shorthand for the following call: `wborder(`*win*`, `*verch*`, `*verch*`, `*horch*`, `*horch*`, 0, 0, 0, 0)`.

`hline()` and `whline()` draw a horizontal (left to right) line using *ch* starting at the current cursor position in the window. The current cursor position is not changed. The line is at most *n* characters long, or as many as fit into the window.

`vline()` and `wvline()` draw a vertical (top to bottom) line using *ch* starting at the current cursor position in the window. The current cursor position is not changed. The line is at most *n* characters long, or as many as fit into the window.

RETURN VALUE

All routines return the integer `OK`, or a non-negative integer if `immedok()` is set.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that `border()` and `box()` may be macros.

SEE ALSO

CURSES(TI_ENV), curs_outopts(TI_LIB).

LEVEL

Level 1.

NAME

curs_clear: erase, werase, clear, wclear, clrtobot, wclrtobot, clrtoeol, wclrtoeol – clear all or part of a CURSES window

SYNOPSIS

```
# include <curses.h>

int erase(void);
int werase(WINDOW *win);
int clear(void);
int wclear(WINDOW *win);
int clrtobot(void);
int wclrtobot(WINDOW *win);
int clrtoeol(void);
int wclrtoeol(WINDOW *win);
```

DESCRIPTION

The `erase()` and `werase()` routines copy blanks to every position in the window.

The `clear()` and `wclear()` routines are like `erase()` and `werase()`, but they also call `clearok()`, so that the screen is cleared completely on the next call to `wrefresh()` for that window and repainted from scratch.

The `clrtobot()` and `wclrtobot()` routines erase all lines below the cursor in the window. Also, the current line to the right of the cursor, inclusive, is erased.

The `clrtoeol()` and `wclrtoeol()` routines erase the current line to the right of the cursor, inclusive.

RETURN VALUE

All routines return the integer `OK`, or a non-negative integer if `immedok()` is set.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that `erase()`, `werase()`, `clear()`, `wclear()`, `clrtobot()`, and `clrtoeol()` may be macros.

SEE ALSO

CURSES(TI_ENV), curs_outopts(TI_LIB), curs_refresh(TI_LIB).

LEVEL

Level 1.

NAME

curs_color: start_color, init_pair, init_color, has_colors, can_change_color, color_content, pair_content – CURSES color manipulation routines

SYNOPSIS

```
# include <curses.h>

int start_color(void);
int init_pair(short pair, short f, short b);
int init_color(short color, short r, short g, short b);
bool has_colors(void);
bool can_change_color(void);
int color_content(short color, short *r, short *g, short *b);
int pair_content(short pair, short *f, short *b);
```

DESCRIPTION

Overview

CURSES provides routines that manipulate color on color alphanumeric terminals. To use these routines `start_color()` must be called, usually right after `initscr()`. Colors are always used in pairs (referred to as color-pairs). A color-pair consists of a foreground color (for characters) and a background color (for the field on which the characters are displayed). A programmer initializes a color-pair with the routine `init_pair()`. After it has been initialized, `COLOR_PAIR`(*n*), a macro defined in `<curses.h>`, can be used in the same ways other video attributes can be used. If a terminal is capable of redefining colors, the programmer can use the routine `init_color()` to change the definition of a color. The routines `has_colors()` and `can_change_color()` return `TRUE` or `FALSE`, depending on whether the terminal has color capabilities and whether the programmer can change the colors. The routine `color_content()` allows a programmer to identify the amounts of red, green, and blue components in an initialized color. The routine `pair_content()` allows a programmer to find out how a given color-pair is currently defined.

Routine Descriptions

The `start_color()` routine requires no arguments. It must be called if the programmer wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after `initscr()`. `start_color()` initializes eight basic colors (black, blue, green, cyan, red, magenta, yellow, and white), and two global variables, `COLORS` and `COLOR_PAIRS` (respectively defining the maximum number of colors and color-pairs the terminal can support). It also restores the colors on the terminal to the values they had when the terminal was just turned on.

The `init_pair()` routine changes the definition of a color-pair. It takes three arguments: the number of the color-pair to be changed, the foreground color number, and the background color number. The value of the first argument must be between `1` and the smaller of 63 and `COLOR_PAIRS-1`. The value of the second and third arguments must be between 0 and `COLORS`. If the color-pair was previously initialized, the screen is refreshed and all occurrences of that color-pair is changed to the new definition.

The init_color() routine changes the definition of a color. It takes four arguments: the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of the first argument must be between 0 and COLORS. (See the section **Colors** for the default color index.) Each of the last three arguments must be a value between 0 and 1000. When init_color() is used, all occurrences of that color on the screen immediately change to the new definition.

The has_colors() routine requires no arguments. It returns TRUE if the terminal can manipulate colors; otherwise, it returns FALSE. This routine facilitates writing terminal-independent programs. For example, a programmer can use it to decide whether to use color or some other video attribute.

The can_change_color() routine requires no arguments. It returns TRUE if the terminal supports colors and can change their definitions; other, it returns FALSE. This routine facilitates writing terminal-independent programs.

The color_content() routine gives users a way to find the intensity of the red, green, and blue (RGB) components in a color. It requires four arguments: the color number, and three addresses of shorts for storing the information about the amounts of red, green, and blue components in the given color. The value of the first argument must be between 0 and COLORS. The values that are stored at the addresses pointed to by the last three arguments are between 0 (no component) and 1000 (maximum amount of component).

The pair_content() routine allows users to find out what colors a given color-pair consists of. It requires three arguments: the color-pair number, and two addresses of shorts for storing the foreground and the background color numbers. The value of the first argument must be between 1 and the smaller of 63 and and COLOR_PAIRS-1. The values that are stored at the addresses pointed to by the second and third arguments are between 0 and COLORS.

Colors

In <curses.h> the following macros are defined. These are the default colors. CURSES also assumes that COLOR_BLACK is the default background color for all terminals.

```
COLOR_BLACK
COLOR_BLUE
COLOR_GREEN
COLOR_CYAN
COLOR_RED
COLOR_MAGENTA
COLOR_YELLOW
COLOR_WHITE
```

RETURN VALUE

All routines that return an integer return ERR upon failure and OK upon successful completion.

USAGE

Application Program.

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

SEE ALSO

CURSES(TI_ENV), curs_initscr(TI_LIB), curs_attr(TI_LIB).

LEVEL

Level 1.

NAME

curs_delch: delch, wdelch, mvdelch, mvwdelch – delete character under cursor in a CURSES window.

SYNOPSIS

```
#include <curses.h>

int delch(void);
int wdelch(WINDOW *win);
int mvdelch(int y, int x);
int mvwdelch(WINDOW *win, int y, int x);
```

DESCRIPTION

With these routines the character under the cursor in the window is deleted; all characters to the right of the cursor on the same line are moved to the left one position and the last character on the line is filled with a blank. The cursor position does not change (after moving to *y*, *x*, if specified). (This does not imply use of the hardware delete character feature.)

RETURN VALUE

All routines return the integer `ERR` upon failure and an integer value other than `ERR` upon successful completion.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that `delch()`, `mvdelch()`, and `mvwdelch()` may be macros.

SEE ALSO

CURSES(TI_ENV).

LEVEL

Level 1.

NAME

curs_deleteln: deleteln, wdeleteln, insdelln, winsdelln, insertln, winsertln – delete and insert lines in a CURSES window

SYNOPSIS

```
#include <curses.h>

int deleteln(void);
int wdeleteln(WINDOW *win);
int insdelln(int n);
int winsdelln(WINDOW *win, int n);
int insertln(void);
int winsertln(WINDOW *win);
```

DESCRIPTION

With the `deleteln()` and `wdeleteln()` routines, the line under the cursor in the window is deleted; all lines below the current line are moved up one line. The bottom line of the window is cleared. The cursor position does not change. (This does not imply use of a hardware delete line feature.)

With the `insdelln()` and `winsdelln()` routines, for positive *n*, insert *n* lines into the specified window above the current line. The *n* bottom lines are lost. For negative *n*, delete *n* lines (starting with the one under the cursor), and move the remaining lines up. The bottom *n* lines are cleared. The current cursor position remains the same.

With the `insertln()` and `insertln()` routines, a blank line is inserted above the current line and the bottom line is lost. (This does not imply use of a hardware insert line feature.)

RETURN VALUE

All routines return the integer `ERR` upon failure and an integer value other than `ERR` upon successful completion.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that all but `winsdelln()` may be a macros.

SEE ALSO

CURSES(TI_ENV).

LEVEL

Level 1.

NAME

curs_getch: getch, wgetch, mvgetch, mvwgetch, ungetch – get (or push back) characters from CURSES terminal keyboard

SYNOPSIS

```
#include <curses.h>

int getch(void);

int wgetch(WINDOW *win);

int mvgetch(int y, int x);

int mvwgetch(WINDOW *win, int y, int x);

int ungetch(int ch);
```

DESCRIPTION

With the `getch()`, `wgetch()`, `mvgetch()` and `mvwgetch()`, routines a character is read from the terminal associated with the window. In no-delay mode, if no input is waiting, the value `ERR` is returned. In delay mode, the program waits until the system passes text through to the program. Depending on the setting of `cbreak()`, this is after one character (cbreak mode), or after the first newline (nocbreak mode). In half-delay mode, the program waits until a character is typed or the specified timeout has been reached. Unless `noecho()` has been set, the character will also be echoed into the designated window.

If the window is not a pad, and it has been moved or modified since the last call to `wrefresh()`, `wrefresh()` will be called before another character is read.

If `keypad()` is `TRUE`, and a function key is pressed, the token for that function key is returned instead of the raw characters. Possible function keys are defined in `<curses.h>` with integers beginning with `0401`, whose names begin with `KEY_`. If a character that could be the beginning of a function key (such as escape) is received, CURSES sets a timer. If the remainder of the sequence does not come in within the designated time, the character is passed through; otherwise, the function key value is returned. For this reason, many terminals experience a delay between the time a user presses the escape key and the escape is returned to the program. Since tokens returned by these routines are outside the ASCII range, they are not printable.

The `ungetch()` routine places *ch* back onto the input queue to be returned by the next call to `wgetch()`.

Function Keys

The following function keys, defined in `<curses.h>`, might be returned by `getch()` if `keypad()` has been enabled. Note that not all of these may be supported on a particular terminal if the terminal does not transmit a unique code when the key is pressed or if the definition for the key is not present in the *terminfo* database.

Name	*Key name*
KEY_BREAK	Break key
KEY_DOWN	The four arrow keys ...
KEY_UP	
KEY_LEFT	
KEY_RIGHT	
KEY_HOME	Home key (upward+left arrow)
KEY_BACKSPACE	Backspace
KEY_F0	Function keys; space for 64 keys is reserved.
KEY_F(*n*)	For $0 \leq n \leq 63$
KEY_DL	Delete line
KEY_IL	Insert line
KEY_DC	Delete character
KEY_IC	Insert char or enter insert mode
KEY_EIC	Exit insert char mode
KEY_CLEAR	Clear screen
KEY_EOS	Clear to end of screen
KEY_EOL	Clear to end of line
KEY_SF	Scroll 1 line forward
KEY_SR	Scroll 1 line backward (reverse)
KEY_NPAGE	Next page
KEY_PPAGE	Previous page
KEY_STAB	Set tab
KEY_CTAB	Clear tab
KEY_CATAB	Clear all tabs
KEY_ENTER	Enter or send
KEY_SRESET	Soft (partial) reset
KEY_RESET	Reset or hard reset
KEY_PRINT	Print or copy

KEY_LL	Home down or bottom (lower left). Keypad is arranged like this:
	`A1 up A3`
	`left B2 right`
	`C1 down C3`

KEY_A1	Upper left of keypad
KEY_A3	Upper right of keypad
KEY_B2	Center of keypad
KEY_C1	Lower left of keypad
KEY_C3	Lower right of keypad
KEY_BTAB	Back tab key
KEY_BEG	Beg(inning) key
KEY_CANCEL	Cancel key
KEY_CLOSE	Close key
KEY_COMMAND	Cmd (command) key
KEY_COPY	Copy key
KEY_CREATE	Create key

Name	*Key name*
KEY_END	End key
KEY_EXIT	Exit key
KEY_FIND	Find key
KEY_HELP	Help key
KEY_MARK	Mark key
KEY_MESSAGE	Message key
KEY_MOVE	Move key
KEY_NEXT	Next object key
KEY_OPEN	Open key
KEY_OPTIONS	Options key
KEY_PREVIOUS	Previous object key
KEY_REDO	Redo key
KEY_REFERENCE	Ref(erence) key
KEY_REFRESH	Refresh key
KEY_REPLACE	Replace key
KEY_RESTART	Restart key
KEY_RESUME	Resume key
KEY_SAVE	Save key
KEY_SBEG	Shifted beginning key
KEY_SCANCEL	Shifted cancel key
KEY_SCOMMAND	Shifted command key
KEY_SCOPY	Shifted copy key
KEY_SCREATE	Shifted create key
KEY_SDC	Shifted delete char key
KEY_SDL	Shifted delete line key
KEY_SELECT	Select key
KEY_SEND	Shifted end key
KEY_SEOL	Shifted clear line key
KEY_SEXIT	Shifted exit key
KEY_SFIND	Shifted find key
KEY_SHELP	Shifted help key
KEY_SHOME	Shifted home key
KEY_SIC	Shifted input key
KEY_SLEFT	Shifted left arrow key
KEY_SMESSAGE	Shifted message key
KEY_SMOVE	Shifted move key
KEY_SNEXT	Shifted next key
KEY_SOPTIONS	Shifted options key
KEY_SPREVIOUS	Shifted prev key
KEY_SPRINT	Shifted print key
KEY_SREDO	Shifted redo key
KEY_SREPLACE	Shifted replace key
KEY_SRIGHT	Shifted right arrow
KEY_SRSUME	Shifted resume key
KEY_SSAVE	Shifted save key

Name	*Key name*
KEY_SSUSPEND	Shifted suspend key
KEY_SUNDO	Shifted undo key
KEY_SUSPEND	Suspend key
KEY_UNDO	Undo key

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

USAGE

Application Program.

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Use of the escape key by a programmer for a single character function is discouraged.

When using `getch()`, `wgetch()`, `mvgetch()`, or `mvwgetch()`, nocbreak mode (`nocbreak()`) and echo mode (`echo()`) should not be used at the same time. Depending on the state of the tty driver when each character is typed, the program may produce undesirable results.

Note that `getch()`, `mvgetch()`, and `mvwgetch()` may be macros.

SEE ALSO

CURSES(TI_ENV), curs_inopts(TI_LIB), curs_move(TI_LIB), curs_refresh(TI_LIB).

LEVEL

Level 1.

NAME

curs_getstr: getstr, wgetstr, mvgetstr, mvwgetstr, wgetnstr – get character strings from CURSES terminal keyboard

SYNOPSIS

```
#include <curses.h>

int getstr(char *str);

int wgetstr(WINDOW *win, char *str);

int mvgetstr(int y, int x, char *str);

int mvwgetstr(WINDOW *win, int y, int x, char *str);

int wgetnstr(WINDOW *win, char *str, int n);
```

DESCRIPTION

The effect of `getstr()` is as though a series of calls to `getch()` were made, until a newline or carriage return is received. The resulting value is placed in the area pointed to by the character pointer *str*. `wgetnstr()` reads at most *n* characters, thus preventing a possible overflow of the input buffer. The user's erase and kill characters are interpreted, as well as any special keys (such as function keys, "home" key, "clear" key, *etc.*).

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that `getstr()`, `mvgetstr()`, and `mvwgetstr()` may be macros.

SEE ALSO

CURSES(TI_ENV), curs_getch(TI_LIB).

LEVEL

Level 1.

NAME

curs_getwch: getwch, wgetwch, mvgetwch, mvwgetwch, ungetwch – get (or push back) **wchar_t** characters from CURSES terminal keyboard

SYNOPSIS

```
#include <curses.h>

int getwch(void);

int wgetwch(WINDOW *win);

int mvgetwch(int y, int x);

int mvwgetwch(WINDOW *win, int y, int x);

int ungetwch(int wch);
```

DESCRIPTION

The **getwch()**, **wgetwch()**, **mvgetwch()** and **mvwgetwch()** routines read an *EUC* character from the terminal associated with the window, transform it into a **wchar_t** character, and return a **wchar_t** character. In no-delay mode, if no input is waiting, the value **ERR** is returned. In delay mode, the program waits until the system passes text through to the program. Depending on the setting of **cbreak()**, this is after one character (**cbreak()** mode), or after the first newline (**nocbreak()** mode). In half-delay mode, the program waits until a character is typed or the specified timeout has been reached. Unless **noecho()** has been set, the character will also be echoed into the designated window.

If the window is not a pad, and it has been moved or modified since the last call to **wrefresh()**, **wrefresh()** will be called before another character is read.

If **keypad()** is **TRUE**, and a function key is pressed, the token for that function key is returned instead of the raw characters. Possible function keys are defined in **<curses.h>** with integers beginning with **0401**, whose names begin with **KEY_.** If a character that could be the beginning of a function key (such as escape) is received, **curses** sets a timer. If the remainder of the sequence does not come in within the designated time, the character is passed through; otherwise, the function key value is returned. For this reason, many terminals experience a delay between the time a user presses the escape key and the escape is returned to the program.

The **ungetwch()** routine places *wch* back onto the input queue to be returned by the next call to **wgetwch()**.

Function Keys

The following function keys, defined in **<curses.h>**, might be returned by **getwch()** if **keypad()** has been enabled. Note that not all of these may be supported on a particular terminal if the terminal does not transmit a unique code when the key is pressed or if the definition for the key is not present in the **terminfo** database.

Name	*Key name*
KEY_BREAK	Break key
KEY_DOWN	The four arrow keys ...
KEY_UP	
KEY_LEFT	
KEY_RIGHT	
KEY_HOME	Home key (upward+left arrow)
KEY_BACKSPACE	Backspace
KEY_F0	Function keys; space for 64 keys is reserved.
KEY_F(*n*)	For $0 \le n \le 63$
KEY_DL	Delete line
KEY_IL	Insert line
KEY_DC	Delete character
KEY_IC	Insert char or enter insert mode
KEY_EIC	Exit insert char mode
KEY_CLEAR	Clear screen
KEY_EOS	Clear to end of screen
KEY_EOL	Clear to end of line
KEY_SF	Scroll 1 line forward
KEY_SR	Scroll 1 line backward (reverse)
KEY_NPAGE	Next page
KEY_PPAGE	Previous page
KEY_STAB	Set tab
KEY_CTAB	Clear tab
KEY_CATAB	Clear all tabs
KEY_ENTER	Enter or send
KEY_SRESET	Soft (partial) reset
KEY_RESET	Reset or hard reset
KEY_PRINT	Print or copy
KEY_LL	Home down or bottom (lower left). Keypad is arranged like this: **A1 up A3** **left B2 right** **C1 down C3**
KEY_A1	Upper left of keypad
KEY_A3	Upper right of keypad
KEY_B2	Center of keypad
KEY_C1	Lower left of keypad
KEY_C3	Lower right of keypad
KEY_BTAB	Back tab key
KEY_BEG	Beg(inning) key
KEY_CANCEL	Cancel key
KEY_CLOSE	Close key
KEY_COMMAND	Cmd (command) key
KEY_COPY	Copy key
KEY_CREATE	Create key

Name	*Key name*
KEY_END	End key
KEY_EXIT	Exit key
KEY_FIND	Find key
KEY_HELP	Help key
KEY_MARK	Mark key
KEY_MESSAGE	Message key
KEY_MOVE	Move key
KEY_NEXT	Next object key
KEY_OPEN	Open key
KEY_OPTIONS	Options key
KEY_PREVIOUS	Previous object key
KEY_REDO	Redo key
KEY_REFERENCE	Ref(erence) key
KEY_REFRESH	Refresh key
KEY_REPLACE	Replace key
KEY_RESTART	Restart key
KEY_RESUME	Resume key
KEY_SAVE	Save key
KEY_SBEG	Shifted beginning key
KEY_SCANCEL	Shifted cancel key
KEY_SCOMMAND	Shifted command key
KEY_SCOPY	Shifted copy key
KEY_SCREATE	Shifted create key
KEY_SDC	Shifted delete char key
KEY_SDL	Shifted delete line key
KEY_SELECT	Select key
KEY_SEND	Shifted end key
KEY_SEOL	Shifted clear line key
KEY_SEXIT	Shifted exit key
KEY_SFIND	Shifted find key
KEY_SHELP	Shifted help key
KEY_SHOME	Shifted home key
KEY_SIC	Shifted input key
KEY_SLEFT	Shifted left arrow key
KEY_SMESSAGE	Shifted message key
KEY_SMOVE	Shifted move key
KEY_SNEXT	Shifted next key
KEY_SOPTIONS	Shifted options key
KEY_SPREVIOUS	Shifted prev key
KEY_SPRINT	Shifted print key
KEY_SREDO	Shifted redo key
KEY_SREPLACE	Shifted replace key
KEY_SRIGHT	Shifted right arrow
KEY_SRSUME	Shifted resume key
KEY_SSAVE	Shifted save key

Name	*Key name*
`KEY_SSUSPEND`	Shifted suspend key
`KEY_SUNDO`	Shifted undo key
`KEY_SUSPEND`	Suspend key
`KEY_UNDO`	Undo key

RETURN VALUE

All routines return the integer **ERR** upon failure and an integer value other than **ERR** upon successful completion.

USAGE

Application Program.

The header file **<curses.h>** automatically includes the header files **<stdio.h>** and **<unctrl.h>**.

Use of the escape key by a programmer for a single character function is discouraged.

When using **getwch()**, **wgetwch()**, **mvgetwch()** or **mvwgetwch()**, **nocbreak()** mode and **echo()** mode should not be used at the same time. Depending on the state of the tty driver when each character is typed, the program may produce undesirable results.

Note that **getwch()**, **mvgetwch()** and **mvwgetwch()** may be macros.

SEE ALSO

CURSES (TI_ENV), curs_inopts(TI_LIB), curs_move(TI_LIB), curs_refresh(TI_LIB).

LEVEL

Level 1.

NAME

curs_getwstr: getwstr, getnwstr, wgetwstr, wgetnwstr, mvgetwstr, mvgetnwstr, mvwgetwstr, mvwgetnwstr – get **wchar_t** character strings from CURSES terminal keyboard

SYNOPSIS

#include <curses.h>

int getwstr(wchar_t **wstr***);**

int getnwstr(wchar_t **wstr*, **int** *n***);**

int wgetwstr(WINDOW **win*, **wchar_t** **wstr***);**

int wgetnwstr(WINDOW **win*, **wchar_t** **wstr*, **int** *n***);**

int mvgetwstr(int *y*, **int** *x*, **wchar_t** **wstr***);**

int mvgetnwstr(int *y*, **int** *x*, **wchar_t** **wstr*, **int** *n***);**

int mvwgetwstr(WINDOW **win*, **int** *y*, **int** *x*, **wchar_t** **wstr***);**

int mvwgetnwstr(WINDOW **win*, **int** *y*, **int** *x*, **wchar_t** **wstr*, **int** *n***);**

DESCRIPTION

The effect of **getwstr()** is as though a series of calls to **getwch()** were made, until a newline and carriage return is received. The resulting value is placed in the area pointed to by the **wchar_t** pointer *str*. **getnwstr()** reads at most *n* **wchar_t** characters, thus preventing a possible overflow of the input buffer. The user's erase and kill characters are interpreted, as well as any special keys (such as function keys, ''home'' key, ''clear'' key, etc.).

RETURN VALUE

All routines return the integer **ERR** upon failure and an integer value other than **ERR** upon successful completion.

USAGE

Application Program.

The header file **<curses.h>** automatically includes the header files **<stdio.h>** and **<unctrl.h>**.

Note that all routines except **wgetnwstr()** may be macros.

SEE ALSO

CURSES(TI_ENV), curs_getwch(TI_LIB).

LEVEL

Level 1.

NAME

curs_getyx: getyx, getparyx, getbegyx, getmaxyx – get CURSES cursor and window coordinates

SYNOPSIS

```
#include <curses.h>

void getyx(WINDOW *win, int y, int x);

void getparyx(WINDOW *win, int y, int x);

void getbegyx(WINDOW *win, int y, int x);

void getmaxyx(WINDOW *win, int y, int x);
```

DESCRIPTION

With the `getyx()` macro, the cursor position of the window is placed in the two integer variables *y* and *x*.

With the `getparyx()` macro, if *win* is a subwindow, the beginning coordinates of the subwindow relative to the parent window are placed into two integer variables, *y* and *x*. Otherwise, `-1` is placed into *y* and *x*.

Like `getyx()`, the `getbegyx()` and `getmaxyx()` macros store the current beginning coordinates and size of the specified window.

RETURN VALUE

The return values of these macros are undefined (*i.e.*, they should not be used as the right-hand side of assignment statements).

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that all of these interfaces are macros and that "&" is not necessary before the variables *y* and *x*.

SEE ALSO

CURSES(TI_ENV).

LEVEL

Level 1.

NAME

curs_inch: inch, winch, mvinch, mvwinch – get a character and its attributes from a CURSES window

SYNOPSIS

```
#include <curses.h>

chtype inch(void);

chtype winch(WINDOW *win);

chtype mvinch(int y, int x);

chtype mvwinch(WINDOW *win, int y, int x);
```

DESCRIPTION

With these routines, the character, of type `chtype`, at the current position in the named window is returned. If any attributes are set for that position, their values are OR-ed into the value returned. Constants defined in `<curses.h>` can be used with the & (logical AND) operator to extract the character or attributes alone.

Attributes

The following bit-masks may be AND-ed with characters returned by `winch()`.

`A_CHARTEXT`	Bit-mask to extract character
`A_ATTRIBUTES`	Bit-mask to extract attributes
`A_COLOR`	Bit-mask to extract color-pair field information

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that all of these routines may be macros.

SEE ALSO

CURSES(TI_ENV).

LEVEL

Level 1.

NAME

curs_inchstr: inchstr, inchnstr, winchstr, winchnstr, mvinchstr, mvinchnstr, mvwinchstr, mvwinchnstr – get a string of characters (and attributes) from a CURSES window

SYNOPSIS

```
#include <curses.h>

int inchstr(chtype *chstr);
int inchnstr(chtype *chstr, int n);
int winchstr(WINDOW *win, chtype *chstr);
int winchnstr(WINDOW *win, chtype *chstr, int n);
int mvinchstr(int y, int x, chtype *chstr);
int mvinchnstr(int y, int x, chtype *chstr, int n);
int mvwinchstr(WINDOW *win, int y, int x, chtype *chstr);
int mvwinchnstr(WINDOW *win, int y, int x, chtype *chstr, int n);
```

DESCRIPTION

With these routines, a string of type `chtype`, starting at the current cursor position in the named window and ending at the right margin of the window, is returned. The four functions with *n* as the last argument, return the string at most *n* characters long. Constants defined in `<curses.h>` can be used with the & (logical AND) operator to extract the character or the attribute alone from any position in the *chstr* [see curs_inch(TI_LIB)].

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that all routines except `winchnstr()` may be macros.

SEE ALSO

CURSES(TI_ENV), curs_inch(TI_LIB).

LEVEL

Level 1.

NAME

curs_initscr: initscr, newterm, endwin, isendwin, set_term, delscreen – CURSES screen initialization and manipulation routines

SYNOPSIS

```
#include <curses.h>

WINDOW *initscr(void);

int endwin(void);

int isendwin(void);

SCREEN *newterm(char *type, FILE *outfd, FILE *infd);

SCREEN *set_term(SCREEN *new);

void delscreen(SCREEN *sp);
```

DESCRIPTION

`initscr()` is almost always the first routine that should be called (the exceptions are `slk_init()`, `filter()`, `ripoffline()`, `use_env()` and, for multiple-terminal applications, `newterm()`.) This determines the terminal type and initializes all CURSES data structures. `initscr()` also causes the first call to `refresh()` to clear the screen. If errors occur, `initscr()` writes an appropriate error message to standard error and exits; otherwise, a pointer is returned to `stdscr`. If the program needs an indication of error conditions, `newterm()` should be used instead of `initscr()`; `initscr()` should only be called once per application.

A program that outputs to more than one terminal should use the `newterm()` routine for each terminal instead of `initscr()`. A program that needs an indication of error conditions, so it can continue to run in a line-oriented mode if the terminal cannot support a screen-oriented program, would also use this routine. The routine `newterm()` should be called once for each terminal. It returns a variable of type `SCREEN *` which should be saved as a reference to that terminal. The arguments are the *type* of the terminal to be used in place of `$TERM`, a file pointer for output to the terminal, and another file pointer for input from the terminal (if *type* is `NULL`, `$TERM` will be used). The program must also call `endwin()` for each terminal being used before exiting from curses. If `newterm()` is called more than once for the same terminal, the first terminal referred to must be the last one for which `endwin()` is called.

A program should always call `endwin()` before exiting or escaping from CURSES mode temporarily. This routine restores tty modes, moves the cursor to the lower left-hand corner of the screen and resets the terminal into the proper non-visual mode. Calling `refresh()` or `doupdate()` after a temporary escape causes the program to resume visual mode.

The `isendwin()` routine returns `TRUE` if `endwin()` has been called without any subsequent calls to `wrefresh()`, and `FALSE` otherwise.

The `set_term()` routine is used to switch between different terminals. The screen reference `new` becomes the new current terminal. The previous terminal is returned by the routine. This is the only routine which manipulates `SCREEN` pointers; all other routines affect only the current terminal.

The `delscreen()` routine frees storage associated with the `SCREEN` data structure. The `endwin()` routine does not do this, so `delscreen()` should be called after `endwin()` if a particular `SCREEN` is no longer needed.

RETURN VALUE

`endwin()` returns the integer `ERR` upon failure and `OK` upon successful completion.

Routines that return pointers always return `NULL` on error.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that `initscr()` and `newterm()` may be macros.

SEE ALSO

CURSES(TI_ENV), curs_kernel(TI_LIB), curs_refresh(TI_LIB), curs_slk(TI_LIB), curs_util(TI_LIB).

LEVEL

Level 1.

NAME

curs_inopts: cbreak, nocbreak, echo, noecho, halfdelay, intrflush, keypad, meta, nodelay, notimeout, raw, noraw, noqiflush, qiflush, timeout, wtimeout, typeahead – CURSES terminal input option control routines

SYNOPSIS

```
#include <curses.h>

int cbreak(void);
int nocbreak(void);
int echo(void);
int noecho(void);
int halfdelay(int tenths);
int intrflush(WINDOW *win, bool bf);
int keypad(WINDOW *win, bool bf);
int meta(WINDOW *win, bool bf);
int nodelay(WINDOW *win, bool bf);
int notimeout(WINDOW *win, bool bf);
int raw(void);
int noraw(void);
void noqiflush(void);
void qiflush(void);
void timeout(int delay);
void wtimeout(WINDOW *win, int delay);
int typeahead(int fd);
```

DESCRIPTION

The `cbreak()` and `nocbreak()` routines put the terminal into and out of `cbreak` mode, respectively. In this mode, characters typed by the user are immediately available to the program, and erase/kill character-processing is not performed. When out of this mode, the tty driver buffers the typed characters until a newline or carriage return is typed. Interrupt and flow control characters are unaffected by this mode. Initially the terminal may or may not be in `cbreak` mode, as the mode is inherited; therefore, a program should call `cbreak()` or `nocbreak()` explicitly. Most interactive programs using CURSES set the `cbreak` mode.

Note that `cbreak()` overrides `raw()`. [See curs_getch(TI_LIB) for a discussion of how these routines interact with `echo()` and `noecho()`.]

The `echo()` and `noecho()` routines control whether characters typed by the user are echoed by `getch()` as they are typed. Echoing by the tty driver is always disabled, but initially `getch()` is in echo mode, so characters typed are echoed. Authors of most interactive programs prefer to do their own echoing in a controlled area of the screen, or not to echo at all, so they disable echoing by calling `noecho()`. [See curs_getch(TI_LIB) for a discussion of how these routines interact with `cbreak()` and `nocbreak()`.]

The `halfdelay()` routine is used for half-delay mode, which is similar to `cbreak()` mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, ERR is returned if nothing has been typed. The value of `tenths` must be a number between 1 and 255. Use `nocbreak()` to leave half-delay mode.

If the `intrflush()` option is enabled, (*bf* is `TRUE`), when an interrupt key is pressed on the keyboard (interrupt, break, quit) all output in the tty driver queue will be flushed, giving the effect of faster response to the interrupt, but causing CURSES to have the wrong idea of what is on the screen. Disabling (*bf* is `FALSE`), the option prevents the flush. The default for the option is inherited from the tty driver settings. The window argument is ignored.

