Tecniche di Progettazione Automatica **Anno Acc. 1995-96**

PROF. ALFREDO DE SANTIS

Implementazioni in C del DES

a cura di Amodeo Vincenzo e Cusumano Massimo *23 Luglio 1996*

Indice

1 Introduzione

Il nostro lavoro analizza 6 implementazioni in linguaggio C dell'algoritmo DES, tutte compatibili con il DES standard: Des Barrett, Des 1.0, D3Des, Plain Des, Libdes e Fast Des. Quello che si è voluto fare è un confronto tra le implementazioni per capire, prima quali sono i diversi modi in cui puó essere realizzato l'algoritmo, poi tra questi rilevare quelli piú efficenti.

1.1 Plain Des

Realizzata dal Dottor Antonio Giorgio Gaggia nel 1993. Questa implementazione è una semplice codifica in linguaggio C dell'algoritmo DES che non realizza nessun tipo di ottimizzazione e quindi risulta essere un ottimo supporto didattico ma anche la più lenta versione tra quelle analizzate.

1.2 Des 1.0

È stata realizzata nel 1990 da Stig Ostholm del Dipartimento di Computer Engineering alla Chalmers University of Technology (Svezia). L'implementazione che contiene una libreria DES a scopi generali e programmi utenti per la cifratura e decifratura di file risulta essere testata sui seguenti sistemi:

$NomeDellaMacchina$	*Sistema operativo*
SUN4	SUN OS 4.1
SUN3	SUN OS 4.1
VAX 11/750	BSD 4.3
IBM RT/PC	AOS 4.3
IBM RS/6000-320	AIX 3.1
IBM PS/2	AIX 1.2+TCF
IBM 3090-1505/VF	AIX/370+TCF

Non è una implementazione molto veloce anche se realizza le permutazioni che caratterizzano l'algoritmo DES in maniera abbastanza veloce (come si vedrà più avanti).

1.3 Fast Des

Realizzata da Stuart Levy nell'Aprile del 1988. Questa implementazione, abbastanza veloce, è caratterizzata dal fatto che il calcolo di una qualsiasi delle permutazioni richieste dall'algoritmo, è eseguito in ordine inverso. Cioé, quando calcoliamo la permutazione di un blocco, il primo bit risultante sarà quello specificato dall'ultimo elemento della tabella della permutazione, il secondo sarà quello specificato dal penultimo elemento della tabella e così via.

1.4 Des Barrett

Questo pacchetto è stato realizzato nel 1991 allo scopo di inviare e ricevere posta elettronica. L'autore David A. Barrett (e-mail: barrett@boulder.Colorado.ED) ha concentrato i propri sforzi per realizzare un programma DES veloce e portabile, anche se rimangono nella sua implementazione considerevoli margini di miglioramento della velocità.

1.5 D3Des

È un'implementazione del DES con la possibilità di doppia e tripla cifratura. È stata realizzata da Richard Outerbridge (e-mail 71755.204@CompuServe.COM) nel 1992 con l'aiuto di Dan Hoey per la realizzazione delle permutazioni IP e IP^{-1}, Jim Gillogly e Phil Karn per la schedulazione delle chiavi. Questa è con Libdes l'implementazione più veloce tra quelle esaminate e non prende molta memoria a tempo di esecuzione. Il motivo della sua velocità è da ricercarsi nella particolarissima rappresentazione delle sedici sottochiavi e nella conseguente realizzazione delle varie permutazioni.

1.6 Libdes

È stata realizzata da Eric Young (e-mail eay@psych.uq.oz.au) nel 1993 seguendo i suggerimenti di Richard Outerbridge su come realizzare le permutazioni IP e IP^{-1} e sullo shift ciclico da applicare alle due metà dell'output della permutazione IP. È un kit costituito da una libreria di cifratura e da un programma di cifratura per il DES, organizzato in modo da poter funzionare su macchine con sistemi operativi UNIX, DOS, VMS. Esso supporta come modalità di cifratura l'ECB, OFB, CFB e *triplo DES*. Contiene routines di supporto per leggere le chiavi da un terminale, generare una chiave casuale e generare una chiave da una stringa di lunghezza arbitraria.

2 Criteri generali d'implementazione

Ricordando la descrizione del DES, i due punti fondamentali dell'algoritmo sono la realizzazione delle diverse *permutazioni* (con *permutazioni* indichiamo anche le espansioni e le riduzioni) e l'indirizzamento e il computo delle S-box.
Cerchiamo di capire come l'algoritmo può essere implementato.

2.1 Le *permutazioni*

Consideriamo:

- M il vettore che individua la *permutazione* da effettuare.
  Nel caso di IP

$$\begin{aligned} M = \quad & [58, 50, 42, 34, 26, 18, 10, 2, 60, 52, 44, 36, 28, 20, 12, 4, \\ & 62, 54, 46, 38, 30, 22, 14, 6, 64, 56, 48, 40, 32, 24, 16, 8, \\ & 57, 49, 41, 33, 25, 17, 9, 1, 59, 51, 43, 35, 27, 19, 11, 3, \\ & 61, 53, 45, 37, 29, 21, 13, 5, 63, 55, 47, 39, 31, 23, 15, 7]. \end{aligned}$$

  Dove $M[1] = 58$ significa che il primo bit dell'output è il 58-esimo bit dell'input, $M[2] = 50$ significa che il secondo bit dell'output è il 50-esimo bit dell'input e cosi via.

- s_in la sequenza binaria da permutare, di dimensione n.

- s_out la sequenza binaria output della permutazione, di dimensione d.

Lo schema generale per realizzare una qualsiasi *permutazione* è dato dalla seguente funzione:

> **Permuta**(M, s_in, d):
> (1) $s\_out = 0$;
> (2) **for** $i = 1$ **to** d
> (3) $\quad k = M[i]$;
> (4) $\quad b = k$-esimo bit di s_in;
> (5) $\quad i$-esimo bit di $s\_out = b$;
> (6) **return** s_out.

La possiamo implementare usando dei vettori di char per memorizzare le due sequenze di bit. Questo è il modo d'agire di Plain Des. Avremo in questo caso:

- la sequenza iniziale $s\_in = s\_in[1] \cdots s\_in[\frac{n}{8}]$ con $s\_in[i]$ costituito da 8 bit. Con i primi 8 bit della sequenza memorizzati in $s\_in[1]$ gli ultimi 8 bit in $s\_in[\frac{n}{8}]$;

- la sequenza d'output $s\_out = s\_out[1] \cdots s\_out[\frac{d}{8}]$ anch'essa memorizzata in un vettore di char in modo che i primi 8 bit dell'output siano memorizzati in $s\_out[1]$ gli ultimi 8 bit in $s\_out[\frac{d}{8}]$.

La specificazione di d e l'uso di vettori di char (un char è costituito da 8 bit) per la memorizzazione di s_in e s_out, rende la funzione molto flessibile. Infatti grazie a d è possibile realizzare oltre alle permutazioni IP, IP^{-1} e P anche l'espansione E e le riduzioni $PC-1$ e $PC-2$ caratterizzate da sequenze d'output che hanno dimensioni differenti dalle sequenze d'input, e grazie all'uso di vettori di char le diverse taglie che s_in e s_out possono assumere vengono gestite in modo semplice. Per IP e IP^{-1} $n = d = 64$, per E $n = 32$ mentre $d = 48$, per P $n = d = 32$, per $PC-1$ $n = 64$ mentre $d = 56$ e per $PC-1$ $n = 56$ mentre $d = 48$. Si osserva che la lunghezza delle varie sequenze è sempre multipla di 8 quindi un opportuno vettore di char riesce a memorizzarle sempre.

Relativamente alla realizzazione della funzione, per prelevare il bit b (passo (4)) si devono fare le seguenti operazioni:

1. Individuare il byte in cui è contenuto il k-esimo bit di s_in che è il byte $\lceil \frac{k}{8} \rceil$.

2. Individuare la posizione del bit all'interno del byte che è $k \ mod \ 8$.

3. Prelevare il k-esimo bit tramite un *and* tra il byte che lo contiene è una *mask* opportuna.

Un discorso analogo vale per inserire il bit b nella sequenza d'output (passo (5)). Si individua il byte in cui occorre inserirlo, la posizione all'interno del byte e quindi si copia il bit x tramite un operazione di *or*.
Tale metodo risulta essere però molto costoso. Per la realizzazione di IP infatti sono necessarie circa 2500 operazioni, contando le operazioni elementari del linguaggio C compresa l'assegnazione.
La funzione può essere realizzata usando anche altre strutture dati per memorizzare le sequenze su cui agisce, ad esempio dei vettori di long. In questo caso le sequenze che hanno una dimensione non multipla di 32 verranno memorizzate nei long riempendo lo spazio mancante

con dei bit 0. Se la sequenza da memorizzare è di 48 bit (ad esempio l'output dell'espansione E) è necessario usare due long e quello che si può fare è dividerla in due sottosequenze da 24 bit e memorizzare ciascuna di queste in un long mettendo a 0 gli otto bit meno significativi o gli otto più significativi.
Comunque anche in questo caso il costo della funzione non cambia.

Nel caso particolare in cui la parola macchina sia tale da contenere per intero tutte le sequenze di bit su cui si agisce (la parola macchina dev'essere di almeno 64 bit) i passi (4) e (5) della funzione possono essere realizzati usando l'istruzione

$$s\_out = [(s\_in \wedge mask_k) << (k-t)] \vee s\_out. \tag{1}$$

$mask_k$ è una sequenza di n bit, tutti a zero tranne il k-esimo che ha valore 1. $a << b$ indica uno shift a sinistra su a di b posizioni, ottenuto eliminando i b bit più significativi di a e inserendo b bit zero nelle posizioni meno significative di a.
Ogni istruzione di questo tipo equivale a fare 4 operazioni quindi per realizzare una *permutazione* ne sono necessarie $4d$. Il costo è sicuramente più basso di quanto visto prima ma c'è il problema che non sono molti i calcolatori caratterizzati da una parola macchina di almeno 64 bit. Si può comunque cercare di sfruttare il modo per copiare i bit tra due sequenze individuato da (1). Lo facciamo individuando del codice non flessibile come quello suddetto ma che funzioni per una specifica permutazione. Diamo un esempio di come, in questo caso, la tecnica descritta dall'istruzione (1) viene utilizzata effettivamente.

Consideriamo la permutazione IP, la realizziamo nel seguente modo:

1. Consideriamo l_{s_in} e r_{s_in} una coppia di long che contiene i 64 bit su cui si deve applicare la matrice di *permutazione*. l_{s_out} e r_{s_out} è una coppia di long usata per contere i 64 bit finali. Con riferimento all'istruzione (1), l_{s_in} e r_{s_in} sono rispettivamente i 32 bit più a sinistra e i 32 bit più a destra della sequenza iniziale s_in, mentre l_{s_out} e r_{s_out} sono rispettivamente i 32 bit più a sinistra e i 32 bit più a destra della sequenza finale s_out;

2. Inizializziamo a zero l_{s_out} e r_{s_out};

3. Realizziamo $M[1] = 58$ con la linea di codice

$$l_{s\_out} = [(r_{s\_in} \wedge mask_{26}) << 25] \vee l_{s\_out}.$$

   Rispetto alla schema generale non abbiamo $mask_{58}$ né lo shift di $58 - 1 = 57$ posizioni. Questo perché nel nostro caso s_in e s_out sono entrambe suddivise in due sottosequenze di 32 bit ciascuna. Per cui, quando consideriamo l'epressione generale $M(k) = t$ il t-esimo bit di s_in è inserito come k-esimo bit di l_{s_out} se $1 \leq k \leq 32$ e come $(k-32)$-esimo bit di r_{s_out} se $33 \leq k \leq 64$. Anche il valore t è preso con riferimento alle due sequenze di 32 bit l_{s_in} e r_{s_in}, invece di 58 consideriamo 26 perché il 58-esimo bit di s_in è il 26-esimo di r_{s_in} $(26 = 58 - 32)$. Ciò significa che consideriamo il t-esimo bit di l_{s_in} quando $1 \leq t \leq 32$ e il $(t-32)$-esimo bit di r_{s_in} quando $33 \leq t \leq 64$;

4. Il secondo scambio che si effettua è $M[2] = 50$, realizzato tramite

$$l_{s\_out} = [(r_{s\_in} \wedge mask_{18}) << 16] \vee l_{s\_out}.$$

   Visto che $18 = 50 - 32$ e dovendo considerare un valore di $k = 18$ lo shift dev'essere di $18 - 2 = 16$ posizioni.

5. Si continua in questo modo usando l_{s_in} o r_{s_in} e l_{s_out} o r_{s_out} in accordo con i valori di k e t, finché la permutazione non è completata.

Ogni istruzione di tipo (1) equivale a fare 4 operazioni quindi occorrono 256 operazioni per realizzare IP contro le circa 2500 della tecnica precedente. Un guadagno notevole, anche se la realizzazione risulta essere ancora lenta se comparata al numero di operazioni (solo 45) con cui IP è effettuata nelle implementazioni D3Des e Libdes.

2.2 Calcolo delle S-box

L'altro punto importante dell'algoritmo DES è come viene calcolato l'input e l'output di ogni S-box. Vediamo un modo diretto per procedere.
Supponiamo di conoscere già i valori B_i $(i = 1, \cdots, 8)$ input delle S-box e che ciascuno di questi sia uguale a $00x_1 \cdots x_6$ e memorizzato in un byte. Per ciascuna S-box bisogna:

1. valutare $i = x_1 x_6$;
2. valutare $j = x_2 \cdots x_5$;
3. accedere alla locazione (i, j) della S-box considerata, ottenendo la sequenza $c_1 \cdots c_4$;
4. inserire $c_1 \cdots c_4$ nella variabile che conterrà l'output delle S-box nella posizione appropriata.

L'implementazione diretta è la seguente:

Il passo 1 è realizzato utilizzando la (1):

$$i = 0;$$

$$i = [(B_i \wedge 00100000) << (-4)] \vee i;$$

$$i = [(B_i \wedge 00000001] \vee i;$$

con un totale di 8 operazioni.

Il passo 2 è realizzato agendo come al passo 1:

$$j = 0;$$

$$j = [(B_i \wedge 00010000) << (-1)] \vee j;$$

$$j = [(B_i \wedge 00001000) << (-1)] \vee j;$$

$$j = [(B_i \wedge 00000100) << (-1)] \vee j;$$

$$j = [(B_i \wedge 00000010) << (-1)] \vee j;$$

con un totale di 17 operazioni. Uno shift a sinistra di b posizioni con b negativo equivale ad uno shift a destra di $|b|$ posizioni.

I passi 3 e 4 possono essere eseguiti in una unica operazione. Per memorizzare l'uscita della k-esima S-box si usa:

$$output\_c[k] = \text{S-box}[i][j];$$

dove $output\_c$ è un vettore di 8 char.
Dobbiamo osservare che l'utilizzo del vettore di char da un lato rende immediata la realizzazione di questi due passi, dall'altro lato comporta l'utilizzo di 64 bit per memorizzarne soltanto 32. Si potrebbe allora pensare di utilizzare al posto del vettore di char un long per memorizzare l'output delle varie S-box, ma questo comporterebbe delle operazioni aggiuntive nell'inserimento di tali output nel long.

I 4 passi vengono realizzati con almeno 25 operazioni; considerato che abbiamo 8 S-box si ha un totale di 200 operazioni.
È un costo molto alto, vedremo infatti come D3Des realizzi tutto ció, P compresa, con solo 29 operazioni.

2.2.1 Calcolo delle S-box e della permutazione P

Un'altro modo di agire, usato da tutte le implementazioni analizzate, è :

- Tenere conto della permutazione P già nelle S-box.
- Strutturare le S-box in modo tale da accedervi direttamente saltando la valutazione dei sei bit d'input così come descritta dall'algoritmo.

Per il primo punto si applica la permutazione P direttamente agli elementi contenuti nelle S-box. Gli array risultanti vengono chiamati SPBOX per distinguerli dalle S-box da cui sono ricavati.
Per ogni S-box ogni possibile uscita (4 bit) viene vista come una stringa di 32 bit, di questi, 4 sono quelli contenuti nelle S-box, mentre i restanti 28 hanno valore 0. La posizione dei 4 bit nella stringa di 32 è dipendente dal numero di S-box di cui sono l'output . Cioè i 4 bit in uscita da S-box1 vengono visti come i 4 più significativi in una stringa di 32, mentre i 4 bit in uscita da S-box8 vengono visti come i 4 meno significativi in una stringa di 32 bit. Ciascuna di queste stringhe di 32 bit è permutata usando la matrice P definita dall'algoritmo del DES. Le SPBOX sono quindi 8 array contenenti elementi di 32 bit.
Per il secondo punto, la costruzione delle SPBOX è fatta in modo tale che dai 6 bit d'input si arrivi in maniera immediata ai 32 bit d'output corrispondenti. Le 4 righe di ogni SPBOX vengono distribuite in un vettore di 64 elementi osservando che per ogni SPBOX:

- le sequenze di 6 bit che danno come output un elemento della prima riga hanno la forma 0xxxx0 (con x bit qualsiasi), sono quindi valori pari compresi tra 0 e 30;
- le sequenze di 6 bit che danno come output un elemento della seconda riga hanno la forma 0xxxx1 (con x bit qualsiasi), sono quindi valori dispari compresi tra 1 e 31;
- le sequenze di 6 bit che danno come output un elemento della terza riga hanno la forma 1xxxx0 (con x bit qualsiasi), sono quindi valori pari compresi tra 32 e 62;
- le sequenze di 6 bit che danno come output un elemento della quarta riga hanno la forma 1xxxx1 (con x bit qualsiasi), sono quindi valori dispari compresi tra 33 e 63.

Quindi le prime due righe occuperanno le prime 32 locazioni, in posizione pari gli elementi della prima riga, in posizione dispari gli elementi della seconda riga. La terza e la quarta riga occuperanno le locazioni del vettore che vanno dalla 32 alla 63 con in posizione pari gli elementi della terza riga e in posizione dispari gli elementi della quarta riga. Ogni elemento delle S-box verrà collocato nel vettore nella posizione individuata dal valore della stringa di 6 bit che lo determina. Cosıfacendo il risultato finale della funzione f è ottenuto effettuando l'*or* dell'output degli otto SPBOX.

Esempio 2.1

Consideriamo il caso in cui l'input della S-box3 è 101100_2 *questo implica che dobbiamo prendere come output l'elemento della S-box3 che si trova nella colonna 7 e nella riga 3, S-box3[2][6]=*0011_2*. A partire da* 101100_2 *otteniamo accedendo alla SPBOX3 alla locazione 44 (*$101100_2 = 44_{10}$*)*

$$SPBOX3[44]=0000\ 0100\ 0000\ 0000\ 0000\ 0000\ 0000\ 0100_2$$

Vediamo perchè SPBOX3[44] contiene tale valore. Partiamo da

$$S\text{-}box3[2][6]=0011_2,$$

questo bisogna innanzitutto vederlo all'interno di una stringa di 32 bit cioè :

0000 0000 0011 0000 0000 0000 0000 0000

e poi permutato tramite P:

0000 0100 0000 0000 0000 0000 0000 0100

che corrisponde a SPBOX3[44].

Si capisce quindi come avendo a disposizione l'input di ciascuna S-box con un costo di 8 operazioni (i 7 *or* più l'assegnamento ad una variabile) si ottiene il valore della f. A questo punto diventa molto importante il modo, e i relativi costi, con cui le diverse implementazioni realizzano le permutazioni ma soprattutto l'espansione E.

Una ultima osservazione da fare sulle SPBOX è relativa allo spazio occupato. Questa organizzazione delle S-box in SPBOX comporta la memorizzazione di 8 tabelle di 64 elementi di 32 bit per un totale di $8 \cdot 64 \cdot 32 = 16384$ bit contro i $8 \cdot 64 \cdot 8 = 4096$ bit delle normali S-box, una differenza di un fattore 4.

3 Analisi generale

Il nostro primo obiettivo è stato quello di stabilire una gerarchia rispetto alla velocità di calcolo delle diverse implementazioni. Per far ciò abbiamo misurato il tempo impiegato da ogni implementazione per la cifratura (decifratura) di vari file testo di dimensioni conosciute. Le misurazioni riguardavano il tempo richiesto per cifrare i k blocchi di 64 bit di cui era costituito il file, più quello necessario alla schedulazione delle chiavi.

In seguito abbiamo diviso le dimensioni dei file testo per i tempi impiegati ed abbiamo mediato tali risultati. Si e lavorato su di una Digital 2100 modello A500MP, dotata di due processori DECchip 21064 con cadenza di clock di 190 MHz, una memoria principale di 128 mega e di sistema operativo DEC OSF/1 V3.2A. Abbiamo ottenuto i seguenti risultati :

Implementazioni	*byte cifrati al secondo*	*byte decifrati al secondo*
D3Des	1096260	1101808
Libdes	925574	944362
Des Barrett	564322	558205
Fast Des	696975	710445
DES 1.0	233261	233676
Plain Des	8984	8892

- Tabella 1 -

Occorre fare una precisazione sui tempi di Libdes. Presentano un costo che nell'altre implementazioni non è stato considerato, il tempo speso per leggere il file da cifrare e portarlo in memoria. I tempi di Libdes risultano così piú lenti rispetto ai tempi delle altre implementazioni. In realtà devono considerarsi dello stesso ordine dei tempi di D3Des.
Gli algoritmi originali delle varie implementazioni sono stati opportunamente modificati in modo da consentire solo la modalità standard di cifratura e decifratura del DES, cioè ECB e in modo che la chiave debba essere inserita in forma di sedici caratteri esadecimali, dove quindi un byte rappresenta un valore esadecimale. E in seguito a ció le dimensioni degli eseguibili individuati erano di circa 40Kbyte per Plain Des, D3Des e Fast Des, di circa 57Kbyte per Libdes e Des Barrett mentre di oltre 200Kbyte per Des 1.0 . Ogni implementazione riorganizzerà la chiave in modo tale da rappresentare con un byte due esadecimali. In particolare il primo esadecimale sarà rappresentato dai 4 bit più significativi del byte ed il secondo dai 4 bit meno significativi. Abbiamo quindi ridotto il numero di bit necessario alla rappresentazione della chiave da 128 a 64, organizzati in 8 byte. Notiamo però che l'algoritmo DES si aspetta una chiave da 56 bit. Tale dimensione è ottenuta trascurando l'ultimo bit di ognuno degli 8 byte (operazione fatta dalle procedure di schedulazione delle chiavi di ciascuna implementazione). Anche se la funzione di scheduling della chiave DES è eseguita una sola volta, prima che inizi la vera e propria fase di cifratura-decifratura e il tempo da essa consumato può essere considerato ininfluente rispetto al tempo totale consumato da ogni implementazione, se ne è tenuto ugualmente conto visto che alcuni implemetazioni usano riorganizzare le sottochiavi per velocizzare le successive operazioni.
Come si nota dalla Tabella 1, anche se abbiamo a che fare con implementazioni dello stesso algoritmo, troviamo differenze di tempo che in alcuni casi assumono dimensioni drammatiche. La conclusione a cui siamo giunti è che la velocità sia strettamente legata al modo in cui sono realizzate le diverse permutazioni richieste dall'algoritmo DES. Per evidenziare ciò abbiamo "contato" le istruzioni 'C' necessarie alle due implementazioni limite (la più lenta Plain Des e la più veloce D3Des), per effettuare le permutazioni necessarie . Il risultato è riassunto nella seguente tabella:

Implementazioni	$IP + E$	IP^{-1}	$Sbox + P$
D3Des	61 *op*	53 *op*	29 *op*
Plain Des	4484 *op*	2562 *op*	1507 *op*

- Tabella 2 -

Notiamo inoltre che Plain Des, oltre ad usare molte più istruzioni di D3Des, fà uso di istruzioni complesse come operazioni modulari che sono molto più costose delle semplici operazioni orientate ai bit usate da D3Des.

