U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology

FIPS PUB 181

FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION

AUTOMATED PASSWORD GENERATOR (APG)

CATEGORY: COMPUTER SECURITY

1993 October 5

FIPS PUB 181

FIPS PUB 181

FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION

AUTOMATED PASSWORD GENERATOR (APG)

CATEGORY: COMPUTER SECURITY

Computer Systems Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899

Issued October 5, 1993

Foreword

The Federal Information Processing Standards Publication Series of the National Institute of Standards and Technology (NIST) is the official publication relating to standards and guidelines adopted and promulgated under the provisions of Section 111(d) of the Federal Property and Administrative Services Act of 1949 as amended by the Computer Security Act of 1987, Public Law 100-235. These mandates have given the Secretary of Commerce and NIST important responsibilities for improving the utilization and management of computer and related telecommunications systems in the Federal Government. The NIST, through its Computer Systems Laboratory, provides leadership, technical guidance, and coordination of Government efforts in the development of standards and guidelines in these areas.

Comments concerning Federal Information Processing Standards Publications are welcomed and should be addressed to the Director, Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899.

James H. Burrows, Director
Computer Systems Laboratory

Abstract

This publication specifies a standard to be used by Federal organizations that require computer generated pronounceable passwords to authenticate the personal identity of an automated data processing (ADP) system user, and to authorize access to system resources. The standard describes an automated password generation algorithm that randomly creates simple pronounceable syllables as passwords. The password generator accepts input from a random number generator based on the Data Encryption Standard (DES) cryptographic algorithm defined in Federal Information Processing Standard 46-1 (FIPS PUB 46-1).

Key words: access control; authentication; Data Encryption Standard (DES); Federal Information Processing Standard (FIPS); identification; passwords; security.

National Institute of Standards and Technology
FIPS PUB 181
56 pages (Oct. 5, 1993)
CODEN: FIPPAT

U.S. Government Printing Office
Washington: 1993

For sale by the National Technical Information Service
U.S. Department of Commerce
Springfield, VA 22161

Federal Information Processing Standards Publication 181

1993 October 5

Announcing the Standard for

Automated Password Generator

1. **Name of Standard.** Automated Password Generator.

2. **Category of Standard.** Computer Security.

3. **Explanation.** A password is a protected character string used to authenticate the identity of a computer system user or to authorize access to system resources. When users are allowed to select their own passwords they often select passwords that are easily compromised. An automated password generator creates random passwords that have no association with a particular user.

This Automated Password Generator Standard specifies an algorithm to generate passwords for the protection of computer resources. This standard is for use in conjunction with FIPS PUB 112, Password Usage Standard, which provides basic security criteria for the design, implementation, and use of passwords. The algorithm uses random numbers to select the characters that form the random pronounceable passwords. The random numbers are generated by a random number subroutine based on the Electronic Codebook mode of the Data Encryption Standard (DES) (FIPS PUB 46-1). The random number subroutine uses a pseudorandom DES key generated in accordance with the procedure described in Appendix C of ANSI X9.17.

Similar to DES, the FIPS for Automated Password Generator is an interoperability standard. Interoperability standards specify functions and formats so that data transmitted can be properly acted upon when received by another computer. This type of standard is independent of physical implementation. Implementors are required to use the algorithm defined in the FIPS, however, they are not constrained in how they package it. For discussion purposes a NIST implementation of the Automated Password Generator is provided. It is expected that commercial implementations will be based on the latest technologies and differ from NIST's, however the results should be logically equivalent to that of this FIPS.

4. **Approving Authority.** Secretary of Commerce.

5. **Maintenance Agency.** U.S. Department of Commerce, National Institute of Standards and Technology (NIST), Computer Systems Laboratory (CSL).