The `keypad()` option enables the keypad of the user's terminal. If enabled (*bf* is `TRUE`), the user can press a function key (such as an arrow key) and `wgetch()` returns a single value representing the function key, as in `KEY_LEFT`. If disabled (*bf* is `FALSE`), CURSES does not treat function keys specially and the program has to interpret the escape sequences itself. If the keypad in the terminal can be turned on (made to transmit) and off (made to work locally), turning on this option causes the terminal keypad to be turned on when `wgetch()` is called. The default value for keypad is false.

Initially, whether the terminal returns 7 or 8 significant bits on input depends on the control mode of the tty driver [see termio(BA_DEV)]. To force 8 bits to be returned, invoke `meta`(*win*, `TRUE`). To force 7 bits to be returned, invoke `meta`(*win*, `FALSE`). The window argument, *win*, is always ignored. If the terminfo capabilities `smm` (meta_on) and `rmm` (meta_off) are defined for the terminal, `smm` is sent to the terminal when `meta`(*win*, `TRUE`) is called and `rmm` is sent when `meta`(*win*, `FALSE`) is called.

The `nodelay()` option causes `getch()` to be a non-blocking call. If no input is ready, `getch()` returns `ERR`. If disabled (*bf* is `FALSE`), `getch()` waits until a key is pressed.

While interpreting an input escape sequence, `wgetch()` sets a timer while waiting for the next character. If `notimeout(`*win*, `TRUE`) is called, then `wgetch()` does not set a timer. The purpose of the timeout is to differentiate between sequences received from a function key and those typed by a user.

With the `raw()` and `noraw()` routines, the terminal is placed into or out of raw mode. Raw mode is similar to `cbreak` mode, in that characters typed are immediately passed through to the user program. The differences are that in raw mode, the interrupt, quit, suspend, and flow control characters are all passed through uninterpreted, instead of generating a signal. The behavior of the BREAK key depends on other bits in the tty driver that are not set by CURSES.

When the `noqiflush()` routine is used, normal flush of input and output queues associated with the `INTR`, `QUIT` and `SUSP` characters will not be done [see termio(BA_DEV)]. When `qiflush()` is called, the queues will be flushed when these control characters are read.

The `timeout()` and `wtimeout()` routines set blocking or non-blocking read for a given window. If *delay* is negative, blocking read is used (*i.e.*, waits indefinitely for input). If *delay* is zero, then non-blocking read is used (*i.e.*, read returns ERR if no input is waiting). If *delay* is positive, then read blocks for *delay* milliseconds, and returns ERR if there is still no input. Hence, these routines provide the same functionality as `nodelay()`, plus the additional capability of being able to block for only *delay* milliseconds (where *delay* is positive).

CURSES does ''line-breakout optimization'' by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update is postponed until `refresh()` or `doupdate()` is called again. This allows faster response to commands typed in advance. Normally, the input FILE pointer passed to `newterm()`, or `stdin` in the case that `initscr()` was used, will be used to do this typeahead checking. The `typeahead()` routine specifies that the file descriptor *fd* is to be used to check for typeahead instead. If *fd* is –1, then no typeahead checking is done.

RETURN VALUE

All routines that return an integer return ERR upon failure and an integer value other than ERR upon successful completion, unless otherwise noted in the preceding routine descriptions.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that `echo`, `noecho()`, `halfdelay()`, `intrflush()`, `meta()`, `nodelay()`, `notimeout()`, `noqiflush()`, `qiflush()`, `timeout()`, and `wtimeout()` may be macros.

SEE ALSO

CURSES(TI_ENV), curs_getch(TI_LIB), curs_initscr(TI_LIB), termio(BA_DEV).

LEVEL

Level 1.

NAME

curs_insch: insch, winsch, mvinsch, mvwinsch – insert a character before the character under the cursor in a CURSES window

SYNOPSIS

```
#include <curses.h>

int insch(chtype ch);

int winsch(WINDOW *win, chtype ch);

int mvinsch(int y, int x, chtype ch);

int mvwinsch(WINDOW *win, int y, int x, chtype ch);
```

DESCRIPTION

With these routines, the character *ch* is inserted before the character under the cursor. All characters to the right of the cursor are moved one space to the right, with the possibility of the rightmost character on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified). (This does not imply use of the hardware insert character feature.)

RETURN VALUE

All routines return the integer `ERR` upon failure and an integer value other than `ERR` upon successful completion.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that `insch()`, `mvinsch()`, and `mvwinsch()` may be macros.

SEE ALSO

CURSES(TI_ENV).

LEVEL

Level 1.

NAME

curs_instr: insstr, insnstr, winsstr, winsnstr, mvinsstr, mvinsnstr, mvwinsstr, mvwinsnstr – insert string before character under the cursor in a CURSES window

SYNOPSIS

```
#include <curses.h>

int insstr(char *str);
int insnstr(char *str, int n);
int winsstr(WINDOW *win, char *str);
int winsnstr(WINDOW *win, char *str, int n);
int mvinsstr(int y, int x, char *str);
int mvinsnstr(int y, int x, char *str, int n);
int mvwinsstr(WINDOW *win, int y, int x, char *str);
int mvwinsnstr(WINDOW *win, int y, int x, char *str, int n);
```

DESCRIPTION

With these routines, a character string (as many characters as will fit on the line) is inserted before the character under the cursor. All characters to the right of the cursor are moved to the right, with the possibility of the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified). (This does not imply use of the hardware insert character feature.) The four routines with *n* as the last argument insert at most *n* characters. If *n*<=0, then the entire string is inserted.

If a character in *str* is a tab, newline, carriage return or backspace, the cursor is moved appropriately within the window. A newline also does a `clrtoeol()` before moving. Tabs are considered to be at every eighth column. If a character in *str* is another control character, it is drawn in the ^X notation. Calling `winch()` after adding a control character (and moving to it, if necessary) does not return the control character, but instead returns the representation of the control character.

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that all but `winsnstr()` may be macros.

SEE ALSO

CURSES(TI_ENV), curs_clear(TI_LIB), curs_inch(TI_LIB).

LEVEL

Level 1.

NAME

curs_instr: instr, innstr, winstr, winnstr, mvinstr, mvinnstr, mvwinstr, mvwinnstr – get a string of characters from a CURSES window

SYNOPSIS

```
#include <curses.h>

int instr(char *str);

int innstr(char *str, int n);

int winstr(WINDOW *win, char *str);

int winnstr(WINDOW *win, char *str, int n);

int mvinstr(int y, int x, char *str);

int mvinnstr(int y, int x, char *str, int n);

int mvwinstr(WINDOW *win, int y, int x, char *str);

int mvwinnstr(WINDOW *win, int y, int x, char *str, int n);
```

DESCRIPTION

These routines return a string of characters in *str*, starting at the current cursor position in the named window and ending at the right margin of the window. Attributes are stripped from the characters. The four functions with *n* as the last argument return the string at most *n* characters long.

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that all routines except `winnstr()` may be macros.

SEE ALSO

CURSES(TI_ENV).

LEVEL

Level 1.

NAME

curs_inswch: inswch, winswch, mvinswch, mvwinswch – insert a **wchar_t** character before the character under the cursor in a CURSES window

SYNOPSIS

#include <curses.h>

int inswch(chtype *wch***);**

int winswch(WINDOW **win***, chtype** *wch***);**

int mvinswch(int *y***, int** *x***, chtype** *wch***);**

int mvwinswch(WINDOW **win***, int** *y***, int** *x***, chtype** *wch***);**

DESCRIPTION

These routines insert the character *wch*, holding a **wchar_t** character, before the character under the cursor. All characters to the right of the cursor are moved one space to the right, with the possibility of the rightmost character on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified). (This does not imply use of the hardware insert character feature.)

RETURN VALUE

All routines return the integer **ERR** upon failure and an integer value other than **ERR** upon successful completion.

USAGE

Application Program.

The header file **<curses.h>** automatically includes the header files **<stdio.h>** and **<unctrl.h>**.

Note that **inswch()**, **mvinswch()** and **mvwinswch()** may be macros.

SEE ALSO

CURSES(TI_ENV).

LEVEL

Level 1.

NAME

curs_inswstr: inswstr, insnwstr, winswstr, winsnwstr, mvinswstr, mvinsnwstr, mvwinswstr, mvwinsnwstr – insert **wchar_t** string before character under the cursor in a CURSES window

SYNOPSIS

```
#include <curses.h>

int inswstr(wchar *wstr);
int insnwstr(wchar *wstr, int n);
int winswstr(WINDOW *win, wchar *wstr);
int winsnwstr(WINDOW *win, wchar *wstr, int n);
int mvinswstr(int y, int x, wchar *wstr);
int mvinsnwstr(int y, int x, wchar *wstr, int n);
int mvwinswstr(WINDOW *win, int y, int x, wchar *wstr);
int mvwinsnwstr(WINDOW *win, int y, int x, wchar *wstr, int n);
```

DESCRIPTION

These routines insert a **wchar_t** character string (as many **wchar_t** characters as will fit on the line) before the character under the cursor. All characters to the right of the cursor are moved to the right, with the possibility of the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified). (This does not imply use of the hardware insert character feature.) The four routines with *n* as the last argument insert at most *n* **wchar_t** characters. If *n*<=0, then the entire string is inserted.

If a character in *wstr* is a tab, newline, carriage return or backspace, the cursor is moved appropriately within the window. A newline also does a **clrtoeol** before moving. Tabs are considered to be at every eighth column. If a character in *wstr* is another control character, it is drawn in the ^*X* notation. Calling **winch()** after adding a control character (and moving to it, if necessary) does not return the control character, but instead returns the representation of the control character.

RETURN VALUE

All routines return the integer **ERR** upon failure and an integer value other than **ERR** upon successful completion.

USAGE

Application Program.

The header file **<curses.h>** automatically includes the header files **<stdio.h>** and **<unctrl.h>**.

Note that all but **winsnwstr()** may be macros.

SEE ALSO

CURSES(TI_ENV), curs_clear(TI_LIB), curs_inwch(TI_LIB).

LEVEL

Level 1.

NAME

curs_inwch: inwch, winwch, mvinwch, mvwinwch – get a **wchar_t** character and its attributes from a CURSES window

SYNOPSIS

```
#include <curses.h>

chtype inwch(void);

chtype winwch(WINDOW *win);

chtype mvinwch(int y, int x);

chtype mvwinwch(WINDOW *win, int y, int x);
```

DESCRIPTION

These routines return the **wchar_t** character, of type **chtype**, at the current position in the named window. If any attributes are set for that position, their values are OR-ed into the value returned. Constants defined in **<curses.h>** can be used with the **&** (logical AND) operator to extract the character or attributes alone.

Attributes

The following bit-masks may be AND-ed with characters returned by **winwch()**.

A_CHARTEXT	Bit-mask to extract character
A_ATTRIBUTES	Bit-mask to extract attributes
A_COLOR	Bit-mask to extract color-pair field information

USAGE

Application Program.

The header file **<curses.h>** automatically includes the header files **<stdio.h>** and **<unctrl.h>**.

Note that all of these routines may be macros.

SEE ALSO

CURSES(TI_ENV).

LEVEL

Level 1.

NAME

curs_inwchstr: inwchstr, inwchnstr, winwchstr, winwchnstr, mvinwchstr, mvinwchnstr, mvwinwchstr, mvwinwchnstr – get a string of **wchar_t** characters (and attributes) from a CURSES window

SYNOPSIS

#include <curses.h>

int inwchstr(chtype ***wchstr*);**

int inwchnstr(chtype ***wchstr*, int** *n***);**

int winwchstr(WINDOW ***win*, chtype** ***wchstr*);**

int winwchnstr(WINDOW ***win*, chtype** ***wchstr*, int** *n***);**

int mvinwchstr(int *y*, **int** *x*, **chtype** ***wchstr*);**

int mvinwchnstr(int *y*, **int** *x*, **chtype** ***wchstr*, int** *n***);**

int mvwinwchstr(WINDOW ***win*, int** *y*, **int** *x*, **chtype** ***wchstr*);**

int mvwinwchnstr(WINDOW ***win*, int** *y*, **int** *x*, **chtype** ***wchstr*, int** *n***);**

DESCRIPTION

These routines return a string of type **chtype**, holding **wchar_t** characters, starting at the current cursor position in the named window and ending at the right margin of the window. The four functions with *n* as the last argument return the string at most *n* **wchar_t** characters long. Constants defined in **<curses.h>** can be used with the **&** (logical AND) operator to extract the **wchar_t** character or the attribute alone from any position in the *chstr* [see curs_inwch(TI_LIB)].

RETURN VALUE

All routines return the integer **ERR** upon failure and an integer value other than **ERR** upon successful completion.

USAGE

Application Program.

The header file **<curses.h>** automatically includes the header files **<stdio.h>** and **<unctrl.h>**.

Note that all routines except **winwchnstr()** may be macros.

SEE ALSO

CURSES(TI_ENV), curs_inwch(TI_LIB).

LEVEL

Level 1.

NAME

curs_inwstr: inwstr, innwstr, winwstr, winnwstr, mvinwstr, mvinnwstr, mvwinwstr, mvwinnwstr – get a string of **wchar_t** characters from a CURSES window

SYNOPSIS

```
#include <curses.h>

int inwstr(char *str);
int innwstr(char *str, int n);
int winwstr(WINDOW *win, char *str);
int winnwstr(WINDOW *win, char *str, int n);
int mvinwstr(int y, int x, char *str);
int mvinnwstr(int y, int x, char *str, int n);
int mvwinwstr(WINDOW *win, int y, int x, char *str);
int mvwinnwstr(WINDOW *win, int y, int x, char *str, int n);
```

DESCRIPTION

These routines return the string of **wchar_t** characters in *str* starting at the current cursor position in the named window and ending at the right margin of the window. Attributes are stripped from the characters. The four functions with *n* as the last argument return the string at most *n* **wchar_t** characters long.

RETURN VALUE

All routines return the integer **ERR** upon failure and an integer value other than **ERR** upon successful completion.

USAGE

Application Program.

The header file **<curses.h>** automatically includes the header files **<stdio.h>** and **<unctrl.h>**.

Note that all routines except **winnwstr()** may be macros.

SEE ALSO

CURSES(TI_ENV).

LEVEL

Level 1.

NAME

curs_kernel: def_prog_mode, def_shell_mode, reset_prog_mode, reset_shell_mode, resetty, savetty, getsyx, setsyx, ripoffline, curs_set, napms – low-level CURSES routines

SYNOPSIS

```
#include <curses.h>

int def_prog_mode(void);
int def_shell_mode(void);
int reset_prog_mode(void);
int reset_shell_mode(void);
int resetty(void);
int savetty(void);
int getsyx(int y, int x);
int setsyx(int y, int x);
int ripoffline(int line, int (*init)(WINDOW *win, int));
int curs_set(int visibility);
int napms(int ms);
```

DESCRIPTION

The following routines give low-level access to various CURSES functionality. Theses routines typically are used inside library routines.

The `def_prog_mode()` and `def_shell_mode()` routines save the current terminal modes as the "program" (in CURSES) or "shell" (not in CURSES) state for use by the `reset_prog_mode()` and `reset_shell_mode()` routines. This is done automatically by `initscr()`.

The `reset_prog_mode()` and `reset_shell_mode()` routines restore the terminal to "program" (in CURSES) or "shell" (out of CURSES) state. These are done automatically by `endwin()` and, after an `endwin()`, by `doupdate()`, so they normally are not called.

The `resetty()` and `savetty()` routines save and restore the state of the terminal modes. `savetty()` saves the current state in a buffer and `resetty()` restores the state to what it was at the last call to `savetty()`.

With the `getsyx()` routine, the current coordinates of the virtual screen cursor are returned in *y* and *x*. If `leaveok()` is currently `TRUE`, then `–1`,`–1` is returned. If lines have been removed from the top of the screen, using `ripoffline()`, *y* and *x* include these lines; therefore, *y* and *x* should be used only as arguments for `setsyx()`.

With the `setsyx()` routine, the virtual screen cursor is set to *y*, *x*. If *y* and *x* are both `–1`, then `leaveok()` is set. The two routines `getsyx()` and `setsyx()` are designed to be used by a library routine, which manipulates CURSES windows but does not want to change the current position of the program's cursor. The library routine would call `getsyx()` at the beginning, do its manipulation of its own windows, do a `wnoutrefresh()` on its windows, call `setsyx()`, and then call `doupdate()`.

The `ripoffline()` routine provides access to the same facility that `slk_init()` [see curs_slk(TI_ENV)] uses to reduce the size of the screen. `ripoffline()` must be called before `initscr()` or `newterm()` is called. If *line* is positive, a line is removed from the top of `stdscr`; if *line* is negative, a line is removed from the bottom. When this is done inside `initscr()`, the routine `init()` (supplied by the user) is called with two arguments: a window pointer to the one-line window that has been allocated and an integer with the number of columns in the window. Inside this initialization routine, the integer variables `LINES` and `COLS` (defined in `<curses.h>`) are not guaranteed to be accurate and `wrefresh()` or `doupdate()` must not be called. It is allowable to call `wnoutrefresh()` during the initialization routine.

`ripoffline()` can be called up to five times before calling `initscr()` or `newterm()`.

With the `curs_set()` routine, the cursor state is set to invisible, normal, or very visible for `visibility` equal to `0`, `1`, or `2` respectively. If the terminal supports the *visibility* requested, the previous *cursor* state is returned; otherwise, `ERR` is returned.

The `napms()` routine is used to sleep for *ms* milliseconds.

RETURN VALUE

Except for `curs_set()`, these routines always return `OK`. `curs_set()` returns the previous cursor state, or `ERR` if the requested *visibility* is not supported.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that `getsyx()` is a macro, so `&` is not necessary before the variables *y* and *x*.

SEE ALSO

CURSES(TI_ENV), curs_initscr(TI_LIB), curs_outopts(TI_LIB), curs_refresh(TI_LIB), curs_scr_dump(TI_LIB), curs_slk(TI_LIB).

LEVEL

Level 1.

NAME

curs_move: move, wmove – move CURSES window cursor

SYNOPSIS

```
#include <curses.h>

int move(int y, int x);

int wmove(WINDOW *win, int y, int x);
```

DESCRIPTION

With these routines, the cursor associated with the window is moved to line *y* and column *x*. This routine does not move the physical cursor of the terminal until `refresh()` is called. The position specified is relative to the upper left-hand corner of the window, which is (0,0).

RETURN VALUE

These routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that `move()` may be a macro.

SEE ALSO

CURSES(TI_ENV), curs_refresh(TI_LIB).

LEVEL

Level 1.

NAME

curs_outopts: clearok, idlok, idcok immedok, leaveok, setscrreg, wsetscrreg, scrollok, nl, nonl – CURSES terminal output option control routines

SYNOPSIS

```
#include <curses.h>

int clearok(WINDOW *win, bool bf);
int idlok(WINDOW *win, bool bf);
void idcok(WINDOW *win, bool bf);
void immedok(WINDOW *win, bool bf);
int leaveok(WINDOW *win, bool bf);
int setscrreg(int top, int bot);
int wsetscrreg(WINDOW *win, int top, int bot);
int scrollok(WINDOW *win, bool bf);
int nl(void);
int nonl(void);
```

DESCRIPTION

These routines set options that deal with output within CURSES. All options are initially `FALSE`, unless otherwise stated. It is not necessary to turn these options off before calling `endwin()`.

With the `clearok()` routine, if enabled (*bf* is `TRUE`), the next call to `wrefresh()` with this window will clear the screen completely and redraw the entire screen from scratch. This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect. If the *win* argument to `clearok()` is the global variable `curscr`, the next call to `wrefresh()` with any window causes the screen to be cleared and repainted from scratch.

With the `idlok()` routine, if enabled (*bf* is `TRUE`), CURSES considers using the hardware insert/delete line feature of terminals so equipped. If disabled (*bf* is `FALSE`), CURSES very seldom uses this feature. (The insert/delete character feature is always considered.) This option should be enabled only if the application needs insert/delete line, for example, for a screen editor. It is disabled by default because insert/delete line tends to be visually annoying when used in applications where it isn't really needed. If insert/delete line cannot be used, CURSES redraws the changed portions of all lines.

With the `idcok()` routine, if enabled (*bf* is `TRUE`), CURSES considers using the hardware insert/delete character feature of terminals so equipped. This is enabled by default.

With the `immedok()` routine, if enabled (*bf* is `TRUE)`, any change in the window image, such as the ones caused by `waddch(), wclrtobot(), wscrl()`, *etc.*, automatically cause a call to `wrefresh()`. However, it may degrade the performance considerably, due to repeated calls to `wrefresh()`. It is disabled by default.

Normally, the hardware cursor is left at the location of the window cursor being refreshed. The `leaveok()` option allows the cursor to be left wherever the update happens to leave it. It is useful for applications where the cursor is not used, since it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled.

The `setscrreg()` and `wsetscrreg()` routines allow the application programmer to set a software scrolling region in a window. *top* and *bot* are the line numbers of the top and bottom margin of the scrolling region. (Line 0 is the top line of the window.) If this option and `scrollok()` are enabled, an attempt to move off the bottom margin line causes all lines in the scrolling region to scroll up one line. Only the text of the window is scrolled. (Note that this has nothing to do with the use of a physical scrolling region capability in the terminal, like that in the VT100. If `idlok()` is enabled and the terminal has either a scrolling region or insert/delete line capability, they will probably be used by the output routines.)

The `scrollok()` option controls what happens when the cursor of a window is moved off the edge of the window or scrolling region, either as a result of a newline action on the bottom line, or typing the last character of the last line. If disabled, (*bf* is `FALSE`), the cursor is left on the bottom line. If enabled, (*bf* is `TRUE`), `wrefresh()` is called on the window, and the physical terminal and window are scrolled up one line. [Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call `idlok()`.]

The `nl()` and `nonl()` routines control whether newline is translated into carriage return and linefeed on output, and whether return is translated into newline on input. Initially, the translations do occur. By disabling these translations using `nonl()`, CURSES is able to make better use of the linefeed capability, resulting in faster cursor motion.

RETURN VALUE

`setscrreg()` and `wsetscrreg()` return `OK` upon success and `ERR` upon failure. All other routines that return an integer always return `OK`.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that `clearok()`, `leaveok()`, `scrollok()`, `idcok()`, `nl()`, `nonl()` and `setscrreg()` may be macros.

The `immedok()` routine is useful for windows that are used as terminal emulators.

SEE ALSO

CURSES(TI_ENV), curs_addch(TI_LIB), curs_clear(TI_LIB), curs_initscr(TI_LIB), curs_scroll(TI_LIB), curs_refresh(TI_LIB).

LEVEL

Level 1.

NAME

curs_overlay: overlay, overwrite, copywin – overlap and manipulate overlapped CURSES windows

SYNOPSIS

```
#include <curses.h>

int overlay(WINDOW *srcwin, WINDOW *dstwin);

int overwrite(WINDOW *srcwin, WINDOW *dstwin);

int copywin(WINDOW *srcwin, WINDOW *dstwin, int sminrow,
     int smincol, int dminrow, int dmincol, int dmaxrow,
     int dmaxcol, int overlay);
```

DESCRIPTION

The `overlay()` and `overwrite()` routines overlay *srcwin* on top of *dstwin*. *scrwin* and *dstwin* are not required to be the same size; only text where the two windows overlap is copied. The difference is that `overlay()` is non-destructive (blanks are not copied) whereas `overwrite()` is destructive.

The `copywin()` routine provides a finer granularity of control over the `overlay()` and `overwrite()` routines. Like in the `prefresh()` routine, a rectangle is specified in the destination window, (*dminrow*, *dmincol*) and (*dmaxrow*, *dmaxcol*), and the upper-left-corner coordinates of the source window, (*sminrow*, *smincol*). If the argument *overlay* is `true`, then copying is non-destructive, as in `overlay()`.

RETURN VALUE

Routines that return an integer return ERR upon failure and an integer value other than ERR upon successful completion.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that `overlay()` and `overwrite()` may be macros.

SEE ALSO

CURSES(TI_ENV), curs_pad(TI_LIB), curs_refresh(TI_LIB).

LEVEL

Level 1.

NAME

curs_pad: newpad, subpad, prefresh, pnoutrefresh, pechochar, pechowchar – create and display CURSES pads

SYNOPSIS

```
#include <curses.h>

WINDOW *newpad(int nlines, int ncols);

WINDOW *subpad(WINDOW *orig, int nlines, int ncols,
     int begin_y, int begin_x);

int prefresh(WINDOW *pad, int pminrow, int pmincol,
     int sminrow, int smincol, int smaxrow, int smaxcol);

int pnoutrefresh(WINDOW *pad, int pminrow, int pmincol,
     int sminrow, int smincol, int smaxrow, int smaxcol);

int pechochar(WINDOW *pad, chtype ch);

int pechowchar(WINDOW *pad, chtype wch);
```

DESCRIPTION

The `newpad()` routine creates and returns a pointer to a new pad data structure with the given number of lines, *nlines*, and columns, *ncols*. A pad is like a window, except that it is not necessarily associated with a viewable part of the screen. Automatic refreshes of pads (e.g., from scrolling or echoing of input) do not occur. It is not legal to call `wrefresh()` with a *pad* as an argument; the routines `prefresh()` or `pnoutrefresh()` should be called instead. Note that these routines require additional parameters to specify the part of the pad to be displayed and the location on the screen to be used for the display.

The `subpad()` routine creates and returns a pointer to a subwindow within a pad with the given number of lines, *nlines*, and columns, *ncols*. Unlike `subwin()`, which uses screen coordinates, the window is at position (*begin_x*, *begin_y*) on the pad. The window is made in the middle of the window *orig*, so that changes made to one window affect both windows. During the use of this routine, it will often be necessary to call `touchwin()` or `touchline()` on *orig* before calling `prefresh()`.

The `prefresh()` and `pnoutrefresh()` routines are analogous to `wrefresh()` and `wnoutrefresh()` except that they relate to pads instead of windows. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left-hand corner of the rectangle to be displayed in the pad. *sminrow, smincol, smaxrow* and *smaxcol* specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *sminrow* or *smincol* are treated as if they were zero.

The `pechochar()` routine is functionally equivalent to a call to `addch()` followed by a call to `refresh()`, a call to `waddch()` followed by a call to `wrefresh()` or a call to `waddch()` followed by a call to `prefresh()`. The knowledge that only a single character is being output is taken into consideration and, for non-control characters, a considerable performance gain might be seen by using these routines instead of their equivalents. In the case of `pechochar()`, the last location of the pad on the screen is reused for the arguments to `prefresh()`.

The `pechowchar()` routine is functionally equivalent to a call to `addwch()` followed by a call to `refresh()`, a call to `waddwch()` followed by a call to `wrefresh()` or a call to `waddwch()` followed by a call to `prefresh()`.

RETURN VALUE

Routines that return an integer return ERR upon failure and an integer value other than ERR upon successful completion.

Routines that return pointers return `NULL` on error.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that `pechochar()` may be a macro.

SEE ALSO

CURSES(TI_ENV), curs_refresh(TI_LIB), curs_touch(TI_LIB), curs_addch(TI_LIB), curs_addwch(TI_LIB).

LEVEL

Level 1.

NAME

curs_printw: printw, wprintw, mvprintw, mvwprintw, vwprintw – print formatted output in CURSES windows

SYNOPSIS

```
#include <curses.h>

int printw(char *fmt [, arg] ...);

int wprintw(WINDOW *win, char *fmt [, arg] ...);

int mvprintw(int y, int x, char *fmt [, arg] ...);

int mvwprintw(WINDOW *win, int y, int x,
     char *fmt [, arg] ...);

#include <varargs.h>

int vwprintw(WINDOW *win, char *fmt, varglist);
```

DESCRIPTION

The `printw()`, `wprintw()`, `mvprintw()` and `mvwprintw()` routines are analogous to `printf()` [see printf(BA_LIB)]. In effect, the string that would be output by `printf()` is output instead as though `waddstr()` were used on the given window.

The `vwprintw()` routine is analogous to `vprintf()` [see vprintf(BA_LIB)] and performs a `wprintw()` using a variable argument list. The third argument is a `va_list`, a pointer to a list of arguments, as defined in `<varargs.h>`.

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

SEE ALSO

CURSES(TI_ENV), printf(BA_LIB), vprintf(BA_LIB).

LEVEL

Level 1.

NAME

curs_refresh: refresh, wrefresh, wnoutrefresh, doupdate, redrawwin, wredrawln – refresh CURSES windows and lines

SYNOPSIS

```
#include <curses.h>

int refresh(void);

int wrefresh(WINDOW *win);

int wnoutrefresh(WINDOW *win);

int doupdate(void);

int redrawwin(WINDOW *win);

int wredrawln(WINDOW *win, int beg_line, int num_lines);
```

DESCRIPTION

The `refresh()` and `wrefresh()` routines (or `wnoutrefresh()` and `doupdate()`) must be called to get any output on the terminal, as other routines merely manipulate data structures. The routine `wrefresh()` copies the named window to the physical terminal screen, taking into account what is already there in order to do optimizations. The `refresh()` routine is the same, using `stdscr` as the default window. Unless `leaveok()` has been enabled, the physical cursor of the terminal is left at the location of the cursor for that window.

The `wnoutrefresh()` and `doupdate()` routines allow multiple updates with more efficiency than `wrefresh()` alone. In addition to all the window structures, CURSES keeps two data structures representing the terminal screen: a physical screen, describing what is actually on the screen, and a virtual screen, describing what the programmer wants to have on the screen.

The routine `wrefresh()` works by first calling `wnoutrefresh()`, which copies the named window to the virtual screen, and then calling `doupdate()`, which compares the virtual screen to the physical screen and does the actual update. If the programmer wishes to output several windows at once, a series of calls to `wrefresh()` results in alternating calls to `wnoutrefresh()` and `doupdate()`, causing several bursts of output to the screen. By first calling `wnoutrefresh()` for each window, it is then possible to call `doupdate()` once, resulting in only one burst of output, with fewer total characters transmitted and less CPU time used. If the *win* argument to `wrefresh()` is the global variable `curscr`, the screen is immediately cleared and repainted from scratch.

The `redrawwin()` routine indicates to CURSES that some screen lines are corrupted and should be thrown away before anything is written over them. These routines could be used for programs such as editors, which want a command to redraw some part of the screen or the entire screen. The routine `redrawln()` is preferred over `redrawwin()` where a noisy communication line exists and redrawing the entire window could be subject to even more communication noise. Just redrawing several lines offers the possibility that they would show up unblemished.

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

USAGE

Application Program.

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that `refresh()` and `redrawwin()` may be macros.

SEE ALSO

CURSES(TI_ENV), curs_outopts(TI_LIB).

LEVEL

Level 1.

NAME

curs_scanw: scanw, wscanw, mvscanw, mvwscanw, vwscanw – convert formatted input from a CURSES widow

SYNOPSIS

`#include <curses.h>`

`int scanw(char *`*fmt* `[,` *arg*`] ...);`

`int wscanw(WINDOW *`*win*`, char *`*fmt* `[,` *arg*`] ...);`

`int mvscanw(int` *y*`, int` *x*`, char *`*fmt* `[,` *arg*`] ...);`

`int mvwscanw(WINDOW *`*win*`, int` *y*`, int` *x*`,`
`char *`*fmt* `[,` *arg*`]` *...);*

*int vwscanw(WINDOW *win, char *fmt, va_list varglist);*

DESCRIPTION

The `scanw()`, `wscanw()` and `mvscanw()` routines correspond to `scanf` [see scanf(BA_LIB)]. The effect of these routines is as though `wgetstr()` were called on the window, and the resulting line used as input for the scan. Fields which do not map to a variable in the *fmt* field are lost.

The `vwscanw()` routine is similar to `vwprintw()` in that it performs a `wscanw()` using a variable argument list. The third argument is a *va_list*, a pointer to a list of arguments, as defined in `<varargs.h>`.

RETURN VALUE

`vwscanw()` returns `ERR` on failure and an integer equal to the number of fields scanned on success.

Applications may interrogate the return value from the `scanw()`, `wscanw()`, `mvscanw()` and `mvwscanw()` routines to determine the number of fields which were mapped in the call.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

SEE ALSO

CURSES(TI_ENV), curs_getstr, curs_printw, scanf(BA_LIB).

LEVEL

Level 1.

NAME

curs_scr_dump: scr_dump, scr_restore, scr_init, scr_set – read (write) a CURSES screen from (to) a file

SYNOPSIS

```
#include <curses.h>

int scr_dump(char *filename);

int scr_restore(char *filename);

int scr_init(char *filename);

int scr_set(char *filename);
```

DESCRIPTION

With the `scr_dump()` routine, the current contents of the virtual screen are written to the file *filename*.