4 Analisi di D3Des

Come risulta dalle Tabelle 1 e 2, D3Des è l'implementazione più efficiente. Per capire come funziona la abbiamo analizzata passo passo, visualizzando in apposite tabelle lo stato e l'organizzazione di alcune variabili. Tutto questo è stato necessario per seguire il flusso e per capire il significato di alcune operazioni in maniera chiara.

4.1 Scheduling della chiave

A partire dalla sequenza di 64 bit vengono calcolate le 16 chiavi di 48 bit, ognuna delle quali viene memorizzata in 2 long. Si usa quindi il vettore di 32 long Kn nel quale l'i-esima sottochiave, $i = 1, \cdots, 16$, è memorizzata negli elementi $Kn[2(i-1)]$ e $Kn[2(i-1)+1]$, nel primo la parte sinistra della i-esima sottochiave nel secondo la parte destra. La i-esima sottochiave

$$\text{key}[i]= kl_{01}kl_{02}kl_{03}kl_{04}...kl_{20}kl_{21}kl_{22}kl_{23}kl_{24}\ kr_{01}kr_{02}kr_{03}kr_{04}...kr_{20}kr_{21}kr_{22}kr_{23}kr_{24}$$

viene allora rappresentata nel modo che segue dove dall'alto verso il basso indichiamo i bit dal piu significativo al meno significativo :

$Kn[2(i-1)]$		$Kn[2(i-1)+1]$	
	0 0 0 0 0 0 0 0		0 0 0 0 0 0 0 0
L_1	kl_{01} kl_{02} kl_{03} kl_{04} kl_{05} kl_{06}	R_1	kr_{01} kr_{02} kr_{03} kr_{04} kr_{05} kr_{06}
L_2	kl_{07} kl_{08} kl_{09} kl_{10} kl_{11} kl_{12}	R_2	kr_{07} kr_{08} kr_{09} kr_{10} kr_{11} kr_{12}
L_3	kl_{13} kl_{14} kl_{15} kl_{16} kl_{17} kl_{18}	R_3	kr_{13} kr_{14} kr_{15} kr_{16} kr_{17} kr_{18}
L_4	kl_{19} kl_{20} kl_{21} kl_{22} kl_{23} kl_{24}	R_4	kr_{19} kr_{20} kr_{21} kr_{22} kr_{23} kr_{24}

Per ogni sottochiave si individuano 8 gruppi di 6 bit, gli L_j, $j = 1, \cdots, 4$, blocchi della sottochiave sinistra e gli R_j, $j = 1, \cdots, 4$, blocchi della sottochiave destra. Le posizioni di tali blocchi, all'interno delle coppia di long che contengono la sottochiave, vengono riorganizzate usando la tecnica descritta dall'istruzione (1), in modo da ottenere come rappresentazione ultima della i-esima sottochiave:

$KnL[2(i-1)]$		$KnL[2(i-1)+1]$	
	0 0		0 0
L_1	kl_{01} kl_{02} kl_{03} kl_{04} kl_{05} kl_{06}	L_2	kl_{07} kl_{08} kl_{09} kl_{10} kl_{11} kl_{12}
	0 0		0 0
L_3	kl_{13} kl_{14} kl_{15} kl_{16} kl_{17} kl_{18}	L_4	kl_{19} kl_{20} kl_{21} kl_{22} kl_{23} kl_{24}
	0 0		0 0
R_1	kr_{01} kr_{02} kr_{03} kr_{04} kr_{05} kr_{06}	R_2	kr_{07} kr_{08} kr_{09} kr_{10} kr_{11} kr_{12}
	0 0		0 0
R_3	kr_{13} kr_{14} kr_{15} kr_{16} kr_{17} kr_{18}	R_4	kr_{19} kr_{20} kr_{21} kr_{22} kr_{23} kr_{24}

Questa rappresentazione della sottoschiave permette di ridurre il numero di operazioni necessarie all'indirizzamento delle S-box (come si vedrà nella Sezione 4.3.1).

4.2 Permutazione IP

Spieghiamo ora istruzione per istruzione come avviene il calcolo della permutazione IP. Partendo da un blocco iniziale di 64 bit il primo passo effettuato è la divisione del suddetto blocco in 2 blocchi di 32 bit, $leftt$ e $right$. Subito dopo è inizializzata una variabile di lavoro, $work$, di 32 bit. Per evidenziare l'effetto che ogni istruzione ha sulle variabili $leftt$, $right$ e $work$ le abbiamo rappresentate in maniera tabulare, seguendo il flusso delle istruzioni, che per comodità riportiamo di seguito :

```
leftt = block[0];
right = block[1];
/* Primo Passo */
work = ((leftt >> 4) ^ right) & 0x0f0f0f0fL;
right ^= work;
```

```
leftt ^= (work << 4);
/* Secondo Passo */
work = ((leftt >> 16) ^ right) & 0x0000ffffL;
right ^= work;
leftt ^= (work << 16);
/* Terzo Passo */
work = ((right >> 2) ^ leftt) & 0x33333333L;
leftt ^= work;
right ^= (work << 2);
/* Quarto Passo */
work = ((right >> 8) ^ leftt) & 0x00ff00ffL;
leftt ^= work;
right ^= (work << 8);
right = ((right << 1) | ((right >> 31) & 1L)) & 0xffffffffL;
/* Quinto Passo */
work = (leftt ^ right) & 0xaaaaaaaaL;
leftt ^= work;
right ^= work;
leftt = ((leftt << 1) | ((leftt >> 31) & 1L)) & 0xffffffffL;
```

Ogni passo realizza uno scambio di bit tra $leftt$ e $right$ usando la seguente proprietà dell'*or* esclusivo: $a \oplus (a \oplus b) = b$. Lo schema seguito è il seguente :

```
work=(( a >> n ) ⊕ b) & mask
a=( work << n ) ⊕ a)
b=work ⊕ b
```

con `n` intero e `mask` sequenza di 8 caratteri esadecimali, parametri che individuano quali bit di `a` e `b` devono essere scambiati. Per capire come funziona questo schema basta osservare la seguente rappresentazione tabulare delle 3 variabili usate.

$lefftt_0$		$right_0$		$work_1$	$right_1$	$lefftt_1$	$work_2$
l_{01}	01 02	r_{01}	33 34	0 0 0 0	r_{01}	r_{03}	0 0 0 0
l_{02}	03 04	r_{02}	35 36		r_{02}	r_{04}	
l_{03}	05 06	r_{03}	37 38	$r_{03}r_{04} \oplus l_{01}l_{02}$	l_{01}	l_{03}	0 0 0 0
l_{04}	07 08	r_{04}	39 40		l_{02}	l_{04}	
l_{05}	09 10	r_{05}	41 42	0 0 0 0	r_{05}	r_{07}	0 0 0 0
l_{06}	11 12	r_{06}	43 44		r_{06}	r_{08}	
l_{07}	13 14	r_{07}	45 46	$r_{07}r_{08} \oplus l_{05}l_{06}$	l_{05}	l_{07}	0 0 0 0
l_{08}	15 16	r_{08}	47 48		l_{06}	l_{08}	
l_{09}	17 18	r_{09}	49 50	0 0 0 0	r_{09}	r_{11}	$r_{09}r_{10} \oplus r_{03}r_{04}$
l_{10}	19 20	r_{10}	51 52		r_{10}	r_{12}	
l_{11}	21 22	r_{11}	53 54	$r_{11}r_{12} \oplus l_{09}l_{10}$	l_{09}	l_{11}	$l_{09}l_{10} \oplus l_{03}l_{04}$
l_{12}	23 24	r_{12}	55 56		l_{10}	l_{12}	
l_{13}	25 26	r_{13}	57 58	0 0 0 0	r_{13}	r_{15}	$r_{13}r_{14} \oplus r_{07}r_{08}$
l_{14}	27 28	r_{14}	59 60		r_{14}	r_{16}	
l_{15}	29 30	r_{15}	61 62	$r_{15}r_{16} \oplus l_{13}l_{14}$	l_{13}	l_{15}	$l_{13}l_{14} \oplus l_{07}l_{08}$
l_{16}	31 32	r_{16}	63 64		l_{14}	l_{16}	

Nella rappresentazione tabulare $lefftt_i$ e $right_i$, $i = 1, \cdots, 6$, rappresentano le sequenze ottenute alla fine dell'i-esimo passo della realizzazione della permutazione, $work_i$ è la sequenza di 32 bit usata per realizzarlo. $lefftt_0$ e $right_0$ rappresentano la parte sinistra e destra del blocco da cifrare. Tutti gli elementi del tipo l_i e r_i presenti nella rappresentazione tabulare, individuano coppie di bit (l_i rappresenta i bit in posizione $2 \cdot (i-1)$ e $2 \cdot i$ del blocco iniziale, mentre r_i rappresenta i bit in posizione $32 + 2 \cdot (i-1)$ e $32 + 2 \cdot i$).

$right_2$	$leftt_2$	$work_3$	$leftt_3$	$right_3$	$work_4$	$leftt_4$	$right_4$
r_{01}	r_{09}	0 0	r_{09}	r_{10}	0 0	r_{09}	r_{13}
r_{02}	r_{10}	$r_{10} \oplus r_{01}$	r_{01}	r_{02}	0 0	r_{01}	r_{05}
l_{01}	l_{09}	0 0	l_{09}	l_{10}	0 0	l_{09}	l_{13}
l_{02}	l_{10}	$l_{10} \oplus l_{01}$	l_{01}	l_{02}	0 0	l_{01}	l_{05}
r_{05}	r_{13}	0 0	r_{13}	r_{14}	$r_{13} \oplus r_{10}$	r_{10}	r_{14}
r_{06}	r_{14}	$r_{14} \oplus r_{05}$	r_{05}	r_{06}	$r_{05} \oplus r_{02}$	r_{02}	r_{06}
l_{05}	l_{13}	0 0	l_{13}	l_{14}	$l_{13} \oplus l_{10}$	l_{10}	l_{14}
l_{06}	l_{14}	$l_{14} \oplus l_{05}$	l_{05}	l_{06}	$l_{05} \oplus l_{02}$	l_{02}	l_{06}
r_{03}	r_{11}	0 0	r_{11}	r_{12}	0 0	r_{11}	r_{15}
r_{04}	r_{12}	$r_{12} \oplus r_{03}$	r_{03}	r_{04}	0 0	r_{03}	r_{07}
l_{03}	l_{11}	0 0	l_{11}	l_{12}	0 0	l_{11}	l_{15}
l_{04}	l_{12}	$l_{12} \oplus l_{03}$	l_{03}	l_{04}	0 0	l_{03}	l_{07}
r_{07}	r_{15}	0 0	r_{15}	r_{16}	$r_{15} \oplus r_{12}$	r_{12}	r_{16}
r_{08}	r_{16}	$r_{16} \oplus r_{07}$	r_{07}	r_{08}	$r_{07} \oplus r_{04}$	r_{04}	r_{08}
l_{07}	l_{15}	0 0	l_{15}	l_{16}	$l_{15} \oplus l_{12}$	l_{12}	l_{16}
l_{08}	l_{16}	$l_{16} \oplus l_{07}$	l_{07}	l_{08}	$l_{07} \oplus l_{04}$	l_{04}	l_{08}

$leftt_4$	$right_4$	$right'_4$	$work_5$	$leftt_5$	$right_5$	$leftt'_5$
49	57	58	$49 \oplus 58$	58	49	50
50	58	41	0	50	41	42
33	41	42	$33 \oplus 42$	42	33	34
34	42	25	0	34	25	26
17	25	26	$17 \oplus 26$	26	17	18
18	26	09	0	18	09	10
01	09	10	$01 \oplus 10$	10	01	02
02	10	59	0	02	59	60
51	59	60	$51 \oplus 60$	60	51	52
52	60	43	0	52	43	44
35	43	44	$35 \oplus 44$	44	35	36
36	44	27	0	36	27	28
19	27	28	$19 \oplus 28$	28	19	20
20	28	11	0	20	11	12
03	11	12	$03 \oplus 12$	12	03	04
04	12	61	0	04	61	62
53	61	62	$53 \oplus 62$	62	53	54
54	62	45	0	54	45	46
37	45	46	$37 \oplus 46$	46	37	38
38	46	29	0	38	29	30
21	29	30	$21 \oplus 30$	30	21	22
22	30	13	0	22	13	14
05	13	14	$05 \oplus 14$	14	05	06
06	14	63	0	06	63	64
55	63	64	$55 \oplus 64$	64	55	56
56	64	47	0	56	47	48
39	47	48	$39 \oplus 48$	48	39	40
40	48	31	0	40	31	32
23	31	32	$23 \oplus 32$	32	23	24
24	32	15	0	24	15	16
07	15	16	$07 \oplus 16$	16	07	08
08	16	57	0	08	57	58

La doppia rappresentazione di $leftt_4$ e $rightt_4$ è stata necessaria per rendere ancora più evidente la posizione dei bit del blocco iniziale a quel punto del calcolo ($right'_4$ è la sequenza alla fine del quarto passo, dopo aver effettuato lo shift circolare).
Una volta calcolata $leftt_5$ ci accorgiamo che questa è la prima metà del risultato dell'applicazione della permutazione IP al nostro blocco iniziale, mentre in $right_5$ otteniamo la seconda metà del risultato, shiftato però circolarmente di una posizione a sinistra. Lo shift è fatto anche sulla parte sinistra ottenendo così i blocchi che vanno in input alla successiva fase: le 16 iterazioni. Questa rappresentazione del risultato della permutazione IP tramite i due blocchi $leftt'_5$ (è la sequenza ottenuta alla fine del quinto passo, dopo aver effettuato lo shift circolare) e $right_5$ risulterà utile per calcolare l'espansione E.

4.3 La realizzazione delle iterazioni

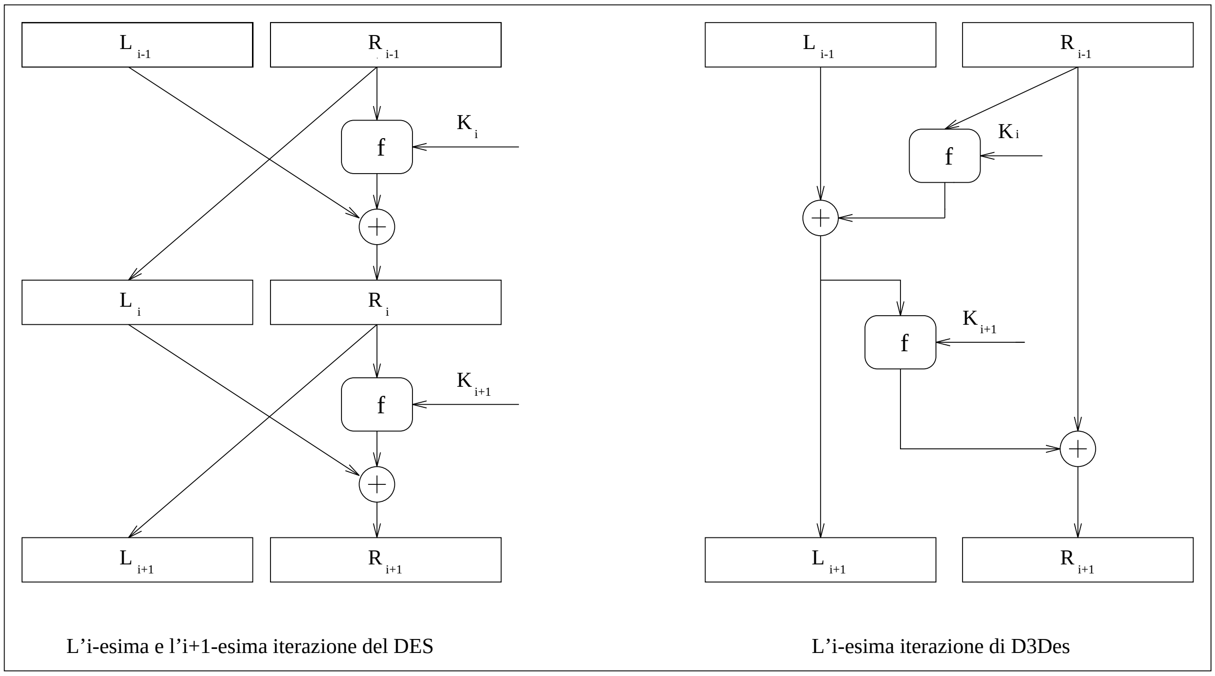

Figura 1: Confronto tra le iterazioni del DES e dell'implementazione D3Des

Dato che in ciascuna delle 16 iterazioni solo metà dei 64 bit vengono modificati, l'implementazione effettua solo 8 iterazioni in ciascuna delle quali vengono sviluppati 2 round che permettono di realizzare due successive iterazioni del DES (nella Figura 1 è mostrato il confronto). Riportiamo di seguito il frammento di codice interessato:

```
for( round = 0; round < 8; round++ ) {
/*
** primo round **
*/
work  = (right << 28) | (right >> 4);
/* work=xxE1xxE3xxE5xxE7 */
work ^= *keys++;
fval  = SP7[ work               & 0x3fL];
fval |= SP5[(work >>  8) & 0x3fL];
fval |= SP3[(work >> 16) & 0x3fL];
fval |= SP1[(work >> 24) & 0x3fL];
work  = right ^ *keys++;
/* right=xxE2xxE4xxE6xxE8 */
fval |= SP8[ work               & 0x3fL];
fval |= SP6[(work >>  8) & 0x3fL];
fval |= SP4[(work >> 16) & 0x3fL];
fval |= SP2[(work >> 24) & 0x3fL];
leftt ^= fval;
/*
** secondo round **
```

```
*/
work  = (leftt << 28) | (leftt >> 4);
/* work=xxE1xxE3xxE5xxE7 */
work ^= *keys++;
fval  = SP7[ work                & 0x3fL];
fval |= SP5[(work >>  8) & 0x3fL];
fval |= SP3[(work >> 16) & 0x3fL];
fval |= SP1[(work >> 24) & 0x3fL];
/* leftt=xxE2xxE4xxE6xxE8 */
work  = leftt ^ *keys++;
fval |= SP8[ work                & 0x3fL];
fval |= SP6[(work >>  8) & 0x3fL];
fval |= SP4[(work >> 16) & 0x3fL];
fval |= SP2[(work >> 24) & 0x3fL];
right ^= fval;
}
```

L'analisi di questa fase verrà fatta in 2 parti analizzando prima come avviene l'espansione E e l'indirizzamento alle S-box poi come è realizzato il calcolo dell'output delle S-box e la permutazione P.

4.3.1 Espansione E ed indirizzamento alle S-box

Ricordiamo che per il calcolo della funzione $f(R_{i-1}, K_i)$ prima R_{i-1} è espansa nel blocco di 48 bit $E(R_{i-1})$ usando la matrice di selezione dei bit E, poi è fatto l'*or* esclusivo dell'espansione con la sottochiave corrispondente ottenendo i blocchi di 6 bit ciascuno $B_1 \cdots B_8$, input delle S-box. Se la sequenza di bit R_{i-1} passata a f è $r_{01}r_{02}\cdots r_{31}r_{32}$ e la sottochiave corrispondente è $K_i = K_{i,01}K_{i,02}\cdots K_{i,47}K_{i,48}$ allora l'input alle S-box è :

$$
\begin{aligned}
B_1 &= \begin{pmatrix} Ki_{01}\cdots Ki_{06} \\ \oplus \\ r_{32}r_{01}\cdots r_{05} \end{pmatrix} & B_2 &= \begin{pmatrix} Ki_{07}\cdots Ki_{12} \\ \oplus \\ r_{04}r_{05}\cdots r_{09} \end{pmatrix} \\
B_3 &= \begin{pmatrix} Ki_{13}\cdots Ki_{18} \\ \oplus \\ r_{08}r_{09}\cdots r_{13} \end{pmatrix} & B_4 &= \begin{pmatrix} Ki_{19}\cdots Ki_{24} \\ \oplus \\ r_{12}r_{13}\cdots r_{17} \end{pmatrix} \\
B_5 &= \begin{pmatrix} Ki_{25}\cdots Ki_{30} \\ \oplus \\ r_{16}r_{17}\cdots r_{21} \end{pmatrix} & B_6 &= \begin{pmatrix} Ki_{31}\cdots Ki_{36} \\ \oplus \\ r_{20}r_{21}\cdots r_{25} \end{pmatrix} \\
B_7 &= \begin{pmatrix} Ki_{37}\cdots Ki_{42} \\ \oplus \\ r_{24}r_{25}\cdots r_{29} \end{pmatrix} & B_8 &= \begin{pmatrix} Ki_{43}\cdots Ki_{48} \\ \oplus \\ r_{28}\cdots r_{32}r_{01} \end{pmatrix}.
\end{aligned}
\tag{2}
$$

A partire da questi si calcola $f(R_{i-1}, K_i) = P(S_1(B_1) \cdots S_8(B_8))$ dove $S_i(B_i)$ è l'output della S-box i e $P(x)$ realizza la permutazione P. Nell'implementazione che stiamo analizzando lo shift circolare effettuato sulle due metà dell'output della permutazione iniziale IP fa si che in ciascun round delle 8 iterazioni l'espansione E possa essere fatta velocemente, anche se in due parti.
Chiamiamo R la sequenza di bit su cui si applica f: corrisponde a *right* nel primo round e a *leftt* nel secondo. Grazie alla rappresentazione di IP a partire dalla prima iterazione la sequenza R è

$$r_{02}r_{03}\mathbf{r_{04}r_{05}r_{06}r_{07}r_{08}r_{09}} \cdots r_{26}r_{27}\mathbf{r_{28}r_{29}r_{30}r_{31}r_{32}r_{01}}$$

e può essere vista nella forma

$$xxE_2xxE_4xxE_6xxE_8 \tag{3}$$

dove E_i è costituito dai 6 bit di R che si trovano nelle posizioni individuate dall'i-esima riga di E, cioè $E(R) = E_1E_2E_3E_4E_5E_6E_7E_8$. Dalla matrice E riportata nell'Appendice A si rileva che $E(R)$ equivale a:

$$r_{32}r_{01}r_{02}r_{03}r_{04}r_{05}r_{04}r_{05}r_{06}r_{07}r_{08}r_{09} \cdots r_{24}r_{25}r_{26}r_{27}r_{28}r_{29}r_{28}r_{29}r_{30}r_{31}r_{32}r_{01}$$

In più effettuando su di R uno shift circolare a destra di 4 posizioni otteniamo la sequenza

$$r_{30}r_{31}\mathbf{r_{32}r_{01}r_{02}r_{03}r_{04}r_{05}r_{06}} \cdots r_{22}r_{23}\mathbf{r_{24}r_{25}r_{26}r_{27}r_{28}r_{29}}$$

che può essere vista come

$$xxE_1xxE_3xxE_5xxE_7 \tag{4}$$

Sfruttando questa struttura l'indirizzamento alle S-box viene calcolato in due passi: prima si calcola l'input alle S-box di indice dispari poi l'input alle S-box di indice pari. Gli output sono mantenuti in *fval* che contiene alla fine dei 2 passi il valore di f. Per ottenere gli input alle S-box di indice dispari si effettua l'*or* esclusivo tra la variabile *work*, contenente l'espressione (4) grazie alla prima istruzione di ogni round, e la prima parte della sottochiave corrispondente.
Ricordiamo che ogni sottochiave è contenuta in 2 long nella forma $00L_100L_300R_100R_3$ e $00L_200L_400R_200R_4$ (vedi Sezione 4.1) e che gli input alle S-box sono dati dalla (2). Si ottiene quindi

$$work = xx(L_1 \oplus E_1)xx(L_3 \oplus E_3)xx(R_1 \oplus E_5)xx(R_3 \oplus E_7).$$

In *work* sono ora contenuti gli input per le S-box 1,3,5,7.
Successivamente calcolando in *work* l'*or* esclusivo tra R, che contiene l'espressione (3), e la seconda parte della sottochiave, *work* sarà nella forma:

$$work = xx(L_2 \oplus E_2)xx(L_4 \oplus E_4)xx(R_2 \oplus E_6)xx(R_4 \oplus E_8)$$

e quindi conterrà gli input per le S-box 2,4,6,8.

