



Cybercrimes.it

Computer Forensics & Crimine Informatico

Dario Scalea

DD

Analisi funzionale, forense ed algoritmica

## DD Tavola sinottica

Fileutils 4.0.36 (nist-tested)

**dd** **-help** **-version** **if=file** **of=file** **ibs=n** **obs=n** **bs=n** **cbs=n** **skip=n** **seek=n** **count=n** **conv=value,value**

COM	OPERAZIONE		
<b>dd</b>	Chiamata dell'algoritmo dd eseguente conversione e copia di file		
	OPERANDO	OPERAZIONE	
	<b>--help</b>	Visualizza la sinossi didascalica del comando	
	<b>--version</b>	Visualizza versione ed autori del file	
	<b>if=file</b>	Specifica il percorso del file d'ingresso	
	<b>of=file</b>	Specifica il percorso del file d'uscita	
	<b>ibs=n</b>	Specifica in bytes la dimensione del blocco in ingresso (valore predefinito 512)	
	<b>obs=n</b>	Specifica in bytes la dimensione del blocco in uscita (valore predefinito 512)	
	<b>bs=n</b>	Specifica in bytes un unico valore sia per la dimensione del blocco in ingresso che per quello in uscita.	
	<b>cbs=n</b>	Specifica in bytes la dimensione del blocco di conversione per il parametro block e unblock dell'operando conv(valore predefinito 0).	
	<b>skip=n</b>	Specifica il numero di blocchi in ingresso da saltare prima di cominciare a copiare	
	<b>seek=n</b>	Specifica il numero di blocchi del file d'uscita da saltare prima di copiare.	
	<b>count=n</b>	Specifica il numero di blocchi in ingresso da copiare.	
	<b>Conv=</b> value, value	Effettua le operazioni specificate dai parametri dell'operando	
		PARAMETRO VALUE	OPERAZIONE
		ascii	Convertire ebcdic in ascii
		ebcdic	Convertire ascii in ebcdic
		ibm	Convertire ascii in ibm ebcdic
		block	Considerare l'ingresso come una sequenza di records di lunghezza variabile terminati da newline o eof indipendentemente dai limiti del blocco in ingresso
		unblock	Convertire i records di lunghezza fissa in records di lunghezza variabile
	lcase	Rilevare i caratteri maiuscoli tramite la parola chiave LC_CTYPE <i>tolower</i> come minuscoli. I caratteri senza mappatura specifica non sono convertiti.	
	ucase	Rilevare i caratteri minuscoli tramite la parola chiave LC_CTYPE <i>toupper</i> come maiuscoli. I caratteri senza mappatura specifica non sono convertiti.	
	swab	Scambiare ogni coppia di bytes in ingresso	
	noerror	Non fermare il processo quando si verifica un errore d'ingresso.	
	notrunc	Non troncato il file d'uscita.	
	sync	Aggiungere n bytes nulli ad ogni blocco d'ingresso fino alla dimensione del buffer specificata nell'operando ibs=.	

## PROTASI

L'algoritmo copiatore dd(data definition) nacque agli inizi degli anni '70. Il port di maggiore diffusione è senza dubbio quello che vide la luce nel 1975 quando il comando dd venne integrato nel sistema operativo Unix V6. Scritto ed implementato in più fasi da Paul Rubin, David MacKenzie e Stuart Kemp, venne progettato per aggiungere funzionalità di conversione alla copia di file. In verità, il comando dd non deve la sua fama alle proprie capacità di conversione ma alla possibilità di definire le quantità di bit per unità di I/O tramite gli operandi ibs, obs e bs. È proprio questa possibilità che ha fatto diventare dd lo standard de facto per la duplicazione delle prove binarie in ambito forense. Infatti, il metodo di duplicazione byte a byte è quello che consente di ottenere i risultati migliori nel tentativo di creare copie identiche all'originale e, conseguentemente, di rispettare l'ineludibile principio d'inalterabilità della prova. In controtendenza col proliferare di articoli, guide e quant'altro che si ispirano alla filosofia del "Tutta l'informatica in 24 ore", continueremo ad affrontare un argomento per volta con una discreta profondità analitica, cercando di limitare il più possibile le corbellerie che tanto vanno di moda.

## DD:IL PERCHÈ DI UNA SCELTA

Pur esistendo una serie di varianti all'algoritmo base dd (dcfldd, ddrescue, dd\_rescue, sdd, etc.), perchè scegliere dd per la funzione di copia? Gli studi di complessità computazionali ci dicono che, a condizioni date, la complessità asintotica  $O(f(n))$  di un algoritmo è proporzionale al margine d'errore dell'algoritmo stesso. Considerando che in computer forensics il margine d'errore deve essere assolutamente minimizzato, la scelta deve ricadere sull'algoritmo con la minore complessità asintotica. Ricordando che, pur esistendo funzioni non calcolabili dalla macchina di Turing, secondo la tesi di Church non esistono formalismi né macchine concrete in grado di calcolarle, l'algoritmo con la minore complessità asintotica è proprio quello generatore ovvero dd.

## IL PRAGMA OPERATIVO

Il comando dd, in ambiente unix, è stato inserito nel software package fileutils che, a sua volta, è diventato parte integrante di coreutils. La versione scelta è la fileutils 4.0.36 perchè nist-tested. Proviamo a comprendere il funzionamento del comando tramite alcuni esempi. Creare sul desktop il file x.txt contenente la stringa "ciao". Da terminale digitiamo il seguente comando per creare una copia y.txt di x.txt:

```
dd if=/root/Desktop/x.txt of=/root/Desktop/y.txt
```

Secondo le direttive POSIX, nell'stderr verrà scritto:

entrati 0+1 record

usciti 0+1 record

L'stderr indica in numero di record I/O secondo la notazione in blocchi

totali+parziali.Siccome non abbiamo indicato la dimensione dei blocchi I/O,l'algoritmo ha assunto la dimensione standard (512 byte).Dal momento che la dimensione del file x.txt è di soli 5 byte,vengono indicati 0 blocchi totali + 1 blocco parziale.Infatti,se aggiungiamo la dimensione del blocco,digitando:

```
dd if=/root/Desktop/x.txt of=/root/Desktop/y.txt bs=1
```

L'stderr sarà:

```
entrati 5+0 record
```

```
usciti 5+0 record
```

Avendo specificato la dimensione di blocco pari ad 1 byte,ed essendo la dimensione del file pari a 5 byte,l'std indica correttamente 5 blocchi totali+ 0 parziali sia in ingresso che in uscita.Infatti, l'operando bs specifica un unico valore di blocksize sia per l'input che per l'output,per per valori I/O differenti bisogna utilizzare gli operandi ibs,obs.Proviamo,digitando:

```
dd if=/root/Desktop/x.txt of=/root/Desktop/y.txt ibs=1 obs=4 skip=2 seek=1
```