With the `scr_restore()` routine, the virtual screen is set to the contents of *filename*, which must have been written using `scr_dump()`. The next call to `doupdate()` restores the screen to the way it looked in the dump file.

With the `scr_init()` routine, the contents of *filename* are read in and used to initialize the CURSES data structures about what the terminal currently has on its screen. If the data is determined to be valid, CURSES bases its next update of the screen on this information rather than clearing the screen and starting from scratch. `scr_init()` is used after `initscr()` or a `system()` [see system(BA_LIB)] call to share the screen with another process which has done a `scr_dump()` after its `endwin()` call. The data is declared invalid if the time-stamp of the tty is old or the terminfo capabilities `rmcup` and `nrrmc` exist.

The `scr_set()` routine is a combination of `scr_restore()` and `scr_init()`. It tells the program that the information in *filename* is what is currently on the screen, and also what the program wants on the screen. This can be thought of as a screen inheritance function.

To read (write) a window from (to) a file, use the `getwin()` and `putwin()` routines [see curs_util(TI_LIB)].

RETURN VALUE

All routines return the integer `ERR` upon failure and `OK` upon success.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that `scr_init()`, `scr_set()`, and `scr_restore()` may be macros.

SEE ALSO

CURSES(TI_ENV), curs_initscr(TI_LIB), curs_refresh(TI_LIB), curs_util(TI_LIB), system(BA_LIB).

LEVEL

Level 1.

NAME

curs_scroll: scroll, srcl, wscrl – scroll a CURSES window

SYNOPSIS

```
#include <curses.h>

int scroll(WINDOW *win);

int scrl(int n);

int wscrl(WINDOW *win, int n);
```

DESCRIPTION

With the `scroll()` routine, the window is scrolled up one line. This involves moving the lines in the window data structure. As an optimization, if the scrolling region of the window is the entire screen, the physical screen is scrolled at the same time.

With the `scrl()` and `wscrl()` routines, for positive *n* scroll the window up *n* lines (line *i+n* becomes *i*); otherwise scroll the window down *n* lines. This involves moving the lines in the window character image structure. The current cursor position is not changed.

For these functions to work, scrolling must be enabled via `scrollok()`.

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that `scrl()` and `scroll()` may be macros.

SEE ALSO

CURSES(TI_ENV), curs_outopts(TI_LIB).

LEVEL

Level 1.

NAME

curs_slk: slk_init, slk_set, slk_refresh, slk_noutrefresh, slk_label, slk_clear, slk_restore, slk_touch, slk_attron, slk_attrset, slk_attroff – CURSES soft label routines

SYNOPSIS

```
#include <curses.h>

int slk_init(int fmt);
int slk_set(int labnum, char *label, int fmt);
int slk_refresh(void);
int slk_noutrefresh(void);
char *slk_label(int labnum);
int slk_clear(void);
int slk_restore(void);
int slk_touch(void);
int slk_attron(chtype attrs);
int slk_attrset(chtype attrs);
int slk_attroff(chtype attrs);
```

DESCRIPTION

CURSES manipulates the set of soft function-key labels that exist on many terminals. For those terminals that do not have soft labels, CURSES takes over the bottom line of `stdscr`, reducing the size of `stdscr` and the variable `LINES`. CURSES standardizes on eight labels of up to eight characters each.

To use soft labels, the `slk_init()` routine must be called before `initscr()` or `newterm()` is called. If `initscr()` eventually uses a line from `stdscr` to emulate the soft labels, then *fmt* determines how the labels are arranged on the screen. Setting *fmt* to `0` indicates a 3-2-3 arrangement of the labels; `1` indicates a 4-4 arrangement.

With the `slk_set()` routine, *labnum* is the label number, from `1` to `8`. *label* is the string to be put on the label, up to eight characters in length. A null string or a null pointer sets up a blank label. *fmt* is either `0`, `1`, or `2`, indicating whether the label is to be left-justified, centered, or right-justified, respectively, within the label.

The `slk_refresh()` and `slk_noutrefresh()` routines correspond to the `wrefresh()` and `wnoutrefresh()` routines.

With the `slk_label()` routine, the current label for label number *labnum* is returned with leading and trailing blanks stripped.

With the `slk_clear()` routine, the soft labels are cleared from the screen.

With the `slk_restore()` routine, the soft labels are restored to the screen after a `slk_clear()` is performed.

With the `slk_touch()` routine, all the soft labels are forced to be output the next time a `slk_noutrefresh()` is performed.

The `slk_attron()`, `slk_attrset()` and `slk_attroff()` routines correspond to `attron()`, `attrset()`, and `attroff()`. They have an effect only if soft labels are simulated on the bottom line of the screen.

RETURN VALUE

Routines that return an integer return ERR upon failure and an integer value other than ERR upon successful completion.

`slk_label()` returns NULL on error.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Most applications would use `slk_noutrefresh()` because a `wrefresh()` is likely to follow soon.

SEE ALSO

CURSES(TI_ENV), curs_attr(TI_LIB), curs_initscr(TI_LIB), curs_refresh(TI_LIB).

LEVEL

Level 1.

NAME

curs_termattrs: baudrate, erasechar, has_ic, has_il, killchar, longname, termattrs, termname – CURSES environment query routines

SYNOPSIS

```
#include <curses.h>

int baudrate(void);
char erasechar(void);
int has_ic(void);
int has_il(void);
char killchar(void);
char *longname(void);
chtype termattrs(void);
char *termname(void);
```

DESCRIPTION

The `baudrate()` routine returns the output speed of the terminal. The number returned is in bits per second, for example `9600`, and is an integer.

With the `erasechar()` routine, the user's current erase character is returned.

The `has_ic()` routine is true if the terminal has insert- and delete-character capabilities.

The `has_il()` routine is true if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This might be used to determine if it would be appropriate to turn on physical scrolling using `scrollok()`.

With the `killchar()` routine, the user's current line kill character is returned.

The `longname()` routine returns a pointer to a static area containing a verbose description of the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to `initscr()` or `newterm()`. The area is overwritten by each call to `newterm()` and is not restored by `set_term()`, so the value should be saved between calls to `newterm()` if `longname()` is going to be used with multiple terminals.

If a given terminal doesn't support a video attribute that an application program is trying to use, CURSES may substitute a different video attribute for it. The `termattrs()` function returns a logical OR of all video attributes supported by the terminal. This information is useful when a CURSES program needs complete control over the appearance of the screen.

The `termname()` routine returns the value of the environmental variable `TERM` (truncated to 14 characters).

RETURN VALUE

`longname()` and `termname()` return `NULL` on error.

Routines that return an integer return `ERR` upon failure and an integer value other than `ERR` upon successful completion.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that `termattrs()` may be a macro.

SEE ALSO

CURSES(TI_ENV), curs_initscr(TI_LIB), curs_outopts(TI_LIB).

LEVEL

Level 1.

NAME

curs_termcap: tgetent, tgetflag, tgetnum, tgetstr, tgoto, tputs – CURSES interfaces (emulated) to the termcap library

SYNOPSIS

```
#include <curses.h>
#include <term.h>

int tgetent(char *bp, char *name);
int tgetflag(char id[2]);
int tgetnum(char id[2]);
char *tgetstr(char id[2], char **area);
char *tgoto(char *cap, int col, int row);
int tputs(char *str, int affcnt, int (*putc)(void));
```

DESCRIPTION

These routines are included as a conversion aid for programs that use the *termcap* library. Their parameters are the same and the routines are emulated using the *terminfo* database. These routines are supported at Level 2 and should not be used in new applications.

The `tgetent()` routine looks up the termcap entry for *name*. The emulation ignores the buffer pointer *bp*.

The `tgetflag()` routine gets the boolean entry for *id*.

The `tgetnum()` routine gets the numeric entry for *id*.

The `tgetstr()` routine returns the string entry for *id*. Use `tputs()` to output the returned string.

The `tgoto()` routine instantiates the parameters into the given capability. The output from this routine is to be passed to `tputs()`.

The `tputs()` routine is described on the curs_terminfo(TI_LIB) manual page.

RETURN VALUE

Routines that return an integer return ERR upon failure and an integer value other than ERR upon successful completion.

Routines that return pointers return `NULL` on error.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

SEE ALSO

CURSES(TI_ENV), curs_terminfo(TI_LIB), putc(BA_LIB).

LEVEL

Level 1: `tputs()`.

Level 2: December 1, 1985, `tgetent()`, `tgetflag()`, `tgetnum()`, `tgetstr()`, `tgoto()`.

NAME

curs_terminfo: setupterm, setterm, set_curterm, del_curterm, restartterm, tparm, tputs, putp, vidputs, vidattr, mvcur, tigetflag, tigetnum, tigetstr – CURSES interfaces to terminfo database

SYNOPSIS

```
#include <curses.h>
#include <term.h>

int setupterm(char *term, int fildes, int *errret);

int setterm(char *term);

TERMINAL *set_curterm(TERMINAL *nterm);

int del_curterm(TERMINAL *oterm);

int restartterm(char *term, int fildes, int *errret);

char *tparm(char *str, long int p1, long int p2, long int p3,
     long int p4, long int p5, long int p6, long int p7,
     long int p8, long int p9);

int tputs(char *str, int affcnt, int (*putc)(int));

int putp(char *str);

int vidputs(chtype attrs, int (*putc)(int));

int vidattr(chtype attrs);

int mvcur(int oldrow, int oldcol, int newrow, int newcol);

int tigetflag(char *capname);

int tigetnum(char *capname);

int tigetstr(char *capname);
```

DESCRIPTION

These low-level routines must be called by programs that have to deal directly with the *terminfo* database to handle certain terminal capabilities, such as programming function keys. For all other functionality, CURSES routines are more suitable and their use is recommended.

Initially, `setupterm()` should be called. Note that `setupterm()` is automatically called by `initscr()` and `newterm()`. This defines the set of terminal-dependent variables [listed in terminfo(TI_ENV)]. The *terminfo* variables `lines` and `columns` are initialized by `setupterm()` as follows: If `use_env(FALSE)` has been called, values for `lines` and `columns` specified in *terminfo* are used. Otherwise, if the environment variables `LINES` and `COLUMNS` exist, their values are used. If these environment variables do not exist and the program is running in a window, the current window size is used. Otherwise, if the environment variables do not exist, the values for `lines` and `columns` specified in the *terminfo* database are used.

The header files <curses.h> and <term.h> should be included (in this order) to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through `tparm()` to instantiate them. All *terminfo* strings [including the output of `tparm()`] should be printed with `tputs()` or `putp()`. Call the `reset_shell_mode()` to restore the tty modes before exiting [see curs_kernel(TI_LIB)]. Programs which use cursor addressing should output `enter_ca_mode` upon startup and should output `exit_ca_mode` before exiting. Programs desiring shell escapes should call `reset_shell_mode()` and output `exit_ca_mode` before the shell is called and should output `enter_ca_mode` and call `reset_prog_mode()` after returning from the shell.

The `setupterm()` routine reads in the *terminfo* database, initializing the *terminfo* structures, but does not set up the output virtualization structures used by CURSES. The terminal type is the character string *term;* if *term* is null, the environment variable `TERM` is used. All output is to file descriptor `fildes` which is initialized for output. If *errret* is not null, then `setupterm()` returns `OK` or `ERR` and stores a status value in the integer pointed to by *errret*. A status of `1` in *errret* is normal, `0` means that the terminal could not be found, and `-1` means that the *terminfo* database could not be found. If *errret* is null, `setupterm()` prints an error message upon finding an error and exits. Thus, the simplest call is:

```
setupterm((char *)0, 1, (int *)0);,
```

which uses all the defaults and sends the output to `stdout`.

The `setterm()` routine is being replaced by `setupterm()`. The call:

`setupterm(`*term*`, 1, (int *)0)`

provides the same functionality as `setterm(`*term*`)`. The `setterm()` routine is included here for compatibility and is supported at Level 2.

The `set_curterm()` routine sets the variable `cur_term` to *nterm*, and makes all of the *terminfo* boolean, numeric, and string variables use the values from *nterm.*

The `del_curterm()` routine frees the space pointed to by *oterm* and makes it available for further use. If *oterm* is the same as `cur_term`, references to any of the *terminfo* boolean, numeric, and string variables thereafter may refer to invalid memory locations until another `setupterm()` has been called.

The `restartterm()` routine is similar to `setupterm()` and `initscr()`, except that it is called after restoring memory to a previous state. It assumes that the windows and the input and output options are the same as when memory was saved, but the terminal type and baud rate may be different.

The `tparm()` routine instantiates the string *str* with parameters *pi*. A pointer is returned to the result of *str* with the parameters applied.

The `tputs()` routine applies padding information to the string *str* and outputs it. The *str* must be a terminfo string variable or the return value from `tparm()`, `tgetstr()`, or `tgoto()`. *affcnt* is the number of lines affected, or 1 if not applicable. *putc()* is a `putchar()`-like routine to which the characters are passed, one at a time.

The `putp()` routine calls `tputs(`*str*`, 1, putchar)`. Note that the output of `putp()` always goes to `stdout`, not to the *fildes* specified in `setupterm()`.

The `vidputs()` routine displays the string on the terminal in the video attribute mode *attrs*, which is any combination of the attributes listed in CURSES(TI_ENV). The characters are passed to the `putchar()`-like routine *putc()*.

The `vidattr()` routine is like the `vidputs()` routine, except that it outputs through `putchar()`.

The `mvcur()` routine provides low-level cursor motion.

The `tigetflag()`, `tigetnum()` and `tigetstr()` routines return the value of the capability corresponding to the *terminfo capname* passed to them, such as `xenl`.

With the `tigetflag()` routine, the value `-1` is returned if *capname* is not a boolean capability.

With the `tigetnum()` routine, the value `-2` is returned if *capname* is not a numeric capability.

With the `tigetstr()` routine, the value `(char *)-1` is returned if *capname* is not a string capability.

The *capname* for each capability is given in the table column entitled *capname* code in the capabilities section of terminfo(TI_ENV).

`char *boolnames`, `*boolcodes`, `*boolfnames`

`char *numnames`, `*numcodes`, `*numfnames`

`char *strnames`, `*strcodes`, `*strfnames`

These null-terminated arrays contain the *capnames*, the *termcap* codes, and the full C names, for each of the *terminfo* variables.

RETURN VALUE

All routines return the integer `ERR` upon failure and an integer value other than `ERR` upon successful completion, unless otherwise noted in the preceding routine descriptions.

Routines that return pointers always return `NULL` on error.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

The `setupterm()` routine should be used in place of `setterm()`.

Note that `vidattr()` and `vidputs()` may be macros.

SEE ALSO

CURSES(TI_ENV), curs_initscr(TI_LIB), curs_kernel(TI_LIB), curs_termcap(TI_LIB), putc(BA_LIB), terminfo(TI_ENV).

LEVEL

Level 1: `setupterm()`, `set_curterm()`, `del_curterm()`, `restartterm()`, `tparm()`, `tputs()`, `putp()`, `vidputs()`, `vidattr()`, `mvcur()`, `tigetflag()`, `tigetnum()`, `tigetstr()`.

Level 2: December 1985, `setterm()`. `setterm() will be removed in the next issue of the SVID.`

NAME

curs_touch: touchwin, touchline, untouchwin, wtouchln, is_linetouched, is_wintouched – CURSES refresh control routines

SYNOPSIS

```
#include <curses.h>

int touchwin(WINDOW *win);
int touchline(WINDOW *win, int start, int count);
int untouchwin(WINDOW *win);
int wtouchln(WINDOW *win, int y, int n, int changed);
int is_linetouched(WINDOW *win, int line);
int is_wintouched(WINDOW *win);
```

DESCRIPTION

The `touchwin()` and `touchline()` routines throw away all optimization information about which parts of the window have been touched, by pretending that the entire window has been drawn on. This is sometimes necessary when using overlapping windows, since a change to one window affects the other window, but the records of which lines have been changed in the other window do not reflect the change. The routine `touchline()` only pretends that *count* lines have been changed, beginning with line *start*.

The `untouchwin()` routine marks all lines in the window as unchanged since the last call to `wrefresh()`.

The `wtouchln()` routine makes *n* lines in the window, starting at line *y*, look as if they have (*changed*=`1`) or have not (*changed*=`0`) been changed since the last call to `wrefresh()`.

The `is_linetouched()` and `is_wintouched()` routines return `TRUE` if the specified line/window was modified since the last call to `wrefresh()`; otherwise they return `FALSE`. In addition, `is_linetouched()` returns `ERR` if *line* is not valid for the given window.

RETURN VALUE

All routines return the integer `ERR` upon failure and an integer value other than `ERR` upon successful completion, unless otherwise noted in the preceding routine descriptions.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that all routines except `wtouchln()` may be macros.

SEE ALSO

CURSES(TI_ENV), curs_refresh(TI_LIB).

LEVEL

Level 1.

NAME

curs_util: unctrl, keyname, filter, use_env, putwin, getwin, delay_output, flushinp – miscellaneous CURSES utility routines

SYNOPSIS

```
#include <curses.h>

char *unctrl(chtype c);

char *keyname(int c);

void filter(void);

void use_env(char bool);

int putwin(WINDOW *win, FILE *filep);

WINDOW *getwin(FILE *filep);

int delay_output(int ms);

int flushinp(void);
```

DESCRIPTION

The `unctrl()` macro expands to a character string which is a printable representation of the character *c*. Control characters are displayed in the ^*X* notation. Printing characters are displayed as is.

With the `keyname()` routine, a character string corresponding to the key *c* is returned.

The `filter()` routine, if used, is called before `initscr()` or `newterm()` are called. It makes CURSES think that there is a one-line screen. CURSES does not use any terminal capabilities that assume that they know on what line of the screen the cursor is positioned.

The `use_env()` routine, if used, is called before `initscr()` or `newterm()` are called. When called with `FALSE` as an argument, the values of `lines` and `columns` specified in the *terminfo* database will be used, even if environment variables `LINES` and `COLUMNS` (used by default) are set, or if CURSES is running in a window (in which case default behavior would be to use the window size if `LINES` and `COLUMNS` are not set).

With the `putwin()` routine, all data associated with window *win* is written into the file to which *filep* points. This information can be later retrieved using the `getwin()` function.

The `getwin()` routine reads window related data stored in the file by `putwin()`. The routine then creates and initializes a new window using that data. It returns a pointer to the new window.

The `delay_output()` routine inserts an *ms* millisecond pause in output. This routine should not be used extensively because padding characters are used rather than a CPU pause.

The flushinp() routine throws away any typeahead that has been typed by the user and has not yet been read by the program.

RETURN VALUE

Except for flushinp(), routines that return an integer return ERR upon failure and an integer value other than ERR upon successful completion.

flushinp() always returns OK.

Routines that return pointers return NULL on error.

USAGE

Application Program.

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that unctrl() is a macro, which is defined in <unctrl.h>.

SEE ALSO

CURSES(TI_ENV), curs_initscr(TI_LIB), curs_scr_dump(TI_LIB).

LEVEL

Level 1.

NAME

curs_window: newwin, delwin, mvwin, subwin, derwin, mvderwin, dupwin, wsyncup, syncok, wcursyncup, wsyncdown – create CURSES windows

SYNOPSIS

```
#include <curses.h>

WINDOW *newwin(int nlines, int ncols, int begin_y,
     int begin_x);

int delwin(WINDOW *win);

int mvwin(WINDOW *win, int y, int x);

WINDOW *subwin(WINDOW *orig, int nlines, int ncols,
     int begin_y, int begin_x);

WINDOW *derwin(WINDOW *orig, int nlines, int ncols,
     int begin_y, int begin_x);

int mvderwin(WINDOW *win, int par_y, int par_x);

WINDOW *dupwin(WINDOW *win);

void wsyncup(WINDOW *win);

int syncok(WINDOW *win, bool bf);

void wcursyncup(WINDOW *win);

void wsyncdown(WINDOW *win);
```

DESCRIPTION

The `newwin()` routine creates and returns a pointer to a new window with the given number of lines, *nlines*, and columns, *ncols*. The upper left-hand corner of the window is at line *begin_y*, column *begin_x*. If either *nlines* or *ncols* is zero, they default to `LINES` – *begin_y* and `COLS` – *begin_x*. A new full-screen window is created by calling `newwin(0,0,0,0)`.

The `delwin()` routine deletes the named window, freeing all memory associated with it. Subwindows must be deleted before the main window can be deleted.

The `mvwin()` routine moves the window so that the upper left-hand corner is at position (*x*, *y*). If the move would cause the window to be off the screen, it is an error and the window is not moved. Moving subwindows is allowed, but should be avoided.

The `subwin()` routine creates and returns a pointer to a new window with the given number of lines, *nlines*, and columns, *ncols*. The window is at position (*begin_y*, *begin_x*) on the screen. (This position is relative to the screen, and not to the window *orig*.) The window is made in the middle of the window *orig*, so that changes made to one window will affect both windows. The subwindow shares memory with the window *orig*. When using this routine, it is necessary to call `touchwin()` or `touchline()` on *orig* before calling `wrefresh()` on the subwindow.

The `derwin()` routine is the same as `subwin()`, except that *begin_y* and *begin_x* are relative to the origin of the window *orig* rather than the screen. There is no difference between the subwindows and the derived windows.

The `mvderwin()` routine moves a derived window (or subwindow) inside its parent window. The screen-relative parameters of the window are not changed. This routine is used to display different parts of the parent window at the same physical position on the screen.

The `dupwin()` routine creates an exact duplicate of the window *win*.

Each CURSES window maintains two data structures: the character image structure and the status structure. The character image structure is shared among all windows in the window hierarchy (*i.e.*, the window with all subwindows). The status structure, which contains information about individual line changes in the window, is private to each window. The routine `wrefresh()` uses the status data structure when performing screen updating. Since status structures are not shared, changes made to one window in the hierarchy may not be properly reflected on the screen.

The routine `wsyncup()` causes the changes in the status structure of a window to be reflected in the status structures of its ancestors. If `syncok()` is called with second argument `TRUE` then `wsyncup()` is called automatically whenever there is a change in the window.

The routine `wcursyncup()` updates the current cursor position of all the ancestors of the window to reflect the current cursor position of the window.

The routine `wsyncdown()` updates the status structure of the window to reflect the changes in the status structures of its ancestors. Applications seldom call this routine because it is called automatically by `wrefresh()`.

RETURN VALUE

Routines that return an integer return the integer `ERR` upon failure and an integer value other than `ERR` upon successful completion.

`delwin()` returns the integer `ERR` upon failure and `OK` upon successful completion.

Routines that return pointers return `NULL` on error.

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

If many small changes are made to the window, the `wsyncup()` option could degrade performance.

Note that `syncok()` may be a macro.

SEE ALSO

CURSES(TI_ENV), curs_refresh(TI_LIB), curs_touch(TI_LIB).

LEVEL

Level 1.

NAME

form_cursor: pos_form_cursor – position FORMS window cursor

SYNOPSIS

```
#include <form.h>

int pos_form_cursor(FORM *form);
```

DESCRIPTION

`pos_form_cursor()` moves the form window cursor to the location required by the form driver to resume form processing. This may be needed after the application calls a CURSES library I/O routine.

RETURN VALUE

`pos_form_cursor()` returns one of the following:

`E_OK`	– The function returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An argument is incorrect.
`E_NOT_POSTED`	– The form is not posted.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV).

LEVEL

Level 1.

NAME

form_data: data_ahead, data_behind – tell if FORMS field has off-screen data ahead or behind

SYNOPSIS

```
#include <form.h>

int data_ahead(FORM *form);

int data_behind(FORM *form);
```

DESCRIPTION

`data_ahead()` returns `TRUE` (`1`) if the current field has more off-screen data ahead; otherwise it returns `FALSE` (`0`).

`data_behind()` returns `TRUE (1)` if the current field has more off-screen data behind; otherwise it returns `FALSE (0)`.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV).

LEVEL

Level 1.

NAME

form_driver – command processor for the FORMS subsystem

SYNOPSIS

```
#include <form.h>

int form_driver(FORM *form, int c);
```

DESCRIPTION

`form_driver()` is the workhorse of the FORMS subsystem; it checks to determine whether the character *c* is a FORMS request or data. If it is a request, the form driver executes the request and reports the result. If it is data (a printable ASCII character), it enters the data into the current position in the current field. If it is not recognized, the form driver assumes it is an application-defined command and returns `E_UNKNOWN_COMMAND`. Application defined commands should be defined relative to `MAX_COMMAND`, the maximum value of a request listed below.

Form driver requests:

`REQ_NEXT_PAGE`	Move to the next page.
`REQ_PREV_PAGE`	Move to the previous page.
`REQ_FIRST_PAGE`	Move to the first page.
`REQ_LAST_PAGE`	Move to the last page.
`REQ_NEXT_FIELD`	Move to the next field.
`REQ_PREV_FIELD`	Move to the previous field.
`REQ_FIRST_FIELD`	Move to the first field.
`REQ_LAST_FIELD`	Move to the last field.
`REQ_SNEXT_FIELD`	Move to the sorted next field.
`REQ_SPREV_FIELD`	Move to the sorted prev field.
`REQ_SFIRST_FIELD`	Move to the sorted first field.
`REQ_SLAST_FIELD`	Move to the sorted last field.
`REQ_LEFT_FIELD`	Move left to field.
`REQ_RIGHT_FIELD`	Move right to field.
`REQ_UP_FIELD`	Move up to field.
`REQ_DOWN_FIELD`	Move down to field.
`REQ_NEXT_CHAR`	Move to the next character in the field.
`REQ_PREV_CHAR`	Move to the previous character in the field.
`REQ_NEXT_LINE`	Move to the next line in the field.
`REQ_PREV_LINE`	Move to the previous line in the field.
`REQ_NEXT_WORD`	Move to the next word in the field.
`REQ_PREV_WORD`	Move to the previous word in the field.
`REQ_BEG_FIELD`	Move to the first char in the field.
`REQ_END_FIELD`	Move after the last char in the field.
`REQ_BEG_LINE`	Move to the beginning of the line.
`REQ_END_LINE`	Move after the last char in the line.
`REQ_LEFT_CHAR`	Move left in the field.
`REQ_RIGHT_CHAR`	Move right in the field.
`REQ_UP_CHAR`	Move up in the field.

REQ_DOWN_CHAR	Move down in the field.
REQ_NEW_LINE	Insert/overlay a new line.
REQ_INS_CHAR	Insert the blank character at the cursor.
REQ_INS_LINE	Insert a blank line at the cursor.
REQ_DEL_CHAR	Delete the character at the cursor.
REQ_DEL_PREV	Delete the character before the cursor.
REQ_DEL_LINE	Delete the line at the cursor.
REQ_DEL_WORD	Delete the word at the cursor.
REQ_CLR_EOL	Clear to the end of the line.
REQ_CLR_EOF	Clear to the end of the field.
REQ_CLR_FIELD	Clear the entire field.
REQ_OVL_MODE	Enter overlay mode.
REQ_INS_MODE	Enter insert mode.
REQ_SCR_FLINE	Scroll the field forward a line.
REQ_SCR_BLINE	Scroll the field backward a line.
REQ_SCR_FPAGE	Scroll the field forward a page.
REQ_SCR_BPAGE	Scroll the field backward a page.
REQ_SCR_FHPAGE	Scroll the field forward half a page.
REQ_SCR_BHPAGE	Scroll the field backward half a page.
REQ_SCR_FCHAR	Horizontal scroll forward a character.
REQ_SCR_BCHAR	Horizontal scroll backward a character.
REQ_SCR_HFLINE	Horizontal scroll forward a line.
REQ_SCR_HBLINE	Horizontal scroll backward a line.
REQ_SCR_HFHALF	Horizontal scroll forward half a line.
REQ_SCR_HBHALF	Horizontal scroll backward half a line.
REQ_VALIDATION	Validate field.
REQ_PREV_CHOICE	Display the previous field choice.
REQ_NEXT_CHOICE	Display the next field choice.

RETURN VALUE

`form_driver()` returns one of the following:

E_OK	– The function returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An argument is incorrect.
E_NOT_POSTED	– The form is not posted.
E_INVALID_FIELD	– The field contents are invalid.
E_BAD_STATE	– The routine was called from an initialization or termination function.
E_REQUEST_DENIED	– The form driver request failed.
E_UNKNOWN_COMMAND	– An unknown request was passed to the the form driver.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV).

LEVEL

Level 1.

NAME

form_field: set_form_fields, form_fields, field_count, move_field – connect fields to FORMS

SYNOPSIS

```
#include <form.h>

int set_form_fields(FORM *form, FIELD **field);

FIELD **form_fields(FORM *form);

int field_count(FORM *form);

int move_field(FIELD *field, int frow, int fcol);
```

DESCRIPTION

`set_form_fields()` changes the fields connected to *form* to *fields*. The original fields are disconnected.

`form_fields()` returns a pointer to the field pointer array connected to *form*.

`field_count()` returns the number of fields connected to *form*.

`move_field()` moves the disconnected *field* to the location *frow, fcol* in the FORMS subwindow.

RETURN VALUE

`form_fields()` returns `NULL` on error.

`field_count()` returns `-1` on error.

`set_form_fields()` and `move_field()` return one of the following:

`E_OK`	– The function returned successfully.
`E_CONNECTED`	– The field is already connected to a form.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An argument is incorrect.
`E_POSTED`	– The form is posted.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV).

LEVEL

Level 1.

NAME

form_field_attributes: set_field_fore, field_fore, set_field_back, field_back, set_field_pad, field_pad – format the general display attributes of FORMS

SYNOPSIS

```
#include <form.h>

int set_field_fore(FIELD *field, chtype attr);
chtype field_fore(FIELD *field);

int set_field_back(FIELD *field, chtype attr);
chtype field_back(FIELD *field);

int set_field_pad(FIELD *field, int pad);
int field_pad(FIELD *field);
```

DESCRIPTION

`set_field_fore()` sets the foreground attribute of *field*. The foreground attribute is the low-level CURSES display attribute used to display the field contents. `field_fore()` returns the foreground attribute of *field*.

`set_field_back()` sets the background attribute of *field*. The background attribute is the low-level CURSES display attribute used to display the extent of the field. `field_back()` returns the background attribute of *field*.

`set_field_pad()` sets the pad character of *field* to *pad*. The pad character is the character used to fill within the field. `field_pad()` returns the pad character of *field*.

RETURN VALUE

`field_fore()`, `field_back()` and `field_pad()` return default values if *field* is `NULL`. If *field* is not `NULL` and is not a valid `FIELD` pointer, the return value from these routines is undefined.

`set_field_fore()`, `set_field_back()` and `set_field_pad()` return one of the following:

`E_OK`	– The function returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An argument is incorrect.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV).

LEVEL

Level 1.

NAME

form_field_buffer: set_field_buffer, field_buffer, set_field_status, field_status, set_max_field – set and get FORMS field attributes

SYNOPSIS

```
#include <form.h>

int set_field_buffer(FIELD *field, int buf, char *value);
char *field_buffer(FIELD *field, int buf);

int set_field_status(FIELD *field, int status);
int field_status(FIELD *field);

int set_max_field(FIELD *field, int max);
```

DESCRIPTION

`set_field_buffer()` sets buffer *buf* of *field* to *value*. Buffer 0 stores the displayed contents of the field. Buffers other than 0 are application specific and not used by the FORMS library routines. `field_buffer()` returns the value of *field* buffer *buf*.

Every field has an associated status flag that is set whenever the contents of field buffer 0 changes. `set_field_status()` sets the status flag of *field* to *status*. `field_status()` returns the status of *field*.

`set_max_field()` sets a maximum growth on a dynamic field, or if *max*=`0` turns off any maximum growth.

RETURN VALUE

`field_buffer()` returns `NULL` on error.

`field_status()` returns `TRUE` or `FALSE`.

`set_field_buffer()`, `set_field_status()` and `set_max_field()` return one of the following:

`E_OK`	– The function returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An argument is incorrect.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV).

LEVEL

Level 1.

NAME

form_field_info: field_info, dynamic_field_info – get FORMS field characteristics

SYNOPSIS

```
#include <form.h>

int field_info(FIELD *field, int *rows, int *cols,
     int *frow, int *fcol, int *nrow, int *nbuf);

int dynamic_field_info(FIELD *field, int *drows, int *dcols,
     int *max);
```

DESCRIPTION

`field_info()` returns the size, position, and other named field characteristics, as defined in the original call to `new_field()`, to the locations pointed to by the arguments *rows*, *cols*, *frow*, *fcol*, *nrow*, and *nbuf*.

`dynamic_field_info()` returns the actual size of the *field* in the pointer arguments *drows*, *dcols* and returns the maximum growth allowed for *field* in *max*. If no maximum growth limit is specified for *field*, *max* will contain 0. A field can be made dynamic by turning off the field option `O_STATIC`.

RETURN VALUE

These routines return one of the following:

`E_OK`	– The function returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An argument is incorrect.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV).