6. **Cross Index.**

a. American National Standards Institute (ANSI) X9.28, *Financial Institution Multiple Center Key Management (Wholesale)* Draft.
b. Department of Defense CSC-STD-002-85, *Password Management Guideline.*
c. Federal Information Processing Standards Publication (FIPS PUB) 48, *Guidelines on Evaluation of Techniques for Automated Personal Identification.*
d. Federal Information Processing Standards Publication (FIPS PUB) 46-1, *Data Encryption Standard.*
e. Federal Information Processing Standards Publication (FIPS PUB) 81, *DES Modes of Operation.*
f. Federal Information Processing Standards Publication (FIPS PUB) 83, *Guideline on User Authentication Techniques for Computer Network Access Control.*
g. Federal Information Processing Standards Publication (FIPS PUB) 112, *Password Usage.*
h. Federal Information Processing Standards Publication (FIPS PUB) 171, *Key Management Using ANSI X9.17.*
i. National Technical Information Service (NTIS) AD A 017676, *A Random Word Generator for Pronounceable Passwords.*

7. **Objectives.** The objectives of this standard are to:

a. improve the administration of password systems that are used for authenticating the identity of individuals accessing computer resources or files;

b. provide a standard automated method for producing pronounceable passwords that have no association with a particular user;

c. produce passwords that are easily remembered, stored, and entered into computer systems, yet not readily susceptible to automated techniques that have been developed to search for and disclose passwords.

8. **Applicability.** This standard is applicable to the development of procurement or design specifications for implementing an automatic password generation algorithm within a computer system. It shall be used by all Federal departments and agencies when there is a requirement for computer generated pronounceable passwords for authenticating users of computer systems, or for authorizing access to resources in those systems.

This standard does not require the use of passwords in a computer system, but establishes an automatic password generation algorithm for use in systems where an agency's computer security policy requires computer generated pronounceable passwords. It should be used in conjunction with FIPS PUB 112, Password Usage Standard, which specifies basic security criteria for the design, implementation, and use of passwords.

9. Export Control. The Bureau of Export Administration, U.S. Department of Commerce, is responsible for administering export controls on cryptographic products used for authentication and access control, which categories would include implementations of the Automated Password Generator. Vendors should contact the following for a product classification:

Bureau of Export Administration
U.S. Department of Commerce
P.O. Box 273
Washington, DC 20044
Telephone: (202) 482-0708

Following this determination, the vendor will be informed whether an export license is required and will be provided further information as appropriate.

10. Specifications. Federal Information Processing Standard (FIPS) 181, Automated Password Generator (affixed);

11. Qualifications. The Automated Password Generator uses the Electronic Codebook (ECB) mode of the Data Encryption Standard (DES), Federal Information Processing Standard 46-1 (FIPS PUB 46-1), as the random number generator. This mode of operation is specified in FIPS 81, DES Modes of Operation.

The protection provided by the DES algorithm against potential threats has been reviewed every 5 years since its adoption in 1977 and has been reaffirmed during each of those reviews. The DES, and the possible threats reducing the security provided by the use of DES, will undergo continual review by NIST and other cognizant Federal organizations. The new technology available at review time will be evaluated to determine its impact on the DES. In addition, the awareness of any breakthrough in technology or any mathematical weakness of the algorithm will cause NIST to reevaluate the DES and provide necessary revisions.

12. Implementation Schedule. This Standard becomes effective March 25, 1994.

13. Waivers. Under certain exceptional circumstances, the heads of Federal departments and agencies may approve waivers to Federal Information Processing Standards (FIPS). The head of such agency may redelegate such authority only to a senior official designated pursuant to section 3506(b) of Title 44, U.S. Code. Waivers shall be granted only when compliance with a standard would:

a. adversely affect the accomplishment of the mission of an operator of a Federal computer system, or

b. cause a major adverse financial impact on the operator which is not offset by Government-wide savings.

Agency heads may act upon a written waiver request containing the information detailed above. Agency heads may also act without a written waiver request when they determine that conditions for meeting the standard cannot be met. Agency heads may approve waivers only by a written decision which explains the basis on which the agency head made the required finding(s). A copy of each such decision, with procurement sensitive or classified portions clearly identified, shall be sent to: National Institute of Standards and Technology; ATTN: FIPS Waiver Decisions; Technology Building, Room B-154; Gaithersburg, MD 20899.