4.3.2 Calcolo delle S-box e della permutazione P

Come già anticipato questa implementazione struttura le S-box nel modo descritto nel paragrafo 2.2.1. Viene fatta però una variazione resa necessaria dal modo con cui è stato rappresentato l'output della permutazione IP. Si vogliono mantenere in tutte le iterazioni gli effetti (velocità del calcolo di $E(R)$) dello shift circolare di una posizione a sinistra fatto sulle due metà dell'output di IP. Per fare ciò i valori in ciascuna entrata delle tabelle SPBOX usate da D3Des risultano essere shiftati circolarmente di una posizione a sinistra rispetto ai valori di cui è stata descritta la costruzione.
L'esempio che segue mostra come avviene la costruzione delle SPBOX usate da D3Des che segue, ad eccezione dell'ultimo passo, la costruzione descritta nell'Esempio 2.1.

Esempio 4.1

Consideriamo il caso in cui l'input della S-box3 è 101100_2. *Questo implica che dobbiamo prendere come output l'elemento della S-box3 che si trova nella colonna 7 e nella riga 3, S-box3[2][6]=*0011_2. *Nel D3Des a partire da* 101100_2 *accedendo alla SPBOX3 alla locazione 44* ($101100_2 = 44_{10}$) *otteniamo*

SPBOX3[1]*[44]=0000 1000 0000 0000 0000 0000 0000 1000*$_2$

Vediamo perchè SPBOX3[44] contiene tale valore. Dato che

S-box3[2][6]=0011$_2$

e ricordando il modo in con cui sono costruite le 8 S-box questo bisogna innanzitutto vederlo all'interno di una stringa di 32 bit cioè :

0000 0000 0011 0000 0000 0000 0000 0000

poi permutato tramite P ottenendo :

0000 0100 0000 0000 0000 0000 0000 0100

che shiftato di una posizione a sinistra corrisponde a SPBOX3[44].

4.4 Calcolo di IP^{-1}

Il calcolo di IP^{-1} è realizzato seguendo uno schema del tutto analogo a quello di IP. Viene effettuata a partire dai valori $leftt$ e $right$ che si hanno alla fine delle 8 iterazioni. Il blocco di istruzioni che segue realizza la permutazione tenendo conto che: prima occorre effettuare lo scambio a 32 bit e quindi permutare la sequenza $R_{16}L_{16}$ (nell'implementazione è la sequenza $right\ leftt$). Ricordiamo che le sequenze da permutare sono state ottenute passando via via per le SPBOX i cui output sono shiftati di una posizione a sinistra. Questo fa si che alla fine $leftt$ e $right$ risultano essere shiftati a sinistra di una posizione rispetto alle sequenze L_{16} e R_{16} dell'algoritmo DES. Quindi per poter ottenere l'output di IP^{-1} ci sono due modi: shiftare ciclicamente le sequenze $leftt$ e $right$ di una posizione a destra e applicare a $right\ leftt$ IP^{-1} oppure, cosa che qui viene fatta, cambiare la matrice di permutazione che diventa

[1] nel codice di D3Des il riferimento a SPBOXi è fatto con SPi

$newIP^{-1}$							
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25
32	64	40	8	48	16	56	24

In questo modo prendere come primo elemento dell'output di IP^{-1} il 40-esimo della sequenza $R_{16}L_{16}$ come indicato dall'algoritmo DES equivale a prendere come primo elemento il 39-esimo della sequenza $right\ leftt$. Cosicché otteniamo come risultato le sequenze:

$$block[0] = l_7r_7l_{15}r_{15}l_{23}r_{23}l_{31}r_{31}l_6r_6l_{14}r_{14}l_{22}r_{22}l_{30}r_{30}l_5r_5l_{13}r_{13}l_{21}r_{21}l_{29}r_{29}l_4r_4l_{12}r_{12}l_{20}r_{20}l_{28}r_{28} \quad (5)$$

$$block[1] = l_3r_3l_{11}r_{11}l_{19}r_{19}l_{27}r_{27}l_2r_2l_{10}r_{10}l_{18}r_{18}l_{26}r_{26}l_1r_1l_9r_9l_{17}r_{17}l_{25}r_{25}l_{32}r_{32}l_8r_8l_{16}r_{16}l_{24}r_{24}. \quad (6)$$

$block[0]$ e $block[1]$ sono due long usati dall'implementazione per contenere l'output di $newIP^{-1}$. In maniera analoga a quanto fatto per la permutazione IP, l'analisi viene fatta mostrando come ad ogni passo della realizzazione vengono scambiati dei bit attraverso una rappresentazione tabulare delle variabili che intervengono: $leftt$, $right$, $work$. A questo scopo si riporta di seguito il codice corrispondente e la rappresentazioni delle variabili.

```
right = (right << 31) | (right >> 1);
/* Primo Passo */
work = (leftt ^ right) & 0xaaaaaaaaL;
leftt ^= work;
right ^= work;
leftt = (leftt << 31) | (leftt >> 1);
/* Secondo Passo */
work = ((leftt >> 8) ^ right) & 0x00ff00ffL;
right ^= work;
leftt ^= (work << 8);
/* Terzo Passo */
work = ((leftt >> 2) ^ right) & 0x33333333L;
right ^= work;
leftt ^= (work << 2);
/* Quarto Passo */
work = ((right >> 16) ^ leftt) & 0x0000ffffL;
leftt ^= work;
right ^= (work << 16);
/* Quinto Passo */
work = ((right >> 4) ^ leftt) & 0x0f0f0f0fL;
leftt ^= work;
right ^= (work << 4);

*block++ = right;
```

```
*block = leftt;
```

Nella rappresentazione $leftt_i$ e $right_i$, $i = 1, \cdots, 5$ rappresentano le sequenze ottenute alla fine dell'i-esimo passo della realizzazione della permutazione, $work_i$ è la sequenza di 32 bit usata per realizzarlo, $leftt_0$ e $right_0$ rappresentano la sequenza iniziale, quella su cui è effettuata la permutazione.

$leftt_0$	$right_0$
l_{01}	r_{32}
l_{02}	r_{01}
l_{03}	r_{02}
l_{04}	r_{03}
l_{05}	r_{04}
l_{06}	r_{05}
l_{07}	r_{06}
l_{08}	r_{07}
l_{09}	r_{08}
l_{10}	r_{09}
l_{11}	r_{10}
l_{12}	r_{11}
l_{13}	r_{12}
l_{14}	r_{13}
l_{15}	r_{14}
l_{16}	r_{15}
l_{17}	r_{16}
l_{18}	r_{17}
l_{19}	r_{18}
l_{20}	r_{19}
l_{21}	r_{20}
l_{22}	r_{21}
l_{23}	r_{22}
l_{24}	r_{23}
l_{25}	r_{24}
l_{26}	r_{25}
l_{27}	r_{26}
l_{28}	r_{27}
l_{29}	r_{28}
l_{30}	r_{29}
l_{31}	r_{30}
l_{32}	r_{31}

$work_1$	$leftt_1$	$right_1$
$l_{01} \oplus r_{32}$	l_{32}	l_{01}
0	r_{32}	r_{01}
$l_{03} \oplus r_{02}$	l_{02}	l_{03}
0	r_{02}	r_{03}
$l_{05} \oplus r_{04}$	l_{04}	l_{05}
0	r_{04}	r_{05}
$l_{07} \oplus r_{06}$	l_{06}	l_{07}
0	r_{06}	r_{07}
$l_{09} \oplus r_{08}$	l_{08}	l_{09}
0	r_{08}	r_{09}
$l_{11} \oplus r_{10}$	l_{10}	l_{11}
0	r_{10}	r_{11}
$l_{13} \oplus r_{12}$	l_{12}	l_{13}
0	r_{12}	r_{13}
$l_{15} \oplus r_{14}$	l_{14}	l_{15}
0	r_{14}	r_{15}
$l_{17} \oplus r_{16}$	l_{16}	l_{17}
0	r_{16}	r_{17}
$l_{19} \oplus r_{18}$	l_{18}	l_{19}
0	r_{18}	r_{19}
$l_{21} \oplus r_{20}$	l_{20}	l_{21}
0	r_{20}	r_{21}
$l_{23} \oplus r_{22}$	l_{22}	l_{23}
0	r_{22}	r_{23}
$l_{25} \oplus r_{24}$	l_{24}	l_{25}
0	r_{24}	r_{25}
$l_{27} \oplus r_{26}$	l_{26}	l_{27}
0	r_{26}	r_{27}
$l_{29} \oplus r_{28}$	l_{28}	l_{29}
0	r_{28}	r_{29}
$l_{31} \oplus r_{30}$	l_{30}	l_{31}
0	r_{30}	r_{31}

$work_2$	$leftt_2$	$right_2$
0	l_{09}	l_{01}
0	r_{09}	r_{01}
0	l_{11}	l_{03}
0	r_{11}	r_{03}
0	l_{13}	l_{05}
0	r_{13}	r_{05}
0	l_{15}	l_{07}
0	r_{15}	r_{07}
$l_{32} \oplus l_{09}$	l_{08}	l_{32}
$r_{32} \oplus r_{09}$	r_{08}	r_{32}
$l_{02} \oplus l_{11}$	l_{10}	l_{02}
$r_{02} \oplus r_{11}$	r_{10}	r_{02}
$l_{04} \oplus l_{13}$	l_{12}	l_{04}
$r_{04} \oplus r_{13}$	r_{12}	r_{04}
$l_{06} \oplus l_{15}$	l_{14}	l_{06}
$r_{06} \oplus r_{15}$	r_{14}	r_{06}
0	l_{25}	l_{17}
0	r_{25}	r_{17}
0	l_{27}	l_{19}
0	r_{27}	r_{19}
0	l_{29}	l_{21}
0	r_{29}	r_{21}
0	l_{31}	l_{23}
0	r_{31}	r_{23}
$l_{16} \oplus l_{25}$	l_{24}	l_{16}
$r_{16} \oplus r_{25}$	r_{24}	r_{16}
$l_{18} \oplus l_{27}$	l_{26}	l_{18}
$r_{18} \oplus r_{27}$	r_{26}	r_{18}
$l_{20} \oplus l_{29}$	l_{28}	l_{20}
$r_{20} \oplus r_{29}$	r_{28}	r_{20}
$l_{22} \oplus l_{31}$	l_{30}	l_{22}
$r_{22} \oplus r_{31}$	r_{30}	r_{22}

$leftt_1$ è la sequenze ottenuta dopo aver effettuato lo shift circolare alla fine del quarto passo della realizzazione.

$work_3$	$leftt_3$	$right_3$
0	l_{03}	l_{01}
0	r_{03}	r_{01}
$l_{03} \oplus l_{09}$	l_{11}	l_{09}
$r_{03} \oplus r_{09}$	r_{11}	r_{09}
0	l_{07}	l_{05}
0	r_{07}	r_{05}
$l_{07} \oplus l_{13}$	l_{15}	l_{13}
$r_{07} \oplus r_{13}$	r_{15}	r_{13}
0	l_{02}	l_{32}
0	r_{02}	r_{32}
$l_{02} \oplus l_{08}$	l_{10}	l_{08}
$r_{02} \oplus r_{08}$	r_{10}	r_{08}
0	l_{06}	l_{04}
0	r_{06}	r_{04}
$l_{06} \oplus l_{12}$	l_{14}	l_{12}
$r_{06} \oplus r_{12}$	r_{14}	r_{12}
0	l_{19}	l_{17}
0	r_{19}	r_{17}
$l_{19} \oplus l_{25}$	l_{27}	l_{25}
$r_{19} \oplus r_{25}$	r_{27}	r_{25}
0	l_{23}	l_{21}
0	r_{23}	r_{21}
$l_{23} \oplus l_{29}$	l_{31}	l_{29}
$r_{23} \oplus r_{29}$	r_{31}	r_{29}
0	l_{18}	l_{16}
0	r_{18}	r_{16}
$l_{18} \oplus l_{24}$	l_{26}	l_{24}
$r_{18} \oplus r_{24}$	r_{26}	r_{24}
0	l_{22}	l_{20}
0	r_{22}	r_{20}
$l_{22} \oplus l_{28}$	l_{30}	l_{28}
$r_{22} \oplus r_{28}$	r_{30}	r_{28}

$work_4$	$leftt_4$	$right_4$
0	l_{03}	l_{19}
0	r_{03}	r_{19}
0	l_{11}	l_{27}
0	r_{11}	r_{27}
0	l_{07}	l_{23}
0	r_{07}	r_{23}
0	l_{15}	l_{31}
0	r_{15}	r_{31}
0	l_{02}	l_{18}
0	r_{02}	r_{18}
0	l_{10}	l_{26}
0	r_{10}	r_{26}
0	l_{06}	l_{22}
0	r_{06}	r_{22}
0	l_{14}	l_{30}
0	r_{14}	r_{30}
$l_{01} \oplus l_{19}$	l_{01}	l_{17}
$r_{01} \oplus r_{19}$	r_{01}	r_{17}
$l_{09} \oplus l_{27}$	l_{09}	l_{25}
$r_{09} \oplus r_{27}$	r_{09}	r_{25}
$l_{05} \oplus l_{23}$	l_{05}	l_{21}
$r_{05} \oplus r_{23}$	r_{05}	r_{21}
$l_{13} \oplus l_{31}$	l_{13}	l_{29}
$r_{13} \oplus r_{31}$	r_{13}	r_{29}
$l_{32} \oplus l_{18}$	l_{32}	l_{16}
$r_{32} \oplus r_{18}$	r_{32}	r_{16}
$l_{08} \oplus l_{26}$	l_{08}	l_{24}
$r_{08} \oplus r_{26}$	r_{08}	r_{24}
$l_{04} \oplus l_{22}$	l_{04}	l_{20}
$r_{04} \oplus r_{22}$	r_{04}	r_{20}
$l_{12} \oplus l_{30}$	l_{12}	l_{28}
$r_{12} \oplus r_{30}$	r_{12}	r_{28}

$work_5$	$leftt_5$	$right_5$
0	l_{03}	l_{07}
0	r_{03}	r_{07}
0	l_{11}	l_{15}
0	r_{11}	r_{15}
$l_{07} \oplus l_{19}$	l_{19}	l_{23}
$r_{07} \oplus r_{19}$	r_{19}	r_{23}
$l_{15} \oplus l_{27}$	l_{27}	l_{31}
$r_{15} \oplus l_{27}$	r_{27}	r_{31}
0	l_{02}	l_{06}
0	r_{02}	r_{06}
0	l_{10}	l_{14}
0	r_{10}	r_{14}
$l_{06} \oplus l_{18}$	l_{18}	l_{22}
$r_{06} \oplus r_{18}$	r_{18}	r_{22}
$l_{14} \oplus l_{26}$	l_{26}	l_{30}
$r_{14} \oplus r_{26}$	r_{26}	r_{30}
0	l_{01}	l_{05}
0	r_{01}	r_{05}
0	l_{09}	l_{13}
0	r_{09}	r_{13}
$l_{05} \oplus l_{17}$	l_{17}	l_{21}
$r_{05} \oplus r_{17}$	r_{17}	r_{21}
$l_{13} \oplus l_{25}$	l_{25}	l_{29}
$r_{13} \oplus r_{25}$	r_{25}	l_{29}
0	l_{32}	l_{04}
0	r_{32}	r_{04}
0	l_{08}	l_{12}
0	r_{08}	r_{12}
$l_{04} \oplus l_{16}$	l_{16}	l_{20}
$r_{04} \oplus r_{16}$	r_{16}	r_{20}
$l_{12} \oplus l_{24}$	l_{24}	l_{28}
$r_{12} \oplus r_{24}$	r_{24}	r_{28}

Le ultime due istruzioni del codice dato ci portano ad ottenere le espressioni (5) e (6).

5 Analisi Libdes

L'altra implementazione del DES che analizziamo è Libdes, l'implementazione più veloce con D3Des.
Prima di addentrarci nell'analisi dobbiamo osservare che in tale implementazione la generica stringa di n bit $s_1 s_2 \cdots s_n$ verrà memorizzata nella variabile da destra verso sinistra: $s_n s_{n-1} \cdots s_1$. I valori che n può assumere sono 8, 16 e 32.

5.1 Calcolo di IP, IP^{-1} e $PC-1$

Vediamo come sono effettuate le permutazioni IP, IP^{-1} e $PC-1$. Riportiamo innanzitutto le relative linee di codice :

Permutazione IP

```
#define IP(l,r,t) \
PERM_OP(r,l,t, 4,0x0f0f0f0f); \
PERM_OP(l,r,t,16,0x0000ffff); \
PERM_OP(r,l,t, 2,0x33333333); \
PERM_OP(l,r,t, 8,0x00ff00ff); \
PERM_OP(r,l,t, 1,0x55555555);
```

Permutazione IP^{-1}

```
#define FP(l,r,t) \
PERM_OP(l,r,t, 1,0x55555555); \
PERM_OP(r,l,t, 8,0x00ff00ff); \
PERM_OP(l,r,t, 2,0x33333333); \
PERM_OP(r,l,t,16,0x0000ffff); \
PERM_OP(l,r,t, 4,0x0f0f0f0f);
```

Permutazione PC-1

```
PERM_OP (d,c,t,4,0x0f0f0f0f);
HPERM_OP(c,t,-2,0xcccc0000);
HPERM_OP(d,t,-2,0xcccc0000);
PERM_OP (d,c,t,1,0x55555555);
PERM_OP (c,d,t,8,0x00ff00ff);
PERM_OP (d,c,t,1,0x55555555);
d=      (((d&0x000000ff)<<16)| (d&0x0000ff00)     |
 ((d&0x00ff0000)>>16)|((c&0xf0000000)>>4));
c&=0x0fffffff;
```

Le funzioni `PERM_OP(a,b,t,n,m)` e `HPERM_OP(a,t,n,m)` sono definite così :

```
#define PERM_OP(a,b,t,n,m) ((t)=((((a)>>(n))^(b))&(m)),\        (i)
(b)^=(t),\                                                      (ii)
(a)=((a)^((t)<<(n))))                                           (iii)

#define HPERM_OP(a,t,n,m) ((t)=((((a)<<(16-(n)))^(a))&(m)),\
(a)=(a)^(t)^(t>>(16-(n))))
```

`HPERM_OP(a,t,-2,0xcccc0000)` effettua lo scambio di posizione dei bit della variabile $a = a_1 \cdots a_{32}$, scambiando il bit a_i con il bit a_{i+18}, $\ i = 1, 2, 5, 6, 9, 10, 13, 14$, come si puó rilevare dalla Tabella 3.

$(a \ll 18) \oplus a$	t	$t \gg 18$	a	$a \oplus t \oplus (t \gg 18)$
$a_{14} \oplus a_{32}$	$a_{14} \oplus a_{32}$	0	a_{32}	a_{14}
$a_{13} \oplus a_{31}$	$a_{13} \oplus a_{31}$	0	a_{31}	a_{13}
$a_{12} \oplus a_{30}$	0	0	a_{30}	a_{30}
$a_{11} \oplus a_{29}$	0	0	a_{29}	a_{29}
$a_{10} \oplus a_{28}$	$a_{10} \oplus a_{28}$	0	a_{28}	a_{10}
$a_{09} \oplus a_{27}$	$a_{09} \oplus a_{27}$	0	a_{27}	a_{09}
$a_{08} \oplus a_{26}$	0	0	a_{26}	a_{26}
$a_{07} \oplus a_{25}$	0	0	a_{25}	a_{25}
$a_{06} \oplus a_{24}$	$a_{06} \oplus a_{24}$	0	a_{24}	a_{06}
$a_{05} \oplus a_{23}$	$a_{05} \oplus a_{23}$	0	a_{23}	a_{05}
$a_{04} \oplus a_{22}$	0	0	a_{22}	a_{22}
$a_{03} \oplus a_{21}$	0	0	a_{21}	a_{21}
$a_{02} \oplus a_{20}$	$a_{02} \oplus a_{20}$	0	a_{20}	a_{02}
$a_{01} \oplus a_{19}$	$a_{01} \oplus a_{19}$	0	a_{19}	a_{01}
a_{18}	0	0	a_{18}	a_{18}
a_{17}	0	0	a_{17}	a_{17}
a_{16}	0	0	a_{16}	a_{16}
a_{15}	0	0	a_{15}	a_{15}
a_{14}	0	$a_{14} \oplus a_{32}$	a_{14}	a_{32}
a_{13}	0	$a_{13} \oplus a_{31}$	a_{13}	a_{31}
a_{12}	0	0	a_{12}	a_{12}
a_{11}	0	0	a_{11}	a_{11}
a_{10}	0	$a_{10} \oplus a_{28}$	a_{10}	a_{28}
a_{09}	0	$a_{09} \oplus a_{27}$	a_{09}	a_{27}
a_{08}	0	0	a_{08}	a_{08}
a_{07}	0	0	a_{07}	a_{07}
a_{06}	0	$a_{06} \oplus a_{24}$	a_{06}	a_{24}
a_{05}	0	$a_{05} \oplus a_{23}$	a_{05}	a_{23}
a_{04}	0	0	a_{04}	a_{04}
a_{03}	0	0	a_{03}	a_{03}
a_{02}	0	$a_{02} \oplus a_{20}$	a_{02}	a_{20}
a_{01}	0	$a_{01} \oplus a_{19}$	a_{01}	a_{19}

– Tabella 3 –

Per capire invece cosa fà `PERM_OP(a,b,t,n,m)` analizziamo la sequenza di operazioni che effettua sui singoli bit.

La (ii) tenendo conto della (i) è :

$$b_i \leftarrow b_i \oplus ((a_i^{'} \oplus b_i) \wedge m_i) \quad i = 1, \cdots, 32 \ ; \ \text{dove} \qquad a_i^{'} = \begin{cases} a_{i+n} & i = 1, \cdots, 32 - n; \\ 0 & i = 32 - n + 1, \cdots, 32. \end{cases}$$

Quindi

$$b_i \quad \leftarrow \quad \begin{cases} b_i & \text{se} \quad m_i = 0; \\ a_i^{'} & \text{se} \quad m_i = 1. \end{cases} \tag{7}$$

La (iii) equivale a:

$$a_i \quad \leftarrow \quad \begin{cases} a_i \oplus ((a_i \oplus b_i^{'}) \wedge m_i^{'}) & i = n + 1, \cdots, 32; \\ a_i & i = 1, \cdots, n; \end{cases} \tag{8}$$

dove

$$\begin{aligned} b_i^{'} &= b_{i-n} \qquad \text{per} \quad i = n + 1, \cdots, 32 \ ; \\ m_i^{'} &= m_{i-n} \qquad \text{per} \quad i = n + 1, \cdots, 32 \ ; \end{aligned} \tag{9}$$

e la (8) equivale per $i = n + 1, \cdots, 32$ a:

$$a_i \quad \leftarrow \quad \begin{cases} a_i & \text{se } m_i^{'} = 0; \\ b_i^{'} & \text{se } m_i^{'} = 1. \end{cases} \tag{10}$$

Quindi si può notare come `PERM_OP` effettua uno scambio di bit tra a e b in funzione della variabile mask m e del valore di n.

Vediamo adesso come viene effettuata la permutazione IP. Seguiamo passo passo come variano i bit dei due long l e r, che inizialmente contengono i 64 bit della variabile $x = x_{01}x_{02} \cdots x_{64}$ da permutare, man mano che vengono effettuate le `PERM_OP` che realizzano IP.
Innanzitutto, prima che venga effettuata $IP(l, r, t)$ i bit di x sono distribuiti su l e r come sotto riportato:

$$l = x_{25} \cdots x_{32} x_{17} \cdots x_{24} x_{09} \cdots x_{16} x_{01} \cdots x_{08}$$

$$r = x_{57} \cdots x_{64} x_{49} \cdots x_{56} x_{41} \cdots x_{48} x_{33} \cdots x_{40}.$$

Vediamo adesso come variano l e r nella funzione `PERM_OP(r,l,t, 4,0x0f0f0f0f)`.