LEVEL

Level 1.

FINAL COPY
June 15, 1995
File: ti_lib/form_f_info
svid

NAME

form_field_just: set_field_just, field_just – format the general appearance of FORMS

SYNOPSIS

```
#include <form.h>

int set_field_just(FIELD *field, int justification);

int field_just(FIELD *field);
```

DESCRIPTION

`set_field_just()` sets the justification for *field*. Justification may be one of:

> `NO_JUSTIFICATION`, `JUSTIFY_RIGHT`, `JUSTIFY_LEFT`, or `JUSTIFY_CENTER`.

The field justification will be ignored if *field* is a dynamic field.

`field_just()` returns the type of justification assigned to *field*.

RETURN VALUE

`field_just()` returns the one of:

> `NO_JUSTIFICATION`, `JUSTIFY_RIGHT`, `JUSTIFY_LEFT`, or `JUSTIFY_CENTER`.

set_field_just() returns one of the following:

`E_OK`	– The function returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An argument is incorrect.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV).

LEVEL

Level 1.

NAME

form_field_new: new_field, dup_field, link_field, free_field, – create and destroy FORMS fields

SYNOPSIS

```
#include <form.h>

FIELD *new_field(int r, int c, int frow, int fcol,
     int nrow, int ncol);

FIELD *dup_field(FIELD *field, int frow, int fcol);

FIELD *link_field(FIELD *field, int frow, int fcol);

int free_field(FIELD *field);
```

DESCRIPTION

`new_field()` creates a new field with *r* rows and *c* columns, starting at *frow*, *fcol*, in the subwindow of a form. *nrow* is the number of off-screen rows and *nbuf* is the number of additional working buffers. This routine returns a pointer to the new field.

`dup_field()` duplicates *field* at the specified location. All field attributes are duplicated, including the current contents of the field buffers.

`link_field()` also duplicates *field* at the specified location. However, unlike `dup_field()`, the new field shares the field buffers with the original field. After creation, the attributes of the new field can be changed without affecting the original field.

`free_field()` frees the storage allocated for *field*.

RETURN VALUE

Routines that return pointers return `NULL` on error. `free_field()` returns one of the following:

`E_OK`	– The function returned successfully.
`E_CONNECTED`	– The field is already connected to a form.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An argument is incorrect.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

FORMS(TI_ENV).

LEVEL

Level 1.

NAME

form_field_opts: set_field_opts, field_opts_on, field_opts_off, field_opts – FORMS field option routines

SYNOPSIS

```
#include <form.h>

int set_field_opts(FIELD *field, OPTIONS opts);
int field_opts_on(FIELD *field, OPTIONS opts);
int field_opts_off(FIELD *field, OPTIONS opts);
OPTIONS field_opts(FIELD *field);
```

DESCRIPTION

`set_field_opts()` turns on the named options of *field* and turns off all remaining options. Options are boolean values that can be OR-ed together.

`field_opts_on()` turns on the named options; no other options are changed.

`field_opts_off()` turns off the named options; no other options are changed.

`field_opts()` returns the options set for *field*.

Field Options:

`O_VISIBLE`	The field is displayed.
`O_ACTIVE`	The field is visited during processing.
`O_PUBLIC`	The field contents are displayed as data is entered.
`O_EDIT`	The field can be edited.
`O_WRAP`	Words not fitting on a line are wrapped to the next line.
`O_BLANK`	The whole field is cleared if a character is entered in the first position.
`O_AUTOSKIP`	Skip to the next field when the current field becomes full.
`O_NULLOK`	A blank field is considered valid.
`O_STATIC`	The field buffers are fixed in size.
`O_PASSOK`	Validate field only if modified by user.

RETURN VALUE

`set_field_opts()`, `field_opts_on()` and `field_opts_off()` return one of the following:

`E_OK`	– The function returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_CURRENT`	– The field is the current field.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV).

 form_field_opts (TI_LIB)

LEVEL

Level 1.

NAME

form_field_userptr: set_field_userptr, field_userptr – associate application data with FORMS

SYNOPSIS

```
#include <form.h>

int set_field_userptr(FIELD *field, char *ptr);
char *field_userptr(FIELD *field);
```

DESCRIPTION

Every field has an associated user pointer that can be used to store pertinent data. `set_field_userptr()` sets the user pointer of *field*. `field_userptr()` returns the user pointer of *field*.

RETURN VALUE

`field_userptr()` returns `NULL` on error. `set_field_userptr()` returns one of the following:

`E_OK`	– The function returned successfully.
`E_SYSTEM_ERROR`	– System error.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV).

LEVEL

Level 1.

NAME

form_field_validation: set_field_type, field_type, field_arg – FORMS field data type validation

SYNOPSIS

```
#include <form.h>

int set_field_type(FIELD *field, FIELDTYPE *type, ...);

FIELDTYPE *field_type(FIELD *field);

char *field_arg(FIELD *field);
```

DESCRIPTION

`set_field_type()` associates the specified field type with *field*. Certain field types take additional arguments. `TYPE_ALNUM`, for instance, requires one, the minimum width specification for the field. The other predefined field types are: `TYPE_ALPHA`, `TYPE_ENUM`, `TYPE_INTEGER`, `TYPE_NUMERIC`, `TYPE_REGEXP`.

`field_type()` returns a pointer to the field type of *field*. `NULL` is returned if no field type is assigned.

`field_arg()` returns a pointer to the field arguments associated with the field type of *field*. `NULL` is returned if no field type is assigned.

RETURN VALUE

`field_type()` and `field_arg()` return `NULL` on error.

`set_field_type()` returns one of the following:

`E_OK`	– The function returned successfully.
`E_SYSTEM_ERROR`	– System error.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV).

LEVEL

Level 1.

NAME

form_fieldtype: new_fieldtype, free_fieldtype, set_fieldtype_arg, set_fieldtype_choice, link_fieldtype – FORMS fieldtype routines

SYNOPSIS

```
#include <form.h>

FIELDTYPE *new_fieldtype(int (* field_check)(FIELD *, char *),
     int (* char_check)(int, char *));

int free_fieldtype(FIELDTYPE *fieldtype);

int set_fieldtype_arg(FIELDTYPE *fieldtype,
     char *(* mak_arg)(va_list *),
     char *(* copy_arg)(char *), void (* free_arg)(char *));

int set_fieldtype_choice(FIELDTYPE *fieldtype,
     int (* next_choice)(FIELD *, char *),
     int (* prev_choice)(FIELD *, char *));

FIELDTYPE *link_fieldtype(FIELDTYPE *type1, FIELDTYPE *type2);
```

DESCRIPTION

`new_fieldtype()` creates a new field type. The application programmer must write the function *field_check()*, which validates the field value, and the function *char_check()*, which validates each character. `free_fieldtype()` frees the space allocated for the field type.

By associating function pointers with a field type, `set_fieldtype_arg()` connects to the field type additional arguments necessary for a `set_field_type()` call. Function *mak_arg()* allocates a structure for the field specific parameters to `set_field_type()` and returns a pointer to the saved data. Function *copy_arg()* duplicates the structure created by *make_arg()*. Function *free_arg()* frees any storage allocated by *make_arg()* or *copy_arg()*.

The `form_driver()` requests `REQ_NEXT_CHOICE` and `REQ_PREV_CHOICE` let the user request the next or previous value of a field type comprising an ordered set of values. `set_fieldtype_choice()` allows the application programmer to implement these requests for the given field type. It associates with the given field type those application-defined functions that return pointers to the next or previous choice for the field.

`link_fieldtype()` returns a pointer to the field type built from the two given types. The constituent types may be any application-defined or pre-defined types.

RETURN VALUE

Routines that return pointers always return `NULL` on error. Routines that return an integer return one of the following:

`E_OK`	– The function returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An argument is incorrect.
`E_CONNECTED`	– Type is connected to one or more fields.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV).

LEVEL

Level 1.

NAME

form_hook: set_form_init, form_init, set_form_term, form_term, set_field_init, field_init, set_field_term, field_term – assign application-specific routines for invocation by FORMS

SYNOPSIS

```
#include <form.h>

int set_form_init(FORM *form, void (*func)(FORM *));
void (*)(FORM *) form_init(FORM *form);

int set_form_term(FORM *form, void (*func)(FORM *));
void (*)(FORM *) form_term(FORM *form);

int set_field_init(FORM *form, void (*func)(FORM *));
void (*)(FORM *) field_init(FORM *form);

int set_field_term(FORM *form, void (*func)(FORM *));
void (*)(FORM *) field_term(FORM *form);
```

DESCRIPTION

These routines allow the programer to assign application specific routines to be executed automatically at initialization and termination points in the FORMS application. The user need not specify any application-defined initialization or termination routines at all, but they may be helpful for displaying messages or page numbers and other chores.

`set_form_init()` assigns an application-defined initialization function to be called when the *form* is posted and just after a page change. `form_init()` returns a pointer to the initialization function, if any.

`set_form_term()` assigns an application-defined function to be called when the *form* is unposted and just before a page change. `form_term()` returns a pointer to the function, if any.

`set_field_init()` assigns an application-defined function to be called when the *form* is posted and just after the current field changes. `field_init()` returns a pointer to the function, if any.

`set_field_term()` assigns an application-defined function to be called when the *form* is unposted and just before the current field changes. `field_term()` returns a pointer to the function, if any.

RETURN VALUE

Routines that return pointers always return `NULL` on error. Routines that return an integer return one of the following:

`E_OK`	– The function returned successfully.
`E_SYSTEM_ERROR`	– System error.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV).

LEVEL

Level 1.

NAME

form_new: new_form, free_form – create and destroy FORMS

SYNOPSIS

```
#include <form.h>

FORM *new_form(FIELD **fields);

int free_form(FORM *form);
```

DESCRIPTION

`new_form()` creates a new form connected to the designated fields and returns a pointer to the form.

`free_form()` disconnects the *form* from its associated field pointer array and deallocates the space for the form.

RETURN VALUE

`new_form()` always returns `NULL` on error. `free_form()` returns one of the following:

`E_OK`	– The function returned successfully.
`E_BAD_ARGUMENT`	– An argument is incorrect.
`E_POSTED`	– The form is posted.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV).

LEVEL

Level 1.

NAME

form_new_page: set_new_page, new_page – FORMS pagination

SYNOPSIS

```
#include <form.h>

int set_new_page(FIELD *field, int bool);

int new_page(FIELD *field);
```

DESCRIPTION

`set_new_page()` marks *field* as the beginning of a new page on the form.

`new_page()` returns a boolean value indicating whether or not *field* begins a new page of the form.

RETURN VALUE

`new_page()` returns `TRUE` or `FALSE`.

`set_new_page()` returns one of the following:

`E_OK`	– The function returned successfully.
`E_CONNECTED`	– The field is already connected to a form.
`E_SYSTEM_ERROR`	– System error.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV).

LEVEL

Level 1.

NAME

form_opts: set_form_opts, form_opts_on, form_opts_off, form_opts – FORMS option routines

SYNOPSIS

```
#include <form.h>

int set_form_opts(FORM *form, OPTIONS opts);
int form_opts_on(FORM *form, OPTIONS opts);
int form_opts_off(FORM *form, OPTIONS opts);
OPTIONS form_opts(FORM *form);
```

DESCRIPTION

`set_form_opts()` turns on the named options for *form* and turns off all remaining options. Options are boolean values which can be OR-ed together.

`form_opts_on()` turns on the named options; no other options are changed.

`form_opts_off()` turns off the named options; no other options are changed.

`form_opts()` returns the options set for *form*.

Form Options:

O_NL_OVERLOAD	Overload the REQ_NEW_LINE form driver request.
O_BS_OVERLOAD	Overload the REQ_DEL_PREV form driver request.

RETURN VALUE

`set_form_opts()`, `form_opts_on()` and `form_opts_off()` return one of the following:

E_OK	– The function returned successfully.
E_SYSTEM_ERROR	– System error.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV).

LEVEL

Level 1.

NAME

form_page: set_form_page, form_page, set_current_field, current_field, field_index – set FORMS current page and field

SYNOPSIS

```
#include <form.h>

int set_form_page(FORM *form, int page);
int form_page(FORM *form);

int set_current_field(FORM *form, FIELD *field);
FIELD *current_field(FORM *form);

int field_index(FIELD *field);
```

DESCRIPTION

`set_form_page()` sets the page number of *form* to *page*. `form_page()` returns the current page number of *form*.

`set_current_field()` sets the current field of *form* to *field*. `current_field()` returns a pointer to the current field of *form*.

`field_index()` returns the index in the field pointer array of *field*.

RETURN VALUE

`form_page()` returns `-1` on error.

`current_field()` returns `NULL` on error.

`field_index()` returns `-1` on error.

`set_form_page()` and `set_current_field()` return one of the following:

`E_OK`	– The function returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An argument is incorrect.
`E_BAD_STATE`	– The routine was called from an initialization or termination function.
`E_INVALID_FIELD`	– The field contents are invalid.
`E_REQUEST_DENIED`	– The form driver request failed.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV).

LEVEL

Level 1.

NAME

form_post: post_form, unpost_form – write or erase FORMS from associated subwindows

SYNOPSIS

```
#include <form.h>

int post_form(FORM *form);

int unpost_form(FORM *form);
```

DESCRIPTION

`post_form()` writes *form* into its associated subwindow. The application programmer must use CURSES library routines to display the form on the physical screen or call `update_panels()` if the PANELS library is being used.

`unpost_form()` erases *form* from its associated subwindow.

RETURN VALUE

These routines return one of the following:

`E_OK`	– The function returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An argument is incorrect.
`E_POSTED`	– The form is posted.
`E_NOT_POSTED`	– The form is not posted.
`E_NO_ROOM`	– The form does not fit in the subwindow.
`E_BAD_STATE`	– The routine was called from an initialization or termination function.
`E_NOT_CONNECTED`	– The field is not connected to a form.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV), PANELS(TI_ENV), panel_update(TI_LIB).

LEVEL

Level 1.

NAME

form_userptr: set_form_userptr, form_userptr – associate application data with FORMS

SYNOPSIS

```
#include <form.h>

int set_form_userptr(FORM *form, char *ptr);
char *form_userptr(FORM *form);
```

DESCRIPTION

Every form has an associated user pointer that can be used to store pertinent data. `set_form_userptr()` sets the user pointer of *form*. `form_userptr()` returns the user pointer of *form*.

RETURN VALUE

`form_userptr()` returns `NULL` on error. `set_form_userptr()` returns one of the following:

`E_OK`	– The function returned successfully.
`E_SYSTEM_ERROR`	– System error.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV).

LEVEL

Level 1.

NAME

form_win: set_form_win, form_win, set_form_sub, form_sub, scale_form – FORMS window and subwindow association routines

SYNOPSIS

```
#include <form.h>

int set_form_win(FORM *form, WINDOW *win);
WINDOW *form_win(FORM *form);

int set_form_sub(FORM *form, WINDOW *sub);
WINDOW *form_sub(FORM *form);

int scale_form(FORM *form, int *rows, int *cols);
```

DESCRIPTION

`set_form_win()` sets the window of *form* to *win*. `form_win()` returns a pointer to the window associated with *form*.

`set_form_sub()` sets the subwindow of *form* to *sub*. `form_sub()` returns a pointer to the subwindow associated with *form*.

`scale_form()` returns the smallest window size necessary for the subwindow of *form*. *rows* and *cols* are pointers to the locations used to return the number of rows and columns for the form.

RETURN VALUE

Routines that return pointers always return `NULL` on error. Routines that return an integer return one of the following:

`E_OK`	– The function returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An argument is incorrect.
`E_NOT_CONNECTED`	– The field is not connected to a form.
`E_POSTED`	– The form is posted.

USAGE

Application Program.

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), FORMS(TI_ENV).

LEVEL

Level 1.

NAME

menu_attributes: set_menu_fore, menu_fore, set_menu_back, menu_back, set_menu_grey, menu_grey, set_menu_pad, menu_pad – control MENUS display attributes

SYNOPSIS

```
#include <menu.h>

int set_menu_fore(MENU *menu, chtype attr);
chtype menu_fore(MENU *menu);

int set_menu_back(MENU *menu, chtype attr);
chtype menu_back(MENU *menu);

int set_menu_grey(MENU *menu, chtype attr);
chtype menu_grey(MENU *menu);

int set_menu_pad(MENU *menu, int pad);
int menu_pad(MENU *menu);
```

DESCRIPTION

`set_menu_fore()` sets the foreground attribute of *menu* — the display attribute for the current item (if selectable) on single-valued menus and for selected items on multi-valued menus. This display attribute is a CURSES library visual attribute. `menu_fore()` returns the foreground attribute of *menu*.

`set_menu_back()` sets the background attribute of `menu` — the display attribute for unselected, yet selectable, items. This display attribute is a CURSES library visual attribute.

`set_menu_grey()` sets the grey attribute of *menu* — the display attribute for non-selectable items in multi-valued menus. This display attribute is a CURSES library visual attribute. `menu_grey()` returns the grey attribute of *menu*.

The pad character is the character that fills the space between the name and description of an item. `set_menu_pad()` sets the pad character for *menu* to *pad*. `menu_pad()` returns the pad character of *menu*.

RETURN VALUE

These routines return one of the following:

E_OK	–	The routine returned successfully.
E_SYSTEM_ERROR	–	System error.
E_BAD_ARGUMENT	–	An incorrect argument was passed to the routine.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV).

LEVEL

Level 1.

NAME

menu_cursor: pos_menu_cursor – correctly position a MENUS cursor

SYNOPSIS

```
#include <menu.h>

int pos_menu_cursor(MENU *menu);
```

DESCRIPTION

`pos_menu_cursor()` moves the cursor in the window of *menu* to the correct position to resume menu processing. This is needed after the application calls a CURSES library I/O routine.

RETURN VALUE

This routine returns one of the following:

`E_OK`	– The routine returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An incorrect argument was passed to the routine.
`E_NOT_POSTED`	– The menu has not been posted.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV), PANELS(TI_ENV), panel_update(TI_LIB).

LEVEL

Level 1.

NAME

menu_driver – command processor for the MENUS subsystem

SYNOPSIS

```
#include <menu.h>

int menu_driver(MENU *menu, int c);
```

DESCRIPTION

`menu_driver()` is the workhorse of the MENUS subsystem. It checks to determine whether the character *c* is a menu request or data. If *c* is a request, the menu driver executes the request and reports the result. If *c* is data (a printable ASCII character), it enters the data into the pattern buffer and tries to find a matching item. If no match is found, the menu driver deletes the character from the pattern buffer and returns `E_NO_MATCH`. If the character is not recognized, the menu driver assumes it is an application-defined command and returns `E_UNKNOWN_COMMAND`.

Menu driver requests:

`REQ_LEFT_ITEM`	Move left to an item.
`REQ_RIGHT_ITEM`	Move right to an item.
`REQ_UP_ITEM`	Move up to an item.
`REQ_DOWN_ITEM`	Move down to an item.
`REQ_SCR_ULINE`	Scroll up a line.
`REQ_SCR_DLINE`	Scroll down a line.
`REQ_SCR_DPAGE`	Scroll up a page.
`REQ_SCR_UPAGE`	Scroll down a page.
`REQ_FIRST_ITEM`	Move to the first item.
`REQ_LAST_ITEM`	Move to the last item.
`REQ_NEXT_ITEM`	Move to the next item.
`REQ_PREV_ITEM`	Move to the previous item.
`REQ_TOGGLE_ITEM`	Select/de-select an item.
`REQ_CLEAR_PATTERN`	Clear the menu pattern buffer.
`REQ_BACK_PATTERN`	Delete the previous character from pattern buffer.
`REQ_NEXT_MATCH`	Move the next matching item.
`REQ_PREV_MATCH`	Move to the previous matching item.

RETURN VALUE

`menu_driver()` returns one of the following:

`E_OK`	– The routine returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An incorrect argument was passed to the routine.
`E_BAD_STATE`	– The routine was called from an initialization or termination function.
`E_NOT_POSTED`	– The menu has not been posted.
`E_UNKNOWN_COMMAND`	– An unknown request was passed to the menu driver.

E_NO_MATCH	–	The character failed to match.
E_NOT_SELECTABLE	–	The item cannot be selected.
E_REQUEST_DENIED	–	The menu driver could not process the request.

USAGE

Application Program.

Application defined commands should be defined relative to (greater than) MAX_COMMAND, the maximum value of a request listed above.

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV).

LEVEL

Level 1.

NAME

menu_format: set_menu_format, menu_format – set and get maximum numbers of rows and columns in MENUS

SYNOPSIS

`#include <menu.h>`

`int set_menu_format(MENU *`*menu*`, int `*rows*`, int `*cols*`);`

`void menu_format(MENU *`*menu*`, int *`*rows*`, int *`*cols*`);`

DESCRIPTION

`set_menu_format()` sets the maximum number of rows and columns of items that may be displayed at one time on a menu. If the menu contains more items than can be displayed at once, the menu will be scrollable.

`menu_format()` returns the maximum number of rows and columns that may be displayed at one time on *menu*. *rows* and *cols* are pointers to the variables used to return these values.

RETURN VALUE

`set_menu_format()` returns one of the following:

`E_OK`	– The routine returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An incorrect argument was passed to the routine.
`E_POSTED`	– The menu is already posted.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV).

LEVEL

Level 1.

NAME

menu_hook: set_item_init, item_init, set_item_term, item_term, set_menu_init, menu_init, set_menu_term, menu_term – assign application-specific routines for automatic invocation by MENUS

SYNOPSIS

```
#include <menu.h>

int set_item_init(MENU *menu, void (*func)(MENU *));
void (*)(MENU *) item_init(MENU *menu);

int set_item_term(MENU *menu, void (*func)(MENU *));
void (*)(MENU *) item_term(MENU *menu);

int set_menu_init(MENU *menu, void (*func)(MENU *));
void (*)(MENU *) menu_init(MENU *menu);

int set_menu_term(MENU *menu, void (*func)(MENU *));
void (*)(MENU *) menu_term(MENU *menu);
```

DESCRIPTION

`set_item_init()` assigns the application-defined function to be called when the *menu* is posted and just after the current item changes. `item_init()` returns a pointer to the item initialization routine, if any, called when the *menu* is posted and just after the current item changes.

`set_item_term()` assigns an application-defined function to be called when the *menu* is unposted and just before the current item changes. `item_term()` returns a pointer to the termination function, if any, called when the *menu* is unposted and just before the current item changes.

`set_menu_init()` assigns an application-defined function to be called when the *menu* is posted and just after the top row changes on a posted menu. `menu_init()` returns a pointer to the menu initialization routine, if any, called when the *menu* is posted and just after the top row changes on a posted menu.

`set_menu_term()` assigns an application-defined function to be called when the *menu* is unposted and just before the top row changes on a posted menu. `menu_term()` returns a pointer to the menu termination routine, if any, called when the *menu* is unposted and just before the top row changes on a posted menu.

RETURN VALUE

Routines that return pointers always return `NULL` on error. Routines that return an integer return one of the following:

`E_OK`	– The routine returned successfully.
`E_SYSTEM_ERROR`	– System error.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV), menu_control(TI_LIB), menu_hook(TI_LIB).

LEVEL

Level 1.

NAME

menu_item_current: set_current_item, current_item, set_top_row, top_row, item_index – set and get current MENUS items

SYNOPSIS

```
#include <menu.h>

int set_current_item(MENU *menu, ITEM *item);
ITEM *current_item(MENU *menu);

int set_top_row(MENU *menu, int row);
int top_row(MENU *menu);

int item_index(ITEM *item);
```

DESCRIPTION

The current item of a menu is the item where the cursor is currently positioned. `set_current_item()` sets the current item of *menu* to *item*. `current_item()` returns a pointer to the the current item in *menu*.

`set_top_row()` sets the top row of *menu* to *row*. The left-most item on the new top row becomes the current item. `top_row()` returns the number of the menu row currently displayed at the top of *menu*.

`item_index()` returns the index to the *item* in the item pointer array. The value of this index ranges from `0` through *N*`-1`, where *N* is the total number of items connected to the menu.

RETURN VALUE

`current_item()` returns `NULL` on error.

`top_row()` and `index_item()` return `-1` on error.

`set_current_item()` and `set_top_row()` return one of the following:

`E_OK`	– The routine returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An incorrect argument was passed to the routine.
`E_BAD_STATE`	– The routine was called from an initialization or termination function.
`E_NOT_CONNECTED`	– No items are connected to the menu.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV).

LEVEL

Level 1.

NAME

menu_item_name: item_name, item_description – get MENUS item name and description

SYNOPSIS

```
#include <menu.h>

char *item_name(ITEM *item);

char *item_description(ITEM *item);
```

DESCRIPTION

`item_name()` returns a pointer to the name of *item*.

`item_description()` returns a pointer to the description of *item*.

RETURN VALUE

These routines return `NULL` on error.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV), menu_new(TI_LIB).

LEVEL

Level 1.

NAME

menu_item_new: new_item, free_item – create and destroy MENUS items

SYNOPSIS

```
#include <menu.h>

ITEM *new_item(char *name, char *desc);

int free_item(ITEM *item);
```

DESCRIPTION

`new_item()` creates a new item from *name* and *description*, and returns a pointer to the new item.

`free_item()` frees the storage allocated for *item*. Once an item is freed, the user can no longer connect it to a menu.

RETURN VALUE

`new_item()` returns `NULL` on error.

`free_item()` returns one of the following:

`E_OK`	– The routine returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An incorrect argument was passed to the routine.
`E_CONNECTED`	– One or more items are already connected to another menu.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV).

LEVEL

Level 1.

NAME

menu_item_opts: set_item_opts, item_opts_on, item_opts_off, item_opts – MENUS item option routines

SYNOPSIS

```
#include <menu.h>

int set_item_opts(ITEM *item, OPTIONS opts);
int item_opts_on(ITEM *item, OPTIONS opts);
int item_opts_off(ITEM *item, OPTIONS opts);
OPTIONS item_opts(ITEM *item);
```

DESCRIPTION

`set_item_opts()` turns on the named options for *item* and turns off all other options. Options are boolean values that can be OR-ed together.

`item_opts_on()` turns on the named options for *item*; no other option is changed.

`item_opts_off()` turns off the named options for *item*; no other option is changed.

`item_opts()` returns the current options of *item*.

Item Options:

`O_SELECTABLE` The item can be selected during menu processing.

RETURN VALUE

Except for `item_opts()`, these routines return one of the following:

`E_OK`	– The routine returned successfully.
`E_SYSTEM_ERROR`	– System error.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV).

LEVEL

Level 1.

NAME

menu_item_userptr: set_item_userptr, item_userptr – associate application data with MENUS items

SYNOPSIS

```
#include <menu.h>

int set_item_userptr(ITEM *item, char *userptr);

char *item_userptr(ITEM *item);
```

DESCRIPTION

Every item has an associated user pointer that can be used to store relevant information. `set_item_userptr()` sets the user pointer of *item*. `item_userptr()` returns the user pointer of *item*.

RETURN VALUE

`item_userptr()` returns `NULL` on error. `set_item_userptr()` returns one of the following:

`E_OK`	– The routine returned successfully.
`E_SYSTEM_ERROR`	– System error.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV).

LEVEL

Level 1.

NAME

menu_item_value: set_item_value, item_value – set and get MENUS item values

SYNOPSIS

```
#include <menu.h>

int set_item_value(ITEM *item, int bool);

int item_value(ITEM *item);
```

DESCRIPTION

Unlike single-valued menus, multi-valued menus enable the end-user to select one or more items from a menu. `set_item_value()` sets the selected value of the *item* — `TRUE` (selected) or `FALSE` (not selected). `set_item_value()` may be used only with multi-valued menus. To make a menu multi-valued, use `set_menu_opts()` or `menu_opts_off()` to turn off the option `O_ONEVALUE`. [see menu_opts(TI_LIB)].

`item_value()` returns the select value of *item*, either `TRUE` (selected) or `FALSE` (unselected).

RETURN VALUE

`set_item_value()` returns one of the following:

`E_OK`	– The routine returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_REQUEST_DENIED`	– The menu driver could not process the request.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV), menu_opts(TI_LIB).

LEVEL

Level 1.

NAME

menu_item_visible: item_visible – tell if MENUS item is visible

SYNOPSIS

```
#include <menu.h>

int item_visible(ITEM *item);
```

DESCRIPTION

A menu item is visible if it currently appears in the subwindow of a posted menu. `item_visible()` returns `TRUE` if *item* is visible, otherwise it returns `FALSE`.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV), menu_new(TI_LIB).

LEVEL

Level 1.

NAME

menu_items: set_menu_items, menu_items, item_count – connect and disconnect items to and from MENUS

SYNOPSIS

```
#include <menu.h>

int set_menu_items(MENU *menu, ITEM **items);

ITEM **menu_items(MENU *menu);

int item_count(MENU *menu);
```

DESCRIPTION

`set_menu_items()` changes the item pointer array connected to *menu* to the item pointer array *items*.

`menu_items()` returns a pointer to the item pointer array connected to *menu*.

`item_count()` returns the number of items in *menu*.

RETURN VALUE

`menu_items()` returns `NULL` on error.

`item_count()` returns -1 on error.

`set_menu_items()` returns one of the following:

`E_OK`	– The routine returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An incorrect argument was passed to the routine.
`E_POSTED`	– The menu is already posted.
`E_CONNECTED`	– One or more items are already connected to another menu.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV).

LEVEL

Level 1.

NAME

menu_mark: set_menu_mark, menu_mark – MENUS mark string routines

SYNOPSIS

```
#include <menu.h>

int set_menu_mark(MENU *menu, char *mark);

char *menu_mark(MENU *menu);
```

DESCRIPTION

MENUS displays mark strings to distinguish selected items in a menu (or the current item in a single-valued menu). `set_menu_mark()` sets the mark string of *menu* to *mark*. `menu_mark()` returns a pointer to the mark string of *menu*.

RETURN VALUE

`menu_mark()` returns `NULL` on error. `set_menu_mark()` returns one of the following:

`E_OK`	– The routine returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An incorrect argument was passed to the routine.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV).

LEVEL

Level 1.

NAME

menu_new: new_menu, free_menu – create and destroy MENUS

SYNOPSIS

```
#include <menu.h>

MENU *new_menu(ITEM **items);

int free_menu(MENU *menu);
```

DESCRIPTION

`new_menu()` creates a new menu connected to the item pointer array *items* and returns a pointer to the new menu.

`free_menu()` disconnects *menu* from its associated item pointer array and frees the storage allocated for the menu.

RETURN VALUE

`new_menu()` returns `NULL` on error.

`free_menu()` returns one of the following:

`E_OK`	– The routine returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An incorrect argument was passed to the routine.
`E_POSTED`	– The menu is already posted.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV).

LEVEL

Level 1.

NAME

menu_opts: set_menu_opts, menu_opts_on, menu_opts_off, menu_opts – MENUS option routines

SYNOPSIS

```
#include <menu.h>

int set_menu_opts(MENU *menu, OPTIONS opts);
int menu_opts_on(MENU *menu, OPTIONS opts);
int menu_opts_off(MENU *menu, OPTIONS opts);
OPTIONS menu_opts(MENU *menu);
```

DESCRIPTION

Menu Options

`set_menu_opts()` turns on the named options for *menu* and turns off all other options. Options are boolean values that can be OR-ed together.

`menu_opts_on()` turns on the named options for *menu*; no other option is changed.

`menu_opts_off()` turns off the named options for *menu*; no other option is changed.

`menu_opts()` returns the current options of *menu*.

Menu Options:

`O_ONEVALUE`	Only one item can be selected from the menu.
`O_SHOWDESC`	Display the description of the items.
`O_ROWMAJOR`	Display the menu in row major order.
`O_IGNORECASE`	Ignore the case when pattern matching.
`O_SHOWMATCH`	Place the cursor within the item name when pattern matching.
`O_NONCYCLIC`	Make certain menu driver requests non-cyclic.

RETURN VALUE

Except for `menu_opts()`, these routines return one of the following:

`E_OK`	– The routine returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_POSTED`	– The menu is already posted.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV).

LEVEL

Level 1.

NAME

menu_pattern: set_menu_pattern, menu_pattern – set and get MENUS pattern match buffer

SYNOPSIS

```
#include <menu.h>

int set_menu_pattern(MENU *menu, char *pat);

char *menu_pattern(MENU *menu);
```

DESCRIPTION

Every menu has a pattern buffer to match entered data with menu items. `set_menu_pattern()` sets the pattern buffer to *pat* and tries to find the first item that matches the pattern. If it does, the matching item becomes the current item. If not, the current item does not change. `menu_pattern()` returns the string in the pattern buffer of *menu*.