Questo comando salta due blocchi di 1 byte in ingresso(skip=2) corrispondenti agli elementi c ed i della stringa ciao e un blocco di 4 byte del file d'uscita(seek=1) corrispondenti agli elementi c,i,a ed o prima di cominciare a copiare.Quindi ci dobbiamo aspettare che nel file y.txt ci sarà la stringa "ciaoao".Ed è proprio così.L'stderr sarà:

```
entrati 3+0 record
```

```
usciti 0+1 record
```

Per ritornare alla stringa ciao,basterà copiare il carattere terminatore nella quinta posizione con il comando:

```
dd if=/root/Desktop/x.txt of=/root/Desktop/y.txt ibs=1 obs=4 skip=4 seek=1 count=1
```

Vediamo l'utilizzo dell'ultimo operando,digitando:

```
dd if=/root/Desktop/x.txt of=/root/Desktop/y.txt ibs=10 obs=2 cbs=1 conv=unblock,ucase,sync
```

Il risultato è un file y.txt dalla dimensione di 15 byte contenente la stringa:

```
C
```

```
I
```

```
A
```

```
O
```

L'stderr è:

```
entrati 0+1 record
```

```
usciti 7+1 record
```

L'algoritmo dd ha bufferizzato il file x.txt aggiungendo tanti byte nulli fino alla

dimensione `ibs(sync)`, convertendo la lunghezza dei record da fissa a variabile(`unblock`) specificandone la dimensione in un 1 byte(`cbs=1`) con relativi caratteri terminatori, cambiando i caratteri minuscoli in maiuscoli (`ucase`).

Questi esempi ci danno la prova delle potenzialità e della versatilità del comando `dd`.

## LA PROSPETTIVA FORENSE

Dal momento che sono sufficienti pochi byte per conservare informazione determinanti (ad es. una password), la duplicazione di memorie di massa (ad es. disco rigido) in ambito forense deve essere effettuato con la minore `blocksize(bs)` possibile. Questa scelta serve a minimizzare il rischio che la mancata o errata trascrizione di un settore durante la duplicazione impedisca, in sede di analisi, il rilevamento di dati probatori. Pensate che, durante una serie di test sull'algoritmo `dd`, nel 25,9% dei casi, un settore non corrisponde (fonte nist). Un'altra scelta importante riguarda i parametri `noerror` e `sync` dell'operando `conv`. L'utilizzo del parametro `noerror` consente al processo, nell'occorrenza di un errore, di riportare l'errore nell'`stderr` ma di non arrestarsi. Questo comporta che gli indirizzi e la dimensione dell'immagine del disco rigido non corrisponderanno. Per ovviare a questo inconveniente si usano in contemporanea i parametri `noerror` e `sync`. Infatti, il `sync` completerà con byte nulli lo spazio rimanente fino alla `blocksize`, consentendo di preservare gli indirizzi. La dimensione, invece, risulterà sempre un multiplo della `blocksize`. È erronea la convinzione secondo cui anche il parametro `notrunc` sia indispensabile, poiché, non utilizzando il `seek`, l'algoritmo `dd` tronca l'`of` prima di iniziare a copiare. Pertanto il comando canonico nella clonazione di un disco rigido potrebbe essere:

```
dd if=/dev/hda of=/dev/hdb bs=1 conv=noerror,sync
```

Una doverosa precisazione

Conoscere il comando `dd` non è sufficiente a garantire una corretta acquisizione della prova informatica. È necessario effettuare una serie di verifiche preventive sulle chiamate di interrupt della bios, come la famigerata `int 13h`, (in ambiente windows, linux ha un accesso diretto) e sulla geometria del disco rigido anche tramite i comandi `ata` perché esistono svariate tecniche di occultamento geometrico dei dati. Inoltre, in seguito alla duplicazione, è opportuno utilizzare algoritmi di compendio (ad es. `md5`) per confrontare l'identità dei risultati. Poiché ognuno di questi argomenti, per vastità e complessità, richiede una trattazione separata e dal momento che questo whitepaper intende essere solo una guida all'algoritmo `dd` e non all'intero processo di acquisizione della prova informatica, non entrerà nei dettagli.