I bit della variabile mask m sono:

$$m_i \quad = \quad \begin{cases} 1 & \text{se} \quad i = 1, \cdots, 4, 9, \cdots, 12, 17, \cdots, 20, 25, \cdots, 28; \\ 0 & \text{se} \quad i = 5, \cdots, 8, 13, \cdots, 16, 21, \cdots, 24, 29, \cdots, 32. \end{cases}$$

Dalla (7) otteniamo

$$l_i \quad \leftarrow \quad \begin{cases} l_i & \text{se} \quad i = 5, \cdots, 8, 13, \cdots, 16, 21, \cdots, 24, 29, \cdots 32; \\ r_i^{'} & \text{se} \quad i = 1, \cdots, 4, 9, \cdots, 12, 17, \cdots, 20, 25, \cdots 28. \end{cases}$$

Dalla (9) otteniamo che

$$m_i^{'} \quad = \quad \begin{cases} 0 & \text{se} \quad i = 9, \cdots, 12, 17, \cdots, 20, 25, \cdots, 28; \\ 1 & \text{se} \quad i = 5, \cdots, 8, 13, \cdots, 16, 21, \cdots, 24, 29, \cdots, 32. \end{cases}$$

Dalle (8) e (10) segue che:

$$r_i \quad \leftarrow \quad \begin{cases} r_i & \text{se} \quad i = 1, \cdots, 4, 9, \cdots, 12, 17, \cdots, 20, 25, \cdots, 28; \\ l_i^{'} = l_{i+4} & \text{se} \quad i = 5, \cdots, 8, 13, \cdots, 16, 21, \cdots, 24, 29, \cdots, 32. \end{cases}$$

Otteniamo così infine

$$l = x_{25} \cdots x_{28} x_{57} \cdots x_{60} x_{17} \cdots x_{20} x_{49} \cdots x_{52} x_{09} \cdots x_{12} x_{41} \cdots x_{44} x_{01} \cdots x_{04} x_{33} \cdots x_{36};$$

$$r = x_{29} \cdots x_{32} x_{61} \cdots x_{64} x_{21} \cdots x_{24} x_{53} \cdots x_{56} x_{13} \cdots x_{16} x_{45} \cdots x_{48} x_{05} \cdots x_{08} x_{37} \cdots x_{40}.$$

Analogo è il funzionamento delle altre `PERM_OP`:

Nella funzione `PERM_OP(r,l,t, 16,0x0000ffff)` si ottiene:

$$l = x_{13} \cdots x_{16} x_{45} \cdots x_{48} x_{05} \cdots x_{08} x_{37} \cdots x_{40} x_{09} \cdots x_{12} x_{41} \cdots x_{44} x_{01} \cdots x_{04} x_{33} \cdots x_{36} \;;$$
$$r = x_{29} \cdots x_{32} x_{61} \cdots x_{64} x_{21} \cdots x_{24} x_{53} \cdots x_{56} x_{25} \cdots x_{28} x_{57} \cdots x_{60} x_{17} \cdots x_{20} x_{49} \cdots x_{52} \;;$$

Nella funzione `PERM_OP(r,l,t, 2,0x33333333)` si ottiene:

$$\begin{aligned} l \quad = \quad & x_{13}x_{14}x_{29}x_{30}x_{45}x_{46}x_{61}x_{62}x_{05}x_{06}x_{21}x_{22}x_{37}x_{38}x_{53}x_{54} \\ & x_{09}x_{10}x_{25}x_{26}x_{41}x_{42}x_{57}x_{58}x_{01}x_{02}x_{17}x_{18}x_{33}x_{34}x_{49}x_{50} \;; \end{aligned}$$

$$\begin{aligned} r \quad = \quad & x_{15}x_{16}x_{31}x_{32}x_{47}x_{48}x_{63}x_{64}x_{07}x_{08}x_{23}x_{24}x_{39}x_{40}x_{55}x_{56} \\ & x_{11}x_{12}x_{27}x_{28}x_{43}x_{44}x_{59}x_{60}x_{03}x_{04}x_{19}x_{20}x_{35}x_{36}x_{51}x_{52} \;; \end{aligned}$$

Nella funzione `PERM_OP(l,r,t, 8,0x00ff00ff)` si ottiene:

$$l = x_{07}x_{08}x_{23}x_{24}x_{39}x_{40}x_{55}x_{56}x_{05}x_{06}x_{21}x_{22}x_{37}x_{38}x_{53}x_{54} x_{03}x_{04}x_{19}x_{20}x_{35}x_{36}x_{51}x_{52}x_{01}x_{02}x_{17}x_{18}x_{33}x_{34}x_{49}x_{50} ;$$

$$r = x_{15}x_{16}x_{31}x_{32}x_{47}x_{48}x_{63}x_{64}x_{13}x_{14}x_{29}x_{30}x_{45}x_{46}x_{61}x_{62} x_{11}x_{12}x_{27}x_{28}x_{43}x_{44}x_{59}x_{60}x_{09}x_{10}x_{25}x_{26}x_{41}x_{42}x_{57}x_{58} ;$$

Nella funzione `PERM_OP(r,l,t, 1,0x55555555)` si ottiene:

$$l = x_{07}x_{15}x_{23}x_{31}x_{39}x_{47}x_{55}x_{63}x_{05}x_{13}x_{21}x_{29}x_{37}x_{45}x_{53}x_{61} x_{03}x_{11}x_{19}x_{27}x_{35}x_{43}x_{51}x_{59}x_{01}x_{09}x_{17}x_{25}x_{33}x_{41}x_{49}x_{57} ;$$

$$r = x_{08}x_{16}x_{24}x_{32}x_{40}x_{48}x_{56}x_{64}x_{06}x_{14}x_{22}x_{30}x_{38}x_{46}x_{54}x_{62} x_{04}x_{12}x_{20}x_{28}x_{36}x_{44}x_{52}x_{60}x_{02}x_{10}x_{18}x_{26}x_{34}x_{42}x_{50}x_{58} .$$

Considerato inoltre che in ogni `PERM_OP` si hanno 3 assegnazioni, 3 *or* esclusivi, 2 shift e 1 and, mentre in `HPERM_OP` si hanno 1 assegnazione, 2 shift, 3 *or* esclusivi e 1 *and* allora si ha che IP e IP^{-1} vengono realizzate ognuna con 45 operazioni. Invece $PC-1$ è realizzata con 65 operazioni.

5.2 Schedulazione delle chiavi

In questa sezione analizziamo come l'implementazione effettua il calcolo delle 16 sottochiavi. Riportiamo le relative linee di codice:

```
for (i=0; i<ITERATIONS; i++)
{
if (shifts2[i])
{ c=((c>>2)|(c<<26)); d=((d>>2)|(d<<26)); }
else
{ c=((c>>1)|(c<<27)); d=((d>>1)|(d<<27)); }
c&=0x0fffffff;
d&=0x0fffffff;
s=      des_skb[0][ (c    )&0x3f                ]|
des_skb[1][((c>> 6)&0x03)|((c>> 7)&0x3c)]|
des_skb[2][((c>>13)&0x0f)|((c>>14)&0x30)]|
des_skb[3][((c>>20)&0x01)|((c>>21)&0x06) |
  ((c>>22)&0x38)];
t=      des_skb[4][ (d    )&0x3f                ]|
des_skb[5][((d>> 7)&0x03)|((d>> 8)&0x3c)]|
des_skb[6][ (d>>15)&0x3f                ]|
des_skb[7][((d>>21)&0x0f)|((d>>22)&0x30)];

/* table contained 0213 4657 */
```

```
*(k++)=((t<<16)|(s&0x0000ffff))&0xffffffff;
s=     ((s>>16)|(t&0xffff0000));

s=(s<<4)|(s>>28);
*(k++)=s&0xffffffff;
}
```

Ricordiamo come avviene la schedulazione delle chiavi.
A partire dalla chiave iniziale, si effettua la riduzione $PC-1$ ottenendo le 2 sequenze di 28 bit c_0 e d_0 che vanno in input alla prima iterazione.
L'iterazione i-esima riceve le sequenze c_{i-1} e d_{i-1}, effettua sulle due sequenze un opportuno shift, applica alla sequenza di 56 bit la riduzione $PC-2$ e ottiene la sottochiave i-esima. Le iterazioni sono 16 una per ogni sottochiave.
In Libdes, le due sequenze c_{i-1} e d_{i-1}, input dell'i-esima iterazione sono memorizzati in due long c e d con

$$c = 0000k_{28}k_{27}\cdots k_{02}k_{01} \text{ e } d = 0000k_{56}k_{55}\cdots k_{29}.$$

Ciascuna delle sottochiavi che si ottiene è memorizzata in due locazioni successive di un vettore di long s di 32 elementi, nel quale la sottochiave i-esima è memorizzata negli elementi $s[2(i-1)]$ e $s[2(i-1)+1]$ con $i = 1, \cdots, 16$.
Considerata la i-esima iterazione, in questa si avrà che lo shift ciclico viene effettuato a destra e non a sinistra, questo per come sono memorizzati i bit nelle variabili, mentre la permutazione $PC-2$ viene effettuata tramite la matrice *des_skb* che oltre ad effettuare tale permutazione dei bit dispone questi in due variabili s e t come sotto riportato:

$$s = 00ki_{24}ki_{23}ki_{22}ki_{21}ki_{20}ki_{19}00ki_{12}\cdots ki_{07}00ki_{18}\cdots ki_{13}00ki_{06}\cdots ki_{01}$$
$$t = 00ki_{48}ki_{47}ki_{46}ki_{45}ki_{44}ki_{43}00ki_{36}\cdots ki_{31}00ki_{42}\cdots ki_{37}00ki_{30}\cdots ki_{25};$$

dove $ki_{48}ki_{47}ki_{46}ki_{45}\cdots ki_{03}ki_{02}ki_{01}$ sono i 48 bit della i-esima sottochiave.
A partire da queste due variabili si ottiene $s[2(i-1)]$ e $s[2(i-1)+1]$. Si effettuano una serie di scambi tra i gruppi di 6 bit delle due variabili s e t. Si shifta ciclicamente a sinistra t di 4 posizioni in modo da ottenere :

$$s[2(i-1)] = 00ki_{42}ki_{41}ki_{40}ki_{39}ki_{38}ki_{37}00ki_{30}\cdots ki_{25}00ki_{18}\cdots ki_{13}00ki_{06}\cdots ki_{01}$$

$$s[(2(i-1)+1] = ki_{46}ki_{45}ki_{44}ki_{43}00ki_{36}\cdots ki_{31}00ki_{24}\cdots ki_{19}00ki_{12}\cdots ki_{07}00k_{48}k_{47}$$

Il perchè di questa particolare organizzazione dei bit delle sottochiavi sarà chiara nella successiva sezione.

5.3 La realizzazione delle iterazioni: espansione E, indirizzamento alle S-box e permutazione P

Anche il Libdes così come il D3Des effettua 8 e non 16 iterazioni sviluppando in ogni iterazione 2 round consecutivi dell'algoritmo DES. Vediamo come l'implementazione effettua il calcolo di $f(R_{i-1}, K_i)$ e l'*or* esclusivo di questa con L_{i-1}. Riportiamo di seguito le linee di codice interessate, con una descrizione delle variabili usate:

```
#define D_ENCRYPT(L,R,S)        \
u=(R^s[S  ]); \
t=R^s[S+1]; \
t=((t>>4)+(t<<28)); \
L^=des_SPtrans[1][(t    )&0x3f]| \
des_SPtrans[3][(t>> 8)&0x3f]| \
des_SPtrans[5][(t>>16)&0x3f]| \
des_SPtrans[7][(t>>24)&0x3f]| \
des_SPtrans[0][(u    )&0x3f]| \
des_SPtrans[2][(u>> 8)&0x3f]| \
des_SPtrans[4][(u>>16)&0x3f]| \
des_SPtrans[6][(u>>24)&0x3f];
```

$L = l_{32}l_{31}\cdots l_{01}$ e $R = r_{32}r_{31}\cdots r_{01}$ rappresentano nell'i-esima iterazione le sequenze L_{i-1} e R_{i-1}, S e s sono rispettivamente il numero del round e il vettore delle sottochiavi mentre `des_SPtrans` è una matrice che rappresenta la $SPBOX$ descritte nella sezione 2.2.1 opportunamente modificate come si vedrà in seguito.
Occorre puntualizzare, prima di proseguire, che Libdes dopo aver calcolato IP e quindi l'input alla fase delle iterazioni, effettua su L e R (le sequenze di 32 bit che rappresentano l'output di IP) uno shift circolare a sinistra di una posizione. Vedremo in seguito la sua utilità.
Vediamo adesso come l'implementazione calcola $E(R_{i-1}) \oplus K_i$ e per far questo analizziamo la Tabella 4 dove le sequenze dei bit appaiono così come sono memorizzate nelle variabili.
Nella prima colonna abbiamo l'*or* esclusivo bit a bit di $E(R_{i-1})$ e della sottochiave K_i, secondo l'algoritmo DES.
Shiftiamo R_{i-1} ciclicamente a sinistra di una posizione ottenendo:

$$R'_{i-1} = r'_{01}, \cdots, r'_{31} \quad \text{con} \quad r'_i = \begin{cases} r_{i-1} & i = 2\cdots 31; \\ r_{01} & i = 32. \end{cases} \tag{11}$$

Ne effettuiamo l'espansione con la matrice E' ottenuta da E dopo averne fatto uno shift circolare di una posizione a sinistra, in modo tale che $E'(R'_{i-1}) = E(R_{i-1})$. La matrice E' è :

E' bit-selection table					
1	2	3	4	5	4
5	6	7	8	9	8
9	10	11	12	13	12
13	14	15	16	17	16
17	18	19	20	21	20
21	22	23	24	25	24
25	26	27	28	29	28
29	30	31	32	1	32

Dell'espansione di $E'(R'_{i-1})$ ne facciamo l' *or* esclusivo con la sottochiave K_i. I valori ottenuti sono rappresentati nella seconda colonna.

Riscriviamo questa serie di *or* esclusivi aggiungendo dei *cuscinetti* tra ogni gruppo di 6 bit, otteniamo così quanto riportato nella terza colonna nella quale abbiamo una sequenza di 64 *or* esclusivi divisi in gruppi di 8 ($A_{01}, A_{02}, A_{03}, A_{04}, A_{05}, A_{06}, A_{07}, A_{08}$). Se consideriamo adesso le sequenze $A_{01}A_{03}A_{05}A_{07}$ e $A_{02}A_{04}A_{06}A_{08}$ e shiftiamo circolarmente a sinistra di 4 posizioni la prima sequenza otteniamo la situazione illustrata nella penultima colonna.
Tenendo conto della rappresentazione delle chiavi vista nella Sezione 5.2, la prima delle due linee di codice che segue determina in u e t le sequenze mostrate nella penultima colonna della Tabella 4, la seconda determina la sequenza mostrata nell'ultima colonna

```
u=(R^s[S  ]); t=R^s[S+1];
t=((t>>4)+(t<<28)).
```

Riportiamo per comodità i valori contenuti nell'ultima colonna della Tabella 4 :

$$u = xxb_{42}b_{41}b_{40}b_{39}b_{38}b_{37}xxb_{30}\cdots b_{25}xxb_{18}\cdots b_{13}xxb_{06}\cdots b_{01} = xxB_{07}xxB_{05}xxB_{03}xxB_{01}$$
$$t = xxb_{48}b_{47}b_{46}b_{45}b_{44}b_{43}xxb_{36}\cdots b_{31}xxb_{24}\cdots b_{19}xxb_{12}\cdots b_{07} = xxB_{08}xxB_{06}xxB_{04}xxB_{02}$$

L'individuazione dei bit b_i fatta nell'ultima colonna è giustificata dallo shift circolare su R_{i-1} che ha determinato R'_{i-1}. Tenendo conto della (11) che lega r'_i a r_i otteniamo che i bit b_i sono calcolati in maniera corretta. Infatti, se consideriamo l'input alla $S{-}box8$ i bit $b_{43}b_{44}b_{45}b_{46}b_{47}b_{48}$ che dalla (2) introdotte nella sezione 4.3.1 sono uguali a:

$$b_{43} = r_{28} \oplus ki_{43}$$
$$b_{44} = r_{29} \oplus ki_{44}$$
$$b_{45} = r_{30} \oplus ki_{45}$$
$$b_{46} = r_{31} \oplus ki_{46}$$
$$b_{47} = r_{32} \oplus ki_{47}$$
$$b_{48} = r_{01} \oplus ki_{48}$$

coincidono con quelli da noi indicati nel 1° blocco dell'ultima colonna.
A questo punto l'input e l'output delle $S-box$, la permutazione P dell'output delle $S-box$ e l'*or* esclusivo di tale valore che rappresenta $f(R_{i-1}, K_i)$, con L_{i-1} è effettuato così

$$R_i = L_i \oplus (des\_SPtrans[0][B_{01}] \vee des\_SPtrans[1][B_{02}] \vee \cdots \vee des\_SPtrans[7][B_{08}])$$

Il valore di uscita di $des\_SPtrans[i-1][B_i]$ con $i = 1, \cdots, 8$ è l'output della $SPBOX$ i-esima che è stata costruita secondo lo schema descritto nella sezione 2.2.1 con due aggiunte: le sequenze di 32 bit vengono considerate da destra verso sinistra e shiftate ciclicamente di una posizione a sinistra.
Per capire meglio cosa restituisce $des\_SPtrans$ basta vedere l'esempio che segue.

Esempio 5.1

Sia $B_{01} = b_{06}b_{05}b_{04}b_{03}b_{02}b_{01} = 010111$ *l'input della S-box1.*
Risulta $b_{01}b_{06} = 10_2 = 2_{10}$ *e* $b_{02}b_{03}b_{04}b_{05} = 1101_2 = 13_{10}$, *seguendo i passi dell'algoritmo DES questo implica che nella S-box1 dobbiamo prendere come output l'elemento di indice riga 1 e di indice colonna 12 cioè* $S-box1[1][12] = 1010_2 = c_{01}c_{02}c_{03}c_{04}$. *In Libdes a partire da* B_{01} *otteniamo*

$$des\_SPtrans[0][B_{01}] = 0000\ 0000\ 1000\ 0000\ 0000\ 0010\ 0000\ 0000$$

Vediamo perchè $des\_SPtrans[0][B_{01}]$ *contiene tale valore. Partiamo da*

S-box1[1][12]=1010$_2$,

questo bisogna innanzitutto vederelo all'interno di una stringa di 32 bit cioè

1010 0000 0000 0000 0000 0000 0000 0000

e poi permutato tramite P:

0000 0000 1000 0000 0000 1000 0000 0000

Dato che i bit sono memorizzati nelle variabili da destra verso sinistra e considerato lo shift ciclico a sinistra che necessita R_i *nella successiva iterazione otteniamo* $des\_SPtrans[0][B_{01}]$.

$E(R_{i-1}) \oplus K_i$
$r_{01} \oplus ki_{48}$
$r_{32} \oplus ki_{47}$
$r_{31} \oplus ki_{46}$
$r_{30} \oplus ki_{45}$
$r_{29} \oplus ki_{44}$
$r_{28} \oplus ki_{43}$
$r_{29} \oplus ki_{42}$
$r_{28} \oplus ki_{41}$
$r_{27} \oplus ki_{40}$
$r_{26} \oplus ki_{39}$
$r_{25} \oplus ki_{38}$
$r_{24} \oplus ki_{37}$
$r_{25} \oplus ki_{36}$
$r_{24} \oplus ki_{35}$
$r_{23} \oplus ki_{34}$
$\vdots$
$r_{20} \oplus ki_{31}$
$r_{21} \oplus ki_{30}$
$r_{20} \oplus ki_{29}$
$r_{19} \oplus ki_{28}$
$\vdots$
$r_{16} \oplus ki_{25}$
$r_{17} \oplus ki_{24}$
$r_{16} \oplus ki_{23}$
$r_{15} \oplus ki_{22}$
$\vdots$
$r_{12} \oplus ki_{19}$
$r_{13} \oplus ki_{18}$
$r_{12} \oplus ki_{17}$
$r_{11} \oplus ki_{16}$
$\vdots$
$r_{08} \oplus ki_{13}$
$r_{09} \oplus ki_{12}$
$r_{08} \oplus ki_{11}$
$r_{07} \oplus ki_{10}$
$\vdots$
$r_{04} \oplus ki_{07}$
$r_{05} \oplus ki_{06}$
$r_{04} \oplus ki_{05}$
$r_{03} \oplus ki_{04}$
$r_{02} \oplus ki_{03}$
$r_{01} \oplus ki_{02}$
$r_{32} \oplus ki_{01}$

$E' \oplus K_i$
$r'_{02} \oplus ki_{48}$
$r'_{01} \oplus ki_{47}$
$r'_{32} \oplus ki_{46}$
$\vdots$
$r'_{29} \oplus ki_{43}$
$r'_{30} \oplus ki_{42}$
$r'_{29} \oplus ki_{41}$
$r'_{28} \oplus ki_{40}$
$\vdots$
$r'_{25} \oplus ki_{37}$
$r'_{26} \oplus ki_{36}$
$r'_{25} \oplus ki_{35}$
$r'_{24} \oplus ki_{34}$
$\vdots$
$r'_{21} \oplus ki_{31}$
$r'_{22} \oplus ki_{30}$
$r'_{21} \oplus ki_{29}$
$r'_{20} \oplus ki_{28}$
$\vdots$
$r'_{17} \oplus ki_{25}$
$r'_{18} \oplus ki_{24}$
$r'_{17} \oplus ki_{23}$
$r'_{16} \oplus ki_{22}$
$\vdots$
$r'_{13} \oplus ki_{19}$
$r'_{14} \oplus ki_{18}$
$r'_{13} \oplus ki_{17}$
$r'_{12} \oplus ki_{16}$
$\vdots$
$r'_{09} \oplus ki_{13}$
$r'_{10} \oplus ki_{12}$
$r'_{09} \oplus ki_{11}$
$r'_{08} \oplus ki_{10}$
$\vdots$
$r'_{05} \oplus ki_{07}$
$r'_{06} \oplus ki_{06}$
$r'_{05} \oplus ki_{05}$
$r'_{04} \oplus ki_{04}$
$r'_{03} \oplus ki_{03}$
$r'_{02} \oplus ki_{02}$
$r'_{01} \oplus ki_{01}$