RETURN VALUE

`menu_pattern()` returns `NULL` on error. `set_menu_pattern()` returns one of the following:

E_OK	– The routine returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An incorrect argument was passed to the routine.
E_NO_MATCH	– The character failed to match.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV).

LEVEL

Level 1.

NAME

menu_post: post_menu, unpost_menu – write or erase MENUS from associated subwindows

SYNOPSIS

```
#include <menu.h>

int post_menu(MENU *menu);

int unpost_menu(MENU *menu);
```

DESCRIPTION

`post_menu()` writes *menu* to the subwindow. The application programmer must use CURSES library routines to display the menu on the physical screen or call `update_panels()` if the PANELS library is being used.

`unpost_menu()` erases *menu* from its associated subwindow.

RETURN VALUE

These routines return one of the following:

`E_OK`	– The routine returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An incorrect argument was passed to the routine.
`E_POSTED`	– The menu is already posted.
`E_BAD_STATE`	– The routine was called from an initialization or termination function.
`E_NO_ROOM`	– The menu does not fit within its subwindow.
`E_NOT_POSTED`	– The menu has not been posted.
`E_NOT_CONNECTED`	– No items are connected to the menu.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV), PANELS(TI_ENV).

LEVEL

Level 1.

NAME

menu_userptr: set_menu_userptr, menu_userptr – associate application data with MENUS

SYNOPSIS

```
#include <menu.h>

int set_menu_userptr(MENU *menu, char *userptr);
char *menu_userptr(MENU *menu);
```

DESCRIPTION

Every menu has an associated user pointer that can be used to store relevant information. `set_menu_userptr()` sets the user pointer of *menu*. `menu_userptr()` returns the user pointer of *menu*.

RETURN VALUE

`menu_userptr()` returns `NULL` on error.

`set_menu_userptr()` returns one of the following:

`E_OK`	– The routine returned successfully.
`E_SYSTEM_ERROR`	– System error.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV).

LEVEL

Level 1.

NAME

menu_win: set_menu_win, menu_win, set_menu_sub, menu_sub, scale_menu – MENUS window and subwindow association routines

SYNOPSIS

```
#include <menu.h>

int set_menu_win(MENU *menu, WINDOW *win);
WINDOW *menu_win(MENU *menu);

int set_menu_sub(MENU *menu, WINDOW *sub);
WINDOW *menu_sub(MENU *menu);

int scale_window(MENU *menu, int *rows, int *cols);
```

DESCRIPTION

`set_menu_win()` sets the window of *menu* to *win*. `menu_win()` returns a pointer to the window of *menu*.

`set_menu_sub()` sets the subwindow of *menu* to *sub*. `menu_sub()` returns a pointer to the subwindow of *menu*.

`scale_window()` returns the minimum window size necessary for the subwindow of *menu*. *rows* and *cols* are pointers to the locations used to return the values.

RETURN VALUE

Routines that return pointers always return `NULL` on error. Routines that return an integer return one of the following:

`E_OK`	– The routine returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An incorrect argument was passed to the routine.
`E_POSTED`	– The menu is already posted.
`E_NOT_CONNECTED`	– No items are connected to the menu.

USAGE

Application Program.

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), MENUS(TI_ENV).

LEVEL

Level 1.

NAME

panel_above: panel_above, panel_below – PANELS deck traversal primitives

SYNOPSIS

```
#include <panel.h>

PANEL *panel_above(PANEL *panel);

PANEL *panel_below(PANEL *panel);
```

DESCRIPTION

`panel_above()` returns a pointer to the panel just above *panel*, or `NULL` if *panel* is the top panel. `panel_below()` returns a pointer to the panel just below *panel*, or `NULL` if *panel* is the bottom panel.

If `NULL` is passed for *panel*, `panel_above()` returns a pointer to the bottom panel in the deck, and `panel_below()` returns a pointer to the top panel in the deck.

RETURN VALUE

`NULL` is returned if an error occurs.

USAGE

Application Program.

These routines allow traversal of the deck of currently visible panels.

The header file `<panel.h>` automatically includes the header file `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), PANELS(TI_ENV).

LEVEL

Level 1.

NAME

panel_move: move_panel – move a PANELS window on the virtual screen

SYNOPSIS

```
#include <panel.h>

int move_panel(PANEL *panel, int starty, int startx);
```

DESCRIPTION

`move_panel()` moves the CURSES window associated with *panel* so that its upper left-hand corner is at *starty*, *startx*. See usage note, below.

RETURN VALUE

`OK` is returned if the routine completes successfully, otherwise `ERR` is returned.

USAGE

Application Program.

For PANELS windows, use `move_panel()` instead of the `mvwin()` CURSES routine. Otherwise, `update_panels()` will not properly update the virtual screen.

The header file `<panel.h>` automatically includes the header file `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), PANELS(TI_ENV), panel_update(TI_LIB).

LEVEL

Level 1.

NAME

panel_new: new_panel, del_panel – create and destroy PANELS

SYNOPSIS

```
#include <panel.h>

PANEL *new_panel(WINDOW *win);

int del_panel(PANEL *panel);
```

DESCRIPTION

`new_panel()` creates a new panel associated with *win* and returns the panel pointer. The new panel is placed on top of the panel deck.

`del_panel()` destroys *panel*, but not its associated window.

RETURN VALUE

`new_panel()` returns `NULL` if an error occurs.

`del_win()` returns `OK` if successful, `ERR` otherwise.

USAGE

Application Program.

The header file `<panel.h>` automatically includes the header file `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), PANELS(TI_ENV), panel_update(TI_LIB).

LEVEL

Level 1.

NAME

panel_show: show_panel, hide_panel, panel_hidden – PANELS deck manipulation routines

SYNOPSIS

```
#include <panel.h>

int show_panel(PANEL *panel);

int hide_panel(PANEL *panel);

int panel_hidden(PANEL *panel);
```

DESCRIPTION

`show_panel()` makes *panel*, previously hidden, visible and places it on top of the deck of panels.

`hide_panel()` removes *panel* from the panel deck and, thus, hides it from view. The internal data structure of the panel is retained.

`panel_hidden()` returns `TRUE` (`1`) or `FALSE` (`0`) indicating whether or not *panel* is in the deck of panels.

RETURN VALUE

`show_panel()` and `hide_panel()` return the integer `OK` upon successful completion or `ERR` upon error.

USAGE

Application Program.

The header file `<panel.h>` automatically includes the header file `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), PANELS(TI_ENV), panel_update(TI_LIB).

LEVEL

Level 1.

NAME

panel_top: top_panel, bottom_panel – PANELS deck manipulation routines

SYNOPSIS

```
#include &lt;panel.h&gt;

int top_panel(PANEL *panel);

int bottom_panel(PANEL *panel);
```

DESCRIPTION

`top_panel()` pulls *panel* to the top of the desk of panels. It leaves the size, location, and contents of its associated window unchanged.

`bottom_panel()` puts *panel* at the bottom of the deck of panels. It leaves the size, location, and contents of its associated window unchanged.

RETURN VALUE

All of these routines return the integer `OK` upon successful completion or `ERR` upon error.

USAGE

Application Program.

The header file `<panel.h>` automatically includes the header file `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), PANELS(TI_ENV), panel_update(TI_LIB).

LEVEL

Level 1.

NAME

panel_update: update_panels – PANELS virtual screen refresh routine

SYNOPSIS

```
#include <panel.h>

void update_panels(void);
```

DESCRIPTION

`update_panels()` refreshes the virtual screen to reflect the depth relationships between the panels in the deck. The user must use the curses library call `doupdate()` [see curs_refresh(TI_LIB)] to refresh the physical screen.

USAGE

Application Program.

The header file `<panel.h>` automatically includes the header file `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), PANELS(TI_ENV), curs_refresh(TI_LIB).

LEVEL

Level 1.

NAME

panel_userptr: set_panel_userptr, panel_userptr – associate application data with a PANELS panel

SYNOPSIS

```
#include <panel.h>

int set_panel_userptr(PANEL *panel, char *ptr);

char * panel_userptr(PANEL *panel);
```

DESCRIPTION

Each panel has a user pointer available for maintaining relevant information.

`set_panel_userptr()` sets the user pointer of *panel* to *ptr*.

`panel_userptr()` returns the user pointer of *panel*.

RETURN VALUE

`set_panel_userptr()` returns `OK` if successful, `ERR` otherwise.

`panel_userptr()` returns `NULL` if there is no user pointer assigned to *panel*.

USAGE

Application Program.

The header file `<panel.h>` automatically includes the header file `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), PANELS(TI_ENV).

LEVEL

Level 1.

NAME

panel_window: panel_window, replace_panel – get or set the current window of a PANELS panel

SYNOPSIS

```
#include <panel.h>

WINDOW *panel_window(PANEL *panel);

int replace_panel(PANEL *panel, WINDOW *win);
```

DESCRIPTION

`panel_window()` returns a pointer to the window of *panel*.

`replace_panel()` replaces the current window of *panel* with *win*.

RETURN VALUE

`panel_window()` returns `NULL` on failure.

`replace_panel()` returns `OK` on successful completion, `ERR` otherwise.

USAGE

Application Program.

The header file `<panel.h>` automatically includes the header file `<curses.h>`.

SEE ALSO

CURSES(TI_ENV), PANELS(TI_ENV).

LEVEL

Level 1.

Terminal Interface Commands And Utilities

The following section contains the manual pages for the TI_CMD routines.

NAME

captoinfo — convert a *termcap* description into a *terminfo* description

SYNOPSIS

`captoinfo [-v ...] [-V] [-1] [-w` *width*`]` *file ...*

DESCRIPTION

`captoinfo` looks in *file* for `termcap` descriptions. For each one found, an equivalent `terminfo` description is written to standard output, along with any comments found. A description which is expressed as relative to another description (as specified in the `termcap tc = field`) will be reduced to the minimum superset before being output.

If no *file* is given, then the environment variable `TERMCAP` is used for the filename or entry. If `TERMCAP` is a full pathname to a file, only the terminal whose name is specified in the environment variable `TERM` is extracted from that file. If the environment variable `TERMCAP` is not set, then the file `/usr/share/lib/termcap` is read.

`-v` print out tracing information on standard error as the program runs. Specifying additional `-v` options will cause more detailed information to be printed.

`-V` print out the version of the program in use on standard error and exit.

`-1` cause the fields to print out one to a line. Otherwise, the fields will be printed several to a line to a maximum width of 60 characters.

`-w` change the output to *width* characters.

FILES

`/usr/share/lib/terminfo/?/*` Compiled terminal description database.

USAGE

Administrator.

`captoinfo` should be used to convert `termcap` entries to `terminfo` entries because the `termcap` database (from earlier versions of UNIX System V) may not be supplied in future releases.

SEE ALSO

CURSES(TI_ENV), infocmp(TI_CMD), terminfo(TI_ENV).

LEVEL

Level 1.

NAME

clear – clear the terminal screen

SYNOPSIS

`clear`

DESCRIPTION

`clear` clears the terminal's screen if possible. It checks the environment for the terminal type and then searches the terminfo database for the correct codes for clearing the screen.

SEE ALSO

tput(TI_CMD).

LEVEL

Level 1.

NAME

infocmp – compare or print out *terminfo* descriptions

SYNOPSIS

infocmp [-d] [-c] [-n] [-I] [-L] [-C] [-r] [-u] [-s d|i|l|c] [-v] [-V]
[-1] [-w *width*] [-A *directory*] [-B *directory*] [*termname* ...]

DESCRIPTION

infocmp can be used to compare a binary terminfo entry with other terminfo entries, rewrite a terminfo description to take advantage of the use= terminfo field, or print out a terminfo description from the binary file (term) in a variety of formats. In all cases, the boolean fields will be printed first, followed by the numeric fields, followed by the string fields.

Default Options

If no options are specified and zero or one *termnames* are specified, the –I option will be assumed. If more than one *termname* is specified, the –d option will be assumed.

Comparison Options [–d] [–c] [–n]

infocmp compares the terminfo description of the first terminal *termname* with each of the descriptions given by the entries for the other terminal's *termnames*. If a capability is defined for only one of the terminals, the value returned will depend on the type of the capability: F for boolean variables, –1 for integer variables, and NULL for string variables.

–d produces a list of each capability that is different between two entries. This option is useful to show the difference between two entries, created by different people, for the same or similar terminals.

–c produces a list of each capability that is common between two entries. Capabilities that are not set are ignored. This option can be used as a quick check to see if the –u option is worth using.

–n produces a list of each capability that is in neither entry. If no *termnames* are given, the environment variable TERM will be used for both of the *termnames*. This can be used as a quick check to see if anything was left out of a description.

Source Listing Options [–I] [–L] [–C] [–r]

The –I, –L, and –C options will produce a source listing for each terminal named.

–I use the terminfo names

–L use the long C variable name listed in <term.h>

–C use the termcap names

–r when using –C, put out all capabilities in termcap form

If no *termnames* are given, the environment variable TERM will be used for the terminal name.

The source produced by the –C option may be used directly as a termcap entry, but not all of the parameterized strings may be changed to the termcap format. infocmp will attempt to convert most of the parameterized information, but anything not converted will be plainly marked in the output and commented out. These should be edited by hand.

All padding information for strings will be collected together and placed at the beginning of the string where `termcap` expects it. Mandatory padding (padding information with a trailing '/') will become optional.

All `termcap` variables no longer supported by `terminfo`, but which are derivable from other `terminfo` variables, will be output. Not all `terminfo` capabilities will be translated; only those variables which were part of `termcap` will normally be output. Specifying the `-r` option will take off this restriction, allowing all capabilities to be output in *termcap* form.

Note that because padding is collected to the beginning of the capability, not all capabilities are output. Mandatory padding is not supported. Because `termcap` strings are not as flexible, it is not always possible to convert a `terminfo` string capability into an equivalent `termcap` format. A subsequent conversion of the `termcap` file back into `terminfo` format will not necessarily reproduce the original `terminfo` source.

Some common `terminfo` parameter sequences, their `termcap` equivalents, and some terminal types which commonly have such sequences, are:

`terminfo`	`termcap`	Representative Terminals
`%p1%c`	`%.`	adm
`%p1%d`	`%d`	hp, ANSI standard, vt100
`%p1%'x'%+%c`	`%+x`	concept
`%i`	`%i`	ANSI standard, vt100
`%p1%?%'x'%>%t%p1%'y'%+%;`	`%>xy`	concept
`%p2` is printed before `%p1`	`%r`	hp

Use= Option [–u]

`-u` produces a `terminfo` source description of the first terminal *termname* which is relative to the sum of the descriptions given by the entries for the other terminals *termnames*. It does this by analyzing the differences between the first *termname* and the other *termnames* and producing a description with `use=` fields for the other terminals. In this manner, it is possible to retrofit generic terminfo entries into a terminal's description. Or, if two similar terminals exist, but were coded at different times or by different people so that each description is a full description, using `infocmp` will show what can be done to change one description to be relative to the other.

A capability will get printed with an at-sign (@) if it no longer exists in the first *termname*, but one of the other *termname* entries contains a value for it. A capability's value gets printed if the value in the first *termname* is not found in any of the other *termname* entries, or if the first of the other *termname* entries that has this capability gives a different value for the capability than that in the first *termname*.

The order of the other *termname* entries is significant. Since the terminfo compiler `tic` does a left-to-right scan of the capabilities, specifying two `use=` entries that contain differing entries for the same capabilities will produce different results depending on the order that the entries are given in. `infocmp` will flag any such inconsistencies between the other *termname* entries as they are found.

Alternatively, specifying a capability *after* a `use=` entry that contains that capability will cause the second specification to be ignored. Using `infocmp` to recreate a description can be a useful check to make sure that everything was specified correctly in the original source description.

Another error that does not cause incorrect compiled files, but will slow down the compilation time, is specifying extra `use=` fields that are superfluous. `infocmp` will flag any other *termname* `use=` fields that were not needed.

Other Options [–s d | i | l | c] [–v] [–V] [–1] [–w *width*]

`–s` sorts the fields within each type according to the argument below:

- `d` leave fields in the order that they are stored in the *terminfo* database.
- `i` sort by *terminfo* name.
- `l` sort by the long C variable name.
- `c` sort by the *termcap* name.

If the `–s` option is not given, the fields printed out will be sorted alphabetically by the `terminfo` name within each type, except in the case of the `–C` or the `–L` options, which cause the sorting to be done by the `termcap` name or the long C variable name, respectively.

`–v` prints out tracing information on standard error as the program runs.

`–V` prints out the version of the program in use on standard error and exit.

`–1` causes the fields to be printed out one to a line. Otherwise, the fields will be printed several to a line to a maximum width of 60 characters.

`–w` changes the output to *width* characters.

Changing Databases [–A *directory*] [–B *directory*]

The location of the compiled `terminfo` database is taken from the environment variable `TERMINFO` . If the variable is not defined, or the terminal is not found in that location, the system `terminfo` database, usually in `/usr/share/lib/terminfo`, will be used. The options `–A` and `–B` may be used to override this location. The `–A` option will set `TERMINFO` for the first *termname* and the `–B` option will set `TERMINFO` for the other *termnames*. With this, it is possible to compare descriptions for a terminal with the same name located in two different databases. This is useful for comparing descriptions for the same terminal created by different people.

FILES

`/usr/share/lib/terminfo/?/*` Compiled terminal description database.

SEE ALSO

CURSES(TI_LIB), captoinfo(TI_CMD), terminfo(TI_ENV), tic(TI_CMD).

LEVEL

Level 1.

NAME

tic – *terminfo* compiler

SYNOPSIS

`tic [-v[n]] [-c]` *file*

DESCRIPTION

The command `tic` translates a `terminfo` file from the source format into the compiled format. The results are placed in the directory `/usr/share/lib/terminfo`. The compiled format is necessary for use with the library routines in CURSES(TI_ENV).

`-v`*n* specifies that (verbose) output be written to standard error trace information showing `tic`'s progress. The optional integer *n* is a number from 1 to 10, inclusive, indicating the desired level of detail of information. If *n* is omitted, the default level is 1. If *n* is specified and greater than 1, the level of detail is increased.

`-c` specifies to check only *file* for errors. Errors in `use=` links are not detected.

file contains one or more `terminfo` terminal descriptions in source format [see terminfo(TI_ENV)]. Each description in the file describes the capabilities of a particular terminal. When a `use=`*entry-name* field is discovered in a terminal entry currently being compiled, `tic` reads in the binary from `/usr/share/lib/terminfo` to complete the entry. (Entries created from *file* will be used first. If the environment variable `TERMINFO` is set, that directory is searched instead of `/usr/share/lib/terminfo`.) `tic` duplicates the capabilities in *entry-name* for the current entry, with the exception of those capabilities that explicitly are defined in the current entry.

If the environment variable `TERMINFO` is set, the compiled results are placed there instead of `/usr/share/lib/terminfo`.

Total compiled entries cannot exceed 4096 bytes. The name field cannot exceed 128 bytes. Terminal names exceeding 14 characters will be truncated to 14 characters and a warning message will be printed.

FILES

`/usr/share/lib/terminfo/?/*` Compiled terminal description database.

USAGE

Administrator.

If the Enhanced Security Utilities are installed and running, privileged use of this command is restricted to maintenance mode. See the *System Administrator's Guide* for a description of maintenance mode.

When an entry, e.g., `entry_name_1`, contains a `use=`*entry_name_2* field, any canceled capabilities in *entry_name_2* must also appear in `entry_name_1` before `use=` for these capabilities to be canceled in `entry_name_1`.

SEE ALSO

CURSES(TI_ENV), captoinfo(TI_CMD), infocmp(TI_CMD), terminfo(TI_ENV).

LEVEL

Level 1.

NAME

tput – initialize a terminal or query the *terminfo* database

SYNOPSIS

`tput` [-T*type*] *capname* [*parms* ...]

tput [-T*type*] `init`

tput [-T*type*] `longname`

tput [-T*type*] `reset`

tput -S

DESCRIPTION

The command `tput` uses the `terminfo` database to make the values of terminal-dependent capabilities and information available to the shell [see sh(BU_CMD)], to initialize or reset the terminal, or return the long name of the requested terminal type. The command `tput` outputs a string if the attribute is of type string, or an integer if the attribute is of type integer. If the attribute is of type boolean, `tput` simply sets the exit code (`0` for true if the terminal has the capability, `1` for false if it does not), and produces no output.

`–T`*type*
: indicates the type of terminal. Normally this option is unnecessary, as the default is taken from the environment variable `TERM`. If `–T` is specified, then the shell variables `LINES` and `COLUMNS` and the layer size will not be referenced.

capname
: indicates the attribute from the `terminfo` database [see terminfo(TI_ENV)].

parms
: If the attribute is a string that takes parameters, the arguments *parms* will be instantiated into the string. An all numeric argument will be passed to the attribute as a number.

`init`
: If the `terminfo` database is present and an entry for the user's terminal exists, then the following will occur: (1) if present, the terminal's initialization strings will be output (`is1, is2, is3, if, iprog`) (2) any delays (*e.g.*, newline) specified in the entry will be set in the tty driver (3) tabs expansion will be turned on or off according to the specification in the entry, and (4) if tabs are not expanded, standard tabs will be set (every 8 spaces). If an entry does not contain the information needed for any of the four above activities, that activity will silently be skipped.

`longname`
: If the `terminfo` database is present and an entry for the user's terminal exists, then the long name of the terminal will be output. The long name is the last name in the first line of the terminal's description in the `terminfo` database.

`reset`
: `reset` behaves identically to `init` with the following exception. Instead of outputting initialization strings, the terminal's reset strings will be output if present (`rs1, rs2, rs3, rf`). If the reset strings are not present, but initialization strings are, the initialization strings will be output.

`–S` allows more than one capability per invocation of `tput`. The capabilities must be passed to `tput` from the standard input instead of from the command line (see Example). Only one *capname* is allowed per line. The `–S` option changes the meaning of the `0` and `1` boolean and string exit codes (see Return Value).

RETURN VALUE

Before using a value returned on standard output, the user should test the exit code [`$?`, see sh()] to be sure it is `0`.

If *capname* is of type boolean, an exit code of `0` is returned for true and `1` for false unless the `–S` option is used.

If *capname* is of type string, an exit code of `0` is returned if the *capname* is defined for this terminal *type* (the value of *capname* is returned on standard output); an exit code of `1` is returned if *capname* is not defined for this terminal *type* (a null value is returned on standard output).

If *capname* is of type boolean or type string and the `–S` option is used, an exit code of `0` is returned to indicate that all lines were successful. No indication of which line failed can be given, therefore, exit code `1` will never appear. Exit codes `2`, `3`, and `4` retain their usual interpretation.

If *capname* is of type integer, exit code of `0` is always set, whether or not *capname* is defined for this terminal *type*. To determine if *capname* is defined for this terminal *type*, the user must test the value of standard output. A value of `–1` means that *capname* is not defined for this terminal.

Any other exit code indicates an error.

`tput` prints the following messages corresponding to the exit codes.

exit code	*message*
0	`-1` (*capname* is a numeric variable that is not specified in the `terminfo` database for this terminal type, e.g. `tput –T450 lines` and `tput –T2621 xmc`)
1	no error message is printed, see above.
2	`usage error`
3	`unknown terminal` *type* or `no terminfo database`
4	`unknown terminfo capability` *capname*

FILES

`/usr/lib/terminfo/?/*` Compiled terminal description database.

USAGE

Application Program.

`tput init` or `tput reset` may clear the user's screen.

EXAMPLES

`tput init`

Initialize the terminal according to the type of terminal in the environmental variable `TERM`. This command should be included in `.profile` after the environmental variable `TERM` has been exported.

`tput –T5620 reset`
Reset an AT&T 5620 terminal, overriding the type of terminal in the environmental variable `TERM`.

`tput clear`
Echo clear-screen sequence for the current terminal.

`tput cols`
Print the number of columns for the current terminal.

`tput –T450 cols`
Print the number of columns for the 450 terminal.

```
bold=`tput smso`
offbold=`tput rmso`
```
Set the shell variables `bold` and `offbold` to begin standout mode sequence and to end standout mode sequence for the current terminal respectively. This may be followed by a prompt, e.g.:

```
echo "${bold}Name: ${offbold}\c"
```

`tput hc`
Set exit code to indicate if the current terminal is a hardcopy terminal.

`tput cup 23 4`
Print the sequence to move the cursor to row 23, column 4.

`tput longname`
Print the long name from the `terminfo` database for the type of terminal specified in the environmental variable `TERM`.

`tput cup 0 0`
Send the sequence to move the cursor to row `0`, column `0` (the upper left corner of the screen, usually known as the "home" cursor position).

```
tput –S <<!
> clear
> cup 10 10
> bold
> !
```
This example shows tput processing several capabilities in one invocation. This example clears the screen, moves the cursor to position 10, 10 and turns on bold (extra bright) mode. The list is terminated by an exclamation mark (`!`) on a line by itself.

SEE ALSO

sh(BU_CMD), stty(BU_CMD), terminfo(TI_ENV).

LEVEL

Level 1.

Window System Introduction

Overview Of The Window System Extension

The Window System Extension supports the creation of application programs that communicate with the user through a windowed user interface. This extension defines low-level libraries and communication protocol. It does not define higher-level graphical toolkits or a specific development environment. In a future issue of the SVID, toolkit interfaces will be added to this extension.

The following are prerequisites for support of the Window System Extension:

- Base System
- Kernel Extension
- Basic Utilities Extension
- Advanced Utilities Extension
- Software Development Extension
- Terminal Interface Extension

OVERVIEW OF THE WINDOW SYSTEM ENVIRONMENT

The System V Window System Extension supports the X11 Window System. The X11 Window System interface is required.

The X11 window system follows the client/server model; a collection of client programs communicate with a window system server that drives high-resolution bit-mapped display devices. Client applications communicate with their server using the appropriate (X11) protocol. Multiple applications can run on the same display screen simultaneously. A server process arbitrates a shared display, a keyboard, and a pointing device, and it performs I/O on behalf of the client applications. Client applications can execute on the local processor, or they can run on a remote processor and communicate with the server through a network connection.

The Window System Extension is network-transparent. A client program can run on any machine in a network, and the client and server programs need not execute on machines that share a common architecture. When the client and server reside on different machines, the window system uses the Transport Interface (TI) library, libnsl, to access the services of a transport provider on a remote machine. When the client and server reside on the same machine, messages are transmitted using a local interprocess communication mechanism.

REQUIRED COMPONENTS OF THE WINDOW SYSTEM EXTENSION

The window system server is not a required component of the Window System Extension. For example, a host might omit the server if its terminals do not have graphics capabilities. In this case, other hosts on the network may have graphics capabilities and may provide servers. Since the Window System Extension is network-transparent, clients built on a host that does not provide a server may connect to a server elsewhere on the network to display their data.

If a host offers a window system server, the server can be an X11 server.

To communicate with the server, X clients use the library Xlib. The components of Xlib are described in the *X11 Library Routines* chapter. Typically, Xlib is installed as libX11.a or libX11.so.

Several subsets of the Window System Extension are acceptable for System V compliance. These options are summarized below.

If graphics hardware is present, the following two options are acceptable:

1. Xlib and its header files, and the X11 server
2.
3. Xlib and its header files

If no graphics hardware is present, option 2) is acceptable.

ORGANIZATION OF TECHNICAL INFORMATION

The information is this extension is now a points to the "X Window System Protocol, Version 11 Specification" (Massachusetts Institute of Technology, 1987, 1988) and *Xlib - C Language Interface, X Window System, X Version 11, Release 5,* (Massachusetts Institute of Technology, 1991). This information will no longer be duplicated in the SVID. The SVID will track upward-compatible future releases of the X library.

X11 WINDOW SYSTEM COMPONENTS

A client application communicates with the X capabilities of the System V window system server using the X11 protocol. The X11 protocol specifies exchange format, rules for data exchange, X11 protocol and message semantics, but is policy-free and does not impose any specific appearance on the interface. The look and feel of a particular interface is defined by the window manager and different toolkits that define a higher-level program interface to the X capabilities.

The X Version 11 protocol defines the format, syntax, common types, errors codes, keyboard keycodes, pointers, predefined atoms, connection setup, requests, connection close, and events. A detailed description of these can be found in the *X Window System Protocol, Version 11 Specification* (Massachusetts Institute of Technology, 1987, 1988).

The X library, **libX**, generates the X11 protocol and buffers traffic between each client application and the server. A full specification of the libX library and its contents can be found in *Xlib - C Language Interface, X Window System, X Version 11, Release 5,* (Massachusetts Institute of Technology, 1991).

The X Toolkit Intrinsics library **libXt** provides a framework for building X-based toolkits. A full specification of the X Toolkit Intrinsics can be found in *X Toolkit Intrinsics - C Language X Interface, X Window System, X Version 11, Release 5,* (Massachusetts Institute of Technology, 1991).

FUTURE DIRECTIONS

The Motif™ Graphical User Interface Release 1.2 will be supported in a future edition of the SVID. Motif is a trademark of the Open Software Foundation Inc. The header files required to use Xlib are listed below. These header files are required for SVID compliance.

Remote Administration Introduction

Remote Administration Overview

The Remote Administration Extension contains additional system management services that provide support in a networked environment. These services are the Remote Operations Interface (ROI) and the Software Distribution Service (SDS).

All of the interfaces and commands in this extension have been moved to Level 2 in the SVID, Fourth Edition. In the future, they are to be phased out in favor of the Distributed Management Functionality. In this introduction, all the commands and utilities are marked with an asterisk (*) to indicate that they have been moved to Level 2.

The following are prerequisite for support of the REMOTE ADMINISTRATION EXTENSION:

- Base System
- Basic Utilities Extension
- Advanced Utilities Extension
- Administered Systems Extension

REMOTE OPERATIONS INTERFACE

The Remote Operations Interface offers uniform networking access for applications that desire network service independence. This allows a service to invoke remote operations following a client/server scenario without regard to which network services will be used. The application controls whether synchronous or queued services are called. The ROI has three components:

- library subroutines to initiate remote operations;
- library subroutines to build additional ROI connections to network services outside of the delivered set;
- commands to administer ROI parameters, and to monitor and cancel ROI jobs.

SOFTWARE DISTRIBUTION SERVICE

The Software Distribution Service (SDS) is a facility that enables computers defined as "servers" to distribute software packages across a network to computers defined as either "clients" or "target servers" (other servers that receive packages and then distribute them to their own clients.)

Features of SDS include:

- support for a hierarchical configuration of authorized clients and servers
- administrative commands for maintaining databases of information needed to run an SDS network
- catalogs (lists of software packages) from which clients can request packages
- three methods of package transfer:
  - broadcast- a server sends a package to a client on its own initiative (that is, not in response to any request by a client)
  - request- a client asks a server to send a specified package, selected from a catalog of available packages
  - subscription- a client sets up a mechanism that automatically requests (from a server) a new version of a specified package as soon as it becomes available
- a mechanism for tracking package distribution

SUMMARY OF LIBRARY ROUTINES

The following routines are supported by the Remote Administration Extension.

mgroup*	**roistat***	**roitosval***	**roijobids***
roigetuser*	**roitosparse***	**remop***	

SUMMARY OF COMMANDS AND UTILITIES

The following commands and utilities are supported by the Remote Administration Extension. All of the commands and utilities in this section have been internationalized and may reference environment variables for localization information. [See envvar(BA_ENV)].

catreq*	**catsend***	**distauth***	**remtab***
distrpt*	**pkgcat***	**pkgdel***	**distconf***
pkgreq*	**pkgsend***	**pkgtrk***	**pkgput***
remalias*	**remclean***	**remkill***	**remadmin***
remop*	**remstat***		

ORGANIZATION OF TECHNICAL INFORMATION

The ''Remote Administration Library Routines'' chapter provides manual page descriptions of library routines supported by this extension.

The ''Remote Administration Commands and Utilities'' chapter provides manual page descriptions of commands and utilities supported by this extension.

Remote Administration Library Routines

The following section contains the manual pages for the RA_LIB routines.

FINAL COPY
June 15, 1995
File: ra_lib.cov
svid

Page: 621

NAME

mgroup – expand aliases to machine names

SYNOPSIS

#include <remop.h>
int mgroup(const char ****aliases*, struct remop *****remopp***);**

DESCRIPTION

mgroup() is used to map machine aliases into machine names required by the remote operation services. Aliases are configured by the administrator via the **remalias** command.

mgroup() takes a list of machines, machine aliases, or both and creates a list of machines as a list of **remop** structures. The *aliases* argument points to a string containing a list of machines, machine aliases, or both separated by spaces or commas. Since applications will normally use this information as input to **remop**(), **mgroup**() packages each remote machine name within a **remop** structure. The *remopp* argument is initialized by **mgroup**() to point to a NULL-terminated linked list of **remop** structures. **mgroup**() also initializes the following fields: the **machine** fields of the **remop** structure are initialized to point to the name of the remote; the **exit_status** field is set to -1; and the **sid**, **adminid**, and **primid** fields are set to 0. The **next** field will point to the next **remop** structure in the list.