In addition, notice of each waiver granted and each delegation of authority to approve waivers shall be sent promptly to the Committee on Government Operations of the House of Representatives and the Committee on Government Affairs of the Senate and shall be published promptly in the Federal Register.

When the determination on a waiver applies to the procurement of equipment and/or services, a notice of the waiver determination must be published in the Commerce Business Daily as a part of the notice of solicitation for offers of an acquisition or, if the waiver determination is made after that notice is published, by amendment to such notice.

A copy of the waiver, any supporting documents, the document approving the waiver, and any supporting and accompanying documents, with such deletions as the agency is authorized and decides to make under 5 U.S.C Sec. 552(b), shall be part of the procurement documentation and retained by the agency.

14. Where to Obtain Copies. Copies of this publication are available for sale by the National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. When ordering, refer to Federal Information Processing Standards Publication 181 (FIPSPUB181), and identify the title. When microfiche is desired, this should be specified. Payment may be made by check, money order, credit card, or deposit account.

Federal Information
Processing Standards Publication 181

1993 October 5

Announcing the Standard for

Automated Password Generator

Contents

1.0 INTRODUCTION

The Automated Password Generator standard is derived from a C-code version of the program described in "A Random Word Generator For Pronounceable Passwords," National Technical Information Service (NTIS) AD A 017676. The original program used Unix system functions to produce the random numbers needed by the password generator. These functions were replaced with a DES-based random number subroutine that uses DES in the Electronic Code Book (ECB) mode. As input, DES uses the old password or user supplied character string, and a pseudorandom key created in accordance with the procedure described in Appendix C of ANSI X9.17. Any change to either the key or input data string causes DES to generate an entirely different random number. Every time this occurs the password generator creates a new random password.

2.0 TECHNICAL EXPLANATION

The Automated Password Generator standard is organized as a main procedure that references three major components: (1) the "unit table"; (2) the "digram table"; and (3) the "random number subroutine." The random password generator works by forming pronounceable syllables and concatenating them to form a word. Rules of pronounceability are stored in a table for every unit and every pair of units (digram). The rules are used to determine whether a given unit is legal or illegal, based on its position within the syllable and adjacent units. Most rules and checks are syllable oriented and do not depend on anything outside the current syllable. The main procedure (algorithm) defines the internal rules used to generate random words. The three components and the algorithm are described below.

Appendix A is the code for the NIST implementation of the Automated Password Generator standard. This code consists of the C-code version of the program described in "A Random Word Generator For Pronounceable Passwords," the code that comprises the DES random number subroutine, the actual DES subroutine, and the code for generating the pseudorandom key. Implementations in other programming languages are acceptable, however, the results obtained must be logically equivalent to those produced by this standard.

In the NIST implementation of the password generator, the values selected for the two DES keys and the seed for the random number generator are readable in the code (Appendix A). In an actual vendor or user developed implementation the values of the keys and the seed would be secret, randomly generated values set by the application.

2.1 Unit Table

The unit table defines the units (alphabetic characters) and specifies rules pertaining to the individual units used in a randomly generated word. For example, the location of vowels in the words generated is determined by these rules. The unit table used in the Automated Password Generator standard is identical to that furnished in the report "A Random Word

Generator For Pronounceable Passwords" (item i in Cross Index).

2.2 Digram Table

The digram table specifies rules about all possible pairs of units and the juxtaposition of units. The table contains one entry for every pair of units (digram), whether that pair is allowed or not. The random word generator ensures that the rules specified in the digram table are satisfied for every two consecutive units in the word being formed. The digram table is also from the original report.

2.3 Random Number Generator Subroutine

The random number generator uses a DES subroutine to produce double precision floating point values between 0 and (excluding) 1. These numbers are multiplied by a program variable **n** which is an integer value. This operation yields a random integer between 0 and (**n**-1) inclusive. The random numbers created by the DES routine serve as input to the random word generator. The subroutine to generate these numbers is called by the word generator each time a character (unit) is needed.