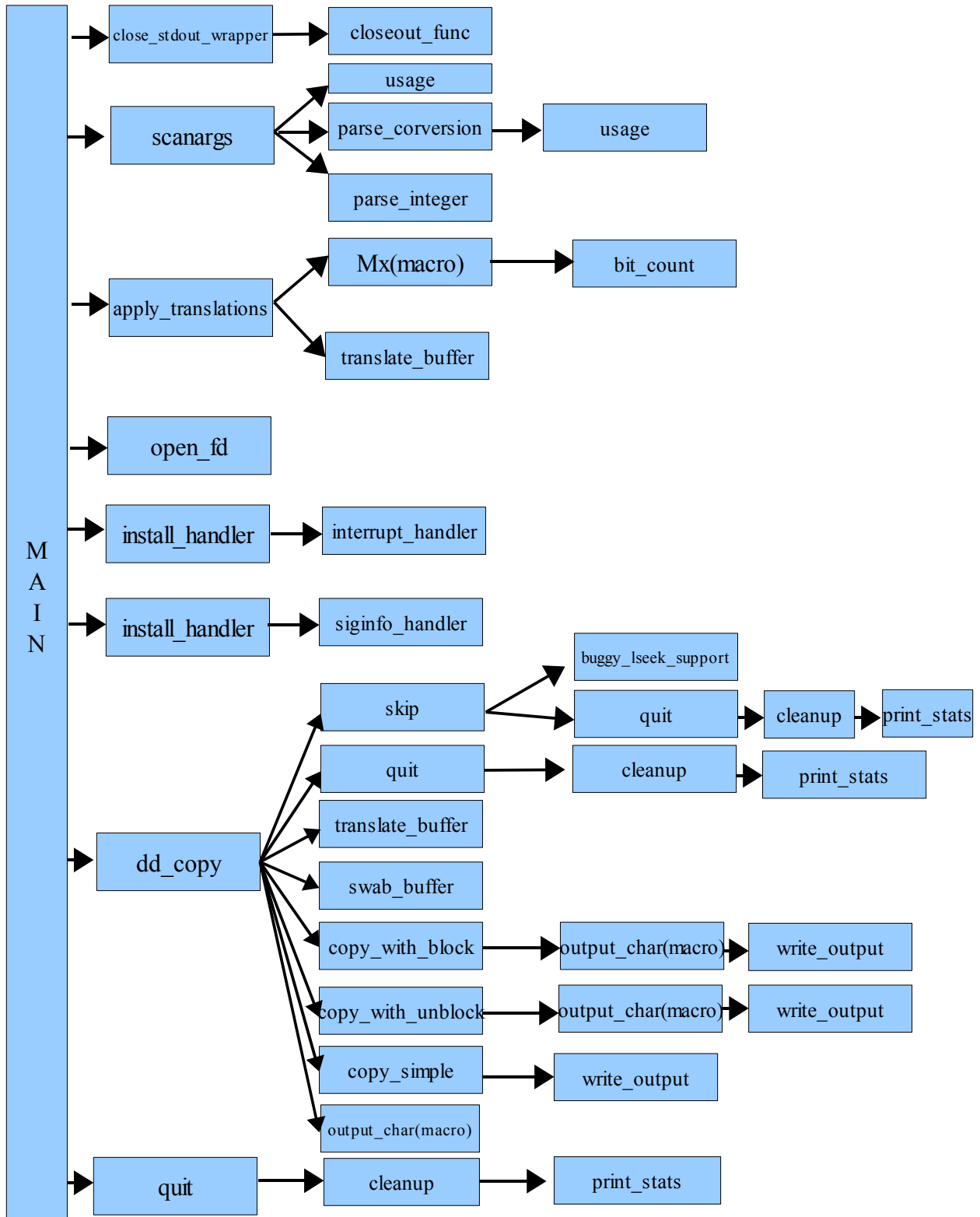
## IL CODICE

Il codice dell'algoritmo `dd` è stato scritto in linguaggio C. Consta di 15 principali direttive d'inclusione, 23 di definizione (5 macro) e 6 condizionali al preprocessore, 6 strutture, 26 funzioni (main compreso). L'uso profuso dell'indicatore di classe di memoria `static` impedisce che le variabili e le funzioni a cui è associato possano essere usati da funzioni non definite nel file stesso. Diffuso è anche l'utilizzo della parola riservata `unsigned` che impedisce alla variabile di assumere valori negativi. Risparmiandovi i dettagli della progettazione, vi rappresento la tabella di finalità funzionale, il diagramma gerarchico per chiamata ed una valutazione per principi programmatori.

Tabella di finalità funzionale

Funzione	Fine
usage	Visualizzazione delle informazioni di supporto.
translate_charset	Conversione del charset tramite ciclo di scorrimento di array.
bit_count	Conteggio di bit tramite ciclo con assegnamento ed and su bit.
print_stats	Visualizzazione delle informazioni di report.
cleanup	Printstats con aggiunta di eventuali informazioni di chiusura I/O.
quit	Cleanup con chiamata di terminazione e restituzione codice d'errore.
interrupt_handler	Gestione evento per cleanup e terminazione programma.
siginfo_handler	Evento per printstats.
install_handler	Gestione condizionale degli eventi.
open_fd	Apertura di un file in corrispondenza di uno specifico descrittore.
write_output	Scrittura del buffer d'uscita.
parse_conversion	Interpretazione dei parametri dell'operando conv.
parse_integer	Analisi stringa come valore decimale non negativo.
scanargs	Analisi degli operandi e relativi parametri del comando
apply_translations	Applicazione delle conversioni
translate_buffer	Applicazione della conversione del charset al buffer d'ingresso
swab_buffer	Scambio di ogni coppia di byte contenuta nel buffer
buggy_lseek_support	Correzione bug su scorrimento del file
skip	Salto di n records di file
copy_simple	Copia priva di conversione
copy_with_block	Copia con il parametro block dell'operando conv attivo
copy_with_unblock	Copia con il parametro unblock dell'operando conv attivo
dd_copy	Ciclo canonico di copia.
close_stdout_wrapper/coseout_func	Chiusura riflessiva dello standardout.
main	Funzione principale dotata di privilegio d'esecuzione

## Topologia gerarchica delle funzioni primarie



VALUTAZIONE DEL CODICE

Efficienza	8
Leggibilità	7
Portabilità	8
Progettazione	8
Ottimizzazione	6
Testing	7
Completezza	6

Dario Scalea

## Appendice:dd,il codice integrale

```
.* dd -- convert a file while copying it.
Copyright (C) 85, 90, 91, 1995-2000 Free Software Foundation, Inc.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software Foundation,
Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. */

/* Written by Paul Rubin, David MacKenzie, and Stuart Kemp. */

#include <config.h>
#include <stdio.h>

#define SWAB_ALIGN_OFFSET 2

#if HAVE_INTTYPES_H
# include <inttypes.h>
#endif
#include <sys/types.h>
#include <signal.h>
#include <getopt.h>

#include "system.h"
#include "closeout.h"
#include "error.h"
#include "getpagesize.h"
#include "human.h"
#include "long-options.h"
#include "quote.h"
#include "safe-read.h"
#include "xstrtol.h"

/* The official name of this program (e.g., no `g' prefix). */
#define PROGRAM_NAME "dd"

#define AUTHORS "Paul Rubin, David MacKenzie, and Stuart Kemp"

#ifndef SIGINFO
# define SIGINFO SIGUSR1
#endif

#ifndef S_TYPEISSHM
# define S_TYPEISSHM(Mode) 0
#endif

#define ROUND_UP_OFFSET(X, M) ((M) - 1 - (((X) + (M) - 1) % (M)))
#define PTR_ALIGN(Ptr, M) ((Ptr) \
                          + ROUND_UP_OFFSET ((char *) (Ptr) - (char *) 0, (M)))

#define max(a, b) ((a) > (b) ? (a) : (b))
#define output_char(c) \
do { \
  obuf[oc++] = (c); if (oc >= output_blocksize) write_output (); \
} while (0)

/* Default input and output blocksize. */
#define DEFAULT_BLOCKSIZE 512

/* Conversions bit masks. */
#define C_ASCII 01
#define C_EBCDIC 02
#define C_IBM 04
#define C_BLOCK 010
#define C_UNBLOCK 020
#define C_LCASE 040
#define C_UCASE 0100
#define C_SWAB 0200
#define C_NOERROR 0400
#define C_NOTRUNC 01000
#define C_SYNC 02000
/* Use separate input and output buffers, and combine partial input blocks. */
#define C_TWOBUFFS 04000

int full_write ();

/* The name this program was run with. */
char *program_name;

/* The name of the input file, or NULL for the standard input. */
static char *input_file = NULL;

/* The name of the output file, or NULL for the standard output. */
static char *output_file = NULL;

/* The number of bytes in which atomic reads are done. */
static size_t input_blocksize = 0;

/* The number of bytes in which atomic writes are done. */
static size_t output_blocksize = 0;
```

```

/* Conversion buffer size, in bytes. 0 prevents conversions. */
static size_t conversion_blocksize = 0;

/* Skip this many records of `input_blocksize' bytes before input. */
static uintmax_t skip_records = 0;

/* Skip this many records of `output_blocksize' bytes before output. */
static uintmax_t seek_records = 0;

/* Copy only this many records. The default is effectively infinity. */
static uintmax_t max_records = (uintmax_t) -1;

/* Bit vector of conversions to apply. */
static int conversions_mask = 0;

/* If nonzero, filter characters through the translation table. */
static int translation_needed = 0;

/* Number of partial blocks written. */
static uintmax_t w_partial = 0;

/* Number of full blocks written. */
static uintmax_t w_full = 0;

/* Number of partial blocks read. */
static uintmax_t r_partial = 0;

/* Number of full blocks read. */
static uintmax_t r_full = 0;

/* Records truncated by conv=block. */
static uintmax_t r_truncate = 0;

/* Output representation of newline and space characters.
   They change if we're converting to EBCDIC. */
static unsigned char newline_character = '\n';
static unsigned char space_character = ' ';

/* Output buffer. */
static unsigned char *obuf;

/* Current index into `obuf'. */
static size_t oc = 0;

/* Index into current line, for `conv=block' and `conv=unblock'. */
static size_t col = 0;

struct conversion
{
    char *convname;
    int conversion;
};

static struct conversion conversions[] =
{
    {"ascii", C_ASCII | C_TWOBUFFS}, /* EBCDIC to ASCII. */
    {"ebcdic", C_EBCDIC | C_TWOBUFFS}, /* ASCII to EBCDIC. */
    {"ibm", C_IBM | C_TWOBUFFS}, /* Slightly different ASCII to EBCDIC. */
    {"block", C_BLOCK | C_TWOBUFFS}, /* Variable to fixed length records. */
    {"unblock", C_UNBLOCK | C_TWOBUFFS}, /* Fixed to variable length records. */
    {"lcase", C_LCASE | C_TWOBUFFS}, /* Translate upper to lower case. */
    {"ucase", C_UCASE | C_TWOBUFFS}, /* Translate lower to upper case. */
    {"swab", C_SWAB | C_TWOBUFFS}, /* Swap bytes of input. */
    {"noerror", C_NOERROR}, /* Ignore i/o errors. */
    {"notrunc", C_NOTRUNC}, /* Do not truncate output file. */
    {"sync", C_SYNC}, /* Pad input records to ibs with NULs. */
    {NULL, 0}
};

/* Translation table formed by applying successive transformations. */
static unsigned char trans_table[256];

static unsigned char const ascii_to_ebcdic[] =
{
    0, 01, 02, 03, 067, 055, 056, 057,
    026, 05, 045, 013, 014, 015, 016, 017,
    020, 021, 022, 023, 074, 075, 062, 046,
    030, 031, 077, 047, 034, 035, 036, 037,
    0100, 0117, 0177, 0173, 0133, 0154, 0120, 0175,
    0115, 0135, 0134, 0116, 0153, 0140, 0113, 0141,
    0360, 0361, 0362, 0363, 0364, 0365, 0366, 0367,
    0370, 0371, 0172, 0136, 0114, 0176, 0156, 0157,
    0174, 0301, 0302, 0303, 0304, 0305, 0306, 0307,
    0310, 0311, 0321, 0322, 0323, 0324, 0325, 0326,
    0327, 0330, 0331, 0342, 0343, 0344, 0345, 0346,
    0347, 0350, 0351, 0112, 0340, 0132, 0137, 0155,
    0171, 0201, 0202, 0203, 0204, 0205, 0206, 0207,
    0210, 0211, 0221, 0222, 0223, 0224, 0225, 0226,
    0227, 0230, 0231, 0242, 0243, 0244, 0245, 0246,
    0247, 0250, 0251, 0300, 0152, 0320, 0241, 07,
    040, 041, 042, 043, 044, 025, 06, 027,
    050, 051, 052, 053, 054, 011, 012, 033,
    060, 061, 032, 063, 064, 065, 066, 010,
    070, 071, 072, 073, 04, 024, 076, 0341,
    0101, 0102, 0103, 0104, 0105, 0106, 0107, 0110,
    0111, 0121, 0122, 0123, 0124, 0125, 0126, 0127,
    0130, 0131, 0142, 0143, 0144, 0145, 0146, 0147,
    0150, 0151, 0160, 0161, 0162, 0163, 0164, 0165,
    0166, 0167, 0170, 0200, 0212, 0213, 0214, 0215,
    0216, 0217, 0220, 0232, 0233, 0234, 0235, 0236,