$r'_{04} \oplus 0$	A_{01}
$r'_{03} \oplus 0$	
$r'_{02} \oplus ki_{48}$	
$r'_{01} \oplus ki_{47}$	
$r'_{32} \oplus ki_{46}$	
$\vdots$	
$r'_{29} \oplus ki_{43}$	
$r'_{32} \oplus 0$	A_{02}
$r'_{31} \oplus 0$	
$r'_{30} \oplus ki_{42}$	
$\vdots$	
$r'_{25} \oplus ki_{37}$	
$r'_{28} \oplus 0$	A_{03}
$r'_{27} \oplus 0$	
$r'_{26} \oplus ki_{36}$	
$\vdots$	
$r'_{21} \oplus ki_{31}$	
$r'_{24} \oplus 0$	A_{04}
$r'_{23} \oplus 0$	
$r'_{22} \oplus ki_{30}$	
$\vdots$	
$r'_{17} \oplus ki_{25}$	
$r'_{20} \oplus 0$	A_{05}
$r'_{19} \oplus 0$	
$r'_{18} \oplus ki_{24}$	
$\vdots$	
$r'_{13} \oplus ki_{19}$	
$r'_{16} \oplus 0$	A_{06}
$r'_{15} \oplus 0$	
$r'_{14} \oplus ki_{18}$	
$\vdots$	
$r'_{09} \oplus ki_{13}$	
$r'_{12} \oplus 0$	A_{07}
$r'_{11} \oplus 0$	
$r'_{10} \oplus ki_{12}$	
$\vdots$	
$r'_{05} \oplus ki_{07}$	
$r'_{08} \oplus 0$	A_{08}
$r'_{07} \oplus 0$	
$r'_{06} \oplus ki_{06}$	
$\vdots$	
$r'_{01} \oplus ki_{01}$	

t
$r'_{32} \oplus ki_{46}$
$r'_{31} \oplus ki_{45}$
$r'_{30} \oplus ki_{44}$
$r'_{29} \oplus ki_{43}$
$r'_{28} \oplus 0$
$r'_{27} \oplus 0$
$r'_{26} \oplus ki_{36}$
$r'_{25} \oplus ki_{35}$
$r'_{24} \oplus ki_{34}$
$\vdots$
$r'_{17} \oplus ki_{23}$
$r'_{16} \oplus ki_{22}$
$\vdots$
$r'_{09} \oplus ki_{11}$
$r'_{08} \oplus ki_{10}$
$\vdots$
$r'_{01} \oplus ki_{47}$
u
$r'_{32} \oplus 0$
$r'_{31} \oplus 0$
$r'_{30} \oplus ki_{42}$
$r'_{29} \oplus ki_{41}$
$r'_{28} \oplus ki_{40}$
$r'_{27} \oplus ki_{39}$
$r'_{26} \oplus ki_{38}$
$r'_{25} \oplus ki_{37}$
$r'_{24} \oplus 0$
$r'_{23} \oplus 0$
$r'_{22} \oplus ki_{30}$
$\vdots$
$r'_{17} \oplus ki_{25}$
$r'_{16} \oplus 0$
$r'_{15} \oplus 0$
$r'_{14} \oplus ki_{18}$
$\vdots$
$r'_{09} \oplus ki_{13}$
$r'_{08} \oplus 0$
$r'_{07} \oplus 0$
$r'_{06} \oplus ki_{06}$
$\vdots$
$r'_{01} \oplus ki_{01}$

t	
$r'_{04} \oplus 0$	x
$r'_{03} \oplus 0$	x
$r'_{02} \oplus ki_{48}$	b_{48}
$r'_{01} \oplus ki_{47}$	b_{47}
$r'_{32} \oplus ki_{46}$	b_{46}
$r'_{31} \oplus ki_{45}$	b_{45}
$r'_{30} \oplus ki_{44}$	b_{44}
$r'_{29} \oplus ki_{43}$	b_{43}
$r'_{28} \oplus 0$	x
$\vdots$	$\vdots$
$r'_{21} \oplus ki_{31}$	b_{31}
$r'_{20} \oplus 0$	x
$\vdots$	$\vdots$
$r'_{13} \oplus ki_{19}$	b_{19}
$r'_{12} \oplus 0$	x
$\vdots$	$\vdots$
$r'_{05} \oplus ki_{07}$	b_{07}
u	
$r'_{32} \oplus 0$	x
$r'_{31} \oplus 0$	x
$r'_{30} \oplus ki_{42}$	b_{42}
$r'_{29} \oplus ki_{41}$	b_{41}
$r'_{28} \oplus ki_{40}$	b_{40}
$r'_{27} \oplus ki_{39}$	b_{39}
$r'_{26} \oplus ki_{38}$	b_{38}
$r'_{25} \oplus ki_{37}$	b_{37}
$r'_{24} \oplus 0$	x
$r'_{23} \oplus 0$	x
$r'_{22} \oplus ki_{30}$	b_{30}
$\vdots$	$\vdots$
$r'_{17} \oplus ki_{25}$	b_{25}
$r'_{16} \oplus 0$	x
$r'_{15} \oplus 0$	x
$r'_{14} \oplus ki_{18}$	b_{18}
$\vdots$	$\vdots$
$r'_{09} \oplus ki_{13}$	b_{13}
$r'_{08} \oplus 0$	x
$r'_{07} \oplus 0$	x
$r'_{06} \oplus ki_{06}$	b_{06}
$\vdots$	$\vdots$
$r'_{01} \oplus ki_{01}$	b_{01}

- Tabella 4 -

6 Cenni di analisi per Des Barrett e Des 1.0

Abbiamo già osservato come l'efficenza delle implemetazioni sia legata alla realizzazione delle permutazioni, ma osservando il modo con cui opera D3Des e Libdes anche il numero di operazioni necessarie ad effettuare l'indirizzamento alle $S-box$ assume un certo peso. Di seguito sono indicate brevemente alcune caratteristiche delle altre implementazioni. L'intezione è di cercare di capire quali sono i motivi che causano i ritardi dalle implementazioni più veloci.

Des 1.0 Per la realizzazione delle permutazioni IP, IP^{-1}, $PC-1$, $PC-2$ si costruisce in fase di preprocessing delle grosse matrici, una per ogni permutazione, in modo da riuscire a calcolare il risultato delle permutazioni con poche istruzioni.
Consideriamo per esempio il calcolo della permutazione IP.
Ricordiamo che IP viene applicata ad un blocco di 64 bit producendo in output un blocco delle stesse dimensioni. Per poter calcolare il risultato di IP in fase di preprocessing sono stati considerati in anticipo tutti i possibili blocchi cui si sarebbe potuto applicare IP. Per fare ciò l'idea immediata sarebbe stata quella di calcolare in anticipo il risultato di IP per tutti i 2^{64} possibili blocchi d'input. Scartata tale strada, per evidenti problemi dovuti alla quantità di memoria necessaria a memorizzare una tale tabella, si è trovata una strada alternativa. Il blocco d'input è diviso in 8 gruppi di un byte $B_{iniziale1} \cdots B_{iniziale8}$. La fase di preprocessing consiste nel calcolare per ogni possibile valore del blocco i-esimo il suo contributo al risultato della permutazione. Cioè i bit di ogni blocco vengono distribuiti opportunamente in base al valore che rappresentano e al numero di quel blocco. Si costruisce, quindi, una matrice $IP[i][j]$ (i indica il numero di blocco, j il valore del blocco) ogni elemento della quale è composto da 2 long $IP[i][j]_l$ e $IP[i][j]_r$ che rappresentano la parte sinistra e destra del risultato parziale della permutazione. Una matrice cosıfatta è costituita da 2^3 (numero di righe) $\cdot\, 2^8$ (numero di colonne) $= 2048$ entrate. Lo spazio occupato è di circa 68Kbyte per la tabella $IP[i][j]$. Considerando anche le altre, questa strategia necessita di oltre 270Kbyte di tabelle precomputate.
L'Esempio 6.1 mostra come viene calcolato il risultato di IP e quanto questo venga calcolato velocemente (solo 16 operazioni).

Esempio 6.1

Supponiamo di voler applicare IP al blocco B dato da:

1100100000111001100010010000010110000100111111110011111100000011

Calcoliamoci il valore decimale di ogni byte : $11001000_2 = 200_{10}$*;* $00111001_2 = 57_{10}$*;* $10001001_2 = 137_{10}$*;* $00000101_2 = 5_{10}$*;* $10000100_2 = 132_{10}$*;* $11111111_2 = 255_{10}$*;* $00111111_2 = 63_{10}$*;* $00000011_2 = 3_{10}$*; Consideriamo il blocco risultante diviso in due long,* $IP(B)_l$ *e* $IP(B)_r$*, abbiamo:*

$$\begin{aligned} IP(B)_l &= IP[1][200]_l + IP[2][57]_l + IP[3][137]_l + IP[4][5]_l + \\ & \quad IP[5][132]_l + IP[6][255]_l + IP[7][63]_l + IP[8][3]_l \end{aligned}$$

$$
\begin{aligned}
IP(B)_r &= IP[1][200]_r + IP[2][57]_r + IP[3][137]_r + IP[4][5]_r + \\
& \quad IP[5][132]_r + IP[6][255]_r + IP[7][63]_r + IP[8][3]_r
\end{aligned}
$$

Con il simbolo + in questo caso intendiamo l'operazione di or bit a bit. Osserviamo che $IP(B)_l$ e $IP(B)_r$ contengono la parte sinistra e destra del risultato di IP con i bit in posizione inversa: $IP(B)_l = B_8 B_{16} \cdots B_{50} B_{58}$ e $IP(B)_r = B_7 B_{15} \cdots B_{49} B_{57}$

La realizzazione delle altre permutazioni avviene in modo analogo.
Ovviamente la velocità di realizzazione di queste permutazioni ha come contrappeso l'uso di molta memoria per poter memorizzare le matrici da usare. Anche questa implementazione coniuga $S - box$ e permutazione P come visto nella Sezione 2.2.1. Quindi visto che le permutazioni sono calcolate molto velocemente la *lentezza* della implementazione va ricercata nel numero di operazioni necessarie per realizzare in ciascuna delle 16 iterazioni il calcolo dell'espansione E e dell'*or* esclusivo tra questa e la chiave (36 + 8 operazioni).

Des Barrett e Fast des Entrambi usano le $SPBOX$ descritte nella Sezione 2.2.1, quindi realizzano velocemente la fase finale del calcolo di f. Il motivo del loro ritardo è legato alle realizzazioni di IP e IP^{-1}. In Des Barrett richiedono entrambe 259 istruzioni e sono le realizzazioni più lente tra quelle osservate (non consideriamo **plain_des** il cui scopo è essenzialmente didattico). In Fast des si utilizzano 80 istruzioni per IP e 63 per IP^{-1}.

7 Appendice A: le matrici IP, IP^{-1}, E, P

IP							
58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

IP^{-1}							
40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

E bit-selection table					
32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

P			
16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

8 Appendice B: il codice di PlainDes.c

```
  /*  DATI   TABELLARI    */

static unsigned char IP1[64] =
 { 58,50,42,34,26,18,10,2,
   60,52,44,36,28,20,12,4,
   62,54,46,38,30,22,14,6,
   64,56,48,40,32,24,16,8,
   57,49,41,33,25,17,9,1,
   59,51,43,35,27,19,11,3,
   61,53,45,37,29,21,13,5,
   63,55,47,39,31,23,15,7 };

static unsigned char IP_1[64]=
{ 40,8,48,16,56,24,64,32,
  39,7,47,15,55,23,63,31,
  38,6,46,14,54,22,62,30,
  37,5,45,13,53,21,61,29,
  36,4,44,12,52,20,60,28,
  35,3,43,11,51,19,59,27,
  34,2,42,10,50,18,58,26,
  33,1,41,9,49,17,57,25 };

static unsigned char EBS_TABLE[48] =
     { 32,1,2,3,4,5,
       4,5,6,7,8,9,
       8,9,10,11,12,13,
       12,13,14,15,16,17,
       16,17,18,19,20,21,
       20,21,22,23,24,25,
       24,25,26,27,28,29,
       28,29,30,31,32,1};

static unsigned char S_tab[8][4][16]=
       { /* S-box 1 */
 14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7,
 0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8,
 4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0,
 15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13,
 /* S-box 2 */
 15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10,
 3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5,
 0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15,
 13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9,
 /* S-box 3 */
 10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8,
 13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1,
 13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7,
 1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12,
 /* S-box 4 */
 7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15,
 13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9,
 10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4,
 3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14,
 /* S-box 5 */
 2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9,
 14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6,
 4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14,
 11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3,
 /* S-box 6 */
 12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11,
 10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8,
 9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6,
 4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13,
 /* S-box 7 */
 4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1,
 13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6,
 1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2,
 6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12,
 /* S-box 8 */
```

```
 13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7,
 1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2,
 7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8,
 2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11};


static unsigned char P[32] =
    { 16,7,20,21,
      29,12,28,17,
      1,15,23,26,
      5,18,31,10,
      2,8,24,14,
      32,27,3,9,
      19,13,30,6,
      22,11,4,25};

static unsigned char PC_1[56] =
   { 57,49,41,33,25,17,9,
     1,58,50,42,34,26,18,
     10,2,59,51,43,35,27,
     19,11,3,60,52,44,36,
     63,55,47,39,31,23,15,
     7,62,54,46,38,30,22,
     14,6,61,53,45,37,29,
     21,13,5,28,20,12,4};
/*
 * vettore che ricorda di quanto shiftare nella schdelulazione delle chiavi
*/
static unsigned char I_N[16] = {1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};

static unsigned char PC_2[48] =
    { 14,17,11,24,1,5,
      3,28,15,6,21,10,
      23,19,12,4,26,8,
      16,7,27,20,13,2,
      41,52,31,37,47,55,
      30,40,51,45,33,48,
      44,49,39,56,34,53,
      46,42,50,36,29,32};

static unsigned char mask[8] = { 0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01 };

static unsigned char conv[16] =
 { 0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,
   0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f};



static unsigned char Key_sched[16][6]={ 0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00};



static unsigned char TEST = 0;
```

```
    /*    PROCEDURE  UTILIZZATE    */


/*     Dimensione dei tipi
    long = 64 bits
    int  = 32 bits
    char = 8 bits
*/


static void row_col(S,i,j)
 unsigned char S,*i,*j;
{
   *i=(S&0x01)|((S&0x20)>>4);
   *j=(S&0x1E)>>1;
}

/*
 * Realizza gli shift circolari nella fase di schedulazione delle chiavi
*/
static void Leftshift(it,C,D)     /* it 0,..15     OK */
 unsigned char it,*C,*D;
{
  unsigned char t1,t2,tmp;
  if(I_N[it]==1) {t1=0x08;t2=0x80;}
    else {t1=0x0C;t2=0xC0;}
  tmp=t2&(*C);
  *C    =(*C<<I_N[it])|((t2&(*(C+1)))>>(8-I_N[it]));
  *(C+1)=(*(C+1)<<I_N[it])|((t2&(*(C+2)))>>(8-I_N[it]));
  *(C+2)=(*(C+2)<<I_N[it])|((t2&(*(C+3)))>>(8-I_N[it]));
  *(C+3)=(*(C+3)&0x0F)|(((*(C+3)&0xF0)<<I_N[it])|(tmp>>(4-I_N[it])));

  tmp=t1&(*D);
  *D=(((*D)&0xF0)|((((*D)&0x0F)<<I_N[it])&0x0F))|((t2&(*(D+1)))>>(8-I_N[it]));
  *(D+1)=(*(D+1)<<I_N[it])|((t2&(*(D+2)))>>(8-I_N[it]));
  *(D+2)=(*(D+2)<<I_N[it])|((t2&(*(D+3)))>>(8-I_N[it]));
  *(D+3)=(*(D+3)<<I_N[it])|(tmp>>(4-I_N[it]));
}


/*
 * funzione che realizza le varie permutazioni prendendo in input
 * la matrice che individua la permutazione e la sequenza da permutare.
 * Non tiene conto delle particolarita' delle varie permutazioni.
*/


static void Permutation(input,output,sizeout,Tab)
 unsigned char *input,*output,sizeout,*Tab;
{
  unsigned char i,j,k,j1,k1;
  for(i=0;i<=(sizeout/8)-1;i++) *(output+i)= 0x00;
  for(i=1;i<=sizeout;i++)
{
   j=*(Tab+i-1)%8;
   k=( j==0? *(Tab+i-1)/8:(*(Tab+i-1)-j)/8+1); /* k= vettore inp. interessato */
   j=*(Tab+i-1)-(8*(k-1));                     /* j= posizione nel vettore k */

 if (*(input+k-1)&mask[j-1])  {
    j1=i%8;
    k1=(j1==0? i/8:(i-j1)/8+1 );             /* k1= vettore perm. interessato */
    j1=i-(8*(k1-1));                         /* j1= posizione nel vettore k1  */
    *(output+k1-1)|=mask[j1-1];              /* settaggio del bit             */
       }
}}

/*
 * Effettua l'or esclusivo tra due sequenze
*/
```

```
static void Xor32(L,F,Xor)
 unsigned int  *L,*F,*Xor;
{
    *Xor=(*L)^(*F);
}


/* ------------------------- FINE  PROCEDURE --------------------------- */



   /*       Scheduling  della secret key      */

static void Key_scheduling(key)
 unsigned char *key;
{
  unsigned char Choice1[7],iter;
  Permutation(key,Choice1,56,PC_1);
  for (iter=0;iter<=15;iter++)
      {
Leftshift(iter,Choice1,(Choice1+3));
Permutation(Choice1,Key_sched[iter],48,PC_2);
      }
}

      /*  funzione  f(R,K)   */


static void f(R,K,Output)
 unsigned char *R,*K,*Output;  /* size R 32 bits  size K 48 bits */

{
unsigned char S[8],Appl[6],k,j1,k1,i,j,Out[4];

Out[0]=0x00; Out[1]=0x00; Out[2]=0x00; Out[3]=0x00;

Permutation(R,Appl,48,EBS_TABLE);    /*  Appl a 48 bits generato da R */

for (i=0;i<=5;i++) Appl[i]^=*(K+i);  /*  Xor di Appl con K a 48  bits */

S[0]=Appl[0]>>2;                     /* valutazione degli input alle  */
S[1]=(Appl[0]<<4)|(Appl[1]>>4);
S[2]=(Appl[1]<<2)|(Appl[2]>>6);
S[3]=Appl[2]&0x3F;
S[4]=Appl[3]>>2;
S[5]=(Appl[3]<<4)|(Appl[4]>>4);
S[6]=(Appl[4]<<2)|(Appl[5]>>6);
S[7]=Appl[5]&0x3F;                   /* primitive S_i,   i=1,..,8     */



/* I primi 6 bits di ogni S contengono l' input alle S_boxes. */

/* Valutazione dell Output     */

for(k=0;k<=7;k++)
{
  row_col(S[k],&i,&j);
  j1=S_tab[k][i][j];
  k1=k%2==0?k/2:(k-1)/2;
  *(Out+k1)=((*(Out+k1))<<4)|conv[j1];  /*        Delle Primitive     */
}

Permutation(Out,Output,32,P);           /* Output di f(R,K).          */

}


       /* Routine del des */

      /* Parametri: Buffer,Crypted,key = indirizzi di 64 bits di memoria
```

```
    cd =  0 modalita' decifra
       <> 0 modalita' cifra                              */
      /* Cifra un blocco di 64 bit */

void ndes(Buffer,Crypted,key,cd)
 unsigned char *Buffer,*Crypted,*key,cd;
{
  unsigned char work_buf[8],*L,*R,F_RK[4],End[8],i,*swap;
  int t;
  if(TEST==0){  Key_scheduling(key);  TEST = 1;}
  Permutation(Buffer,work_buf,64,IP1);
  L=work_buf;  R=&work_buf[4];
  if(cd) {
       for(i=0;i<=15;i++)
{ swap=L;
  L=R;
  f(R,Key_sched[i],F_RK);
Xor32((unsigned int *)swap,(unsigned int *)F_RK,(unsigned int *)swap);
  R=swap;
}
      }

else {
       for(t=15;t>=0;t--)
{ swap=L;
  L=R;
  f(R,Key_sched[t],F_RK);
Xor32((unsigned int *)swap,(unsigned int *)F_RK,(unsigned int *)swap);
  R=swap;
}
      }

  for(i=0;i<=3;i++) End[i]=*(R+i);
  for(i=4;i<=7;i++) End[i]=*(L+(i-4));
  Permutation(End,Crypted,64,IP_1);
}


char    *des_scan_format1 = "%2x%2x%2x%2x%2x%2x%2x%2x";

int hexdigit1(ch)
   register char ch;
{
   if (ch >= '0' && ch <= '9') return ch - '0';
   if (ch >= 'a' && ch <= 'f') return ch - 'a' + 10;
   if (ch >= 'A' && ch <= 'F') return ch - 'A' + 10;
   return -1;
}

/*
 * Converte un numero ASCII esadecimale al sua corrispondente stringa di bytes
 */
unsigned hex1(d, s, size)
   register char *d, *s;
   register unsigned size;
{
   register unsigned count = 0;
   register int lo, hi;

   if (size > 0) do {

      hi = hexdigit1(*s++);
      if (hi < 0) break;
      lo = hexdigit1(*s++);
      if (lo < 0) break;
      *d++ = (hi << 4) | lo;
      count++;
   } while (count < size);
   return count;
}
```

```
/*
 * Procedura da noi usata per permettere la cifratura di messaggi
 * aventi una dimensione superiore ai 64 bit
*/

void mainnew(scelta,strkey,buffin,buffout,size)
 char *scelta,*buffin,*buffout;
 int size;
 register char *strkey;

{
 register char key[8];
 int j,opzione,a;
printf("************ NEW_DES ***************\n");

hex1(key,strkey,8);
if (*scelta=='c' || *scelta=='C') {opzione=1;}
   else if (*scelta=='d' || *scelta=='D') {opzione=0;}
for (j=0;j<size;j=j+8) {
     ndes(buffin+j,buffout+j,key,opzione);
   }}
```