RETURN VALUE

Upon failure, the value of -1 is returned; otherwise, it returns the number of machines in the **remopp** list.

EXAMPLE

The following example expands the supplied alias and machine names and creates a NULL-terminated link list of **remop** structures.

```
#include <remop.h>

struct     remop  *out;
int   ret;

ret = mgroup("aliasa,aliasb,aliasc,macha", &out);
```

SEE ALSO

remop(RA_LIB), remalias(RA_CMD), remop(RA_CMD)

FUTURE DIRECTIONS

Certain of the Remote Operations Interface library and commands have been moved to Level 2 for the following reasons. On going standards and industry direction has converged around an object-oriented approach to distributed, as opposed to remote, system administration. The ROI interfaces will become obsolete as standards and consensus mature within the workings of the Object Management Group, X/Open Systems Management, and IEEE P1003.7. It is expected that the ROI interfaces will be replaced by standard API's and command-line interfaces as part of the ongoing efforts in the area of Distributed Systems and Network Management.

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

remop – initiate a remote operation

SYNOPSIS

```
#include <remop.h>
int remop(const char *type, const char *primitive,
          const char *operand, struct remop *remreq,
          const char *svc, const char *notify);
```

DESCRIPTION

remop() provides interactive and queued remote operation capabilities on one or more machines by initiating a primitive remote operation on each system specified in the **machine** argument of the **remop** structure. Mapping from machine aliases to machine names (supplied in the **remreq** list) will need to be done by the calling the **mgroup**() function prior to calling **remop**().

The *remreq* argument points to a structure of type **remop** which contains the following elements:

> **char** *machine;
>
> **int** *sid*;
>
> **int** *adminid*;
>
> **int** *primid*;
>
> **int** *depend_flg*;
>
> **int** *dependid*;
>
> **int** *exit_status*;
>
> **int** *filler*;
>
> **struct remop** **next*;

The *remreq* argument points to a NULL-terminated, linked list of **remop** structures. It is a good practice to call the **mgroup**() function before calling **remop**() because **mgroup**() will provide a **remreq** list. The **remreq** list returned by **mgroup**() sets the following fields and values for the **remop** structure: **machine**, which points to the name of the remote system; **exit_status**, which is set to -1; and **sid**, **adminid** and **primid**, which are set to 0.

type, *primitive* and *operand* are NULL-terminated strings; their meaning and use are described below.

The *type* argument specifies whether the remote operation will execute in synchronous or batch fashion. The letter **s** is used for synchronous operations and the application will receive the exit status interactively. Only one remote system may be specified with **s** (that is, the **remreq** list contains one member). The letter **q** is used for queued operations and the application will receive an indication of whether the primitive was successfully queued.

Job identifier assignments are performed on three levels: service jobs, administrative jobs, and primitive jobs. All job identifiers assigned by **remop**() are unique per user and assigned from the same allocator, in ascending order from **1** to **INT_MAX** (defined in **<limits.h>**). When the maximum number is reached, the sequence wraps around.

The **remop** structure will assign new service job identifiers and administrative job identifiers if their values are zero upon the call to **remop**() and the specified job type is **q**. For example, if the values for the **sid** and **adminid** fields of the **remop** structure are zero, each field is assigned job identifiers from **1** to **INT_MAX**. If the values are positive integers, they become the actual job identifier values. If the values are negative, an error message is generated. The job identifier for the **primid** field of the **remop** structure is assigned by the system from the counter using **1** to **INT_MAX**.

The **depend_flg** and **dependid** fields provide applications with the means to specify job dependencies. Possible values for **depend_flg** are defined in **remop.h**. They are **DEP_NONE**, **DEP_START**, **DEP_MID** and **DEP_END**.

An application may indicate that there are no job dependencies by initializing the **depend_flg** and **dependid** fields to **DEP_NONE**. When set to **DEP_START**, **depend_flg** marks the first job in a dependency list; when set to **DEP_MID**, it marks any job in the middle; when set to **DEP_END**, it marks the last job to execute. Applications must use the **dependid** field to tell **remop**() the primitive job identifier on which the current field depends. Dependency lists cannot be used with linked **remop** structure lists; that is, the **next** field of the **remop** structure must be NULL.

The *primitive* argument specifies the remote operation to be invoked. Three *primitive* strings are supported: **ft** will invoke a file transfer, **re** will invoke a remote command execution, and **dt** will invoke a directory transfer.

The *operand* argument provides data to direct the remote operation. Its meaning is dependent on the value of the *primitive* argument. Where *primitive* is **ft**, *operand* contains a pathname specification to transfer a local file to the remote system. *operand* is of the form **path1** or **path1=path2**, where **path1** is the local pathname and **path2**, if supplied, is the full remote pathname. If **path2** is not supplied, the Remote Operation Interface uses its standard naming convention of **/var/spool/roi/users/***logname***/receive/***svc***/***mach*. (See FILES section.)

Where *primitive* is **re**, *operand* contains a command to be remotely executed. The **operand** string is interpreted with **sh**(BU_CMD) rules on the remote system.

Where *primitive* is **dt**, *operand* contains a pathname specification to copy a local directory tree structure to a remote system. The pathname conventions described for the **ft** *primitive* apply; the character string can be of the form **path1** or **path1=path2**. The **dt** *primitive* transfers the complete directory structure to each remote system. In the case where **path1** is a file rather than a directory, the **dt** *primitive* is equivalent to an **ft**.

The *svc* argument specifies an identifier used to indicate membership in an administrative service. The *svc* argument is used in naming the destination directory, **/var/spool/roi/users/***logname***/receive/***svc***/***mach* on the remote system if **path2** was not specified in the *operand*. It also can be used as a status report selection option on the local system (see **remstat**(RA_CMD)).

The last argument, *notify*, can be specified if the application has selected the queued type of operation. *notify* is the full pathname of an executable on the local system. One or more space characters must separate the name of the executable from additional arguments or shell command separators. When the operation reaches the final state, *notify* can be used to report the success or failure of a remote job to each destination machine. If *notify* is not specified, no corresponding executable will be invoked. Certain environment variables are available to the *notify* executable as shown by the table below.

Variable	Description
REMEXIT	Exit value of a remote primitive
REMSTAT	Status of the job
REMMACH	The destination machine name
REMSVC	The service job ID
REMADM	The administrative job ID
REMPRIM	The primitive job ID
STDOUT	Full pathname to **stdout** file
STDERR	Full pathname to **stderr** file

Possible values for **REMSTAT** are defined in **remop.h**. They are **ROI_FAILED, ROI_REJECT**, **ROI_SUCC**, **ROI_TMOUT**, **ROI_INPROG**, **ROI_QUEUED**, **ROI_CANCEL**, and **ROI_NOSTAT**. **STDOUT** and **STDERR** are set to the full pathnames of local files containing, respectively, the standard output and standard error output of the remotely executed operation.

The order for trying network services can be determined by the environment variables **REMOPS** and **REMOPQ**. **REMOPS** specifies the order for trying synchronous network services; **REMOPQ** specifies the order for trying queued network services. The value of each environment variable is a colon-separated list of network services. If these variables are not set, the default order of the network services is the order set by the **remtab** administrative command.

When invoked, **remop**() will interrogate the **_ROIDEBUG** environment variable, which assists users in debugging ROI. **_ROIDEBUG** provides output to standard output so users can follow the execution of primitive operations. To enable this tracing, **_ROIDEBUG** must be set to **yes** and exported.

RETURN VALUE

Upon successful completion of a synchronous operation, the value of 0 is returned. Otherwise, a value of -1 is returned, and the **exit_status** field of the **remop** structure is set to indicate the exit status. When all operations are successfully queued, a value of 0 is returned and the fields **sid**, **adminid**, and **primid** of the **remop** structure are set to the assigned job IDs. Otherwise, a value of -1 is returned and the **exit_status** field of the **remop** structure is set to indicate the exit status.

EXAMPLE

Below are examples for using the **remop**() function for synchronous operation, queued operations, and queued operations with dependencies. All examples use the following declaration statements:

```
#include <remop.h>

static const struct remop empty={0};
struct remop remreq = empty;
int   ret;
char  *svc = "dist";
char  *notify = "echo job completed | /usr/bin/mail user1";
remreq.machine = "intl";
remreq.exit_status = -1;
```

EXAMPLE 1 — Synchronous Operation

The following function call will copy the file **/fs1/user1/foo1.c** to the remote system **intl** over a synchronous network service. The returned value indicates the completion status of the operation.

```
ret = remop("s", "ft", "/fs1/user1/foo1.c", &remreq, svc, (char
*)0);
```

EXAMPLE 2 — Queued Operation

The following function call will copy the file **/fs1/user1/foo1.c** to the remote system **intl** over a queued network service. The user **user1** will be notified via **mail**(BU_CMD) on the completion of the operation.

```
ret = remop("q", "ft", "/fs1/user1/foo1.c", &remreq, svc, notify);
```

EXAMPLE 3 — Queued Operations With Dependencies

The following function call will copy the files **/fs1/user1/foo1.c** and **/fs1/user2/foo2.c** into the **/tmp** directory on the remote system **intl**. Finally, a compilation will be initiated on the system **intl** and the file **foo** will be created in the **/tmp** directory on **intl**.

```
remreq.depend_flg = DEP_START;
remreq.dependid = DEP_NONE;
ret = remop("q", "ft",
     "/fs1/user1/foo1.c=/tmp/foo1.c",
     &remreq, svc, notify);
     .
     .
remreq.depend_flg = DEP_MID;     /* depends on foo1.c transfer */
remreq.dependid = remreq.primid;
ret = remop("q", "ft",
     "/fs1/user1/foo2.c=/tmp/foo2.c",
     &remreq, svc, notify);
     .
     .
remreq.depend_flg = DEP_END;     /* depends on foo2.c transfer */
remreq.dependid = remreq.primid;
ret = remop("q", "re",
   "cd /tmp; /usr/bin/cc -o /tmp/foo /tmp/foo1.c /tmp/foo2.c",
     &remreq, svc, notify);
```

SEE ALSO

mgroup(RA_LIB), remop(RA_CMD), remtab(RA_CMD), remclean(RA_CMD)

FUTURE DIRECTIONS

Certain of the Remote Operations Interface library and commands have been moved to Level 2 for the following reasons. On going standards and industry direction has converged around an object-oriented approach to distributed, as opposed to remote, system administration. The ROI interfaces will become obsolete as standards and consensus mature within the workings of the Object Management Group, X/Open Systems Management, and IEEE P1003.7. It is expected that the

ROI interfaces will be replaced by standard API's and command-line interfaces as part of the ongoing efforts in the area of Distributed Systems and Network Management.

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

roigetuser – get login name of the user

SYNOPSIS

#include <remop.h>
char ∗roigetuser (char ***logname)**

DESCRIPTION

roigetuser() returns the login name from the password file that matches the effective ID of the current process. The argument *logname* must point to a pre-allocated character array. As a result, **roigetuser**() returns a pointer to the login name and puts the login name in *logname*.

RETURN VALUE

If the login name cannot be found in the **/etc/passwd** file, **roigetuser** returns a NULL pointer; otherwise, **roigetuser**() copies the login name to the character array pointed to by *logname* and returns a pointer to the argument *logname*.

EXAMPLE

This example gets a user's login name and loads it into a character array.

```
#include <stdio.h>
#include <remop.h>

char  logname[20];

if (roigetuser(logname) != (char *)NULL)
```

SEE ALSO

remop(RA_CMD), roijobids(RA_CMD), getpwuid(BA_LIB)

FUTURE DIRECTIONS

Certain of the Remote Operations Interface library and commands have been moved to Level 2 for the following reasons. On going standards and industry direction has converged around an object-oriented approach to distributed, as opposed to remote, system administration. The ROI interfaces will become obsolete as standards and consensus mature within the workings of the Object Management Group, X/Open Systems Management, and IEEE P1003.7. It is expected that the ROI interfaces will be replaced by standard API's and command-line interfaces as part of the ongoing efforts in the area of Distributed Systems and Network Management.

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

roijobids – get unique remote job identifiers

SYNOPSIS

```
#include <remop.h>
int roijobids(int count, int *ids);
```

DESCRIPTION

roijobids is used by remote administrative programs to pre-assign job IDs before calling the **remop** routine. While **remop**() includes this capability, some applications may have a need for pre-assignment. **roijobids** returns unique job identifiers and initializes the array *ids* with their values. It will set as many unique remote job identifiers as the value of the argument *count*.

The remote job identifiers assigned by **roijobids** are unique per user.

RETURN VALUE

Upon failure, the value -1 is returned. The value -1 is also returned if the supplied value of *count* is less than or equal to 0. Otherwise, 0 is returned.

EXAMPLE

This example gets a remote job identifier and initializes the array *ids*.

```
#include <remop.h>

int   ids[2];
int   ret;

ret = roijobids(2, ids);
```

SEE ALSO

remop(RA_CMD)

FUTURE DIRECTIONS

Certain of the Remote Operations Interface library and commands have been moved to Level 2 for the following reasons. On going standards and industry direction has converged around an object-oriented approach to distributed, as opposed to remote, system administration. The ROI interfaces will become obsolete as standards and consensus mature within the workings of the Object Management Group, X/Open Systems Management, and IEEE P1003.7. It is expected that the ROI interfaces will be replaced by standard API's and command-line interfaces as part of the ongoing efforts in the area of Distributed Systems and Network Management.

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

roistat – update job status record

SYNOPSIS

```
#include <remop.h>

int roistat(int optype, const char *logname,
     struct job_record *jobinfop);
```

DESCRIPTION

roistat updates or reads the remote operations job status file associated with the user specified by *logname*. This function is used by network service interface applications to update job status records for a remote job.

The *jobinfop* argument points to a structure of type **job_record** which contains the following elements:

long *rtime*;

int *sid*;

int *adminid*;

int *primid*;

int *stat*;

char *dst[DST_LEN]*;

char *svc[SVC_LEN]*;

char *prim[PRIM_LEN]*;

char *ns[NS_LEN]*;

The operations prescribed by the values of *optype* are as follows:

APPEND Append the job status record referenced by *jobinfop* to the end of the job status file.

UPDATE Change the state of the job in the job status file. The **primid** field of the **roistat** structure indicates the job entry in the file, and the **stat** field of the **roistat** structure indicates the new state. The states are: **ST_QUEUED**, **ST_INPROGRESS**, **ST_SUCCEEDED**, **ST_FAILED**, **ST_CANCELLED**, **ST_TIMEOUT**, and **ST_REJECTED**.

READ Read the job status record from the job status file. The record is copied into the structure pointed to by *jobinfop*. The **primid** field of the **roistat** structure indicates the job entry in the file.

RETURN VALUE

Upon successful completion, **roistat**, returns a value of 0; otherwise, it returns a value of -1.

EXAMPLE

The following example will append a job status record to the job status file for user **user1**.

```
#include <remop.h>
      .
      .
int   ret;
struct     job_record jobinfo;
      .
      .
/* Initialize elements of jobinfo structure */

time(&jobinfo.rtime);
jobinfo.sid = 10;
jobinfo.adminid = 11;
jobinfo.primid = 12;
jobinfo.stat = ST_SUCCEEDED;
(void)strcpy(jobinfo.dst, "intl");
(void)strcpy(jobinfo.svc, "bck");
(void)strcpy(jobinfo.prim, "ft");
(void)strcpy(jobinfo.ns, "rexec");
      .
      .
ret = roistat(APPEND, "user1", &jobinfo);
```

SEE ALSO

remop(RA_LIB), remstat(RA_CMD)

FUTURE DIRECTIONS

Certain of the Remote Operations Interface library and commands have been moved to Level 2 for the following reasons. On going standards and industry direction has converged around an object-oriented approach to distributed, as opposed to remote, system administration. The ROI interfaces will become obsolete as standards and consensus mature within the workings of the Object Management Group, X/Open Systems Management, and IEEE P1003.7. It is expected that the ROI interfaces will be replaced by standard API's and command-line interfaces as part of the ongoing efforts in the area of Distributed Systems and Network Management.

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

roitosparse – parse a Transaction Operation Script (TOS) file

SYNOPSIS

#include <remop.h>
**int roitosparse(const char ****tos_file*,
    struct stringll ***tos_list*);

DESCRIPTION

roitosparse is a Remote Operation Interface (ROI) function that parses a TOS file created by **remop**() and provides a NULL-terminated list of **stringll** structures.

A TOS file is created by **remop**() for communicating job information to all network service modules. Each network service primitive may also use the TOS file to store network service-specific information.

The *tos_file* argument points to the pathname of the TOS file. The TOS file format is a newline-separated list of strings of the form **name=value**. The file name is the same as the primitive job ID. *tos_list* is a NULL-terminated list of **stringll** structures. Note that **malloc**(BA_OS) is used to allocate space for elements in *tos_list*.

The argument *tos_list* points to a structure of type **stringll** which contains the following elements:

```
char    *name;
char    *value;
struct  stringll *next;
```

roitosparse initializes the field **name** to point to the name part of a string; the **value** field of the **roitosparse** structure points to the value part. The field **next** will point to the next **stringll** structure in the list.

Any line with a ‘‘**#**’’ character in column 1 is treated as comment and ignored by **roitosparse**. Blank lines may be inserted at any point.

RETURN VALUE

Upon successful completion, **roitosparse** returns 0. Otherwise, the value -1 is returned.

EXAMPLES

The following example shows how to use the **roitosparse** function.

```
#include <remop.h>
#include <stdio.h>
          .
          .
stringll_t *tos_list;
char *sjidp, *nextp;
const char tos_file[] = "/var/spool/roi/users/dist/tos/34";
          .
          .
if (roitosparse(tos_file, &tos_list) == -1) {
          .
          <failed to parse the tos file>
          .
```

```
}

/* Extract the values of SJID and NEXT from the TOS file */

sjidp = roitosval(tos_list, "SJID");
nextp  = roitosval(tos_list, "NEXT");

/* Check if the values are found in the TOS file */

if ((sjidp == (char *)NULL) || (nextp == (char *)NULL)) {
      .
      <failed to find the values>
      .
}
```

SEE ALSO

remop(RA_CMD), remop(RA_LIB), mgroup(RA_LIB), roitosval(RA_LIB), malloc(BA_OS)

FUTURE DIRECTIONS

Certain of the Remote Operations Interface library and commands have been moved to Level 2 for the following reasons. On going standards and industry direction has converged around an object-oriented approach to distributed, as opposed to remote, system administration. The ROI interfaces will become obsolete as standards and consensus mature within the workings of the Object Management Group, X/Open Systems Management, and IEEE P1003.7. It is expected that the ROI interfaces will be replaced by standard API's and command-line interfaces as part of the ongoing efforts in the area of Distributed Systems and Network Management.

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

roitosval – get a value for a variable name

SYNOPSIS

```
#include <remop.h>
char *roitosval(struct stringll *tos_list,
      const char *name);
```

DESCRIPTION

roitosval searches the *tos_list* created by **roitosparse**() for a string with a name equal to the argument *name*. It returns a pointer to the **value** field, if such a name is found. Otherwise, it returns a NULL pointer. When a name is defined in a Transaction Operation Script (TOS) file, **roitosval** returns the last instance.

The argument *tos_list* is of structure type **stringll** which contains the following elements:

```
char    *name;
char    *value;
struct  stringll *next;
```

EXAMPLES

The following example shows how to use the **roitosval** function.

```
#include <remop.h>
#include <stdio.h>
      .
      .
stringll_t *tos_list;
char *sjidp, *nextp;
const char tos_file[] = "/var/spool/roi/users/dist/tos/34";
      .
      .
if (roitosparse(tos_file, &tos_list) == -1) {
      .
      <failed to parse the tos file>
      .
}

/* Extract the values of SJID and NEXT names from the TOS file */

sjidp = roitosval(tos_list, "SJID");
nextp  = roitosval(tos_list, "NEXT");

/* Check if the values are found in the TOS file */

if ((sjidp == (char *)NULL) || (nextp == (char *)NULL)) {
      .
      <failed to find the values>
      .
}
```

SEE ALSO

remop(RA_LIB), **roitosparse**(RA_LIB), **remop**(RA_CMD)

FUTURE DIRECTIONS

Certain of the Remote Operations Interface library and commands have been moved to Level 2 for the following reasons. On going standards and industry direction has converged around an object-oriented approach to distributed, as opposed to remote, system administration. The ROI interfaces will become obsolete as standards and consensus mature within the workings of the Object Management Group, X/Open Systems Management, and IEEE P1003.7. It is expected that the ROI interfaces will be replaced by standard API's and command-line interfaces as part of the ongoing efforts in the area of Distributed Systems and Network Management.

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

Remote Administration Commands And Utilities

The following section contains the manual pages for the RA_CMD routines.

NAME

catreq – request a catalog of packages from a server

SYNOPSIS

catreq [**-qt**] *server* . . .

DESCRIPTION

catreq requests that one or more servers on the software distribution network send a catalog of currently available packages to the invoking system. When the catalog arrives on the invoking system, it is installed, and subscriptions previously set up via the **distauth** command are checked; **pkgreq** is invoked for any subscriptions found.

The requesting system must previously have configured all specified servers in its configuration database (via the **distconf** command). Servers may be invoked by server name, alias, or the token **all.**

The following options are available:

-q Queue the request via ROI, which returns a job ID. The default invocation (**catreq** without the **-q** option) processes the request in real time.

You can use the **remstat** command to check the status of the queued catalog request. See remstat(RA_CMD).

–t Test mode: cause **catsend** to display the catalog on **stdout** or, if the **-q** option is also included, to mail the catalog to the user(s) specified by the **NOTIFYUSER** parameter. (The **NOTIFYUSER** parameter is set using the **distconf** command.) The catalog is not installed and subscriptions are not enabled.

server Specify the name of the server(s) or server machine alias(es) from which the catalog should be requested. The token **all** may be used to specify that catalogs should be requested from all known server machines. Use a space-separated list to specify multiple arguments.

RETURN VALUES

Upon successful completion, **catreq** returns a value of **0**. Otherwise, it returns a non-zero value.

USAGE

This command is available on client and full system configurations.

Subscription may occur as a side effect of a successful **catreq** invocation (assuming that subscription lists have been set up with **distauth**). If many subscriptions are triggered, **catreq** execution might take a long time; you might therefore want to use the **–q** option.

If you use this command on a regular basis, you might want to arrange for **catreq** to be executed periodically by **cron**. See crontab(AU_CMD).

If the Enhanced Security Extension is implemented on your system, you cannot run this command unless you are logged in as **dist**.

EXAMPLES

Example 1:

This example requests a catalog:

```
 catreq snoopy
# Catalog requested from <snoopy>...
# Installing catalog from <snoopy>...
# Subscription triggered... ordering...
# Package <spell, 1.0> requested from <snoopy>...
# Package <spell> spooled in <dist_snoopy>
```

This **catreq** command requests a catalog from server **snoopy,** and installs it while you wait. When the catalog arrives, the authorization database is checked to determine whether there are any subscriptions for any of the packages in the new catalog. If subscriptions are found, requests are sent to the appropriate servers. Because this process may take a long time, you may want to queue the command (with the **-q** option), as shown in the next example.

Example 2:

This example shows a queued **catreq** command:

```
 catreq -q snoopy
# ROI job ID to machine <snoopy> is <a-22>
# Request for catalog from <snoopy> queued.
```

The appropriate user(s), as specified by the NOTIFYUSER parameter, will be notified when the catalog is installed. [See distconf(RA_CMD).] Use the **remstat** command to track the job's progress. [See remstat(RA_CMD).]

Example 3:

This example shows the test mode of **catreq**:

```
 catreq -t charlie
# Catalog requested from <charlie>...
Test catalog from <charlie> contains:
cds            C Development Set
               (i386) 5.0
lp             LP Print Service
               (i386) 4.1
terminf        Terminal Information Utilities
               (i386) 4.0 k18

Subscriptions for the following packages would have been triggered:
cds            C Development Set            client update
               (i386) 5.0
lp             LP Print Service             target server request
               (i386) 4.1
```

The catalog is requested, but not installed. Instead it is displayed to the user along with any subscriptions which would have been triggered.

SEE ALSO

catsend(RA_CMD), distauth(RA_CMD), distconf(RA_CMD), pkgreq(RA_CMD), remalias(RA_CMD).

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

catsend – send a catalog of packages to a client or target server

SYNOPSIS

catsend [**-q** | **-t**] *client* . . .

DESCRIPTION

catsend sends a catalog of currently available packages from a server to the specified clients and/or target servers. **catsend** may be invoked by server administrators or indirectly as a result of a remote **catreq** command issued on a client or a target server machine.

catsend inspects the catalog information on the server to create a personalized catalog of packages available to the *client* or *target server.* If invoked with the token **all**, **catsend** uses the list of known clients and target servers in the configuration database.

The following options are available:

-q Queue the delivery via ROI, which returns a job ID. The default invocation (**catsend** without the **-q** option) processes the delivery in real time.

You can use remstat(RA_CMD) to check the status of the queued catalog delivery.

-t Test mode: cause **catsend** to display the catalog on standard output. If more than one client or target server is specified on the command line, each section of the display is preceded by the machine name of the appropriate client or target server. When **-t** is not specified, ROI is invoked to send the catalog to the *client*.

client Specify the name(s) or alias(es) of the client and/or target server machine(s) to whom the catalog(s) will be sent. The token **all** may be specified, where **all** indicates all configured clients and target servers, as previously defined in the configuration database. [See distconf(RA_CMD).] Use a space-separated list to specify multiple arguments.

RETURN VALUES

Upon successful completion, **catsend** returns a value of **0**. Otherwise, it returns a non-zero value. If subscription occurs during a **catsend** invocation, you may receive **pkgreq** return values.

USAGE

This command is available on server and full system configurations.

Subscription on the client may occur as a side effect of a successful **catsend** invocation (assuming that subscription lists have been set up on the client with **distauth**). If many subscriptions are triggered, **catsend** execution might take a long time; you might therefore want to use the **–q** option.

If you use this command on a regular basis, you might want to arrange for **catsend** to be executed periodically by **cron**. See crontab(AU_CMD).

If the Enhanced Security Extension is implemented on your system, you cannot run this command unless you are logged in as **dist**.

EXAMPLES

Example 1:

This example sends a catalog:

```
 catsend lucy
# Transferring catalog to client <lucy>...
# Installing catalog... checking for subscriptions...
# Catalog installed.
```

The **catsend** command sends a catalog to client **lucy**, and installs it there while you wait. When the catalog arrives, the client authorization database is checked to determine whether there are any subscriptions for any of the packages in the new catalog. If subscriptions are found, requests are sent to the appropriate servers. Because this process may take a long time, you may want to queue the command (with the **-q** option), as shown in the next example.

Example 2:

This example shows a queued **catsend** command:

```
 catsend -q lucy
# ROI job ID to machine <lucy> is <a-22>.
# Distribution of catalog to <lucy> queued.
```

Example 3:

This example shows the test mode of **catsend**:

```
 catsend -t all
<linus>:
cds             C Development Set
                (i386) 5.0
usrenv          User Environment Utilities
                (i386) 4.0
<lucy>:
cds             C Development Set
                (i386) 5.0
sysadm          System Administration Utilities
                (i386) 4.0 k18
terminf         Terminal Information Utilities
                (i386) 4.0 k18
```

The catalogs for all configured clients and target servers are created, formatted, and displayed, but are not sent.

SEE ALSO

catreq(RA_CMD), distconf(RA_CMD), pkgput(RA_CMD).

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

distauth – authorize subscription and broadcast of packages

SYNOPSIS

distauth [**-S**]
distauth [**-i** [**-f** *admin*]] [**-a** *arch*] *pkg server*
distauth -s [**-U**] [**-i** [**-f** *admin*] [**-r** *resp*]] [**-a** *arch*] *pkg server*
distauth -S [**-s**] [**-i** [**-f** *catadmin*]] [**-a** *arch*] *pkg server*
distauth -d [**-Ssn**] *pkg server*

DESCRIPTION

distauth is used to administer the authorization database, which stores authorizations for acceptance of broadcast packages and subscriptions to packages. Broadcast packages are those sent (via **pkgsend**) by servers without a preceding request by the client or target server receiving them. Broadcast authorization is the default for the **distauth** command. Package subscription—that is, specifying **-s** to **distauth**—automatically generates a request [pkgreq(RA_CMD)] when a new version of a package appears in the catalog.

Invoking the command without options or with only the **-S** option displays authorization information on **stdout.**

Each entry in the **distauth** output includes the following fields:

SERVER the name or alias of the authorized server (or **all**)

AUTH type of authorization: **b** (broadcast), **s** (subscription), **U** (subscription update)

PKG the package abbreviation, category, or the token **all**

ARCH the specified package architecture(s) or the token **all**

ACTION install, initiate, or spool

ADMIN | CATADMIN
the name of the **admin** file (for installation) or **catadmin** file (for initiation) or **default**, if none is specified. Will be blank if authorization is to spool.

RESPONSE file that provides responses to prompts during non-interactive package installation. Will be blank if none is specified or authorization is to spool. Heading will not appear for **-S** option.

The options for **distauth** are:

-S Indicate that the authorization is for the target server role only. Without **-S**, **distauth** authorizes for the client role.

-s Subscribe to a package. The default—**distauth** invoked without the **-s** option—is to authorize acceptance of a broadcast package.

-U Request updates for the package during subscription [**pkgreq -U**].

-i Install (or initiate if **-S** was specified) the package automatically. By default, a package is spooled when received.

If the Enhanced Security Extension is implemented on your system, do not use this option unless you also specify the **-S** option (which requests initiation).

-f *admin* | *catadmin*
: Use *admin* when installing the package. If the **-S** option has been specified, then use *catadmin* when initiating the package. This option is valid only with the **-i** option.

  If **-f** is not supplied, the default *admin* or *catadmin* file will be used.

-a *arch*
: Authorize the specified package architecture *arch*.

-r *resp*
: Use the **response** file *resp* when installing an interactive package on a client. This option is valid only with the **-i** and **-s** (subscription) options; do not use it with the **-S** option or for broadcast.

  If *resp* is the token **+** (plus sign), a default **response** file (if available from the server, as indicated by the catalog entry triggering the subscription) will be sent and used during the installation.

  Specifying a local **response** file is valid only if a single package is specified and a version of the package is already installed. A **response** file is generated by invoking **pkgask** *pkginst*, where *pkginst* is the instance identifier for an interactive package. See pkgask(RA_CMD) for more information.

-d
: Delete the specified authorization(s) from the database. For each entry matching the specified deletion criteria, **distauth** will prompt for confirmation (unless the **-n** option has also been specified).

-n
: Delete without prompting for confirmation.

pkg
: Specify one of the following values: (1) the package abbreviation that is authorized; (2) the token **all**, to authorize all packages; or (3) a *category* of package. A category is distinguished from a package name by its having a per cent prefix (**%***category*). For more information, see pkginfo(AS_CMD).

server
: Authorize *server* according to the permissions described by other options. A server will be validated against the list of known servers as defined using the **distconf** command. The token **all** may be used to specify the set of all known servers, or a server alias may be specified. [See remalias(RA_CMD).]

RETURN VALUES

Upon successful completion, **distauth** returns a value of **0**. Otherwise, it returns a non-zero value.

USAGE

This command is found on all client configurations.

Subscription assumes ''well formed'' version names such that sequencing can be determined between versions of the same package. If the progression of version names is ambiguous, **pkgreq** may not be invoked. Subscription for a package will not be triggered if the same (or a newer) version of the package is already installed or initiated on the system.

distauth does not modify existing entries. To modify an entry, use **distauth -d** to delete an entry; then use another **distauth** invocation to add the entry in the modified form.

If the Enhanced Security Extension is implemented on your system, you can run this command only if you have the appropriate administrative privileges, and you can request only initiation (not installation) of packages.

EXAMPLES

Example 1:

This example authorizes all broadcasts from server **charlie**, and displays the resulting authorization database:

```
# distauth all charlie
# distauth
SERVER     AUTH PKG     ARCH    ACTION    ADMIN   RESPONSE
charlie    b    all     all     spool
```

Example 2:

This example subscribes to updates for package **spell** from server **snoopy**:

```
# distauth -sU -i -f myadmin spell snoopy
# distauth
SERVER     AUTH PKG     ARCH    ACTION    ADMIN    RESPONSE
charlie    b    all     all     spool
snoopy     sU   spell   all     install   myadmin
```

The package will be installed on arrival using the file **myadmin**.