```

```

0237, 0240, 0252, 0253, 0254, 0255, 0256, 0257,
0260, 0261, 0262, 0263, 0264, 0265, 0266, 0267,
0270, 0271, 0272, 0273, 0274, 0275, 0276, 0277,
0312, 0313, 0314, 0315, 0316, 0317, 0332, 0333,
0334, 0335, 0336, 0337, 0352, 0353, 0354, 0355,
0356, 0357, 0372, 0373, 0374, 0375, 0376, 0377
};

```

```

static unsigned char const ascii_to_ibm[] =
{
0, 01, 02, 03, 067, 055, 056, 057,
026, 05, 045, 013, 014, 015, 016, 017,
020, 021, 022, 023, 074, 075, 062, 046,
030, 031, 077, 047, 034, 035, 036, 037,
0100, 0132, 0177, 0173, 0133, 0154, 0120, 0175,
0115, 0135, 0134, 0116, 0153, 0140, 0113, 0141,
0360, 0361, 0362, 0363, 0364, 0365, 0366, 0367,
0370, 0371, 0172, 0136, 0114, 0176, 0156, 0157,
0174, 0301, 0302, 0303, 0304, 0305, 0306, 0307,
0310, 0311, 0321, 0322, 0323, 0324, 0325, 0326,
0327, 0330, 0331, 0342, 0343, 0344, 0345, 0346,
0347, 0350, 0351, 0255, 0340, 0275, 0137, 0155,
0171, 0201, 0202, 0203, 0204, 0205, 0206, 0207,
0210, 0211, 0221, 0222, 0223, 0224, 0225, 0226,
0227, 0230, 0231, 0242, 0243, 0244, 0245, 0246,
0247, 0250, 0251, 0300, 0117, 0320, 0241, 07,
040, 041, 042, 043, 044, 025, 06, 027,
050, 051, 052, 053, 054, 011, 012, 033,
060, 061, 032, 063, 064, 065, 066, 010,
070, 071, 072, 073, 04, 024, 076, 0341,
0101, 0102, 0103, 0104, 0105, 0106, 0107, 0110,
0111, 0121, 0122, 0123, 0124, 0125, 0126, 0127,
0130, 0131, 0142, 0143, 0144, 0145, 0146, 0147,
0150, 0151, 0160, 0161, 0162, 0163, 0164, 0165,
0166, 0167, 0170, 0200, 0212, 0213, 0214, 0215,
0216, 0217, 0220, 0232, 0233, 0234, 0235, 0236,
0237, 0240, 0252, 0253, 0254, 0255, 0256, 0257,
0260, 0261, 0262, 0263, 0264, 0265, 0266, 0267,
0270, 0271, 0272, 0273, 0274, 0275, 0276, 0277,
0312, 0313, 0314, 0315, 0316, 0317, 0332, 0333,
0334, 0335, 0336, 0337, 0352, 0353, 0354, 0355,
0356, 0357, 0372, 0373, 0374, 0375, 0376, 0377
};