9 Appendice C: il codice di D3Des.c

```
/* d3des.h -
 *
 *      Headers and defines for d3des.c
 *      Graven Imagery, 1992.
 *
 * Copyright (c) 1988,1989,1990,1991,1992 by Richard Outerbridge
 *      (GEnie : OUTER; CIS : [71755,204])
 */
#define D2_DES          /* include double-length support */
#define D3_DES          /* include triple-length support */
#ifdef D3_DES
#ifndef D2_DES
#define D2_DES          /* D2_DES is needed for D3_DES */
#endif
#endif
#define EN0     0       /* MODE == encrypt */
#define DE1     1       /* MODE == decrypt */

/* A useful alias on 68000-ish machines, but NOT USED HERE. */

typedef union {
unsigned long blok[2];
unsigned short word[4];
unsigned char byte[8];
} M68K;

extern void deskey();
/*                    hexkey[8]     MODE unsigned char *, shor
 * Sets the internal key register according to the hexadecimal
 * key contained in the 8 bytes of hexkey, according to the DES,
 * for encryption or decryption according to MODE.
 */

extern void usekey();
/*                  cookedkey[32] unsigned long *
 * Loads the internal key register with the data in cookedkey.
 */

extern void cpkey();
/*                 cookedkey[32] unsigned long *
 * Copies the contents of the internal key register into the storage
 * located at &cookedkey[0].
 */

extern void desd3();
/*                  from[8]           to[8] unsigned char *, unsigned char
 * Encrypts/Decrypts (according to the key currently loaded in the
 * internal key register) one block of eight bytes at address 'from'
 * into the block at address 'to'.  They can be the same.
 */

#ifdef D2_DES

#define desDkey(a,b)    des2key((a),(b))
extern void des2key();
/*                    hexkey[16]     MODE unsigned char *, short
 * Sets the internal key registerS according to the hexadecimal
 * keyS contained in the 16 bytes of hexkey, according to the DES,
 * for DOUBLE encryption or decryption according to MODE.
 * NOTE: this clobbers all three key registers!
 */

extern void Ddes();
/*                  from[8]           to[8] unsigned char *, unsigned char
 * Encrypts/Decrypts (according to the keyS currently loaded in the
 * internal key registerS) one block of eight bytes at address 'from'
 * into the block at address 'to'.  They can be the same.
 */
```

```
extern void D2des();
/*                 from[16]          to[16] unsigned char *, unsigned char
 * Encrypts/Decrypts (according to the keyS currently loaded in the
 * internal key registerS) one block of SIXTEEN bytes at address 'from'
 * into the block at address 'to'.  They can be the same.
 */

extern void makekey();
/*             *password,      single-length key[8] char *, unsigned char *
 * With a double-length default key, this routine hashes a NULL-terminated
 * string into an eight-byte random-looking key, suitable for use with the
 * deskey() routine.
 */

#define makeDkey(a,b)   make2key((a),(b))
extern void make2key();
/*             *password,      double-length key[16] char *, unsigned char *
 * With a double-length default key, this routine hashes a NULL-terminated
 * string into a sixteen-byte random-looking key, suitable for use with the
 * des2key() routine.
 */

#ifndef D3_DES  /* D2_DES only */

#define useDkey(a)      use2key((a))
#define cpDkey(a)       cp2key((a))

extern void use2key();
/*                 cookedkey[64] unsigned long *
 * Loads the internal key registerS with the data in cookedkey.
 * NOTE: this clobbers all three key registers!
 */
extern void cp2key();
/*                cookedkey[64] unsigned long *
 * Copies the contents of the internal key registerS into the storage
 * located at &cookedkey[0].
 */

#else   /* D3_DES too */

#define useDkey(a)      use3key((a))
#define cpDkey(a)       cp3key((a))

extern void des3key();
/*                   hexkey[24]     MODE unsigned char *, short
 * Sets the internal key registerS according to the hexadecimal
 * keyS contained in the 24 bytes of hexkey, according to the DES,
 * for DOUBLE encryption or decryption according to MODE.
 */

extern void use3key();
/*                 cookedkey[96] unsigned long *
 * Loads the 3 internal key registerS with the data in cookedkey.
 */

extern void cp3key();
/*                cookedkey[96] unsigned long *
 * Copies the contents of the 3 internal key registerS into the storage
 * located at &cookedkey[0].
 */

extern void make3key();
/*             *password,      triple-length key[24] char *, unsigned char *
 * With a triple-length default key, this routine hashes a NULL-terminated
 * string into a twenty-four-byte random-looking key, suitable for use with
 * the des3key() routine. */
#endif  /* D3_DES */
#endif  /* D2_DES */
/* d3des.h V5.09 rwo 9208.04 15:06 Graven Imagery*/
```

```
/* D3DES (V5.09) -
 *
 * A portable, public domain, version of the Data Encryption Standard.
 *
 * Written with Symantec's THINK (Lightspeed) C by Richard Outerbridge.
 * Thanks to: Dan Hoey for his excellent Initial and Inverse permutation
 * code;  Jim Gillogly & Phil Karn for the DES key schedule code; Dennis
 * Ferguson, Eric Young and Dana How for comparing notes; and Ray Lau,
 * for humouring me on.
 *
 * Copyright (c) 1988,1989,1990,1991,1992 by Richard Outerbridge.
 * (GEnie : OUTER; CIS : [71755,204]) Graven Imagery, 1992.
 */

#include "d3des.h"

static void scrunch();
static void unscrun();
static void desfunc();
static void cookey();
static unsigned long KnL[32] = { 0L };
static unsigned long KnR[32] = { 0L };
static unsigned long Kn3[32] = { 0L };
static unsigned char Df_Key[24] = {
0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef,
0xfe,0xdc,0xba,0x98,0x76,0x54,0x32,0x10,
0x89,0xab,0xcd,0xef,0x01,0x23,0x45,0x67 };

static unsigned short bytebit[8]       = {
0200, 0100, 040, 020, 010, 04, 02, 01 };

static unsigned long bigbyte[24] = {
0x800000L,      0x400000L,      0x200000L,      0x100000L,
0x80000L,       0x40000L,       0x20000L,       0x10000L,
0x8000L,        0x4000L,        0x2000L,        0x1000L,
0x800L,         0x400L,         0x200L,         0x100L,
0x80L,          0x40L,          0x20L,          0x10L,
0x8L,           0x4L,           0x2L,           0x1L    };

/* Use the key schedule specified in the Standard (ANSI X3.92-1981). */

static unsigned char pc1[56] = {
56, 48, 40, 32, 24, 16,  8,      0, 57, 49, 41, 33, 25, 17,
 9,  1, 58, 50, 42, 34, 26,     18, 10,  2, 59, 51, 43, 35,
62, 54, 46, 38, 30, 22, 14,      6, 61, 53, 45, 37, 29, 21,
13,  5, 60, 52, 44, 36, 28,     20, 12,  4, 27, 19, 11,  3 };

static unsigned char totrot[16] = {
1,2,4,6,8,10,12,14,15,17,19,21,23,25,27,28 };

static unsigned char pc2[48] = {
13, 16, 10, 23,  0,  4,  2, 27, 14,  5, 20,  9,
22, 18, 11,  3, 25,  7, 15,  6, 26, 19, 12,  1,
40, 51, 30, 36, 46, 54, 29, 39, 50, 44, 32, 47,
43, 48, 38, 55, 33, 52, 45, 41, 49, 35, 28, 31 };


void deskey(key, edf)   /* Thanks to James Gillogly & Phil Karn! */
unsigned char *key;
short edf;
/*
 * Effettua la schedulazione delle sottochiavi, da PC1 a PC2,
 * dispondendo la sottochiave sinistra e la sottoschiave
 * destra in 2 long.
*/
{
register int i, j, l, m, n;
unsigned char pc1m[56], pcr[56];
unsigned long kn[32];

for ( j = 0; j < 56; j++ ) {
```

```
l = pc1[j];
m = l & 07;
pc1m[j] = (key[l >> 3] & bytebit[m]) ? 1 : 0;
}
for( i = 0; i < 16; i++ ) {
if( edf == DE1 ) m = (15 - i) << 1;
else m = i << 1;
n = m + 1;
kn[m] = kn[n] = 0L;
for( j = 0; j < 28; j++ ) {
l = j + totrot[i];
if( l < 28 ) pcr[j] = pc1m[l];
else pcr[j] = pc1m[l - 28];
}
for( j = 28; j < 56; j++ ) {
    l = j + totrot[i];
    if( l < 56 ) pcr[j] = pc1m[l];
    else pcr[j] = pc1m[l - 28];
    }
for( j = 0; j < 24; j++ ) {
if( pcr[pc2[j]] ) kn[m] |= bigbyte[j];
if( pcr[pc2[j+24]] ) kn[n] |= bigbyte[j];
}
}
cookey(kn);
return;
}



static void cookey(raw1)
register unsigned long *raw1;
/*
 * La seguente funzione riorganizza la rappresentazione delle sottochiavi DES
 * mischiando i blocchi della sottochiave destra e della sottochaive sinistra
 * nel modo che e' stato mostrato.
*/
{
register unsigned long *cook, *raw0;
unsigned long dough[32];
register int i;

cook = dough;
for( i = 0; i < 16; i++, raw1++ ) {
raw0 = raw1++;
*cook    = (*raw0 & 0x00fc0000L) << 6;
*cook   |= (*raw0 & 0x00000fc0L) << 10;
*cook   |= (*raw1 & 0x00fc0000L) >> 10;
*cook++ |= (*raw1 & 0x00000fc0L) >> 6;
*cook    = (*raw0 & 0x0003f000L) << 12;
*cook   |= (*raw0 & 0x0000003fL) << 16;
*cook   |= (*raw1 & 0x0003f000L) >> 4;
*cook++ |= (*raw1 & 0x0000003fL);
}
usekey(dough);
return;
}


/*
 * la funzione che segue non viene chiamata
*/
void cpkey(into)
register unsigned long *into;
{
register unsigned long *from, *endp;

from = KnL, endp = &KnL[32];
while( from < endp ) *into++ = *from++;
return;
}
```

```
void usekey(from)
register unsigned long *from;
/* E' usata per copiare il contenuto del vettore 'from' che contiene le 16 sottochiavi
   schedulate nel vettore, variabile globale, KnL
*/
{
register unsigned long *to, *endp;

to = KnL, endp = &KnL[32];
while( to < endp ) *to++ = *from++;
return;
}


void desd3(inblock, outblock)
unsigned char *inblock, /* e' il blocco di 64 bit da cifrare */
      *outblock; /* conterra' il blocco di 64 bit cifrato */

/*
 * E' la funzione centrale: chiamata subito dopo la
 *  schedulazione delle sottochiavi.
*/
{
unsigned long work[2];

scrunch(inblock, work);
desfunc(work, KnL);      /* KnL e' una variabile globale. Contiene le sottochiavi */
unscrun(work, outblock);
return;
}


static void scrunch(outof, into)
register unsigned char *outof; /* contiene l'input alla funzione */
register unsigned long *into;  /* conterra' alla fine l'output */
/*
 * Trasferisce il vettore di 8 caratteri d'input
 * in due long, quelli sui quali si agira' nel corso della
 * cifratura.
*/
{
*into    = (*outof++ & 0xffL) << 24;
*into   |= (*outof++ & 0xffL) << 16;
*into   |= (*outof++ & 0xffL) << 8;
*into++ |= (*outof++ & 0xffL);
*into    = (*outof++ & 0xffL) << 24;
*into   |= (*outof++ & 0xffL) << 16;
*into   |= (*outof++ & 0xffL) << 8;
*into   |= (*outof   & 0xffL);
return;
}



static void unscrun(outof, into)
register unsigned long *outof;/* input della funzione, contiene il blocco cifrato */
register unsigned char *into; /* output della funzione */
/*
 * Trasferisce i 64 bit di testo cifrato, (contenuto in 2 long)
 * in un vettore di 8 caratteri
*/

{
*into++ = (*outof >> 24) & 0xffL;
*into++ = (*outof >> 16) & 0xffL;
*into++ = (*outof >>  8) & 0xffL;
*into++ =  *outof++      & 0xffL;
*into++ = (*outof >> 24) & 0xffL;
*into++ = (*outof >> 16) & 0xffL;
*into++ = (*outof >>  8) & 0xffL;
```

```
*into   =  *outof        & 0xffL;
return;
}


/*
 * I seguenti 8 vettori costituiscono le 8 SPBOX utilizzate dall'implementazione
*/


static unsigned long SP1[64] = {
0x01010400L, 0x00000000L, 0x00010000L, 0x01010404L,
0x01010004L, 0x00010404L, 0x00000004L, 0x00010000L,
0x00000400L, 0x01010400L, 0x01010404L, 0x00000400L,
0x01000404L, 0x01010004L, 0x01000000L, 0x00000004L,
0x00000404L, 0x01000400L, 0x01000400L, 0x00010400L,
0x00010400L, 0x01010000L, 0x01010000L, 0x01000404L,
0x00010004L, 0x01000004L, 0x01000004L, 0x00010004L,
0x00000000L, 0x00000404L, 0x00010404L, 0x01000000L,
0x00010000L, 0x01010404L, 0x00000004L, 0x01010000L,
0x01010400L, 0x01000000L, 0x01000000L, 0x00000400L,
0x01010004L, 0x00010000L, 0x00010400L, 0x01000004L,
0x00000400L, 0x00000004L, 0x01000404L, 0x00010404L,
0x01010404L, 0x00010004L, 0x01010000L, 0x01000404L,
0x01000004L, 0x00000404L, 0x00010404L, 0x01010400L,
0x00000404L, 0x01000400L, 0x01000400L, 0x00000000L,
0x00010004L, 0x00010400L, 0x00000000L, 0x01010004L };

static unsigned long SP2[64] = {
0x80108020L, 0x80008000L, 0x00008000L, 0x00108020L,
0x00100000L, 0x00000020L, 0x80100020L, 0x80008020L,
0x80000020L, 0x80108020L, 0x80108000L, 0x80000000L,
0x80008000L, 0x00100000L, 0x00000020L, 0x80100020L,
0x00108000L, 0x00100020L, 0x80008020L, 0x00000000L,
0x80000000L, 0x00008000L, 0x00108020L, 0x80100000L,
0x00100020L, 0x80000020L, 0x00000000L, 0x00108000L,
0x00008020L, 0x80108000L, 0x80100000L, 0x00008020L,
0x00000000L, 0x00108020L, 0x80100020L, 0x00100000L,
0x80008020L, 0x80100000L, 0x80108000L, 0x00008000L,
0x80100000L, 0x80008000L, 0x00000020L, 0x80108020L,
0x00108020L, 0x00000020L, 0x00008000L, 0x80000000L,
0x00008020L, 0x80108000L, 0x00100000L, 0x80000020L,
0x00100020L, 0x80008020L, 0x80000020L, 0x00100020L,
0x00108000L, 0x00000000L, 0x80008000L, 0x00008020L,
0x80000000L, 0x80100020L, 0x80108020L, 0x00108000L };

static unsigned long SP3[64] = {
0x00000208L, 0x08020200L, 0x00000000L, 0x08020008L,
0x08000200L, 0x00000000L, 0x00020208L, 0x08000200L,
0x00020008L, 0x08000008L, 0x08000008L, 0x00020000L,
0x08020208L, 0x00020008L, 0x08020000L, 0x00000208L,
0x08000000L, 0x00000008L, 0x08020200L, 0x00000200L,
0x00020200L, 0x08020000L, 0x08020008L, 0x00020208L,
0x08000208L, 0x00020200L, 0x00020000L, 0x08000208L,
0x00000008L, 0x08020208L, 0x00000200L, 0x08000000L,
0x08020200L, 0x08000000L, 0x00020008L, 0x00000208L,
0x00020000L, 0x08020200L, 0x08000200L, 0x00000000L,
0x00000200L, 0x00020008L, 0x08020208L, 0x08000200L,
0x08000008L, 0x00000200L, 0x00000000L, 0x08020008L,
0x08000208L, 0x00020000L, 0x08000000L, 0x08020208L,
0x00000008L, 0x00020208L, 0x00020200L, 0x08000008L,
0x08020000L, 0x08000208L, 0x00000208L, 0x08020000L,
0x00020208L, 0x00000008L, 0x08020008L, 0x00020200L };

static unsigned long SP4[64] = {
0x00802001L, 0x00002081L, 0x00002081L, 0x00000080L,
0x00802080L, 0x00800081L, 0x00800001L, 0x00002001L,
0x00000000L, 0x00802000L, 0x00802000L, 0x00802081L,
0x00000081L, 0x00000000L, 0x00800080L, 0x00800001L,
0x00000001L, 0x00002000L, 0x00800000L, 0x00802001L,
0x00000080L, 0x00800000L, 0x00002001L, 0x00002080L,
```

```
0x00800081L, 0x00000001L, 0x00002080L, 0x00800080L,
0x00002000L, 0x00802080L, 0x00802081L, 0x00000081L,
0x00800080L, 0x00800001L, 0x00802000L, 0x00802081L,
0x00000081L, 0x00000000L, 0x00000000L, 0x00802000L,
0x00002080L, 0x00800080L, 0x00800081L, 0x00000001L,
0x00802001L, 0x00002081L, 0x00002081L, 0x00000080L,
0x00802081L, 0x00000081L, 0x00000001L, 0x00002000L,
0x00800001L, 0x00002001L, 0x00802080L, 0x00800081L,
0x00002001L, 0x00002080L, 0x00800000L, 0x00802001L,
0x00000080L, 0x00800000L, 0x00002000L, 0x00802080L };

static unsigned long SP5[64] = {
0x00000100L, 0x02080100L, 0x02080000L, 0x42000100L,
0x00080000L, 0x00000100L, 0x40000000L, 0x02080000L,
0x40080100L, 0x00080000L, 0x02000100L, 0x40080100L,
0x42000100L, 0x42080000L, 0x00080100L, 0x40000000L,
0x02000000L, 0x40080000L, 0x40080000L, 0x00000000L,
0x40000100L, 0x42080100L, 0x42080100L, 0x02000100L,
0x42080000L, 0x40000100L, 0x00000000L, 0x42000000L,
0x02080100L, 0x02000000L, 0x42000000L, 0x00080100L,
0x00080000L, 0x42000100L, 0x00000100L, 0x02000000L,
0x40000000L, 0x02080000L, 0x42000100L, 0x40080100L,
0x02000100L, 0x40000000L, 0x42080000L, 0x02080100L,
0x40080100L, 0x00000100L, 0x02000000L, 0x42080000L,
0x42080100L, 0x00080100L, 0x42000000L, 0x42080100L,
0x02080000L, 0x00000000L, 0x40080000L, 0x42000000L,
0x00080100L, 0x02000100L, 0x40000100L, 0x00080000L,
0x00000000L, 0x40080000L, 0x02080100L, 0x40000100L };

static unsigned long SP6[64] = {
0x20000010L, 0x20400000L, 0x00004000L, 0x20404010L,
0x20400000L, 0x00000010L, 0x20404010L, 0x00400000L,
0x20004000L, 0x00404010L, 0x00400000L, 0x20000010L,
0x00400010L, 0x20004000L, 0x20000000L, 0x00004010L,
0x00000000L, 0x00400010L, 0x20004010L, 0x00004000L,
0x00404000L, 0x20004010L, 0x00000010L, 0x20400010L,
0x20400010L, 0x00000000L, 0x00404010L, 0x20404000L,
0x00004010L, 0x00404000L, 0x20404000L, 0x20000000L,
0x20004000L, 0x00000010L, 0x20400010L, 0x00404000L,
0x20404010L, 0x00400000L, 0x00004010L, 0x20000010L,
0x00400000L, 0x20004000L, 0x20000000L, 0x00004010L,
0x20000010L, 0x20404010L, 0x00404000L, 0x20400000L,
0x00404010L, 0x20404000L, 0x00000000L, 0x20400010L,
0x00000010L, 0x00004000L, 0x20400000L, 0x00404010L,
0x00004000L, 0x00400010L, 0x20004010L, 0x00000000L,
0x20404000L, 0x20000000L, 0x00400010L, 0x20004010L };

static unsigned long SP7[64] = {
0x00200000L, 0x04200002L, 0x04000802L, 0x00000000L,
0x00000800L, 0x04000802L, 0x00200802L, 0x04200800L,
0x04200802L, 0x00200000L, 0x00000000L, 0x04000002L,
0x00000002L, 0x04000000L, 0x04200002L, 0x00000802L,
0x04000800L, 0x00200802L, 0x00200002L, 0x04000800L,
0x04000002L, 0x04200000L, 0x04200800L, 0x00200002L,
0x04200000L, 0x00000800L, 0x00000802L, 0x04200802L,
0x00200800L, 0x00000002L, 0x04000000L, 0x00200800L,
0x04000000L, 0x00200800L, 0x00200000L, 0x04000802L,
0x04000802L, 0x04200002L, 0x04200002L, 0x00000002L,
0x00200002L, 0x04000000L, 0x04000800L, 0x00200000L,
0x04200800L, 0x00000802L, 0x00200802L, 0x04200800L,
0x00000802L, 0x04000002L, 0x04200802L, 0x04200000L,
0x00200800L, 0x00000000L, 0x00000002L, 0x04200802L,
0x00000000L, 0x00200802L, 0x04200000L, 0x00000800L,
0x04000002L, 0x04000800L, 0x00000800L, 0x00200002L };

static unsigned long SP8[64] = {
0x10001040L, 0x00001000L, 0x00040000L, 0x10041040L,
0x10000000L, 0x10001040L, 0x00000040L, 0x10000000L,
0x00040040L, 0x10040000L, 0x10041040L, 0x00041000L,
0x10041000L, 0x00041040L, 0x00001000L, 0x00000040L,
0x10040000L, 0x10000040L, 0x10001000L, 0x00001040L,
```

```
0x00041000L, 0x00040040L, 0x10040040L, 0x10041000L,
0x00001040L, 0x00000000L, 0x00000000L, 0x10040040L,
0x10000040L, 0x10001000L, 0x00041040L, 0x00040000L,
0x00041040L, 0x00040000L, 0x10041000L, 0x00001000L,
0x00000040L, 0x10040040L, 0x00001000L, 0x00041040L,
0x10001000L, 0x00000040L, 0x10000040L, 0x10040000L,
0x10040040L, 0x10000000L, 0x00040000L, 0x10001040L,
0x00000000L, 0x10041040L, 0x00040040L, 0x10000040L,
0x10040000L, 0x10001000L, 0x10001040L, 0x00000000L,
0x10041040L, 0x00041000L, 0x00041000L, 0x00001040L,
0x00001040L, 0x00040040L, 0x10000000L, 0x10041000L };
```

```
static void desfunc(block, keys)
register unsigned long *block, /* E' un vettore di 2 long ed e' l'output di 'scrunch' */
*keys; /* contiene le 16 sottochiavi */
/*
* E' la procedura che effettua la cifratura di un blocco di 64 bit 'block' conoscendo il blocco
* e le sottochiavi schedulate
*/
{
register unsigned long fval, work, right, leftt;
register int round;
/*
 * CALCOLO DI IP
*/
leftt = block[0];
right = block[1];
/* Primo Passo */
work = ((leftt >> 4) ^ right) & 0x0f0f0f0fL;
right ^= work;
leftt ^= (work << 4);
/* Secondo Passo */
work = ((leftt >> 16) ^ right) & 0x0000ffffL;
right ^= work;
leftt ^= (work << 16);
/* Terzo Passo */
work = ((right >> 2) ^ leftt) & 0x33333333L;
leftt ^= work;
right ^= (work << 2);
/* Quarto Passo */
work = ((right >> 8) ^ leftt) & 0x00ff00ffL;
leftt ^= work;
right ^= (work << 8);
/* Quinto Passo */
right = ((right << 1) | ((right >> 31) & 1L)) & 0xffffffffL;
work = (leftt ^ right) & 0xaaaaaaaaL;
leftt ^= work;
/* Sesto Passo */
right ^= work;
leftt = ((leftt << 1) | ((leftt >> 31) & 1L)) & 0xffffffffL;
/*
 * Le iterazioni
*/
for( round = 0; round < 8; round++ ) {
/*
** primo round **
*/
work  = (right << 28) | (right >> 4);
/* work=xxE1xxE3xxE5xxE7 */
work ^= *keys++;
fval  = SP7[ work                & 0x3fL];
fval |= SP5[(work >>  8) & 0x3fL];
fval |= SP3[(work >> 16) & 0x3fL];
fval |= SP1[(work >> 24) & 0x3fL];
/* right=xxE2xxE4xxE6xxE8 */
work  = right ^ *keys++;
fval |= SP8[ work                & 0x3fL];
fval |= SP6[(work >>  8) & 0x3fL];
```

```
fval |= SP4[(work >> 16) & 0x3fL];
fval |= SP2[(work >> 24) & 0x3fL];
leftt ^= fval;
/*
 ** secondo round **
*/
work  = (leftt << 28) | (leftt >> 4);
/* work=xxE1xxE3xxE5xxE7 */
work ^= *keys++;
fval  = SP7[ work                & 0x3fL];
fval |= SP5[(work >>  8) & 0x3fL];
fval |= SP3[(work >> 16) & 0x3fL];
fval |= SP1[(work >> 24) & 0x3fL];
/* leftt=xxE2xxE4xxE6xxE8 */
work  = leftt ^ *keys++;
fval |= SP8[ work                & 0x3fL];
fval |= SP6[(work >>  8) & 0x3fL];
fval |= SP4[(work >> 16) & 0x3fL];
fval |= SP2[(work >> 24) & 0x3fL];
right ^= fval;
}/* fine delle iterazioni */

/*
 * CALCOLO DI IP -1
*/

right = (right << 31) | (right >> 1);
/* Primo Passo */
work = (leftt ^ right) & 0xaaaaaaaaL;
leftt ^= work;
right ^= work;
leftt = (leftt << 31) | (leftt >> 1);
/* Secondo Passo */
work = ((leftt >> 8) ^ right) & 0x00ff00ffL;
right ^= work;
leftt ^= (work << 8);
/* Terzo Passo */
work = ((leftt >> 2) ^ right) & 0x33333333L;
right ^= work;
leftt ^= (work << 2);
/* Quarto Passo */
work = ((right >> 16) ^ leftt) & 0x0000ffffL;
leftt ^= work;
right ^= (work << 16);
/* Quinto Passo */
work = ((right >> 4) ^ leftt) & 0x0f0f0f0fL;
leftt ^= work;
right ^= (work << 4);

*block++ = right;
*block = leftt;
return;
}


int hexdigit2(ch)
   register char ch;
{
   if (ch >= '0' && ch <= '9') return ch - '0';
   if (ch >= 'a' && ch <= 'f') return ch - 'a' + 10;
   if (ch >= 'A' && ch <= 'F') return ch - 'A' + 10;
   return -1;
}


unsigned hex2(d, s, size)
   register char *d, *s;
   register unsigned size;
/*
 * Converte un numero ASCII esadecimale nella sua corrispondente stringa of
bytes
```

```
 * E' utilizzata per la chiave K fornita
 */
{
   register unsigned count = 0;
   register int lo, hi;

   if (size > 0) do {

      hi = hexdigit2(*s++);
      if (hi < 0) break;
      lo = hexdigit2(*s++);
      if (lo < 0) break;
      *d++ = (hi << 4) | lo;
      count++;
   } while (count < size);
   return count;
}




void maind3(scelta,strkey,buffin,buffout,size)
   char *scelta,*buffin,*buffout;
   register char *strkey;
   int size;
/*
* E' la procedura iniziale di d3_des
*/

{
   register char key[8];
   char x;
   int i,j,opzione;
   clock_t tempo;
   struct tms time_in,time_fin;

if (times(&time_in)==-1){printf("\nERRORE times\n");return -1.0;}

printf("************ D3_DES ***************\n");


    hex2(key,strkey,8);

   if (*scelta=='c' || *scelta=='C') {opzione=0;}
     else if (*scelta=='d' || *scelta=='D') {opzione=1;}

   deskey(key,(short)opzione);

   for (j=0;j<size;j=j+8) {
   desd3(buffin+j,buffout+j);
   }
 if (times(&time_fin)==-1){printf("\nERRORE times\n");return -1.0;}
 tempo=(time_fin.tms_stime - time_in.tms_stime)+
       (time_fin.tms_utime - time_in.tms_utime);  /* tempo utente */

 return tempo;
}
```