Example 3:

This example displays the contents of the authorization database relevant to the target server, then deletes all target server subscription authorizations from the authorization database:

```
# distauth -S
SERVER       AUTH PKG          ARCH       ACTION    CATADMIN
patty        s    terminf      i386       spool
marcie       b    %system      i386       initiate  default

# distauth -S -d -s all all
patty        s    terminf      i386       spool

Do you want to delete this authorization entry [y,n,?,q] y
```

SEE ALSO

catreq(RA_CMD), catsend(RA_CMD), distconf(RA_CMD), pkgask(AS_CMD), pkginfo(AS_CMD), pkgreq(RA_CMD), pkgsend(RA_CMD).

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

distconf – add machine and notification entries to software distribution configuration database

SYNOPSIS

distconf [**-d**] [**-u** *user* [,*user* . . .]] [**-e** *client_event* [,*client_event* . . .]]
[**-s** *server* [,*server* . . .]]

distconf -S [**-d**] [**-u** *user* [,*user* . . .]] [**-e** *server_event* [,*server_event* . . .]]
[**-s** *target_server* [,*target_server* . . .]] [**-c** *client* [,*client* . . .]]

DESCRIPTION

distconf provides an administrative interface to the distribution configuration database, which stores data such as known servers, clients, target servers, and notification event handling. Servers who originate packages for their network of dependent clients/target servers use the **-S** option. Clients and target servers use **distconf** without the **-S** option. When used without any options or with only the **-S** option, **distconf** displays configuration information on **stdout**.

Information is organized in the client and server configuration databases by the following parameters:

NOTIFYEVENT	**NOTIFYEVENT**
NOTIFYUSER	**NOTIFYUSER**
SERVERS	**CLIENTS**
	TSERVERS

Use a comma-separated list (no internal spaces) to specify multiple arguments to an option. (See SYNOPSIS above.)

The options for this command are:

-S Indicate that the invocation of **distconf** applies to the originating server role only, that is, to servers that originate packages. (Note that the target server role does not include origination of packages.) In the absence of the **–S** option, the invocation of **distconf** applies to the client role and target server role.

–d Delete specified information; must include one or more parameter(s) on the command line.

–c *client* Add (or delete) the specified *client* to (or from) the **CLIENTS** parameter.

-s *server* | *target_server*
Add (or delete) the specified *server* to (or from) the **SERVERS** parameter. (If the **–S** option is included on the command line, the specified *server* is added to or deleted from the **TSERVERS**—‘‘target servers’’—parameter.)

-u *user* Identify *user*(s) to whom electronic mail is sent in the case of a **NOTIFYEVENT**. The specified list is to be added to (or deleted from) the **NOTIFYUSER** parameter.

–e *client_event* | *server_event*
Indicate that the subsequent list of notification event(s) should be added to (or deleted from) any existing list associated with the **NOTIFYEVENT** parameter.

Valid client and target server events:

all All of the following events are configured as notification events.

received A package (requested or broadcast) has been received from a server.

broadcast A broadcast package (a package not requested by this machine) has been received from a server.

subscribe A request has been automatically issued for a subscribed package.

catalog A catalog has arrived from a server.

catfail A request for a catalog from a server has failed.

rejected A request for a package has been rejected by a server due to authorization problems.

unauthorized A server has attempted to broadcast a package which is not authorized by this machine.

spooled A requested or broadcast package has been delivered. It has been spooled for future installation or initiation.

installed A requested or broadcast package has been delivered to this client (or target server) and successfully installed (or initiated).

partial A requested or broadcast package has been delivered to this client (or target server), but automatic installation (or initiation) of the package has failed.

failed An attempt to request a package from a server has been unsuccessful. The request was successfully queued with the server, but the package could not be delivered.

held A request has been logged on the server because objects were not readily available; manual intervention by the server administrator is required.

cleanspool The spool areas have been cleaned.

Valid server events:

all All of the following events are configured as server notification events.

unauthorized A request to distribute a package is received from an unauthorized client (or target server).

rejected An attempt to send a package to a client (or target server) is rejected because of authorization problems.

failed An attempt to send a package to a client (or target server) is not successful.

installed	A package is successfully installed on a client (or initiated on a target server).
spooled	A package is successfully spooled on a client (or target server).
partial	A package was delivered, but automatic installation (or initiation in the case of a target server) failed.
held	A request was placed in the **waitlog** because objects were not available in the server spool area (that is, **pkgput -o** must be run to service the request).
catfail	An attempt to deliver a catalog has failed.

RETURN VALUES

Upon successful completion, **distconf** returns a value of **0**. Otherwise, it returns a non-zero value.

USAGE

This command is available on all configurations.

The specification of an invalid notification event will result in command failure and an error message which lists and describes each valid notification event. An attempt to remove a non–active attribute value will result in a warning message.

When an administrator for a machine in the client role/target server role uses **distconf** to specify a valid server for this machine, a device alias is automatically created for that server's ROI spool directory on this machine. The alias is in the form

dist_*server*

where *server* is the name or alias of the configured server.

If the Enhanced Security Extension is implemented on your system, you can run this command only if you have the appropriate administrative privileges.

EXAMPLES

Example 1:

This example displays a client's configuration database:

```
# distconf
NOTIFYEVENT=broadcast installed
NOTIFYUSER=root
SERVERS=snoopy
```

snoopy is this client's only server, and when a package is broadcast to or installed on this client, notification (mail) is sent to root.

Example 2:

The following example adds a new server to a client's configuration database, and displays the new configuration:

```
# distconf -s lucy
## Adding alias <dist_lucy> to device table...

# distconf
NOTIFYEVENT=all
NOTIFYUSER=linus!jane
SERVERS=charlie lucy
```

Example 3:

In this example, a server administrator cancels notification of all server events as well as mail notification to user **joe**; the administrator adds a subset of events for notification—those that indicate some failure—back to the configuration database.

```
# distconf -S -d -e all -u joe
# distconf -S -e failed,rejected,catfail
```

SEE ALSO

distauth(RA_CMD), distrpt(RA_CMD), pkgput(RA_CMD), remalias(RA_CMD).

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

distrpt – report on the contents of the software distribution administrative databases

SYNOPSIS

distrpt [**-p** *pkg* [,*pkg* . . .]] [*mach* . . .]

DESCRIPTION

distrpt displays summary information about this system's software distribution databases on **stdout**. The administrator may specify a package and/or machine name as selection criteria to limit the display. Run without options, the command displays all configuration, authorization, and catalog databases.

Options include:

-p *pkg* Request that only information about the package *pkg* be shown. The token **all** may be used to specify the instances when all packages are allowed. Please note that **all** is not the same as any package.

Optionally, you can specify a category of packages. A category is distinguished from a package name by its per cent prefix (**%***category*). For more information, see pkginfo(AS_CMD).

Use commas with no internal spaces to separate multiple arguments.

mach Request that only information about the machine *mach* (specified by name or alias) be shown. If you specify more than one machine, separate the items in your list with spaces.

RETURN VALUES

Upon successful completion, **distrpt** returns a value of **0**. Otherwise, it returns a non-zero value.

USAGE

This command is found on all configurations.

If the Enhanced Security Extension is implemented on your system, you can run this command only if you have the appropriate administrative privileges.

EXAMPLES

The following example shows output that **distrpt** might generate on a system configured as a client.

```
# distrpt
============================================================
Software Distribution Database Report for machine <hal>:
============================================================

----------------- Start of CLIENT info ----------------------

When I RECEIVE packages as a CLIENT:

    From       For          I accept        I auto-
    SERVER     PACKAGE      BROADCAST       SUBSCRIBE
    ---------  ----------   -------------   -------------
    charlie    all          spool           install
    linus      dwb          NO              spool
               pwb          install         NO
```

```
Other SERVERS I may manually request packages from:
    snoopy, woodstock

NOTIFICATION is sent to:
    root, sysadm

For EVENTS:
    Received an unsolicited package
    Received a package
    Unauthorized broadcast

EVENTS ignored:
    Received a catalog
    Unsuccessful package request
    Unsuccessful catalog request
    Request is waiting on a server
    Package successfully installed
    Automatic package installation failed
    A server rejected a package request
    Subscription package requested

------------------ End of CLIENT info ----------------------
============================================================
End of Software Distribution Database Report for machine <hal>
============================================================
```

Different output appears for differently configured systems.

The output is sorted within each role (that is, client/target server, server) by machine, then within each machine by package. The only exception to this rule is the server role catalog information, which is sorted first by package, and then by machine under each role for each package.

The event descriptions map to notification event tokens. For definitions of notification events, see distconf(RA_CMD).

The appearance of **initiate**, **install**, or **spool** in the **BROADCAST** and **SUBSCRIBE** columns of the output is determined by authorizations previously set up via **distauth**.

SEE ALSO

distauth(RA_CMD), distconf(RA_CMD), pkgput(RA_CMD).

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

pkgcat – display a catalog of packages available to a client or target server

SYNOPSIS

pkgcat [**-Sd**] [**-x** | **-l**] [**-p** *pkg* [*,pkg* . . .]] [**-n** *days*] [*server* . . .]

DESCRIPTION

pkgcat displays on **stdout** a catalog of packages that are available for request (via **pkgreq**) by the invoking system. **pkgcat** produces output similar in format to that of the **pkginfo** command. (See the EXAMPLES section.)

When invoked without options or arguments, **pkgcat** lists all packages available to the invoking machine in its role as a client. The default format lists category, package instance, and package name; one line per package is produced. Other formats can be selected—see the descriptions of the **-x** and **–l** options.

Options for **pkgcat** include the following:

-S Display a ''target-server-role'' view of the catalog—that is, a list of packages that can be ordered by **pkgreq -S**. If **-S** is not specified, a ''client-role'' view of the catalog will be displayed.

-d Display only the catalog entries for packages not already installed (or initiated, if the **-S** option has also been specified) on this machine.

-x Display an extracted listing of package information. Output in this format contains the package abbreviation, package name, package architecture, and package version.

-l Display a listing in long format. Output in this format contains all available information about the designated package(s).

-p *pkg* Display only the catalog entries for the specified *pkg*. The token **all** may be used to specify the instances when all packages are allowed. *pkg* may also be a category, which is distinguished from a package name by a prepended per cent sign (**%***category*). For more information, see pkginfo(AS_CMD).

Use a comma-separated list (no internal spaces) to specify multiple packages.

-n *days* Display only the catalog entries for packages that have been added to the catalog in the last *days*, where *days* is an integer (for example, 30).

server Display catalog entries for only those packages available from a specified server. Use a space-separated list to specify multiple servers.

RETURN VALUES

Upon successful completion, **pkgcat** returns a value of **0**. Otherwise, it returns a non-zero value.

USAGE

This command is available on client and full system configurations.

EXAMPLES

Example 1:

This example displays all entries from all server catalogs:

```
$ pkgcat
system      bnu      Basic Networking Utilities
system      cds      C Development Set
utilities   dfs      dfs utilities
system      sysadm   System Administration Utilities
application tstpkg   tstpkg Test Package
```

Example 2:

This example displays packages for a specified package category:

```
$ pkgcat -p %system
system      ed       Editing Utilities
system      ipc      Inter-Process Communication Utilities
system      sys      System Header Files
```

Example 3:

This example shows entries newer than seven days in long format:

```
$ pkgcat -l -n 7
        PKG:  tstpkg
       NAME:  tstpkg Test Package
   CATEGORY:  application
       ARCH:  i386
    VERSION:  Dev Release 01/04/90
       DESC:  Local testing package
  SERIALNUM:  77103-94g
     SERVER:  snoopy
   RESPONSE:  yes
 UPDATEABLE:  yes
```

Example 4:

This example displays, in extracted format, the packages from server **linus** that can be ordered in the server role:

```
$ pkgcat -S -x linus
cds             C Development Set
                (i386) 5.0
terminf         Terminal Information Utilities
                (i386) 4.0
```

SEE ALSO

pkginfo(AS_CMD), pkgreq(RA_CMD).

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

pkgdel – remove a previously initiated package

SYNOPSIS

pkgdel [**-on**] *pkginst* . . .
pkgdel -i [-n] [**-Nru**] *pkginst* . . .
pkgdel -i [-n] [**-s** *target_server*[**,** *target_server* . . .]]
[**-c** *client* [**,** *client* . . .] [**-ru**]] *pkginst* . . .

DESCRIPTION

On a server, **pkgdel** deletes a package that has been previously initiated for distribution by the **pkgput** command.

By default, information about the specified package is removed from the catalog database, and its objects are removed from the server's spool area (**distspool**). The **-i** option can be specified to only remove catalog information.

pkgdel does not remove tracking information for the specified package [see pkgtrk(RA_CMD)].

The options for this command are:

-n Specify that removal will occur without prompting for confirmation; invoking **pkgdel** without this option gives the administrator the opportunity to verify by responding to a prompt.

-o Remove package objects only; leave catalog information intact.

-i Remove specified catalog information only (leave package objects intact). For multiple arguments to **-s** or **-c**, use a comma-separated list with no internal spaces.

- **-N** Remove the serial number from all catalog entries for the package.
- **-s** Remove entries for the specified target servers (or aliases).
- **-c** Remove entries for the specified clients (or aliases).
- **-u** Remove the restriction on updates for specified clients or, if none are specified, all clients (clients will now be able to use **pkgreq -U**).
- **-r** Remove the response file for specified clients or, if none are specified, all clients.

pkginst Specify the package instance(s) to be deleted. A package instance is a variation of a software package, distinguished from other package instances by version or architecture or both; each package instance on a device or in a directory has a unique identifier, composed of either the package abbreviation (such as **pkgA**) or the package abbreviation plus a numerical suffix (such as **pkgA.2**).

Use a space-separated list to specify multiple package instances.

RETURN VALUES

Upon successful completion, **pkgdel** returns a value of 0. Otherwise, it returns a non-zero value.

USAGE

This command is available on server and full system configurations.

Use **pkgput -l** to see package instances in the server spool directory.

If the Enhanced Security Extension is implemented on your system, you can run this command only if you have the appropriate administrative privileges.

EXAMPLES

Example 1:

This example deletes the package **spell** from the server's distribution spool area and deletes information for the package from the catalog database:

```
# pkgdel spell
## The following package is currently initiated:

spell           Spell Utilities
                (i386) 4.0

Do you want to delete this package [y,n,?,q] y
 - Deleting target server <linus> catalog entry for <spell,4.0>
 - Deleting client <lucy> catalog entry for <spell,4.0>
## Deleting package <spell> from server spool area . . .
```

Example 2:

This example operates non-interactively on catalog database information for package **spell**. It deletes authorization for client **lucy** to order the package:

```
# pkgdel -n -i -c lucy spell
```

SEE ALSO

pkgput(RA_CMD).

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

pkgput – initiate a package on a server

SYNOPSIS

pkgput [**-x|-l**] [**-p** *pkg* [, . . .]]

pkgput [**-I**] [**-d** *device*] [**-c** *client* [,*client* . . .] [**-u**] [**-r** *resp*]] [**-s** *server* [**,***server* . . .]] [**-N** *serial*] *pkginst* . . .

pkgput [**-I**] [**-d** *device*] **-f catadmin** *pkginst* . . .

pkgput -i [**-c** *client* [,*client* . . .] [**-u**] [**-r** *resp*]] [**-s** *server* [,*server* . . .]] *pkginst* . . .

pkgput -i -N *serial pkginst*

pkgput -o [**-d** *device*] *pkginst* . . .

DESCRIPTION

pkgput initiates a package for request. Package initiation consists of:

entering package information in the catalog database

spooling the package objects into the server spool area (by default)

The server spool area can be referenced by the device alias **distspool**.

The catalog database contains information about all packages advertised to clients and target servers through catalogs [sent via **catsend** or **catreq**]. Clients and target servers can then order packages in catalogs via the **pkgreq** command.

pkgput can also do the following: (1) display catalog entries for initiated packages (first synopsis), (2) update catalog entries for initiated packages (fourth and fifth synopses), and (3) spool objects to initiated packages (sixth synopsis).

The specified actions are executed for each *pkginst.* (Note, however, that options **-r** and **-N** may be specified only if a single *pkginst* has been specified.) Use a comma-separated list (no internal spaces) to specify multiple arguments to the **-p**, **-c**, and **-s** options.

Options for **pkgput** include the following:

-x Display an extracted listing (package abbreviation, name, architecture, and version) of catalog database entries for initiated packages.

-l Display a long format listing (all available package information) of catalog database entries for initiated packages.

-p *pkg* Display only the catalog database entries for the specified *pkg*. *pkg* may also be a category, which is distinguished from a package name by its per cent prefix (**%***category*).

-I Initiate only the catalog information for the package instance specified (do not transfer objects).

-d *device* Initiate package from *device*. *device* can be a full pathname to a directory (such as **/var/tmp**), a device identifier (such as **/dev/diskette** for a removable disk), or a device alias (such as **qtape1** for a tape). [See putdev(ES_CMD).] In the absence of the option **-d**, **pkgput** looks for the package in the default installation spool directory referenced by the device alias **spool**.

Catalog options for **pkgput** include the following:

-c *client(s)* Make the package available to the listed *client* machine(s) and/or machine alias(es). The token **all** may be used to indicate that all client machines are authorized to request the package. Client names and aliases must have been configured previously, using the **distconf** command. The absence of this option means no client machines are authorized to receive the package.

At least one client and/or target server must be specified when initiating a package (either via **-s** or **-c**, or in the **catadmin** file being used).

-u Prevent clients from ordering updates of the package. By default, updates can be ordered for any initiated package.

-r *resp* Send *resp* as the default response file when a *client* orders an interactive package for installation. Valid only if a single *pkginst* has been specified.

An appropriate response file is generated by invoking **pkgask -r** *resp pkginst*, where *pkginst* is the instance identifier for an interactive package. See pkgask(AS_CMD) for more information.

-s *target_server*(s)
Make the package available to the listed *target_server* machine(s) and/or machine alias(es). See the description of the **-c** option above.

-N *serial* Specify *serial* as the serial number for a package. Valid only if a single *pkginst* has been specified. (Client and target server entries added later will not inherit the serial number.)

Other options for **pkgput** include the following:

-f *catadmin* Use *catadmin* as input to the catalog entry.

If you invoke **pkgput** with no display options and no catalog input options (that is, with none of the following: **-c**, **-s**, **-r**, **-N**, **-u**, **-f**), the default **catadmin** file will be used as input to the package initiation process.

pkginst Specify the package instance(s) to be initiated. A package instance is a variation of a software package, distinguished from other package instances by version or architecture or both; each package instance on a device or in a directory has a unique identifier, composed of either the package abbreviation (such as **pkgA**) or the package abbreviation plus a numerical suffix (such as **pkgA.2**).

Use a space-separated list to specify multiple package instances.

-i Update the catalog information for the initiated package instance specified. At least one catalog option (**-c, -s, -r, -N, -u**) must be specified. If **-r** and **-u** are specified without the **-c** option, the update will affect all client catalog entries for that *pkginst*. **-N** may not be combined with **-c** or **-s**; the serial number for all entries of the *pkginst* will be updated.

-o Add objects to an initiated package instance. Useful when rejuvenating a client request that has been held because package objects were not previously spooled online; can also be used to overwrite objects of an initiated package.

RETURN VALUES

Upon successful completion, **pkgput** returns a value of **0**. Otherwise, it returns a non-zero value.

USAGE

This command is available on server and full system configurations.

To modify existing catalog information, use

pkgdel -i *old_catalog_info pkginst*
pkgput -i *new_catalog_info pkginst*

If the Enhanced Security Extension is implemented on your system, you can run this command only if you have the appropriate administrative privileges.

EXAMPLES

Example 1:

This example initiates the package **spell**:

```
# pkgput -c lucy,linus -N C453 spell
## Initiating package <spell> . . .
   - Transferring <spell> package instance to
     </var/spool/dist> in file system format
   - Making package instance <spell> accessible to id <dist> . . .
   - Creating client <lucy> catalog entry for <spell,2.0>
   - Creating client <linus> catalog entry for <spell,2.0>
## Package <spell> initiated in server spool area
```

The **pkgput** command line specifies that clients **lucy** and **linus** may order the package **spell** and that the serial number for this package is **C453**. **pkgput** places the package in the server's distribution spool area and adds information about the package to the catalog database.

Example 2:

This example adds target server **woodstock** to the catalog information for package **spell**:

```
# pkgput -i -c woodstock spell
## Updating info for package <spell> . . .
  - Creating client <woodstock> catalog entry for <spell,2.0>
```

Example 3:

This example initiates the package **lp** without objects (to save disk space) and uses the **catadmin** file called **myadmin** as catalog input:

```
# pkgput -I -f myadmin lp
## Initiating package <lp> . . .
   - Transferring information files for <lp> package instance to
     </var/spool/dist> in file system format
   - Making package instance <lp> accessible to id <dist> . . .
   - Creating client <all> catalog entry for <lp, Release 3> . . .
## Package <lp> initiated in server spool area
```

SEE ALSO

catsend(RA_CMD), distconf(RA_CMD), pkgdel(RA_CMD), pkgreq(RA_CMD).

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

pkgreq – request delivery of a software package

SYNOPSIS

pkgreq [**-q**] [**-U**] [**-s** *server*] [**-i** [**-f** *admin*] [**-r** *resp*]] [**-a** *arch*] [**-v** *version*] *pkg*
pkgreq -S [**-q**] [**-s** *server*] [**-i** [**-f** *catadmin*]] [**-a** *arch*] [**-v** *version*] *pkg*
pkgreq -l [*jobid* | *server* | *pkg*]
pkgreq -k *jobid*

DESCRIPTION

pkgreq requests delivery of a package from a server to this client (or this target server). It can also request an update to a package currently installed. It can provide status information on package requests and allow a queued request to be canceled.

pkgreq verifies the requested package is listed in the current catalog. Once validated, the request is sent to the appropriate server. The package is delivered to the directory referenced by the device alias **dist**_*server*, and optionally installed or initiated from there.

Other processing occurs as specified for each of the following options:

-q Queue the request via the Remote Operation Interface (ROI); ROI will return a *jobid*. The default invocation (**pkgreq** without the **-q** option) processes the request in real time.

You can use **pkgreq -l** or the **remstat** command to check the status of the queued package request. [See remstat(RA_CMD).]

-S Indicate that the request is being made by a target server. The package should be sent in a form such that it can be initiated for further distribution rather than installed on the calling machine. The default invocation of **pkgreq** (without **-S**) requests that the package be sent in a form suitable for installation on a client.

-s Request *pkg* from the specified *server*. If the token **all** is specified, **pkgreq** will examine the catalog for all entries that match the requested package; the server in the first catalog entry it finds will be used. If the token was not specified and multiple servers are available, the user is interactively prompted to make a selection.

-i Request installation or initiation of the package after it has been received. In the case of a target server role (that is, if **-S** has been specified on the command line), **pkgput** will be executed for this package (package initiation). In the case of a client role, **pkgadd** will be invoked (package installation).

If the Enhanced Security Extension is implemented on your system, do not use this option unless you also specify the **-S** option (which requests initiation).

-f *admin* | *catadmin*
Use *admin* when installing the package. If the **-S** option has been specified, use *catadmin* when executing **pkgput**. This option is valid only with the **-i** option.

If **-f** is not supplied, the default *admin* or *catadmin* file will be used.

-r *resp* Specify *resp*, which is the **response** file to be used when installing an interactive package on a client. This option is valid only with the **-i** option; do not use it with the **-S** option.

If *resp* is the token **+** (plus sign), a default **response** file (if available) will be sent and used during the installation. Check the **RESPONSE** parameter in the display produced by **pkgcat -lp** *pkg* to see if a default **response** file is available from the server.

Specifying a local **response** file is valid only when a version of the package is already installed. A **response** file is generated by invoking **pkgask** *pkginst*, where *pkginst* is the instance identifier for a package. See pkgask(AS_CMD) for more information.

-a Specify *arch* as the architecture of the requested package.

-v Specify *version* as the version of the requested package.

-U Request an update to the currently installed version. **-v** *version* specifies the new version. By default (without the **-v** option), the currently installed version is updated to the new version in the catalog; if several new versions are available, the user is prompted to select one.

pkg Specify the package abbreviation of the requested package. The package abbreviation, architecture, and version uniquely identify a package instance. Specify all three to avoid **pkgreq** prompting when there are multiple instances of a package available.

-l Display status of job associated with *jobid*, *client*, or *pkg*. If the **-l** option is used without any arguments, the status of all recent package requests will be displayed in short format. (See EXAMPLES below.)

-k *jobid* Cancel (kill) the queued request specified by *jobid*.

server Request the package from *server*.

jobid Specify the ROI job identifier of a package request.

RETURN VALUES

Upon successful completion, **pkgreq** returns a value of **0**. Otherwise, it returns a non-zero value.

USAGE

This command is available on client and full system configurations.

If you invoke **pkgreq** in real time (that is, without the **-q** option), package delivery or update may take a long time; it could take even longer with the **-i** option since initiation or installation takes place while you wait. To avoid waiting, use the **-q** option.

If the Enhanced Security Extension is implemented on your system, you can run this command only if you are logged in as **dist**, and you can request only initiation (not installation) of packages.

EXAMPLES

Example 1:

This example requests that the package **mystuff** be sent and installed in real time:

```
# pkgreq -i mystuff
## Package <mystuff, Issue 1> requested from <snoopy>...
## Installing package <mystuff>...

My Favorite Stuff
(i386) Issue 1

Enter path to package base directory (default: /tmp) [?,q]
Using </tmp> as the package base directory.
## Processing package information.
  .
  .
  .
## Package <mystuff> successfully installed
## Deleting package <mystuff> from client spool area...
```

Example 2:

This example verifies that the package **mypkg** is already installed on the system (**pkginfo** command), and that a new version of it is available in the catalog (**pkgcat** command). The **pkgreq** invocation then queues an update request for the new version:

```
# pkginfo -x mypkg
mypkg        My Favorite Package
             (i386) Release 1.0
# pkgcat -x -p mypkg
mypkg        My Favorite Package
             (i386) Release 1.1

# pkgreq -q -U -v "Release 1.1" mypkg
## ROI job ID to machine <snoopy> is <a-216>
## Request for <mypkg, Release 1.1> pkgmap from
   <snoopy> queued
```

Example 3:

This example shows the current status of recent **pkgreq** commands:

```
su:2$ pkgreq -l
REQID        SERVER   PKG          DATE         STATUS
a-192        snoopy   mystuff      Feb 20 15:16 installed
a-205        linus    lp           Feb 21 09:45 spooled
a-216        snoopy   mypkg        Feb 21 11:49 map ordered
```

Example 4:

This example cancels a queued package request:

```
# pkgreq -k a-216
Job (r-214 a-216 p-217) to machine snoopy has been canceled.
UX:reqstat: WARNING: Request of <mypkg> from <snoopy> canceled
```

SEE ALSO

pkgadd(RA_CMD), pkgcat(RA_CMD), pkginfo(AS_CMD), pkgput(RA_CMD).

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

pkgsend – deliver packages to client or target server machine(s)

SYNOPSIS

pkgsend [**-q**] [**-i** [**-r** *resp*]] [**-d** *device*] [**-a** *arch*] [**-v** *version*] *pkg client* . . .
pkgsend -S [**-q**] [**-i**] [**-d** *device*] [**-a** *arch*] [**-v** *version*] *pkg target_server* . . .
pkgsend -l [*jobid* | *client* | *pkg*]
pkgsend -k *jobid*

DESCRIPTION

pkgsend delivers a package from this server to the specified client or target server machine(s). It can also provide the status of package deliveries and allow a queued delivery to be canceled. The package is delivered to a directory on the destination machine referenced by the device alias **dist**_*this_server*, and optionally installed or initiated from there.

Options for this command include:

-q Queue the delivery via the Remote Operation Interface (ROI); ROI returns a *jobid* for each job (to each valid recipient machine). The default invocation (**pkgsend** without the **-q** option) processes the delivery in real time.

You can use **pkgsend -l** or the **remstat** command to check the status of the queued package delivery. [See remstat(RA_CMD).]

-S Indicates recipients are target servers.

-i Install or initiate the package on the target machine(s) using **pkgadd** for specified *client(s)* or **pkgput** for specified *target server(s)*.

If the Enhanced Security Extension is implemented on your system, do not use this option unless you also specify the **-S** option (which requests initiation).

-r *resp* Use *resp* (a response file) to give responses during installation. Valid only with **-i**.

A response file is generated by invoking **pkgask** *pkginst*, where *pkginst* is the instance identifier for an interactive package. See pkgask(AS_CMD) for more information.

-d *device*
Send package from *device*. *device* can be a full pathname to a directory (such as **/var/tmp**), a device identifier (such as **/dev/diskette** for a removable disk), or a device alias (such as **qtape1** for a tape). [See putdev(ES_CMD).] The default device is **distspool**.

-a *arch* Specify *arch* as the architecture of the package to send.

-v *version*
Specify *version* as the version of the package to send.

pkg Specify the package abbreviation of the package to send. The package abbreviation, architecture, and version uniquely identify a package instance. Specify all three to avoid **pkgsend** prompting when there are multiple instances of a package available.

-l Display status of job associated with *jobid*/*client*/*pkg*. If the **-l** option is used without any arguments, the status of all recent deliveries is displayed in short format.

-k *jobid*
Cancel (kill) queued job associated with the indicated *jobid.*

client Client that will be the recipient of the **pkgsend** (or if the **-S** option is specified, *target_server*); may be a machine alias or the token **all,** in which case delivery to all configured clients or target servers is requested. Use a space-separated list to specify multiple arguments.

jobid Specify the ROI job identifier associated with a package delivery.

RETURN VALUES

Upon successful completion, **pkgsend** returns a value of **0**. Otherwise, it returns a non-zero value.

USAGE

This command is found on all server configurations.

If you invoke **pkgsend** in real time (that is, without the **-q** option), package delivery may take a long time; it could take even longer with the **-i** option, since initiation or installation takes place while you wait. To avoid waiting, use the **-q** option.

If the Enhanced Security Extension is implemented on your system, you cannot run this command unless you are logged in as **dist**, and you can request only initiation (not installation) of packages.

EXAMPLES

Example 1:

This example broadcasts the package **spell** to client **linus**:

```
# pkgsend spell linus
## Polling <linus> for package acceptance . . .
## Transferring package <spell> to client <linus> . . .
## Spooling package <spell> on <linus> . . .
## Package <spell> successfully spooled on <linus>.
```

Example 2:

This example creates a response file for package **lp**, then queues delivery of the package and its newly created response file:

```
# pkgask -r /var/sadm/dist/response -d distspool lp
.
.
.
Response file </var/sadm/dist/response/lp> was created.
Processing of request script was successful.
# pkgsend -q -i -r lp lp lucy
## ROI job ID to machine <lp> is <a-566>
## Distribution of package <lp> queued to <lucy>
```

Assuming **lucy** has authorized the broadcast and installation of the package **lp**, it will be automatically installed using the response file **lp** and under the control of the **admin** file specified in the broadcast authorization entry. [See distauth(RA_CMD).]

Example 3:

This example lists current status of recent **pkgsend** commands:

```
# pkgsend -l
SRVID       CLIENT   PKG         DATE         STATUS
a-560       linus    spell       Feb 19 13:17 spooled
a-566       lucy     lp          Feb 19 13:20 polling
```

Example 4:

This example cancels a queued package delivery:

```
# pkgsend -k a-566
Job (r-565 a-566 p-567) to machine lucy has been canceled
UX:sndstat: WARNING: Distribution of <lp> to <lucy> canceled.
```

SEE ALSO

distauth(RA_CMD), distconf(RA_CMD), pkgadd(AS_CMD), pkgask(AS_CMD), pkgput(RA_CMD), pkgreq(RA_CMD).

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

pkgtrk – display/delete tracking information for delivered packages

SYNOPSIS

pkgtrk [**-x** | **-l** [**-s**]] [*pkg* . . .]
pkgtrk -d [**-n**] [**-a** *days*] *pkg* . . .

DESCRIPTION

pkgtrk displays on **stdout** the names of successfully delivered packages and the number of machines to which they were delivered. Delivered packages are any packages sent via either the **pkgsend** command or in response to a client's invocation of **pkgreq**. When used as shown in the second synopsis line (see SYNOPSIS above), **pkgtrk** deletes tracking information for the specified package(s). **pkgtrk** prompts the user to confirm each deletion, unless the **-n** option is included on the command line.

The default format lists package abbreviation, architecture, version, and the number of recipient machines.

Fields for long format are:

package information (package abbreviation, package name, category, architecture, version, serial number)

currently held requests (requesting machine name, time, client or target server, update)

currently sending (machine name, time, client or target server, update)

successful broadcasts (machine, delivery date, client or target server, installation status)

successful requests (machine, delivery date, client or target server, status, update)

client total, target server total

The options for this command are:

-l Print long format (all fields printed in a report format), which includes all available information about the designated package(s).