```

```

static unsigned char const ebcidic_to_ascii[] =
{
0, 01, 02, 03, 0234, 011, 0206, 0177,
0227, 0215, 0216, 013, 014, 015, 016, 017,
020, 021, 022, 023, 0235, 0205, 010, 0207,
030, 031, 0222, 0217, 034, 035, 036, 037,
0200, 0201, 0202, 0203, 0204, 012, 027, 033,
0210, 0211, 0212, 0213, 0214, 05, 06, 07,
0220, 0221, 026, 0223, 0224, 0225, 0226, 04,
0230, 0231, 0232, 0233, 024, 025, 0236, 032,
040, 0240, 0241, 0242, 0243, 0244, 0245, 0246,
0247, 0250, 0133, 056, 074, 050, 053, 041,
046, 0251, 0252, 0253, 0254, 0255, 0256, 0257,
0260, 0261, 0135, 044, 052, 051, 073, 0136,
055, 057, 0262, 0263, 0264, 0265, 0266, 0267,
0270, 0271, 0174, 054, 045, 0137, 076, 077,
0272, 0273, 0274, 0275, 0276, 0277, 0300, 0301,
0302, 0140, 072, 043, 0100, 047, 075, 042,
0303, 0141, 0142, 0143, 0144, 0145, 0146, 0147,
0150, 0151, 0304, 0305, 0306, 0307, 0310, 0311,
0312, 0152, 0153, 0154, 0155, 0156, 0157, 0160,
0161, 0162, 0313, 0314, 0315, 0316, 0317, 0320,
0321, 0176, 0163, 0164, 0165, 0166, 0167, 0170,
0171, 0172, 0322, 0323, 0324, 0325, 0326, 0327,
0330, 0331, 0332, 0333, 0334, 0335, 0336, 0337,
0340, 0341, 0342, 0343, 0344, 0345, 0346, 0347,
0173, 0101, 0102, 0103, 0104, 0105, 0106, 0107,
0110, 0111, 0350, 0351, 0352, 0353, 0354, 0355,
0175, 0112, 0113, 0114, 0115, 0116, 0117, 0120,
0121, 0122, 0356, 0357, 0360, 0361, 0362, 0363,
0134, 0237, 0123, 0124, 0125, 0126, 0127, 0130,
0131, 0132, 0364, 0365, 0366, 0367, 0370, 0371,
060, 061, 062, 063, 064, 065, 066, 067,
070, 071, 0372, 0373, 0374, 0375, 0376, 0377
};

```

```

void
usage (int status)
{
if (status != 0)
    fprintf(stderr, _("\"Try '%s --help' for more information.\n\""),
            program_name);
else
    {
    printf (_("Usage: %s [OPTION]...\n"), program_name);
    printf (_("\n
Copy a file, converting and formatting according to the options.\n\n
\n
bs=BYTES      force ibs=BYTES and obs=BYTES\n
cbs=BYTES     convert BYTES bytes at a time\n
conv=KEYWORDS convert the file as per the comma separated keyword list\n
count=BLOCKS  copy only BLOCKS input blocks\n
ibs=BYTES     read BYTES bytes at a time\n
if=FILE       read from FILE instead of stdin\n
obs=BYTES     write BYTES bytes at a time\n
of=FILE       write to FILE instead of stdout\n
seek=BLOCKS   skip BLOCKS obs-sized blocks at start of output\n\n

```

```

skip=BLOCKS skip BLOCKS ibs-sized blocks at start of input\n
--help display this help and exit\n
--version output version information and exit\n

```

```

\n
BLOCKS and BYTES may be followed by the following multiplicative suffixes:\n
xM M, c 1, w 2, b 512, kD 1000, k 1024, MD 1,000,000, M 1,048,576,\n
GD 1,000,000,000, G 1,073,741,824, and so on for T, P, E, Z, Y.\n
Each KEYWORD may be:\n

```

```

\n
ascii from EBCDIC to ASCII\n
ebcdic from ASCII to EBCDIC\n
ibm from ASCII to alternated EBCDIC\n
block pad newline-terminated records with spaces to cbs-size\n
unblock replace trailing spaces in cbs-size records with newline\n
lcase change upper case to lower case\n
notrunc do not truncate the output file\n
ucase change lower case to upper case\n
swab swap every pair of input bytes\n
noerror continue after read errors\n
sync pad every input block with NULs to ibs-size\n

```

```

));
puts (_("\nReport bugs to <bug-fileutils@gnu.org>."));
}
exit (status);
}

```

```

static void
translate_charset (const unsigned char *new_trans)
{
unsigned int i;

for (i = 0; i < 256; i++)
trans_table[i] = new_trans[trans_table[i]];
translation_needed = 1;
}

```

```

/* Return the number of 1 bits in 'i'. */

```

```

static int
bit_count (register unsigned int i)
{
register int set_bits;

for (set_bits = 0; i != 0; set_bits++)
i &= i - 1;
return set_bits;
}

```

```

static void
print_stats (void)
{
char buf[2][LONGEST_HUMAN_READABLE + 1];
fprintf (stderr, _("%s+%s records in\n"),
human_readable (r_full, buf[0], 1, 1),
human_readable (r_partial, buf[1], 1, 1));
fprintf (stderr, _("%s+%s records out\n"),
human_readable (w_full, buf[0], 1, 1),
human_readable (w_partial, buf[1], 1, 1));
if (r_truncate > 0)
{
fprintf (stderr, "%s %s\n",
human_readable (r_truncate, buf[0], 1, 1),
(r_truncate == 1
? _("truncated record")
: _("truncated records")));
}
}

```

```

static void
cleanup (void)
{
print_stats ();
if (close (STDIN_FILENO) < 0)
error (1, errno, _("closing input file %s"), quote (input_file));
if (close (STDOUT_FILENO) < 0)
error (1, errno, _("closing output file %s"), quote (output_file));
}

```

```

static inline void
quit (int code)
{
cleanup ();
exit (code);
}

```

```

static RETSIGTYPE
interrupt_handler (int sig)
{
#ifdef SA_NOCLDSTOP
struct sigaction sigact;

sigact.sa_handler = SIG_DFL;
sigemptyset (&sigact.sa_mask);
sigact.sa_flags = 0;
sigaction (sig, &sigact, NULL);
#else
signal (sig, SIG_DFL);
#endif
cleanup ();
kill (getpid (), sig);
}

```

```

static RETSIGTYPE
siginfo_handler (int sig ATTRIBUTE_UNUSED)
{
    print_stats ();
}

/* Encapsulate portability mess of establishing signal handlers. */

static void
install_handler (int sig_num, RETSIGTYPE (*sig_handler) (int sig))
{
#ifdef SA_NOCLDSTOP
    struct sigaction sigact;
    sigaction (sig_num, NULL, &sigact);
    if (sigact.sa_handler != SIG_IGN)
    {
        sigact.sa_handler = sig_handler;
        sigemptyset (&sigact.sa_mask);
        sigact.sa_flags = 0;
        sigaction (sig_num, &sigact, NULL);
    }
#else
    if (signal (sig_num, SIG_IGN) != SIG_IGN)
        signal (sig_num, sig_handler);
#endif
}

/* Open a file to a particular file descriptor. This is like standard
`open`, except it always returns DESIRED_FD if successful. */
static int
open_fd (int desired_fd, char const *filename, int options, mode_t mode)
{
    int fd;
    close (desired_fd);
    fd = open (filename, options, mode);
    if (fd < 0)
        return -1;

    if (fd != desired_fd)
    {
        if (dup2 (fd, desired_fd) != desired_fd)
            desired_fd = -1;
        if (close (fd) != 0)
            return -1;
    }

    return desired_fd;
}

/* Write, then empty, the output buffer `obuf`. */

static void
write_output (void)
{
    int nwritten = full_write (STDOUT_FILENO, obuf, output_blocksize);
    if (nwritten != output_blocksize)
    {
        error (0, errno, ("writing to %s"), quote (output_file));
        if (nwritten > 0)
            w_partial++;
        quit (1);
    }
    else
        w_full++;
    oc = 0;
}

/* Interpret one "conv=..." option.
As a by product, this function replaces each `,' in STR with a NUL byte. */

static void
parse_conversion (char *str)
{
    char *new;
    unsigned int i;

    do
    {
        new = strchr (str, ',');
        if (new != NULL)
            *new++ = '\0';
        for (i = 0; conversions[i].convname != NULL; i++)
            if (STREQ (conversions[i].convname, str))
            {
                conversions_mask |= conversions[i].conversion;
                break;
            }
        if (conversions[i].convname == NULL)
            error (0, 0, ("invalid conversion: %s"), quote (str));
            usage (1);
        }
        str = new;
    } while (new != NULL);
}

/* Return the value of STR, interpreted as a non-negative decimal integer,
optionally multiplied by various values.
Assign nonzero to *INVALID if STR does not represent a number in
this format. */