10 Appendice D: il codice di Libdes.c

```
/* des.h */
/* Copyright (C) 1993 Eric Young - see README for more details */
#ifndef DES_DEFS
#define DES_DEFS

typedef unsigned char des_cblock[8];
typedef struct des_ks_struct
{
union   {
des_cblock _;
/* make sure things are correct size on machines with
 * 8 byte longs */
unsigned long pad[2];
} ks;
#define _       ks._
} des_key_schedule[16];

#define DES_KEY_SZ      (sizeof(des_cblock))
#define DES_ENCRYPT     1
#define DES_DECRYPT     0

#define DES_CBC_MODE    0
#define DES_PCBC_MODE   1

#define C_Block des_cblock
#define Key_schedule des_key_schedule
#define ENCRYPT DES_ENCRYPT
#define DECRYPT DES_DECRYPT
#define KEY_SZ DES_KEY_SZ
#define string_to_key des_string_to_key
#define read_pw_string des_read_pw_string
#define random_key des_random_key
#define pcbc_encrypt des_pcbc_encrypt
#define set_key des_set_key
#define key_sched des_key_sched
#define ecb_encrypt des_ecb_encrypt
#define cbc_encrypt des_cbc_encrypt
#define cbc_cksum des_cbc_cksum
#define quad_cksum des_quad_cksum

/* For compatibility with the MIT lib - eay 20/05/92 */
typedef struct des_ks_struct bit_64;

extern int des_check_key;       /* defaults to false */
extern int des_rw_mode;         /* defaults to DES_PCBC_MODE */

/* The next line is used to disable full ANSI prototypes, if your
 * compiler has problems with the prototypes, make sure this line always
 * evaluates to true :-) */
#if !defined(MSDOS) && !defined(__STDC__)
#ifndef KERBEROS
int des_3ecb_encrypt();
int des_cbc_encrypt();
int des_3cbc_encrypt();
int des_cfb_encrypt();
int des_ecb_encrypt();
int des_encrypt();
int des_enc_read();
int des_enc_write();
int des_ofb_encrypt();
int des_pcbc_encrypt();
int des_random_key();
int des_read_password();
int des_read_2passwords();
int des_read_pw_string();
int des_is_weak_key();
int des_set_key_l();
int des_key_sched();
int des_string_to_key();
```

```
int des_string_to_2keys();
#endif
char *crypt();
unsigned long des_cbc_cksum();
unsigned long des_quad_cksum();
unsigned long des_cbc_cksum();
void des_set_odd_parity();
#else /* PROTO */
int des_3ecb_encrypt(des_cblock *input,des_cblock *output,\
des_key_schedule ks1,des_key_schedule ks2,int encrypt);
unsigned long des_cbc_cksum(des_cblock *input,des_cblock *output,\
long length,des_key_schedule schedule,des_cblock *ivec);
int des_cbc_encrypt(des_cblock *input,des_cblock *output,long length,\
des_key_schedule schedule,des_cblock *ivec,int encrypt);
int des_3cbc_encrypt(des_cblock *input,des_cblock *output,long length,\
des_key_schedule sk1,des_key_schedule sk2,\
des_cblock *ivec1,des_cblock *ivec2,int encrypt);
int des_cfb_encrypt(unsigned char *in,unsigned char *out,int numbits,\
long length,des_key_schedule schedule,des_cblock *ivec,int encrypt);
int des_ecb_encrypt(des_cblock *input,des_cblock *output,\
des_key_schedule ks,int encrypt);
int des_encrypt(unsigned long *input,unsigned long *output,
des_key_schedule ks, int encrypt);
int des_enc_read(int fd,char *buf,int len,des_key_schedule sched,\
des_cblock *iv);
int des_enc_write(int fd,char *buf,int len,des_key_schedule sched,\
des_cblock *iv);
char *crypt(const char *buf,const char *salt);
int des_ofb_encrypt(unsigned char *in,unsigned char *out,\
int numbits,long length,des_key_schedule schedule,des_cblock *ivec);
int des_pcbc_encrypt(des_cblock *input,des_cblock *output,long length,\
des_key_schedule schedule,des_cblock *ivec,int encrypt);
unsigned long des_quad_cksum(des_cblock *input,des_cblock *output,\
long length,int out_count,des_cblock *seed);
int des_random_key(des_cblock ret);
int des_read_password(des_cblock *key,char *prompt,int verify);
int des_read_2passwords(des_cblock *key1,des_cblock *key2, \
char *prompt,int verify);
int des_read_pw_string(char *buf,int length,char *prompt,int verify);
void des_set_odd_parity(des_cblock *key);
int des_is_weak_key(des_cblock *key);
int des_set_key(des_cblock *key,des_key_schedule schedule);
int des_key_sched(des_cblock *key,des_key_schedule schedule);
int des_string_to_key(char *str,des_cblock *key);
int des_string_to_2keys(char *str,des_cblock *key1,des_cblock *key2);
#endif
#endif
```

```
/* des_locl.h */

/* Copyright (C) 1993 Eric Young - see README for more details */
#include <stdio.h>
#include "des_li.h"

#if defined(__STDC__) || defined(VMS) || defined(M_XENIX) || defined(MSDOS)
#include <string.h>
#define bcopy(b1,b2,len) memcpy(b2, b1, (size_t)(len))
#define bzero(b,len) memset(b, 0, (size_t)(len))
#define bcmp(b1,b2,len) memcmp(b1, b2, (size_t)(len))
#define index(s1,char) strchr(s1,char)
#endif

#ifdef MSDOS
#define getpid() 2
#define RAND
extern int errno;
```

```
#define PROTO
#endif

#ifdef __STDC__
#define PROTO
#endif

#ifdef RAND
#define random() rand()
#define srandom(s) srand(s)
#endif

#define ITERATIONS 16
#define HALF_ITERATIONS 8

/* used in des_read and des_write */
#define MAXWRITE        (1024*16)
#define BSIZE           (MAXWRITE+4)

#define c2l(c,l)        (l =((unsigned long)(*((c)++)))    , \
 l|=((unsigned long)(*((c)++)))<< 8, \
 l|=((unsigned long)(*((c)++)))<<16, \
 l|=((unsigned long)(*((c)++)))<<24)

/* NOTE - c is not incremented as per c2l */
#define c2ln(c,l1,l2,n) { \
c+=n; \
l1=l2=0; \
switch (n) { \
case 8: l2 =((unsigned long)(*(--(c))))<<24; \
case 7: l2|=((unsigned long)(*(--(c))))<<16; \
case 6: l2|=((unsigned long)(*(--(c))))<< 8; \
case 5: l2|=((unsigned long)(*(--(c))));     \
case 4: l1 =((unsigned long)(*(--(c))))<<24; \
case 3: l1|=((unsigned long)(*(--(c))))<<16; \
case 2: l1|=((unsigned long)(*(--(c))))<< 8; \
case 1: l1|=((unsigned long)(*(--(c))));     \
} \
}

#define l2c(l,c)        (*((c)++)=(unsigned char)(((l)    )&0xff), \
 *((c)++)=(unsigned char)(((l)>> 8)&0xff), \
 *((c)++)=(unsigned char)(((l)>>16)&0xff), \
 *((c)++)=(unsigned char)(((l)>>24)&0xff))

/* replacements for htonl and ntohl since I have no idea what to do
 * when faced with machines with 8 byte longs. */
#define HDRSIZE 4

#define n2l(c,l)        (l =((unsigned long)(*((c)++)))<<24, \
 l|=((unsigned long)(*((c)++)))<<16, \
 l|=((unsigned long)(*((c)++)))<< 8, \
 l|=((unsigned long)(*((c)++))))

#define l2n(l,c)        (*((c)++)=(unsigned char)(((l)>>24)&0xff), \
 *((c)++)=(unsigned char)(((l)>>16)&0xff), \
 *((c)++)=(unsigned char)(((l)>> 8)&0xff), \
 *((c)++)=(unsigned char)(((l)    )&0xff))

/* NOTE - c is not incremented as per l2c */
#define l2cn(l1,l2,c,n) { \
c+=n; \
switch (n) { \
case 8: *(--(c))=(unsigned char)(((l2)>>24)&0xff); \
case 7: *(--(c))=(unsigned char)(((l2)>>16)&0xff); \
case 6: *(--(c))=(unsigned char)(((l2)>> 8)&0xff); \
case 5: *(--(c))=(unsigned char)(((l2)    )&0xff); \
case 4: *(--(c))=(unsigned char)(((l1)>>24)&0xff); \
case 3: *(--(c))=(unsigned char)(((l1)>>16)&0xff); \
case 2: *(--(c))=(unsigned char)(((l1)>> 8)&0xff); \
case 1: *(--(c))=(unsigned char)(((l1)    )&0xff); \
```

```
} \
}

/* The changes to this macro may help or hinder, depending on the
 * compiler and the achitecture.  gcc2 always seems to do well :-).
 * Inspired by Dana How <how@isl.stanford.edu>
 * DO NOT use the alternative version on machines with 8 byte longs. */
#ifdef ALT_ECB
#define D_ENCRYPT(L,R,S) \
u=((R^s[S  ])<<2);      \
t= R^s[S+1]; \
t=((t>>2)+(t<<30)); \
L^= \
*(unsigned long *)(des_SP+0x0100+((t    )&0xfc))+ \
*(unsigned long *)(des_SP+0x0300+((t>> 8)&0xfc))+ \
*(unsigned long *)(des_SP+0x0500+((t>>16)&0xfc))+ \
*(unsigned long *)(des_SP+0x0700+((t>>24)&0xfc))+ \
*(unsigned long *)(des_SP+       ((u    )&0xfc))+ \
*(unsigned long *)(des_SP+0x0200+((u>> 8)&0xfc))+ \
*(unsigned long *)(des_SP+0x0400+((u>>16)&0xfc))+ \
*(unsigned long *)(des_SP+0x0600+((u>>24)&0xfc));
#else /* original version */
#ifdef MSDOS
#define D_ENCRYPT(L,R,S)        \
U.l=R^s[S+1]; \
T.s[0]=((U.s[0]>>4)|(U.s[1]<<12))&0x3f3f; \
T.s[1]=((U.s[1]>>4)|(U.s[0]<<12))&0x3f3f; \
U.l=(R^s[S  ])&0x3f3f3f3f; \
L^=     des_SPtrans[1][(T.c[0])]| \
des_SPtrans[3][(T.c[1])]| \
des_SPtrans[5][(T.c[2])]| \
des_SPtrans[7][(T.c[3])]| \

des_SPtrans[0][(U.c[0])]| \
des_SPtrans[2][(U.c[1])]| \
des_SPtrans[4][(U.c[2])]| \
des_SPtrans[6][(U.c[3])];
#else
#define D_ENCRYPT(L,R,S)        \
u=(R^s[S  ]); \
t=R^s[S+1]; \
t=((t>>4)+(t<<28)); \
L^=     des_SPtrans[1][(t    )&0x3f]| \
des_SPtrans[3][(t>> 8)&0x3f]| \
des_SPtrans[5][(t>>16)&0x3f]| \
des_SPtrans[7][(t>>24)&0x3f]| \
des_SPtrans[0][(u    )&0x3f]| \
des_SPtrans[2][(u>> 8)&0x3f]| \
des_SPtrans[4][(u>>16)&0x3f]| \
des_SPtrans[6][(u>>24)&0x3f];
#endif
#endif

/* IP and FP
 * The problem is more of a geometric problem that random bit fiddling.
 0  1  2  3  4  5  6  7      62 54 46 38 30 22 14  6
 8  9 10 11 12 13 14 15      60 52 44 36 28 20 12  4
16 17 18 19 20 21 22 23      58 50 42 34 26 18 10  2
24 25 26 27 28 29 30 31  to  56 48 40 32 24 16  8  0

32 33 34 35 36 37 38 39      63 55 47 39 31 23 15  7
40 41 42 43 44 45 46 47      61 53 45 37 29 21 13  5
48 49 50 51 52 53 54 55      59 51 43 35 27 19 11  3
56 57 58 59 60 61 62 63      57 49 41 33 25 17  9  1

The output has been subject to swaps of the form
0 1 -> 3 1 but the odd and even bits have been put into
2 3    2 0
different words.  The main trick is to remember that
t=((l>>size)^r)&(mask);
r^=t;
```

```
l^=(t<<size);
can be used to swap and move bits between words.

So l =  0  1  2  3  r = 16 17 18 19
4  5  6  7      20 21 22 23
8  9 10 11      24 25 26 27
       12 13 14 15      28 29 30 31
becomes (for size == 2 and mask == 0x3333)
   t =   2^16  3^17 -- --   l =  0  1 16 17  r =  2  3 18 19
 6^20  7^21 -- --        4  5 20 21       6  7 22 23
10^24 11^25 -- --        8  9 24 25      10 11 24 25
14^28 15^29 -- --       12 13 28 29      14 15 28 29

Thanks for hints from Richard Outerbridge - he told me IP&FP
could be done in 15 xor, 10 shifts and 5 ands.
When I finally started to think of the problem in 2D
I first got 42 operations without xors.  When I remembered
how to use xors :-) I got it to its final state.
*/
#define PERM_OP(a,b,t,n,m) ((t)=((((a)>>(n))^(b))&(m)),\
(b)^=(t),\
(a)^=((t)<<(n)))

#define IP(l,r,t) \
PERM_OP(r,l,t, 4,0x0f0f0f0f); \
PERM_OP(l,r,t,16,0x0000ffff); \
PERM_OP(r,l,t, 2,0x33333333); \
PERM_OP(l,r,t, 8,0x00ff00ff); \
PERM_OP(r,l,t, 1,0x55555555);\


#define FP(l,r,t) \
PERM_OP(l,r,t, 1,0x55555555); \
PERM_OP(r,l,t, 8,0x00ff00ff); \
PERM_OP(l,r,t, 2,0x33333333); \
PERM_OP(r,l,t,16,0x0000ffff); \
PERM_OP(l,r,t, 4,0x0f0f0f0f);



/* sk.h */

/* Copyright (C) 1993 Eric Young - see README for more details */
static unsigned long des_skb[8][64]={
/* for C bits (numbered as per FIPS 46) 1 2 3 4 5 6 */
0x00000000,0x00000010,0x20000000,0x20000010,
0x00010000,0x00010010,0x20010000,0x20010010,
0x00000800,0x00000810,0x20000800,0x20000810,
0x00010800,0x00010810,0x20010800,0x20010810,
0x00000020,0x00000030,0x20000020,0x20000030,
0x00010020,0x00010030,0x20010020,0x20010030,
0x00000820,0x00000830,0x20000820,0x20000830,
0x00010820,0x00010830,0x20010820,0x20010830,
0x00080000,0x00080010,0x20080000,0x20080010,
0x00090000,0x00090010,0x20090000,0x20090010,
0x00080800,0x00080810,0x20080800,0x20080810,
0x00090800,0x00090810,0x20090800,0x20090810,
0x00080020,0x00080030,0x20080020,0x20080030,
0x00090020,0x00090030,0x20090020,0x20090030,
0x00080820,0x00080830,0x20080820,0x20080830,
0x00090820,0x00090830,0x20090820,0x20090830,
/* for C bits (numbered as per FIPS 46) 7 8 10 11 12 13 */
0x00000000,0x02000000,0x00002000,0x02002000,
0x00200000,0x02200000,0x00202000,0x02202000,
0x00000004,0x02000004,0x00002004,0x02002004,
0x00200004,0x02200004,0x00202004,0x02202004,
0x00000400,0x02000400,0x00002400,0x02002400,
0x00200400,0x02200400,0x00202400,0x02202400,
0x00000404,0x02000404,0x00002404,0x02002404,
0x00200404,0x02200404,0x00202404,0x02202404,
0x10000000,0x12000000,0x10002000,0x12002000,
```

```
0x10200000,0x12200000,0x10202000,0x12202000,
0x10000004,0x12000004,0x10002004,0x12002004,
0x10200004,0x12200004,0x10202004,0x12202004,
0x10000400,0x12000400,0x10002400,0x12002400,
0x10200400,0x12200400,0x10202400,0x12202400,
0x10000404,0x12000404,0x10002404,0x12002404,
0x10200404,0x12200404,0x10202404,0x12202404,
/* for C bits (numbered as per FIPS 46) 14 15 16 17 19 20 */
0x00000000,0x00000001,0x00040000,0x00040001,
0x01000000,0x01000001,0x01040000,0x01040001,
0x00000002,0x00000003,0x00040002,0x00040003,
0x01000002,0x01000003,0x01040002,0x01040003,
0x00000200,0x00000201,0x00040200,0x00040201,
0x01000200,0x01000201,0x01040200,0x01040201,
0x00000202,0x00000203,0x00040202,0x00040203,
0x01000202,0x01000203,0x01040202,0x01040203,
0x08000000,0x08000001,0x08040000,0x08040001,
0x09000000,0x09000001,0x09040000,0x09040001,
0x08000002,0x08000003,0x08040002,0x08040003,
0x09000002,0x09000003,0x09040002,0x09040003,
0x08000200,0x08000201,0x08040200,0x08040201,
0x09000200,0x09000201,0x09040200,0x09040201,
0x08000202,0x08000203,0x08040202,0x08040203,
0x09000202,0x09000203,0x09040202,0x09040203,
/* for C bits (numbered as per FIPS 46) 21 23 24 26 27 28 */
0x00000000,0x00100000,0x00000100,0x00100100,
0x00000008,0x00100008,0x00000108,0x00100108,
0x00001000,0x00101000,0x00001100,0x00101100,
0x00001008,0x00101008,0x00001108,0x00101108,
0x04000000,0x04100000,0x04000100,0x04100100,
0x04000008,0x04100008,0x04000108,0x04100108,
0x04001000,0x04101000,0x04001100,0x04101100,
0x04001008,0x04101008,0x04001108,0x04101108,
0x00020000,0x00120000,0x00020100,0x00120100,
0x00020008,0x00120008,0x00020108,0x00120108,
0x00021000,0x00121000,0x00021100,0x00121100,
0x00021008,0x00121008,0x00021108,0x00121108,
0x04020000,0x04120000,0x04020100,0x04120100,
0x04020008,0x04120008,0x04020108,0x04120108,
0x04021000,0x04121000,0x04021100,0x04121100,
0x04021008,0x04121008,0x04021108,0x04121108,
/* for D bits (numbered as per FIPS 46) 1 2 3 4 5 6 */
0x00000000,0x10000000,0x00010000,0x10010000,
0x00000004,0x10000004,0x00010004,0x10010004,
0x20000000,0x30000000,0x20010000,0x30010000,
0x20000004,0x30000004,0x20010004,0x30010004,
0x00100000,0x10100000,0x00110000,0x10110000,
0x00100004,0x10100004,0x00110004,0x10110004,
0x20100000,0x30100000,0x20110000,0x30110000,
0x20100004,0x30100004,0x20110004,0x30110004,
0x00001000,0x10001000,0x00011000,0x10011000,
0x00001004,0x10001004,0x00011004,0x10011004,
0x20001000,0x30001000,0x20011000,0x30011000,
0x20001004,0x30001004,0x20011004,0x30011004,
0x00101000,0x10101000,0x00111000,0x10111000,
0x00101004,0x10101004,0x00111004,0x10111004,
0x20101000,0x30101000,0x20111000,0x30111000,
0x20101004,0x30101004,0x20111004,0x30111004,
/* for D bits (numbered as per FIPS 46) 8 9 11 12 13 14 */
0x00000000,0x08000000,0x00000008,0x08000008,
0x00000400,0x08000400,0x00000408,0x08000408,
0x00020000,0x08020000,0x00020008,0x08020008,
0x00020400,0x08020400,0x00020408,0x08020408,
0x00000001,0x08000001,0x00000009,0x08000009,
0x00000401,0x08000401,0x00000409,0x08000409,
0x00020001,0x08020001,0x00020009,0x08020009,
0x00020401,0x08020401,0x00020409,0x08020409,
0x02000000,0x0A000000,0x02000008,0x0A000008,
0x02000400,0x0A000400,0x02000408,0x0A000408,
0x02020000,0x0A020000,0x02020008,0x0A020008,
0x02020400,0x0A020400,0x02020408,0x0A020408,
```

```
0x02000001,0x0A000001,0x02000009,0x0A000009,
0x02000401,0x0A000401,0x02000409,0x0A000409,
0x02020001,0x0A020001,0x02020009,0x0A020009,
0x02020401,0x0A020401,0x02020409,0x0A020409,
/* for D bits (numbered as per FIPS 46) 16 17 18 19 20 21 */
0x00000000,0x00000100,0x00080000,0x00080100,
0x01000000,0x01000100,0x01080000,0x01080100,
0x00000010,0x00000110,0x00080010,0x00080110,
0x01000010,0x01000110,0x01080010,0x01080110,
0x00200000,0x00200100,0x00280000,0x00280100,
0x01200000,0x01200100,0x01280000,0x01280100,
0x00200010,0x00200110,0x00280010,0x00280110,
0x01200010,0x01200110,0x01280010,0x01280110,
0x00000200,0x00000300,0x00080200,0x00080300,
0x01000200,0x01000300,0x01080200,0x01080300,
0x00000210,0x00000310,0x00080210,0x00080310,
0x01000210,0x01000310,0x01080210,0x01080310,
0x00200200,0x00200300,0x00280200,0x00280300,
0x01200200,0x01200300,0x01280200,0x01280300,
0x00200210,0x00200310,0x00280210,0x00280310,
0x01200210,0x01200310,0x01280210,0x01280310,
/* for D bits (numbered as per FIPS 46) 22 23 24 25 27 28 */
0x00000000,0x04000000,0x00040000,0x04040000,
0x00000002,0x04000002,0x00040002,0x04040002,
0x00002000,0x04002000,0x00042000,0x04042000,
0x00002002,0x04002002,0x00042002,0x04042002,
0x00000020,0x04000020,0x00040020,0x04040020,
0x00000022,0x04000022,0x00040022,0x04040022,
0x00002020,0x04002020,0x00042020,0x04042020,
0x00002022,0x04002022,0x00042022,0x04042022,
0x00000800,0x04000800,0x00040800,0x04040800,
0x00000802,0x04000802,0x00040802,0x04040802,
0x00002800,0x04002800,0x00042800,0x04042800,
0x00002802,0x04002802,0x00042802,0x04042802,
0x00000820,0x04000820,0x00040820,0x04040820,
0x00000822,0x04000822,0x00040822,0x04040822,
0x00002820,0x04002820,0x00042820,0x04042820,
0x00002822,0x04002822,0x00042822,0x04042822,
};


/* spr.h */

/* Copyright (C) 1993 Eric Young - see README for more details */
static unsigned long des_SPtrans[8][64]={
/* nibble 0 */
0x00820200, 0x00020000, 0x80800000, 0x80820200,
0x00800000, 0x80020200, 0x80020000, 0x80800000,
0x80020200, 0x00820200, 0x00820000, 0x80000200,
0x80800200, 0x00800000, 0x00000000, 0x80020000,