-x Print extracted format, suitable for parsing, which contains the same fields in the same order as in long format. Fields are delimited by carets (^).

-s Summarize in extracted or long format. Must be specified in conjunction with **-x** or **-l**.

pkg Package abbreviation or category. The token **all** may be used to specify the instances when all packages are allowed. *pkg* may also be a category, which is distinguished from a package name by its per cent prefix (**%***category*). For more information, see pkginfo(AS_CMD).

Use a space-separated list to specify multiple packages.

-d Delete tracking information for the specified package abbreviation or **%***category*.

-a *days* Delete tracking information older than *days* for the specified *pkg* (or for all packages if the token **all** is specified).

-n Delete tracking information without any further interaction (to be used with the **-d** option).

RETURN VALUES

Upon successful completion, **pkgtrk** returns a value of **0**. Otherwise, it returns a non-zero value.

USAGE

This command is found on all server configurations.

If the Enhanced Security Extension is implemented on your system, you can run this command only if you have the appropriate administrative privileges.

EXAMPLES

Example 1:

With no options or arguments, **pkgtrk** produces output such as the following:

```
#pkgtrk
spell       (i386) 4.0        7
terminf     (i386) 1.0        8
```

Example 2:

This example displays tracking information in long format. (For a display of the same information in extracted format, see Example 3.)

```
#pkgtrk -l spell
              PKG: spell
             NAME: SPELL Utilities
         CATEGORY: system
             ARCH: i386
          VERSION: 4.0
        SERIALNUM: 22B-567
  CURRENTLY -      (objects not on server)
          WAITING: woodstock since  Jul 18 10:36
                   spike   since  Jul 18 10:50
  DELIVERED -
       BROADCASTS: charlie  May 19 15:00  client   installed
                   lucy     May 20 11:03  client   spooled
                   snoopy   Jun 20 09:47  client   partial
                   linus    Jun 21 08:01  target server   spooled
         REQUESTS: patty    May 19 10:42  client   installed
                   marcie   May 20 11:01  client   installed
                   sally    May 21 02:33  target server   initiated
      CLIENT TOTAL: 5
      SERVER TOTAL: 2
```

Example 3:

This example displays the same information shown in Example 2 in extracted format:

```
#pkgtrk -x spell
spell^SPELL Utilities^system^i386^1.0^22B-567^woodstock(Jul 18 1
0:36),spike(Jul 18 10:50)^^charlie(May 1915:00)client-installed,
....^patty(May 1910:42)client-installed-update,...^5^2
```

Example 4:

This example displays information in summarized long format (**-ls**):

```
# pkgtrk -ls inst backup
PKG      ARCH   VERSION       WAITING SENDING CLIENTS SERVERS
inst     i386   Release 1.0         2       0       5       2
backup   i386   Release 2.2         0       3       2       6
```

SEE ALSO

pkginfo(AS_CMD), pkgput(RA_CMD), pkgsend(RA_CMD).

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

remadmin – control remote operation environment

SYNOPSIS

remadmin [-a *aging***] [-t** *timeout***] [-u|-d|-r** *logname***[,***logname***...]]**

DESCRIPTION

The **remadmin** utility allows the administrator to set or display the remote operations attributes, remove log files and sub-directories, and reset the job identifier counter for one or more users.

The meanings of the options are as follows:

–a *aging* Set the remote job aging parameter. This is the amount of time that complete jobs should be carried in the remote administration status report.

–t *timeout* Set the remote job timeout parameter. This is the amount of time a job will run before it is considered to have failed.

–u *logname***[,***logname***...]**
Authorize users to access the remote operation environment.

–d *logname***[,***logname***...]**
Delete users from a list of users who can access the remote operation interface.

–r *logname***[,***logname***...]**
Removes the Remote Operation Interface files and directories in **/var/spool/roi/users/***logname* and resets the job identifier counter for the list of *lognames*. The operation will succeed if no jobs are queued or in progress.

While administrators can invoke **remadmin** functions with a list of users as arguments, users themselves can only invoke **remadmin -r** with their own login ID as an argument. Thus, users can remove only their own ROI files and sub-directories and reset only their own job ID counter.

Invoking **remadmin** with no options displays all current values. If an option and argument are provided, the corresponding parameter is modified.

The *aging* and *timeout* arguments must be of the form *dd.hh.mm*.

With the *logname* as an argument, the **-u** flag creates an administrative file structure enabling the specified user to access the remote operation interface.

USAGE

System administrators can use **remadmin** with all its options. Users can invoke **remadmin** with the **-r** option and must specify their own *logname* as an argument.

SEE ALSO

mgroup(RA_LIB), remop(RA_LIB), remop(RA_CMD)

FUTURE DIRECTIONS

Certain of the Remote Operations Interface library and commands have been moved to Level 2 for the following reasons. On going standards and industry direction has converged around an object-oriented approach to distributed, as opposed to remote, system administration. The ROI interfaces will become obsolete as standards and consensus mature within the workings of the Object Management Group, X/Open Systems Management, and IEEE P1003.7. It is expected that the ROI interfaces will be replaced by standard API's and command-line interfaces as part of the ongoing efforts in the area of Distributed Systems and Network Management.

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

remalias – administer machine aliases

SYNOPSIS

remalias
remalias [-x] -m *alias***[,***alias...]*
remalias -a|-A -m *alias***[,***alias***...]** *machine_or_alias***[,***machine_or_alias***...]**
remalias -d *alias***[,***alias***...]**
remalias -d -m *alias***[,***alias***...]** *machine_or_alias***[,***machine_or_alias***...]**
remalias -l *machine_or_alias***[,***machine_or_alias***...]**

DESCRIPTION

The **remalias** utility is used to administer machine aliases that contain one or more machine names, aliases, or both. The **mgroup**() function expands machine aliases to support remote administration.

The following are options to **remalias**:

–x Expand aliases to component machines.

–m *alias***[,***alias***...]** Select one or more aliases.

–a *machine_or_alias***[,***machine_or_alias***...]**
Add machine names or aliases. This is used with **-m** to select target aliases.

–A *machine_or_alias***[,***machine_or_alias***...]**
Overwrite machine names or aliases with new machine names or aliases. This is used with **-m** to select target aliases.

–d *machine_or_alias***[,***machine_or_alias***...]**
Delete machine names or aliases. Can be used with **-m** to select target aliases.

–l *machine_or_alias***[,***machine_or_alias***...]**
List machines names or aliases.

remalias with no options or arguments lists all known machine aliases.

Invoking **remalias** with the **-m** and **-x** options expands machine aliases into machine names. Machines are listed once. Invoking **remalias** with only the **-m** option lists the contents (machine and alias) of the specified alias; it does not expand any component aliases into machine names.

remalias -a adds machine names or aliases to the file containing specified aliases. the new machine names or aliases are appended to machine names or aliases already included. The **-A** option adds machine names or aliases after deleting any machine names or aliases already included.

remalias -d deletes the specified aliases. **remalias -d** with the **-m** option deletes the specified machine names or aliases from the file containing specified aliases.

remalias -l shows all aliases to which a machine or alias belongs. One machine name and alias are listed per line.

USAGE

System administrators can use **remalias** with all its options. Users can invoke **remalias** with no options, or with the **-x**, **-m**, and **-l** options only.

SEE ALSO

mgroup(RA_LIB)

FUTURE DIRECTIONS

Certain of the Remote Operations Interface library and commands have been moved to Level 2 for the following reasons. On going standards and industry direction has converged around an object-oriented approach to distributed, as opposed to remote, system administration. The ROI interfaces will become obsolete as standards and consensus mature within the workings of the Object Management Group, X/Open Systems Management, and IEEE P1003.7. It is expected that the ROI interfaces will be replaced by standard API's and command-line interfaces as part of the ongoing efforts in the area of Distributed Systems and Network Management.

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

remclean – remote operation interface clean-up program

SYNOPSIS

remclean

DESCRIPTION

remclean scans the remote operation log directories and removes jobs that have completed or have timed out.

Complete jobs exceeding the aging parameter are removed from the tracking log and no longer show up in the output of the **remstat** command. This applies to jobs with the status succeeded, failed, canceled, timeout, or rejected.

remclean cancels queued jobs exceeding the timeout parameter via the appropriate network-specific cancel operation, and changes the status of the job in the status log to timeout. **remclean** invokes the **-n** "*notify*" argument to the **remop**(RA_CMD) command, if specified when the job was initiated, and it informs the user via **mail**(BU_CMD) that the job timed out. This applies to jobs with the status inprogress and queued. If the job is part of a dependency list, the remaining jobs in the list are canceled and the **-n** "*notify*" argument to **remop**(RA_CMD) is invoked if specified when the job was started.

USAGE

remclean is available only to administrators.

SEE ALSO

remadmin(RA_CMD), remstat(1), cron(RA_CMD), remop(RA_LIB), remop(RA_CMD)

FUTURE DIRECTIONS

Certain of the Remote Operations Interface library and commands have been moved to Level 2 for the following reasons. On going standards and industry direction has converged around an object-oriented approach to distributed, as opposed to remote, system administration. The ROI interfaces will become obsolete as standards and consensus mature within the workings of the Object Management Group, X/Open Systems Management, and IEEE P1003.7. It is expected that the ROI interfaces will be replaced by standard API's and command-line interfaces as part of the ongoing efforts in the area of Distributed Systems and Network Management.

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

remkill – cancel remote operation jobs

SYNOP

remkill -a [-u *user***]**
remkill -j *jobid* **[-u** *user***]**

DESCRIPTION

The **remkill** utility cancels remote jobs in the queued state.

The options are as follows:

–a Cancel all jobs belonging to the user. If the **-u** flag is not used, then the current user login is assumed. Although administrators can cancel jobs for all users, non-privileged users can cancel only their own jobs.

–j *jobid* Cancel the specified job known to the machine by its service job identifiers, administrative job identifiers, or primitive job identifiers. Job identifiers take the format *c-x* where *c* is the job identifier type and *x* is a value from 1 to **INT_MAX** (defined in the header file **limits.h**). Service job identifiers take the format **r-***x*, administrative job identifiers take the format **a-***x*, and primitive job identifiers take the format **p-***x*.

–u *user* Cancel a job for the specified *user*. If the **-u** flag is not used, then the current user login is assumed. Although administrators can cancel jobs for all users, non-privileged users can cancel only their own jobs. Since **-u** must take an argument, non-privileged users must include only their own logins.

If the job is part of a dependency list and the **remkill** operation was successful, the remaining jobs in the list are also canceled.

Successfully canceling a job depends on how far the job has proceeded, which in turn depends on how the network service interface is designed on the initiating machine. The success or failure of an operation is reported to the user.

USAGE

Only system administrators can cancel remote jobs for other users. Users can cancel their own jobs only, and must specify their login as the *user* argument to
-u.

SEE ALSO

remop(RA_LIB), remstat(RA_CMD)

FUTURE DIRECTIONS

Certain of the Remote Operations Interface library and commands have been moved to Level 2 for the following reasons. On going standards and industry direction has converged around an object-oriented approach to distributed, as opposed to remote, system administration. The ROI interfaces will become obsolete as standards and consensus mature within the workings of the Object Management Group, X/Open Systems Management, and IEEE P1003.7. It is expected that the ROI interfaces will be replaced by standard API's and command-line interfaces as part of the ongoing efforts in the area of Distributed Systems and Network Management.

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

remop – command interface to **remop** for remote operations

SYNOPSIS

remop -e *cmd* **[-q[-n** *notify***]] -m** *machine_or_alias***[,***machine_or_alias***...]**
[-s *svc***]**
remop -t *local* **[-d** *remote***] [-q[-n** *notify***]]**
-m *machine_or_alias***[,***machine_or_alias***...] [-s** *svc***]**

DESCRIPTION

The **remop** utility provides a command interface to the remote operations library routines **remop**(RA_LIB) and **mgroup**(RA_LIB).

The options are as follows:

–e *cmd* Execute a command on a remote system. *cmd* is an executable command, with arguments, that is enclosed with double quote characters. If the command does not take arguments, double quote characters are unnecessary.

–t *local* Transfer a file or directory to a remote system.

–q Perform remote operation over queued network services. The default is to use synchronous network services.

–n *notify* *notify* is the full pathname of an executable called after the job is completed. Environment variables are available to *notify* as described in **remop**(RA_LIB) This option works over queued network services only and can be used only in conjunction with the **-q** option. If one or more space characters separates the executable from arguments or shell special characters, enclose the entire string with double quote characters.

–s *svc* Use a service identifier to place output on remote machines according to Remote Operations Interface (ROI) standard directory naming conventions of **/var/spool/roi/users/***logname***/receive/***svc***/***mach* where *svc* is a service identifier and *mach* is the originating machine. If *svc* is not specified, the value **default** is substituted where the *svc* identifier normally appears.

–m *machine_or_alias***[,***machine_or_alias***...]**
Provide a comma-separated list of machines names, machine aliases, or both on which remote jobs will be executed using queued network services. For jobs executed over synchronous network services, only one machine can be specified.

–d *remote* Designate a destination file or directory. If the *remote* file name or directory exists, the local file or directory is copied into the specified destination. If you do not specify a destination on the remote machine, **remop**(RA_LIB) creates it based on ROI's standard

directory naming convention,
/var/spool/roi/users/*logname***/receive/***svc***/***mach*
where *svc* is the service identifier and *mach* is the originating machine. If the **-d** *remote* destination does not exist, you receive an error message.

remop satisfies the request via available network services.

The order for accessing network services may be influenced by two user-defined environment variables—**REMOPS** and **REMOPQ**. These variables override the order specified by the system administrator using the **remtab**(RA_CMD) command. **REMOPS** specifies the order for trying synchronous network services, and **REMOPQ** specifies the order for queued network services. The value of each environment variable is expected to be a colon-separated list of network services. If more than one remote machine is specified for file or directory transfer, each machine receives a copy of the file or directory (queued mode only).

USAGE

Both users and system administrators can use **remop** with all its options.

EXAMPLE

Below are examples for using the **remop** command for synchronous and queued operations.

Example 1 — Synchronous Operation

The following example shows how to execute the command **/sbin/mount** on the remote system **intl** and receive the output on the local system and terminal. The operation will be initiated over a synchronous network service.

```
remop -e /sbin/mount -m intl
```

Example 2 — Queued Operation

The following example shows how to execute the command **/sbin/mount** on the remote system **intl** and receive the output on the local system. The **/home/user1/bin/mynotify** command on the local system will be executed when the operation completes. The operation will be initiated over a queued network service.

```
remop -e /sbin/mount -q -n /home/user1/bin/mynotify -m intl
```

Unless redirected, the standard output of the *notify* script will be in the file **/var/spool/roi/users/***login***/notify/o***primid* on the local system. The **stderr** will be found in the file, **/var/spool/roi/users/***login***/notify/e***primid*.

Unless redirected, the standard output of the *notify* script will be in the file **/var/spool/roi/users/***logname***/notify/o***primid*. The standard error will be found in the file, **/var/spool/roi/users/***logname***/notify/e***primid*.

SEE ALSO

mgroup(RA_LIB), remop(RA_LIB), remalias(RA_CMD), remkill(RA_CMD), remtab(RA_CMD), remstat(RA_CMD)

FUTURE DIRECTIONS

Certain of the Remote Operations Interface library and commands have been moved to Level 2 for the following reasons. On going standards and industry direction has converged around an object-oriented approach to distributed, as opposed to remote, system administration. The ROI interfaces will become obsolete as standards and consensus mature within the workings of the Object Management Group, X/Open Systems Management, and IEEE P1003.7. It is expected that the ROI interfaces will be replaced by standard API's and command-line interfaces as part of the ongoing efforts in the area of Distributed Systems and Network Management.

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

remstat – track the status and retrieve output of remote jobs

SYNOPSIS

remstat [-v | -p] [-l] [-a | -u *user***] [-i** *svc***] [-s** *state***]**
[-m *machine_or_alias***[,***machine_or_alias***..]] [-j** *jobid***]**
remstat -c [-u *user***] -j** *jobid*

DESCRIPTION

The **remstat** utility tracks the status and retrieves output of remote jobs. With no options, **remstat** displays the status of the invoking user's administrative jobs.

The options are as follows:

–a Produce a report on all users. Without this or the **-u** option, the report defaults to those ROI jobs initiated by the current user.

–c This option displays **stdout** and **stderr** from a selected job and must be used with the **-j** *jobid* option. Without the **-u** option, the current user login is assumed.

–i *svc* Select a specific administrative service. *svc* is a token that associates a remote operation with a specific administrative service. For example, you could use **dist** to represent the Software Distribution service.

–j *jobid* Specify the service, administrative or primitive job id. Job identifiers take the format *c*-*x* where *c* is the job identifier type and *x* is a value from 1 to **INT_MAX** (defined in the header file **limits.h**). Service job identifiers take the format **r-***x*, administrative job identifiers take the format **a-***x*, and primitive job identifiers take the format **p-***x*. A short-hand notation is allowed for the job id specification, where leading zeroes may be omitted on the command line.

–l Produce a list of colon-separated fields rather than formatted output.

–m *machine_or_alias***[,***machine_or_alias***...]**
Select one or more machines or aliases. *machine_or_alias* is a comma-separated list of machine names or aliases.

–p List all remote jobs associated with each network service primitive. The report contains the status and the originated time of a remote operation.

–s *state* Select a specific state. Defined states are: queued, inprogress, succeeded, failed, canceled, rejected, and timeout. The first letter in the name of each defined state can be used as a status argument. An administrative remote job runs until its component remote primitives are done; it is considered failed as soon as one primitive remote operation fails.

–u *user* — Produce a report on the specified user only. Without this or the **-a** option, the report defaults to those ROI jobs initiated by the current user.

–v — List all primitive remote jobs associated with each administrative job and destination machine.

USAGE

Both users and system administrators can use **remstat** with all its options.

EXAMPLE

Below is an example of using **remstat** with the **-l** option alone and combined with other options.

Sample output from **remstat -l**:

```
usera:r-l:a-4:succ:dist:machb
usera:r-l:a-4:queued:dist:machb
usera:r-l:a-7:inprog:dist:machc
usera:r-l:a-7:queued:dist:machc
usera:r-l:a-10:inprog:dist:machd
usera:r-l:a-10:queued:dist:machd
usera:r-l:a-12:inprog:bck:mache
```

This display takes the output for **remstat** with no options and produces a list of colon-separated fields without headings. The first field is the user followed by the fields for the service identifier, administrative identifier, status, type of service, and destination machine. Note that leading zeros are omitted for service and administrative identifiers.

Sample output from **remstat -l -v**:

```
usera:r-1:a-4:p-2:ft:succ:dist:machb
usera:r-1:a-4:p-3:ft:queued:dist:machb
usera:r-1:a-7:p-5:re:inprog:dist:machc
usera:r-1:a-7:p-6:ft:queued:dist:machc
usera:r-1:a-10:p-8:ft:inprog:dist:machd
usera:r-1:a-10:p-9:dt:queued:dist:machd
usera:r-11:a-12:p-13:dt:inprog:bck:mache
```

This display takes the output from **remstat -v** and produces a list of colon-separated fields without headings. The first field is the user followed by the fields for the service identifier, administrative identifier, primitive identifier, primitive module, status, type of service and destination machine. Note that leading zeros are omitted for service, administrative, and primitive identifiers.

Sample output from **remstat -lp**:

```
usera:p-2:ft.rexec:succ:Thu Nov 30 13:47:48:machb
usera:p-3:ft.uux:queued:Thu Nov 30 13:48:00:machc
usera:p-5:ft.uux:queued:Thu Nov 30 13:49:50:machd
usera:p-9:dt.uux:queued:Thu Nov 30 13:50:00:machd
```

This display takes the output from **remstat -p** and produces a list of colon-separated fields without headings. The first field identifies the user followed by fields for the primitive identifier, network service primitive module, status, time of the remote operation, and destination machine. Note that leading zeros are omitted for primitive identifiers.

Sample output from **remstat -c -j** *jobid*:

```
$ remstat -c -j p-5

                  Svc ID: 0000000005
                Admin ID: 0000000006
                 Prim ID: 0000000007
Standard output
---------------
xyzpkg installed on machb

Error output
------------
running command: echo redirecting output on ‘uname -n‘ >&1
        standard output is directed to the file
     (/var/spool/roi/users/uucp/remop/fred/stdout/o7)
running reply: uux fred!/usr/sadm/roi/nsi/uux/rcv.uux
     -r -n sfmad -j 7 -u uucp -x 0
     -o !/var/spool/roi/users/uucp/remop/fred/stdout/o7
     -e !/var/spool/roi/users/uucp/remop/fred/stderr/e7
/usr/bin/cp o7 /var/spool/roi/users/uucp/stdout/7
/usr/bin/cp e7 /var/spool/roi/users/uucp/stderr/7
```

This output shows the **stdout** and **stderr** from the remote command execution using the **p-5** primitive ID. **remstat -c -j jobid** will also display any error output from subsequent network service commands that ROI may run.

If there is more than one primitive ID associated with a service ID or an administrative ID, and the service or administrative ID is the argument to **remstat -c -j** *jobid*, the output for all the primitive jobs are given, one at a time. In this example, there was only one primitive associated with the service ID.

In this case, **remstat -c -j r-5, remstat -c -j a-6**, and **remstat -c -j p-7** would have given identical output.

SEE ALSO

remop(RA_LIB), remkill(RA_CMD)

FUTURE DIRECTIONS

Certain of the Remote Operations Interface library and commands have been moved to Level 2 for the following reasons. On going standards and industry direction has converged around an object-oriented approach to distributed, as opposed to remote, system administration. The ROI interfaces will become obsolete as standards and consensus mature within the workings of the Object Management Group, X/Open Systems Management, and IEEE P1003.7. It is expected that the ROI interfaces will be replaced by standard API's and command-line interfaces as part of the ongoing efforts in the area of Distributed Systems and Network Management.

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

NAME

remtab – specify the order in which the function **remop**() accesses network services

SYNOPSIS

remtab -a|**-A** *net_service*[**,***net_service***...] -s|-q**
remtab -d *net_service*[**,***net_service***...] -s|-q**

DESCRIPTION

System administrators use the **remtab** utility to specify the order in which **remop**() accesses network services for remote administrative jobs. Users can invoke **remtab** with no options for a display of available network services.

The options listed below are available only if you are an administrator.

–a *net_service*[**,***net_services***...]**
: Append one or more new network service specifications to an existing list of network services.

–A *net_service*[**,***net_services***...]**
: Add network services after deleting old specifications.

–d *net_service*[**,***net_services***...]**
: Delete network services.

–s
: Indicate that the network service entry is for synchronous operation only.

–q
: Indicate that the network service entry is for queued operation only.

With no arguments, **remtab** displays current network service selection information. Non-privileged users can not invoke **remtab** with arguments.

USAGE

System administrators can use **remtab** with all its options. Users can only invoke **remtab** without options.

SEE ALSO

remop(RA_LIB)

FUTURE DIRECTIONS

Certain of the Remote Operations Interface library and commands have been moved to Level 2 for the following reasons. On going standards and industry direction has converged around an object-oriented approach to distributed, as opposed to remote, system administration. The ROI interfaces will become obsolete as standards and consensus mature within the workings of the Object Management Group, X/Open Systems Management, and IEEE P1003.7. It is expected that the ROI interfaces will be replaced by standard API's and command-line interfaces as part of the ongoing efforts in the area of Distributed Systems and Network Management.

LEVEL

Level 2, September 30, 1993. In the future, the RA_ extension will be phased out in favor of distributed management functionality.

Permuted Index

uuencode, uudecode encode a	binary file, or decode its ASCII/	uuencode(AU_CMD) VOL 2
fread, fwrite	binary input/output	fread(BA_OS) VOL 1
bsearch	binary search on a sorted table	bsearch(BA_LIB) VOL 1
tfind, tdelete, twalk manage	binary search trees tsearch,	tsearch(BA_LIB) VOL 1
endpoint t_bind	bind an address to a transport	t_bind(BA_LIB) VOL 1
rpcb_unset library routines for RPC	bind service /rpcb_set,	rpcbind(RS_LIB) VOL 3
exception list for incremental/	bkexcept change or display an	bkexcept(AS_CMD) VOL 2
curs_bkgd: bkgdset, wbkgdset,	bkgd, wbkgd CURSES window/	curs_bkgd(TI_LIB) VOL 3
CURSES window/ curs_bkgd:	bkgdset, wbkgdset, bkgd, wbkgd	curs_bkgd(TI_LIB) VOL 3
backup operations	bkhistory report on completed	bkhistory(AS_CMD) VOL 2
operations to service media/	bkoper interact with backup	bkoper(AS_CMD) VOL 2
contents of a backup table	bkreg change or display the	bkreg(AS_CMD) VOL 2
backup operations	bkstatus display the status of	bkstatus(AS_CMD) VOL 2
aiocb Asynchronous I/O Control	Block	aiocb(MT_LIB) VOL 1
sum print checksum and	block count of a file	sum(BU_CMD) VOL 2
sigpending examine signals that are	blocked and pending	sigpending(BA_OS) VOL 1
barrier_destroy destroy a	blocking barrier	barrier_destroy(MT_LIB) VOL 1
barrier_init initialize a	blocking barrier	barrier_init(MT_LIB) VOL 1
barrier_wait wait at a	blocking barrier	barrier_wait(MT_LIB) VOL 1
df report number of free disk	blocks and i-nodes	df(BU_CMD) VOL 2
wvline create CURSES/ curs_border:	border, wborder, box, whline,	curs_border(TI_LIB) VOL 3
/box, whline, wvline create CURSES	borders, horizontal and vertical/	curs_border(TI_LIB) VOL 3
manipulation/ panel_top: top_panel,	bottom_panel PANELS deck	panel_top(TI_LIB) VOL 3
two wide character strings with	bound wcsncat concatenate	wcsncat(BA_LIB) VOL 1
two wide character strings with	bound wcsncmp compare	wcsncmp(BA_LIB) VOL 1
copy a wide character string with	bound wcsncpy	wcsncpy(BA_LIB) VOL 1
curs_border: border, wborder,	box, whline, wvline create CURSES/	curs_border(TI_LIB) VOL 3
waiting on a/ cond_broadcast	broadcast a wake up to all threads	cond_broadcast(MT_LIB) VOL 1
distauth authorize subscription and	broadcast of packages	distauth(RA_CMD) VOL 3
more, page	browse or page through a text file	more(BU_CMD) VOL 2
table	bsearch binary search on a sorted	bsearch(BA_LIB) VOL 1
auditbuf manipulate the audit	buffer	auditbuf(AT_LIB) VOL 3
write audit record to audit	buffer auditdmp	auditdmp(AT_LIB) VOL 3
set and get MENUS pattern match	buffer /menu_pattern	menu_pattern(TI_LIB) VOL 3
stdio: stdio.h standard	buffered input/output	stdio(BA_ENV) VOL 1
stdio standard	buffered input/output package	stdio(BA_LIB) VOL 1
setbuf, setvbuf assign	buffering to a stdio-stream	setbuf(BA_LIB) VOL 1
sync flush system	buffers	sync(AS_CMD) VOL 2
mknod	build special file	mknod(AS_CMD) VOL 2
wctob wide character to	byte conversion	wctob(BA_LIB) VOL 1
swab swap	bytes	swab(BA_LIB) VOL 1
cc	C compiler	cc(SD_CMD) VOL 3
cflow generate	C flowgraph	cflow(SD_CMD) VOL 3
lint a	C program checker	lint(SD_CMD) VOL 3
cxref generate	C program cross-reference	cxref(SD_CMD) VOL 3
	cal print calendar	cal(BU_CMD) VOL 2
cal print	calendar	cal(BU_CMD) VOL 2
	calendar reminder service	calendar(BU_CMD) VOL 2

FINAL COPY
June 15, 1995
File: PI.master
svid

FINAL COPY
June 15, 1995
File: PI.master
svid

Page: 703

FINAL COPY
June 15, 1995
File: PI.master
svid

Page: 705

FINAL COPY
June 15, 1995
File: PI.master
svid

FINAL COPY
June 15, 1995
File: PI.master
svid

FINAL COPY
June 15, 1995
File: PI.master
svid

FINAL COPY
June 15, 1995
File: PI.master
svid

FINAL COPY
June 15, 1995
File: PI.master
svid

Page: 739

FINAL COPY
June 15, 1995
File: PI.master
svid

Page: 743

FINAL COPY
June 15, 1995
File: PI.master
svid

Page: 750

FINAL COPY
June 15, 1995
File: PI.master
svid

FINAL COPY
June 15, 1995
File: PI.master
svid

FINAL COPY
June 15, 1995
File: PI.master
svid

FINAL COPY
June 15, 1995
File: PI.master
svid

Page: 773

FINAL COPY
June 15, 1995
File: PI.master
svid

Page: 777

FINAL COPY
June 15, 1995
File: PI.master
svid

FINAL COPY
June 15, 1995
File: PI.master
svid

Page: 786

FINAL COPY
June 15, 1995
File: PI.master
svid

	ucontext user context	ucontext(BA_ENV) VOL 1
ulimit: ulimit.h	ulimit commands	ulimit(BA_ENV) VOL 1
	ulimit get and set user limits	ulimit(BA_OS) VOL 1
	ulimit: ulimit.h ulimit commands	ulimit(BA_ENV) VOL 1
ulimit:	ulimit.h ulimit commands	ulimit(BA_ENV) VOL 1
mask	umask set and get file creation	umask(BA_OS) VOL 1
	umask set file-creation mode mask	umask(BU_CMD) VOL 2
systems and remote/ mount,	umount mount or unmount file	mount(AS_CMD) VOL 2
	umount unmount a file system	umount(BA_OS) VOL 1
system	uname get name of current operating	uname(uname(BA_OS)) VOL 1
	uname print name of current system	uname(BU_CMD) VOL 2
unshare make local resource	unavailable for sharing by remote/	unshare(RS_CMD) VOL 3
/zcat compress data for storage,	uncompress and display compressed/	compress(BU_CMD) VOL 2
storage, uncompress and/ compress,	uncompress, zcat compress data for	compress(BU_CMD) VOL 2
putwin, getwin,/ curs_util:	unctrl, keyname, filter, use_env,	curs_util(TI_LIB) VOL 3
unget	undo a previous get of an SCCS file	unget(SD_CMD) VOL 3
SCCS file	unget undo a previous get of an	unget(SD_CMD) VOL 3
input stdio-stream	ungetc push character back into	ungetc(BA_LIB) VOL 1
/getch, wgetch, mvgetch, mvwgetch,	ungetch get (or push back)/	curs_getch(TI_LIB) VOL 3
into input stream	ungetwc push wchar_t character back	ungetwc(BA_LIB) VOL 1
/wgetwch, mvgetwch, mvwgetwch,	ungetwch get (or push back) wchar_t/	curs_getwch(TI_LIB) VOL 3
/srand48, seed48, lcong48 generate	uniformly distributed pseudo-random/	drand48(BA_LIB) VOL 1
file	uniq report repeated lines in a	uniq(BU_CMD) VOL 2
mktemp make a	unique filename	mktemp(BA_LIB) VOL 1
roijobids get	unique remote job identifiers	roijobids(RA_LIB) VOL 3
connld line discipline for	unique stream connections	connld(BA_DEV) VOL 1
constants and structures	unistd: unistd.h standard symbolic	unistd(BA_ENV) VOL 1
constants and structures unistd:	unistd.h standard symbolic	unistd(BA_ENV) VOL 1
t_rcvudata receive a data	unit	t_rcvudata(BA_LIB) VOL 1
t_sndudata send a data	unit	t_sndudata(BA_LIB) VOL 1
t_rcvuderr receive a	unit data error indication	t_rcvuderr(BA_LIB) VOL 1
number mapper rpcbind	universal addresses to RPC program	rpcbind(RS_CMD) VOL 3
system calls link,	unlink exercise link and unlink	link(AS_CMD) VOL 2
	unlink remove directory entry	unlink(BA_OS) VOL 1
link, unlink exercise link and	unlink system calls	link(AS_CMD) VOL 2
demand KE_OS) moduload	unload a loadable kernel module on	moduload(KE_OS) VOL 1
mutex_unlock	unlock a mutex	mutex_unlock(MT_LIB) VOL 1
master/slave pair unlockpt	unlock a pseudo-terminal	unlockpt(BA_LIB) VOL 1
rmutex_unlock	unlock a recursive mutex	rmutex_unlock(MT_LIB) VOL 1
mlockall, munlockall lock or	unlock address space	mlockall(RT_OS) VOL 3
mlock, munlock lock (or	unlock) pages in memory	mlock(RT_OS) VOL 3

FINAL COPY
June 15, 1995
File: PI.master
svid

Page: 804

FINAL COPY
June 15, 1995
File: PI.master
svid

Page: 810