```

```

static uintmax_t
parse_integer (const char *str, int *invalid)
{
    uintmax_t n;
    char *suffix;
    enum strtol_error e = strtoumax (str, &suffix, 10, &n, "bcEGkMPTwYZ0");

    if (e == LONGINT_INVALID_SUFFIX_CHAR && *suffix == 'x')
    {
        uintmax_t multiplier = parse_integer (suffix + 1, invalid);

        if (multiplier != 0 && n * multiplier / multiplier != n)
        {
            *invalid = 1;
            return 0;
        }

        n *= multiplier;
    }
    else if (e != LONGINT_OK)
    {
        *invalid = 1;
        return 0;
    }

    return n;
}

static void
scanargs (int argc, char **argv)
{
    int i;

    --argc;
    ++argv;

    for (i = optind; i < argc; i++)
    {
        char *name, *val;

        name = argv[i];
        val = strchr (name, '=');
        if (val == NULL)
        {
            error (0, 0, _("unrecognized option %s"), quote (name));
            usage (1);
        }
        *val++ = '\0';

        if (STREQ (name, "if"))
            input_file = val;
        else if (STREQ (name, "of"))
            output_file = val;
        else if (STREQ (name, "conv"))
            parse_conversion (val);
        else
        {
            int invalid = 0;
            uintmax_t n = parse_integer (val, &invalid);

            if (STREQ (name, "ibs"))
            {
                input_blocksize = n;
                invalid |= input_blocksize != n || input_blocksize == 0;
                conversions_mask |= C_TWOBUFFS;
            }
            else if (STREQ (name, "obs"))
            {
                output_blocksize = n;
                invalid |= output_blocksize != n || output_blocksize == 0;
                conversions_mask |= C_TWOBUFFS;
            }
            else if (STREQ (name, "bs"))
            {
                output_blocksize = input_blocksize = n;
                invalid |= output_blocksize != n || output_blocksize == 0;
            }
            else if (STREQ (name, "cbs"))
            {
                conversion_blocksize = n;
                invalid |= (conversion_blocksize != n
                           || conversion_blocksize == 0);
            }
            else if (STREQ (name, "skip"))
                skip_records = n;
            else if (STREQ (name, "seek"))
                seek_records = n;
            else if (STREQ (name, "count"))
                max_records = n;
            else
            {
                error (0, 0, _("unrecognized option %s=%s"),
                       quote_n (0, name), quote_n (1, val));
                usage (1);
            }
        }

        if (invalid)
            error (1, 0, _("invalid number %s"), quote (val));
    }
}

```

```

/* If bs= was given, both `input_blocksize' and `output_blocksize' will
   have been set to positive values. If either has not been set,
   bs= was not given, so make sure two buffers are used. */
if (input_blocksize == 0 || output_blocksize == 0)
    conversions_mask |= C_TWOBUFFS;
if (input_blocksize == 0)
    input_blocksize = DEFAULT_BLOCKSIZE;
if (output_blocksize == 0)
    output_blocksize = DEFAULT_BLOCKSIZE;
if (conversion_blocksize == 0)
    conversions_mask &= ~(C_BLOCK | C_UNBLOCK);
}

/* Fix up translation table. */

static void
apply_translations (void)
{
    unsigned int i;

#define MX(a) (bit_count (conversions_mask & (a)))
    if ((MX (C_ASCII | C_EBCDIC | C_IBM) > 1)
        || (MX (C_BLOCK | C_UNBLOCK) > 1)
        || (MX (C_LCASE | C_UCASE) > 1)
        || (MX (C_UNBLOCK | C_SYNC) > 1))
        {
            error (1, 0, _("
only one conv in {ascii,ebcdic,ibm}, {lcase,ucase}, {block,unblock}, {unblock,sync}"));
        }
#undef MX

    if (conversions_mask & C_ASCII)
        translate_charset (ebcdic_to_ascii);

    if (conversions_mask & C_UCASE)
    {
        for (i = 0; i < 256; i++)
            if (ISLOWER (trans_table[i]))
                trans_table[i] = TOUPPER (trans_table[i]);
        translation_needed = 1;
    }
    else if (conversions_mask & C_LCASE)
    {
        for (i = 0; i < 256; i++)
            if (ISUPPER (trans_table[i]))
                trans_table[i] = TOLOWER (trans_table[i]);
        translation_needed = 1;
    }

    if (conversions_mask & C_EBCDIC)
    {
        translate_charset (ascii_to_ebcdic);
        newline_character = ascii_to_ebcdic["\n"];
        space_character = ascii_to_ebcdic[' '];
    }
    else if (conversions_mask & C_IBM)
    {
        translate_charset (ascii_to_ibm);
        newline_character = ascii_to_ibm["\n"];
        space_character = ascii_to_ibm[' '];
    }
}

/* Apply the character-set translations specified by the user
   to the NREAD bytes in BUF. */

static void
translate_buffer (unsigned char *buf, size_t nread)
{
    unsigned char *cp;
    size_t i;

    for (i = nread, cp = buf; i; i--, cp++)
        *cp = trans_table[*cp];
}

/* If nonzero, the last char from the previous call to `swab_buffer'
   is saved in `saved_char'. */
static int char_is_saved = 0;

/* Odd char from previous call. */
static unsigned char saved_char;

/* Swap NREAD bytes in BUF, plus possibly an initial char from the
   previous call. If NREAD is odd, save the last char for the
   next call. Return the new start of the BUF buffer. */

static unsigned char *
swab_buffer (unsigned char *buf, size_t *nread)
{
    unsigned char *bufstart = buf;
    register unsigned char *cp;
    register int i;

    /* Is a char left from last time? */
    if (char_is_saved)
    {
        *--bufstart = saved_char;
        (*nread)++;
        char_is_saved = 0;
    }
}