0x00020000, 0x80000000, 0x00800200, 0x00020200,
0x80820200, 0x00820000, 0x80000200, 0x00800200,
0x80000000, 0x00000200, 0x00020200, 0x80820000,
0x00000200, 0x80800200, 0x80820000, 0x00000000,
0x00000000, 0x80820200, 0x00800200, 0x80020000,
0x00820200, 0x00020000, 0x80000200, 0x00800200,
0x80820000, 0x00000200, 0x00020200, 0x80800000,
0x80020200, 0x80000000, 0x80800000, 0x00820000,
0x80820200, 0x00020200, 0x00820000, 0x80800200,
0x00800000, 0x80000200, 0x80020000, 0x00000000,
0x00020000, 0x00800000, 0x80800200, 0x00820200,
0x80000000, 0x80820000, 0x00000200, 0x80020200,

/* nibble 1 */
0x10042004, 0x00000000, 0x00042000, 0x10040000,
0x10000004, 0x00002004, 0x10002000, 0x00042000,
0x00002000, 0x10040004, 0x00000004, 0x10002000,
0x00040004, 0x10042000, 0x10040000, 0x00000004,
0x00040000, 0x10002004, 0x10040004, 0x00002000,
0x00042004, 0x10000000, 0x00000000, 0x00040004,
0x10002004, 0x00042004, 0x10042000, 0x10000004,
```

```
0x10000000, 0x00040000, 0x00002004, 0x10042004,
0x00040004, 0x10042000, 0x10002000, 0x00042004,
0x10042004, 0x00040004, 0x10000004, 0x00000000,
0x10000000, 0x00002004, 0x00040000, 0x10040004,
0x00002000, 0x10000000, 0x00042004, 0x10002004,
0x10042000, 0x00002000, 0x00000000, 0x10000004,
0x00000004, 0x10042004, 0x00042000, 0x10040000,
0x10040004, 0x00040000, 0x00002004, 0x10002000,
0x10002004, 0x00000004, 0x10040000, 0x00042000,

/* nibble 2 */
0x41000000, 0x01010040, 0x00000040, 0x41000040,
0x40010000, 0x01000000, 0x41000040, 0x00010040,
0x01000040, 0x00010000, 0x01010000, 0x40000000,
0x41010040, 0x40000040, 0x40000000, 0x41010000,
0x00000000, 0x40010000, 0x01010040, 0x00000040,
0x40000040, 0x41010040, 0x00010000, 0x41000000,
0x41010000, 0x01000040, 0x40010040, 0x01010000,
0x00010040, 0x00000000, 0x01000000, 0x40010040,
0x01010040, 0x00000040, 0x40000000, 0x00010000,
0x40000040, 0x40010000, 0x01010000, 0x41000040,
0x00000000, 0x01010040, 0x00010040, 0x41010000,
0x40010000, 0x01000000, 0x41010040, 0x40000000,
0x40010040, 0x41000000, 0x01000000, 0x41010040,
0x00010000, 0x01000040, 0x41000040, 0x00010040,
0x01000040, 0x00000000, 0x41010000, 0x40000040,
0x41000000, 0x40010040, 0x00000040, 0x01010000,

/* nibble 3 */
0x00100402, 0x04000400, 0x00000002, 0x04100402,
0x00000000, 0x04100000, 0x04000402, 0x00100002,
0x04100400, 0x04000002, 0x04000000, 0x00000402,
0x04000002, 0x00100402, 0x00100000, 0x04000000,
0x04100002, 0x00100400, 0x00000400, 0x00000002,
0x00100400, 0x04000402, 0x04100000, 0x00000400,
0x00000402, 0x00000000, 0x00100002, 0x04100400,
0x04000400, 0x04100002, 0x04100402, 0x00100000,
0x04100002, 0x00000402, 0x00100000, 0x04000002,
0x00100400, 0x04000400, 0x00000002, 0x04100000,
0x04000402, 0x00000000, 0x00000400, 0x00100002,
0x00000000, 0x04100002, 0x04100400, 0x00000400,
0x04000000, 0x04100402, 0x00100402, 0x00100000,
0x04100402, 0x00000002, 0x04000400, 0x00100402,
0x00100002, 0x00100400, 0x04100000, 0x04000402,
0x00000402, 0x04000000, 0x04000002, 0x04100400,

/* nibble 4 */
0x02000000, 0x00004000, 0x00000100, 0x02004108,
0x02004008, 0x02000100, 0x00004108, 0x02004000,
0x00004000, 0x00000008, 0x02000008, 0x00004100,
0x02000108, 0x02004008, 0x02004100, 0x00000000,
0x00004100, 0x02000000, 0x00004008, 0x00000108,
0x02000100, 0x00004108, 0x00000000, 0x02000008,
0x00000008, 0x02000108, 0x02004108, 0x00004008,
0x02004000, 0x00000100, 0x00000108, 0x02004100,
0x02004100, 0x02000108, 0x00004008, 0x02004000,
0x00004000, 0x00000008, 0x02000008, 0x02000100,
0x02000000, 0x00004100, 0x02004108, 0x00000000,
0x00004108, 0x02000000, 0x00000100, 0x00004008,
0x02000108, 0x00000100, 0x00000000, 0x02004108,
0x02004008, 0x02004100, 0x00000108, 0x00004000,
0x00004100, 0x02004008, 0x02000100, 0x00000108,
0x00000008, 0x00004108, 0x02004000, 0x02000008,

/* nibble 5 */
0x20000010, 0x00080010, 0x00000000, 0x20080800,
0x00080010, 0x00000800, 0x20000810, 0x00080000,
0x00000810, 0x20080810, 0x00080800, 0x20000000,
0x20000800, 0x20000010, 0x20080000, 0x00080810,
0x00080000, 0x20000810, 0x20080010, 0x00000000,
0x00000800, 0x00000010, 0x20080800, 0x20080010,
```

```
0x20080810, 0x20080000, 0x20000000, 0x00000810,
0x00000010, 0x00080800, 0x00080810, 0x20000800,
0x00000810, 0x20000000, 0x20000800, 0x00080810,
0x20080800, 0x00080010, 0x00000000, 0x20000800,
0x20000000, 0x00000800, 0x20080010, 0x00080000,
0x00080010, 0x20080810, 0x00080800, 0x00000010,
0x20080810, 0x00080800, 0x00080000, 0x20000810,
0x20000010, 0x20080000, 0x00080810, 0x00000000,
0x00000800, 0x20000010, 0x20000810, 0x20080800,
0x20080000, 0x00000810, 0x00000010, 0x20080010,

/* nibble 6 */
0x00001000, 0x00000080, 0x00400080, 0x00400001,
0x00401081, 0x00001001, 0x00001080, 0x00000000,
0x00400000, 0x00400081, 0x00000081, 0x00401000,
0x00000001, 0x00401080, 0x00401000, 0x00000081,
0x00400081, 0x00001000, 0x00001001, 0x00401081,
0x00000000, 0x00400080, 0x00400001, 0x00001080,
0x00401001, 0x00001081, 0x00401080, 0x00000001,
0x00001081, 0x00401001, 0x00000080, 0x00400000,
0x00001081, 0x00401000, 0x00401001, 0x00000081,
0x00001000, 0x00000080, 0x00400000, 0x00401001,
0x00400081, 0x00001081, 0x00001080, 0x00000000,
0x00000080, 0x00400001, 0x00000001, 0x00400080,
0x00000000, 0x00400081, 0x00400080, 0x00001080,
0x00000081, 0x00001000, 0x00401081, 0x00400000,
0x00401080, 0x00000001, 0x00001001, 0x00401081,
0x00400001, 0x00401080, 0x00401000, 0x00001001,

/* nibble 7 */
0x08200020, 0x08208000, 0x00008020, 0x00000000,
0x08008000, 0x00200020, 0x08200000, 0x08208020,
0x00000020, 0x08000000, 0x00208000, 0x00008020,
0x00208020, 0x08008020, 0x08000020, 0x08200000,
0x00008000, 0x00208020, 0x00200020, 0x08008000,
0x08208020, 0x08000020, 0x00000000, 0x00208000,
0x08000000, 0x00200000, 0x08008020, 0x08200020,
0x00200000, 0x00008000, 0x08208000, 0x00000020,
0x00200000, 0x00008000, 0x08000020, 0x08208020,
0x00008020, 0x08000000, 0x00000000, 0x00208000,
0x08200020, 0x08008020, 0x08008000, 0x00200020,
0x08208000, 0x00000020, 0x00200020, 0x08008000,
0x08208020, 0x00200000, 0x08200000, 0x08000020,
0x00208000, 0x00008020, 0x08008020, 0x08200000,
0x00000020, 0x08208000, 0x00208020, 0x00000000,
0x08000000, 0x08200020, 0x00008000, 0x00208020};
```

```
/* des.c */

/* Copyright (C) 1993 Eric Young - see README for more details */
#include <stdio.h>

#ifdef VMS
#include <types.h>
#include <stat.h>
#else
#ifndef _IRIX
#include <sys/types.h>
#endif
#include <sys/stat.h>
#endif
#include "des_li.h"
#include "sys/types.h"
#include "sys/times.h"
#if defined(__STDC__) || defined(VMS) || defined(M_XENIX) || defined(MSDOS)
#include <string.h>
```

```
#define bcopy(f,t,n)    memcpy(t,f,(size_t)(n))
#define bzero(s,n)      memset(s,0,(size_t)(n))
#define bcmp(a,b,n)     memcmp(a, b,(size_t)(n))
#define index(s,c)      strchr(s,c)
#endif

#ifdef PROTO
int usage(void);
int doencryption(void);
int uufwrite(char *data, int size, int num, FILE *fp);
int uufwriteEnd(FILE *fp);
int uufread(char *out,int size,int num,FILE *fp);
int uuencode(unsigned char *in,int num,unsigned char *out);
int uudecode(unsigned char *in,int num,unsigned char *out);
#else
int usage();
int doencryption();
int uufwrite();
int uufwriteEnd();
int uufread();
int uuencode();
int uudecode();
#endif

#ifdef VMS
#define EXIT(a) exit(a&0x10000000)
#else
#define EXIT(a) exit(a)
#endif

#define BUFSIZE (8*1024)
#define VERIFY  1
#define KEYSIZ  8
#define KEYSIZB 1024 /* should hit tty line limit first :-) */
char key[KEYSIZB+1];
int do_encrypt,longk=0;
char *in=NULL,*out=NULL;
FILE *DES_IN,*DES_OUT,*CKSUM_OUT;
char uuname[200];
char uubuf[50];
int uubufnum;
#define INUUBUFN        (45*100)
#define OUTUUBUF        (65*100)
char b[OUTUUBUF];
char bb[300];
des_cblock cksum={0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};
char cksumname[200]="";

int vflag,cflag,eflag,dflag,kflag,bflag,fflag,sflag,uflag,flag3,hflag,error;



clock_t mainldes(edflag,skey,f_IN,f_OUT)
char *skey;
FILE *f_IN,*f_OUT;
char *edflag;
{
int i;
char *p;
struct stat ins,outs;
struct tms time_in,time_fin;
clock_t tempo;
if (times(&time_in)==-1){ printf("\nERRORE times\n"); return -1.0;}

strcpy(key,skey);
DES_IN=f_IN;
DES_OUT=f_OUT;
vflag=cflag=eflag=dflag=kflag=hflag=bflag=fflag=sflag=uflag=flag3=0;
error=0;

bflag=1;hflag=1;kflag=1;
```

```
  if (*edflag=='c' || *edflag=='C') {eflag=1;}
     else if (*edflag=='d' || *edflag=='D') {dflag=1;}

if (((eflag+dflag) == 1) || cflag)
{
if (eflag) do_encrypt=DES_ENCRYPT;
if (dflag) do_encrypt=DES_DECRYPT;
}
if (    (in != NULL) &&
(out != NULL) &&
#ifndef MSDOS
(stat(in,&ins) != -1) &&
(stat(out,&outs) != -1) &&
(ins.st_dev == outs.st_dev) &&
(ins.st_ino == outs.st_ino))
#else /* MSDOS */
(strcmp(in,out) == 0))
#endif
{
fputs("input and output file are the same\n",stderr);
EXIT(3);
}

#ifdef MSDOS
/* This should set the file to binary mode. */
{
#include <fcntl.h>
if (!(uflag && dflag))
setmode(fileno(DES_IN),O_BINARY);
if (!(uflag && eflag))
setmode(fileno(DES_OUT),O_BINARY);
}
#endif

doencryption();
 if (times(&time_fin)==-1){printf("\nERRORE times\n");return -1.0;}
 tempo=(time_fin.tms_stime - time_in.tms_stime)+  /* tempo di sistema */
       (time_fin.tms_utime - time_in.tms_utime);  /* tempo utente */
return tempo;
      }




doencryption()
       {
register int i;
des_key_schedule ks,ks2;
unsigned char iv[8],iv2[8];
char *p;
int num=0,j,k,l,rem,ll,len,last,ex=0;
des_cblock kk,k2;
FILE *O;
int Exit=0;
#ifndef MSDOS
static unsigned char buf[BUFSIZE+8],obuf[BUFSIZE+8];
#else
static unsigned char *buf=NULL,*obuf=NULL;

if (buf == NULL)
{
if (    (( buf=(unsigned char *)malloc(BUFSIZE+8)) == NULL) ||
((obuf=(unsigned char *)malloc(BUFSIZE+8)) == NULL))
{
fputs("Not enough memory\n",stderr);
Exit=10;
goto problems;
}
}
#endif
```

```
if (hflag)
{
j=(flag3?16:8);
p=key;
for (i=0; i<j; i++)
{
k=0;
if ((*p <= '9') && (*p >= '0'))
k=(*p-'0')<<4;
else if ((*p <= 'f') && (*p >= 'a'))
k=(*p-'a'+10)<<4;
else if ((*p <= 'F') && (*p >= 'A'))
k=(*p-'A'+10)<<4;
else
{
fputs("Bad hex key\n",stderr);
Exit=9;
goto problems;
}
p++;
if ((*p <= '9') && (*p >= '0'))
k|=(*p-'0');
else if ((*p <= 'f') && (*p >= 'a'))
k|=(*p-'a'+10);
else if ((*p <= 'F') && (*p >= 'A'))
k|=(*p-'A'+10);
else
{
fputs("Bad hex key\n",stderr);
Exit=9;
goto problems;
}
p++;
if (i < 8)
kk[i]=k;
else
k2[i-8]=k;
}
des_set_key_l((C_Block *)k2,ks2);
bzero(k2,sizeof(k2));
}

des_set_key_l((C_Block *)kk,ks);
bzero(key,sizeof(key));
bzero(kk,sizeof(kk));
/* woops - A bug that does not showup under unix :-( */
bzero(iv,sizeof(iv));
bzero(iv2,sizeof(iv2));

l=1;
rem=0;
/* first read */
if (eflag || (!dflag && cflag))
{
for (;;)
{
num=l=fread(&(buf[rem]),1,BUFSIZE,DES_IN);
l+=rem;
num+=rem;
if (l < 0)
{
perror("read error");
Exit=6;
goto problems;
}

rem=l%8;
len=l-rem;
if (feof(DES_IN))
{
```

```
srand(time(NULL));
for (i=7-rem; i>0; i--)
buf[l++]=rand()&0xff;
buf[l++]=rem;
ex=1;
len+=rem;
}
else
l-=rem;


if (bflag && !flag3)
for (i=0; i<l; i+=8)
des_ecb_encrypt(
(des_cblock *)&(buf[i]),
(des_cblock *)&(obuf[i]),
ks,do_encrypt);

if (rem) bcopy(&(buf[l]),buf,rem);

i=0;
while (i < l)
{

j=fwrite(obuf,1,l-i,DES_OUT);
if (j == -1)
{
perror("Write error");
Exit=7;
goto problems;
}
i+=j;
}
if (feof(DES_IN))
{
break;
}
}
}
else /* decrypt */
{
ex=1;
for (;;)
{
if (ex) {

l=fread(buf,1,BUFSIZE,DES_IN);
ex=0;
rem=l%8;
l-=rem;
}
if (l < 0)
{
perror("read error");
Exit=6;
goto problems;
}

if (bflag && !flag3)
for (i=0; i<l; i+=8)
{
des_ecb_encrypt(
(des_cblock *)&(buf[i]),
(des_cblock *)&(obuf[i]),
ks,do_encrypt);
        }

ll=fread(&(buf[rem]),1,BUFSIZE,DES_IN);
ll+=rem;
rem=ll%8;
ll-=rem;
```

```
if (feof(DES_IN) && (ll == 0))
{
last=obuf[l-1];

if ((last > 7) || (last < 0))
{
fputs("The file was not decrypted correctly.\n",
stderr);
/*Exit=8;
goto problems;*/
last=0;
}
l=l-8+last;
}
i=0;

while (i != l)
{
j=fwrite(obuf,1,l-i,DES_OUT);
if (j == -1)
{
perror("Write error");
Exit=7;
goto problems;
}
i+=j;
}
l=ll;
if ((l == 0) && feof(DES_IN)) break;
}
}

problems:
bzero(buf,sizeof(buf));
bzero(obuf,sizeof(obuf));
bzero(ks,sizeof(ks));
bzero(ks2,sizeof(ks2));
bzero(iv,sizeof(iv));
bzero(iv2,sizeof(iv2));
bzero(kk,sizeof(kk));
bzero(k2,sizeof(k2));
bzero(uubuf,sizeof(uubuf));
bzero(b,sizeof(b));
bzero(bb,sizeof(bb));
bzero(cksum,sizeof(cksum));
if (Exit) EXIT(Exit);
}



/* set_key.c */

/* Copyright (C) 1993 Eric Young - see README for more details */
/* set_key.c v 1.4 eay 24/9/91
 * 1.4 Speed up by 400% :-)
 * 1.3 added register declarations.
 * 1.2 unrolled make_key_sched a bit more
 * 1.1 added norm_expand_bits
 * 1.0 First working version
 */
#include "des_locl.h"
#include "sk.h"


/* NOW DEFINED IN des_local.h
 * See ecb_encrypt.c for a pseudo description of these macros.
 * #define PERM_OP(a,b,t,n,m) ((t)=((((a)>>(n))^(b))&(m)),\
 *      (b)^=(t),\
 *      (a)=((a)^((t)<<(n))))
 */
```

```
#define HPERM_OP(a,t,n,m) ((t)=((((a)<<(16-(n)))^(a))&(m)),\
(a)=(a)^(t)^(t>>(16-(n))))

static char shifts2[16]={0,0,1,1,1,1,1,1,0,1,1,1,1,1,1,0};

/* return 0 if key parity is odd (correct),
 * return -1 if key parity error,
 * return -2 if illegal weak key.
 */
int des_set_key_l(key,schedule)
des_cblock *key;
des_key_schedule schedule;
{
register unsigned long c,d,t,s;
register unsigned char *in;
register unsigned long *k;
register int i;

k=(unsigned long *)schedule;
in=(unsigned char *)key;

c2l(in,c);
c2l(in,d);

/* do PC1 in 60 simple operations */
/*      PERM_OP(d,c,t,4,0x0f0f0f0f);
HPERM_OP(c,t,-2, 0xcccc0000);
HPERM_OP(c,t,-1, 0xaaaa0000);
HPERM_OP(c,t, 8, 0x00ff0000);
HPERM_OP(c,t,-1, 0xaaaa0000);
HPERM_OP(d,t,-8, 0xff000000);
HPERM_OP(d,t, 8, 0x00ff0000);
HPERM_OP(d,t, 2, 0x33330000);
d=((d&0x00aa00aa)<<7)|((d&0x55005500)>>7)|(d&0xaa55aa55);
d=(d>>8)|((c&0xf0000000)>>4);
c&=0x0fffffff; */

/* I now do it in 47 simple operations :-)
 * Thanks to John Fletcher (john_fletcher@lccmail.ocf.llnl.gov)
 * for the inspiration. :-) */
PERM_OP (d,c,t,4,0x0f0f0f0f);
HPERM_OP(c,t,-2,0xcccc0000);
HPERM_OP(d,t,-2,0xcccc0000);
PERM_OP (d,c,t,1,0x55555555);
PERM_OP (c,d,t,8,0x00ff00ff);
PERM_OP (d,c,t,1,0x55555555);
d=      (((d&0x000000ff)<<16)| (d&0x0000ff00)     |
 ((d&0x00ff0000)>>16)|((c&0xf0000000)>>4));
c&=0x0fffffff;
/* calcola le 16 sottochiavi */
for (i=0; i<ITERATIONS; i++)
{
if (shifts2[i])
{ c=((c>>2)|(c<<26)); d=((d>>2)|(d<<26)); }
else
{ c=((c>>1)|(c<<27)); d=((d>>1)|(d<<27)); }
c&=0x0fffffff;
d&=0x0fffffff;
/* could be a few less shifts but I am to lazy at this
 * point in time to investigate */
s=      des_skb[0][ (c    )&0x3f                ]|
des_skb[1][((c>> 6)&0x03)|((c>> 7)&0x3c)]|
des_skb[2][((c>>13)&0x0f)|((c>>14)&0x30)]|
des_skb[3][((c>>20)&0x01)|((c>>21)&0x06) |
  ((c>>22)&0x38)];
t=      des_skb[4][ (d    )&0x3f                ]|
des_skb[5][((d>> 7)&0x03)|((d>> 8)&0x3c)]|
des_skb[6][ (d>>15)&0x3f                ]|
des_skb[7][((d>>21)&0x0f)|((d>>22)&0x30)];

/* table contained 0213 4657 */
```

```
*(k++)=((t<<16)|(s&0x0000ffff))&0xffffffff;
s=     ((s>>16)|(t&0xffff0000));

s=(s<<4)|(s>>28);
*(k++)=s&0xffffffff;
}
return(0);
}

int des_key_sched(key,schedule)
des_cblock *key;
des_key_schedule schedule;
{
return(des_set_key_l(key,schedule));
}
```

```
/* ecb_enc.c */
/* Copyright (C) 1993 Eric Young - see README for more details */
#include "des_locl.h"
#include "spr.h"

int des_ecb_encrypt(input,output,ks,encrypt)
des_cblock *input;
des_cblock *output;
des_key_schedule ks;
int encrypt;
{
register unsigned long l0,l1,t;
register unsigned char *in,*out;
unsigned long ll[2];

in=(unsigned char *)input;
out=(unsigned char *)output;
c2l(in,l0);
c2l(in,l1);
ll[0]=l0;
ll[1]=l1;
des_encrypt(ll,ll,ks,encrypt);
l0=ll[0];
l1=ll[1];
l2c(l0,out);
l2c(l1,out);
l0=l1=ll[0]=ll[1]=0;
return(0);
}

int des_encrypt(input,output,ks,encrypt)
unsigned long *input;
unsigned long *output;
des_key_schedule ks;
int encrypt;
{
register unsigned long l,r,t,u;
#ifdef ALT_ECB
register unsigned char *des_SP=(unsigned char *)des_SPtrans;
#endif
#ifdef MSDOS
union fudge {
unsigned long  l;
unsigned short s[2];
unsigned char  c[4];
} U,T;
#endif
register int i;
register unsigned long *s;

l=input[0];
```

```
r=input[1];

IP(l,r,t);   /* Effettua la permutazione IP */
/* Things have been modified so that the initial rotate is
 * done outside the loop.  This required the
 * des_SPtrans values in sp.h to be rotated 1 bit to the right.
 * One perl script later and things have a 5% speed up on a sparc2.
 * Thanks to Richard Outerbridge <71755.204@CompuServe.COM>
 * for pointing this out. */
t=(r<<1)|(r>>31);
r=(l<<1)|(l>>31);
l=t;

/* clear the top bits on machines with 8byte longs */
l&=0xffffffff;
r&=0xffffffff;

s=(unsigned long *)ks;
/* I don't know if it is worth the effort of loop unrolling the
 * inner loop */
if (encrypt)
{  /* si effettua la cifratura */
for (i=0; i<32; i+=4)
{
D_ENCRYPT(l,r,i+0); /*  1 */
D_ENCRYPT(r,l,i+2); /*  2 */
}
}
else
{ /* si effettua la decifratura */
for (i=30; i>0; i-=4)
{
D_ENCRYPT(l,r,i-0); /* 16 */
D_ENCRYPT(r,l,i-2); /* 15 */
}
}
l=(l>>1)|(l<<31);
r=(r>>1)|(r<<31);
/* clear the top bits on machines with 8byte longs */
l&=0xffffffff;
r&=0xffffffff;

FP(r,l,t); /* Effettua la permutazione IP^-1 */
output[0]=l;
output[1]=r;
l=r=t=u=0;
return(0);
}
```