```

```

}

if (*nread & 1)
{
    /* An odd number of chars are in the buffer. */
    saved_char = bufstart[--*nread];
    char_is_saved = 1;
}

/* Do the byte-swapping by moving every second character two
positions toward the end, working from the end of the buffer
toward the beginning. This way we only move half of the data. */

cp = bufstart + *nread; /* Start one char past the last. */
for (i = *nread / 2; i--; cp -= 2)
    *cp = *(cp - 2);

return ++bufstart;
}

/* Return nonzero iff the file referenced by FDESC is of a type for
which lseek's return value is known to be invalid on some systems.
Otherwise, return zero.
For example, return nonzero if FDESC references a character device
(on any system) because the lseek on many Linux systems incorrectly
returns an offset implying it succeeds for tape devices, even though
the function fails to perform the requested operation. In that case,
lseek should return nonzero and set errno. */

static int
buggy_lseek_support (int fdesc)
{
    /* We have to resort to this because on some systems, lseek doesn't work
on some special files but doesn't return an error, either.
In particular, the Linux tape drivers are a problem.
For example, when I did the following using dd-4.0y or earlier on a
Linux-2.2.17 system with an Exabyte SCSI tape drive:

    dev=/dev/nst0
    reset=mt -f $dev rewind; mt -f $dev fsf 1'
    eval $reset; dd if=$dev bs=32k of=out1
    eval $reset; dd if=$dev bs=32k of=out2 skip=1

the resulting files, out1 and out2, would compare equal. */

    struct stat stats;

    return (fstat (fdesc, &stats) == 0
        && (S_ISCHR (stats.st_mode)));
}

/* Throw away RECORDS blocks of BLOCKSIZE bytes on file descriptor FDESC,
which is open with read permission for FILE. Store up to BLOCKSIZE
bytes of the data at a time in BUF, if necessary. RECORDS must be
nonzero. */

static void
skip (int fdesc, char *file, uintmax_t records, size_t blocksize,
    unsigned char *buf)
{
    off_t offset = records * blocksize;

    /* Try lseek and if an error indicates it was an inappropriate
operation, fall back on using read. Some broken versions of
lseek may return zero, so count that as an error too as a valid
zero return is not possible here. */

    if (offset / blocksize != records
        || buggy_lseek_support (fdesc)
        || lseek (fdesc, offset, SEEK_CUR) <= 0)
    {
        while (records--)
        {
            ssize_t nread = safe_read (fdesc, buf, blocksize);
            if (nread < 0)
            {
                error (0, errno, _("reading %s"), quote (file));
                quit (1);
            }
            /* POSIX doesn't say what to do when dd detects it has been
asked to skip past EOF, so I assume it's non-fatal.
FIXME: maybe give a warning. */
            if (nread == 0)
                break;
        }
    }
}

/* Copy NREAD bytes of BUF, with no conversions. */

static void
copy_simple (unsigned char const *buf, int nread)
{
    int nfree; /* Number of unused bytes in `obuf'. */
    const unsigned char *start = buf; /* First uncopied char in BUF. */

    do
    {
        nfree = output_blocksize - oc;
        if (nfree > nread)
            nfree = nread;
    }
}

```

```

memcpy ((char *) (obuf + oc), (char *) start, nfree);

nread -= nfree;                /* Update the number of bytes left to copy. */
start += nfree;
oc += nfree;
if (oc >= output_blocksize)
    write_output ();
}
while (nread > 0);
}

/* Copy NREAD bytes of BUF, doing conv=block
(pad newline-terminated records to 'conversion_blocksize',
replacing the newline with trailing spaces). */

static void
copy_with_block (unsigned char const *buf, size_t nread)
{
    size_t i;

    for (i = nread; i > 0, buf++)
    {
        if (*buf == newline_character)
        {
            if (col < conversion_blocksize)
            {
                size_t j;
                for (j = col; j < conversion_blocksize; j++)
                    output_char (space_character);
            }
            col = 0;
        }
        else
        {
            if (col == conversion_blocksize)
                r_truncate++;
            else if (col < conversion_blocksize)
                output_char (*buf);
            col++;
        }
    }
}

/* Copy NREAD bytes of BUF, doing conv=unblock
(replace trailing spaces in 'conversion_blocksize'-sized records
with a newline). */

static void
copy_with_unblock (unsigned char const *buf, size_t nread)
{
    size_t i;
    unsigned char c;
    static int pending_spaces = 0;

    for (i = 0; i < nread; i++)
    {
        c = buf[i];

        if (col++ >= conversion_blocksize)
        {
            col = pending_spaces = 0; /* Wipe out any pending spaces. */
            i--;                    /* Push the char back; get it later. */
            output_char (newline_character);
        }
        else if (c == space_character)
            pending_spaces++;
        else
        {
            /* `c' is the character after a run of spaces that were not
            at the end of the conversion buffer. Output them. */
            while (pending_spaces)
            {
                output_char (space_character);
                --pending_spaces;
            }
            output_char (c);
        }
    }
}

/* The main loop. */

static int
dd_copy (void)
{
    unsigned char *ibuf, *bufstart; /* Input buffer. */
    unsigned char *real_buf; /* real buffer address before alignment */
    unsigned char *real_obuf;
    ssize_t nread; /* Bytes read in the current block. */
    int exit_status = 0;
    size_t page_size = getpagesize ();
    size_t n_bytes_read;

    /* Leave at least one extra byte at the beginning and end of `ibuf'
    for conv=swab, but keep the buffer address even. But some peculiar
    device drivers work only with word-aligned buffers, so leave an
    extra two bytes. */

    /* Some devices require alignment on a sector or page boundary
    (e.g. character disk devices). Align the input buffer to a

```

page boundary to cover all bases. Note that due to the swab algorithm, we must have at least one byte in the page before the input buffer; thus we allocate 2 pages of slop in the real buffer. 8k above the blocksize shouldn't bother anyone.

The page alignment is necessary on any linux system that supports either the SGI raw I/O patch or Steven Tweedies raw I/O patch. It is necessary when accessing raw (i.e. character special) disk devices on Unixware or other SVR4-derived system. \*/

```

real_buf = (unsigned char *) xmalloc (input_blocksize
                                     + 2 * SWAB_ALIGN_OFFSET
                                     + 2 * page_size - 1);

ibuf = real_buf;
ibuf += SWAB_ALIGN_OFFSET; /* allow space for swab */

ibuf = PTR_ALIGN (ibuf, page_size);

if (conversions_mask & C_TWOBUFFS)
{
    /* Page-align the output buffer, too. */
    real_obuf = (unsigned char *) xmalloc (output_blocksize + page_size - 1);
    obuf = PTR_ALIGN (real_obuf, page_size);
}
else
{
    real_obuf = NULL;
    obuf = ibuf;
}

if (skip_records != 0)
skip (STDIN_FILENO, input_file, skip_records, input_blocksize, ibuf);

if (seek_records != 0)
{
    /* FIXME: this loses for
       % ./dd if=dd seek=1 |:
       ./dd: standard output: Bad file number
       0+0 records in
       0+0 records out
       */

    skip (STDOUT_FILENO, output_file, seek_records, output_blocksize, obuf);
}

if (max_records == 0)
quit (exit_status);

while (1)
{
    if (r_partial + r_full >= max_records)
        break;

    /* Zero the buffer before reading, so that if we get a read error,
       whatever data we are able to read is followed by zeros.
       This minimizes data loss. */
    if ((conversions_mask & C_SYNC) && (conversions_mask & C_NOERROR))
        memset ((char *) ibuf, 0, input_blocksize);

    nread = safe_read (STDIN_FILENO, ibuf, input_blocksize);

    if (nread == 0)
        break; /* EOF. */

    if (nread < 0)
    {
        error (0, errno, ("reading %s"), quote (input_file));
        if (conversions_mask & C_NOERROR)
        {
            print_stats ();
            /* Seek past the bad block if possible. */
            lseek (STDIN_FILENO, (off_t) input_blocksize, SEEK_CUR);
            if (conversions_mask & C_SYNC)
                /* Replace the missing input with null bytes and
                   proceed normally. */
                nread = 0;
            else
                continue;
        }
        else
        {
            /* Write any partial block. */
            exit_status = 2;
            break;
        }
    }

    n_bytes_read = nread;

    if (n_bytes_read < input_blocksize)
    {
        r_partial++;
        if (conversions_mask & C_SYNC)
        {
            if (!(conversions_mask & C_NOERROR))
                /* If C_NOERROR, we zeroed the block before reading. */
                memset ((char *) (ibuf + n_bytes_read), 0,
                        input_blocksize - n_bytes_read);
            n_bytes_read = input_blocksize;
        }
    }
}

```

```

else
    r_full++;

if (ibuf == obuf) /* If not C_TWOBUFS. */
{
    int nwritten = full_write(STDOUT_FILENO, obuf, n_bytes_read);
    if (nwritten < 0)
    {
        error(0, errno, _("writing %s"), quote(output_file));
        quit(1);
    }
    else if (n_bytes_read == input_blocksize)
        w_full++;
    else
        w_partial++;
    continue;
}

/* Do any translations on the whole buffer at once. */

if (translation_needed)
    translate_buffer(ibuf, n_bytes_read);

if (conversions_mask & C_SWAB)
    bufstart = swab_buffer(ibuf, &n_bytes_read);
else
    bufstart = ibuf;

if (conversions_mask & C_BLOCK)
    copy_with_block(bufstart, n_bytes_read);
else if (conversions_mask & C_UNBLOCK)
    copy_with_unblock(bufstart, n_bytes_read);
else
    copy_simple(bufstart, n_bytes_read);
}

/* If we have a char left as a result of conv=swab, output it. */
if (char_is_saved)
{
    if (conversions_mask & C_BLOCK)
        copy_with_block(&saved_char, 1);
    else if (conversions_mask & C_UNBLOCK)
        copy_with_unblock(&saved_char, 1);
    else
        output_char(saved_char);
}

if ((conversions_mask & C_BLOCK) && col > 0)
{
    /* If the final input line didn't end with a 'n', pad
       the output block to 'conversion_blocksize' chars. */
    unsigned int i;
    for (i = col; i < conversion_blocksize; i++)
        output_char(space_character);
}

if ((conversions_mask & C_UNBLOCK) && col == conversion_blocksize)
/* Add a final 'n' if there are exactly 'conversion_blocksize'
   characters in the final record. */
output_char(newline_character);

/* Write out the last block. */
if (oc != 0)
{
    int nwritten = full_write(STDOUT_FILENO, obuf, oc);
    if (nwritten > 0)
        w_partial++;
    if (nwritten < 0)
    {
        error(0, errno, _("writing %s"), quote(output_file));
        quit(1);
    }
}

free(real_buf);
if (real_obuf)
    free(real_obuf);

return exit_status;
}

/* This is gross, but necessary, because of the way close_stdout
works and because this program closes STDOUT_FILENO directly. */
static void (*closeout_func)(void) = close_stdout;

static void
close_stdout_wrapper(void)
{
    if (closeout_func)
        (*closeout_func)();
}

int
main(int argc, char **argv)
{
    int i;
    int exit_status;

    program_name = argv[0];
    setlocale(LC_ALL, "");
    bindtextdomain(PACKAGE, LOCALEDIR);

```

```

textdomain (PACKAGE);

/* Arrange to close stdout if parse_long_options exits. */
atexit (close_stdout_wrapper);

parse_long_options (argc, argv, PROGRAM_NAME, GNU_PACKAGE, VERSION,
                   AUTHORS, usage);

/* Don't close stdout on exit from here on. */
closeout_func = NULL;

/* Initialize translation table to identity translation. */
for (i = 0; i < 256; i++)
  trans_table[i] = i;

/* Decode arguments. */
scanargs (argc, argv);

apply_translations ();

if (input_file != NULL)
  {
  if (open_fd (STDIN_FILENO, input_file, O_RDONLY, 0) < 0)
    error (1, errno, _("opening %s"), quote (input_file));
  }
else
  input_file = _("standard input");

if (output_file != NULL)
  {
  mode_t perms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH;
  int opts
    = (O_CREAT
       | (seek_records || (conversions_mask & C_NOTRUNC) ? 0 : O_TRUNC));

  /* Open the output file with *read* access only if we might
     need to read to satisfy a 'seek=' request. If we can't read
     the file, go ahead with write-only access; it might work. */
  if (!(seek_records
        || open_fd (STDOUT_FILENO, output_file, O_RDWR | opts, perms) < 0
        && open_fd (STDOUT_FILENO, output_file, O_WRONLY | opts, perms) < 0)
      error (1, errno, _("opening %s"), quote (output_file));

#ifdef HAVE_FTRUNCATE
  if (seek_records != 0 && !(conversions_mask & C_NOTRUNC))
    {
    struct stat stdout_stat;
    off_t o = seek_records * output_blocksize;
    if (o / output_blocksize != seek_records)
      error (1, 0, _("file offset out of range"));

    if (fstat (STDOUT_FILENO, &stdout_stat) != 0)
      error (1, errno, _("cannot fstat %s"), quote (output_file));

    /* Complain only when truncate fails on a regular file, a
       directory, or a shared memory object, as the 2000-08
       POSIX draft specifies truncate's behavior only for these
       file types. For example, do not complain when Linux 2.4
       truncate fails on /dev/fd0. */
    if (ftruncate (STDOUT_FILENO, o) != 0
        && (S_ISREG (stdout_stat.st_mode)
            || S_ISDIR (stdout_stat.st_mode)
            || S_TYPEISSHM (stdout_stat.st_mode)))
      {
      char buf[LONGEST_HUMAN_READABLE + 1];
      error (1, errno, _("advancing past %s bytes in output file %s"),
            human_readable (o, buf, 1, 1),
            quote (output_file));
      }
    }
#endif
  }
else
  {
  output_file = _("standard output");
  }

install_handler (SIGINT, interrupt_handler);
install_handler (SIGQUIT, interrupt_handler);
install_handler (SIGPIPE, interrupt_handler);
install_handler (SIGINFO, siginfo_handler);

exit_status = dd_copy ();

quit (exit_status);
}

```