Not all characters generated will be acceptable to the word generator in every position of the word. Each character is checked for appropriateness using the rules defined by the unit and digram tables. Therefore the random number generator subroutine will be repeatedly called until an acceptable character is returned. An upper limit of 100 calls is placed on generating any particular character. If that number is reached the whole word is discarded and the program starts over.

The actual distribution of legal units is different for every position in a particular word which, for any unit, depends on the units that precede it as well as the units and digram tables. The random number subroutine itself makes no tests for legal units.

As its input DES accepts two 64 bit data blocks. One consists of the old password or a data string; the other is a 64 bit (56 bits + 8 parity bits) pseudorandom key derived using the procedure described in Appendix C of ANSI X9.17. The old password is entered manually from the keyboard. An input array is created from the first eight bytes of the password or input string. The program will accept a null string (carriage return). All characters past the eighth are disregarded. If the input block is less than eight characters long the extra elements in the input array are filled with ASCII 0. The Electronic Codebook (ECB) mode of DES described in FIPS 81 ("DES Modes of Operation," December 2, 1980) is then used to encrypt the input data. The output is a 64-bit random number which is the encrypted form of the input.

The first function in the DES structure is *setkey*(), which converts the pseudorandom key to a format used by DES for the encryption. The command-line options sent to *setkey* are (0, 0, **key**). The first 0 is set so that *setkey*() does not generate parity; the second 0 tells *setkey*() that encryption (rather than decryption) is required. **Key** is a pointer to the beginning of the key array. After *setkey*(), the *des*() function is called. For input it uses the addresses of the input and output arrays. Both input and output are defined as unsigned character arrays of length 8 bytes.

The output array, **out**, is sent to a function, *answer*(), which returns the final required number. The function *answer*() takes in the address of the output array as an unsigned char pointer and the integer **n** for which a value of 0 to (**n**-1) is needed by the random word program. This function creates a variable **sum**, defined as an unsigned integer. To obtain a numerical value from the output character array, it adds the ASCII values of the first three elements in the **out** array and stores the sum in the variable **sum**. Thus, **sum** = **out[0]** + **out[1]** + **out[2]**, which is an integer. To obtain a number with the required range of 0 to **n**-1 from **sum**, the function takes the modulus of **sum** and **n**, (**sum%n**). This value is then returned to the calling function within the random word program.

2.4 Random Word Algorithm

The algorithm used to generate random words is fixed and cannot be modified without changing the logic of the program. The function of the algorithm is to determine whether a given unit, generated by the random unit subroutine, can be appended to the end of the partial word formed so far. Rules of pronounceability are stored in the unit and digram tables discussed above. The rules are used to check if a given unit is legal or illegal. If illegal, the unit is discarded and the random unit subroutine is called again. Once a unit is accepted, various state variables are updated and a unit for the next position in the word is tried. Most rules and checks are syllable oriented and do not depend on anything outside the current syllables. When the end of the word is reached, additional checks are made before the algorithm terminates.

Passwords created by this automated password generator are composed of the 26 characters of the English alphabet. Although numbers and special characters are not permitted, the password space, which is a function of the number of characters in the password, is very large. Approximately 18 million 6-character, 5.7 billion 8-character, and 1.6 trillion 10-character passwords can be created by the program. Users should select a password space commensurate with the level of security required for the information being protected.

The password algorithm does not preclude the generation of words found in a standard English dictionary. If required, a computerized dictionary could be used to check for English words, and the implementation could include software tests to prevent them from being offered to users as passwords.

2.5 NIST Implementation

Figure 1 is a block diagram of the NIST implementation of the automated password generation algorithm. Appendix A contains the C-code for the DES, random key generation, and random word generation routines that were used in the implementation (see shaded boxes in Fig. 1). The personal computer used by NIST to demonstrate the standard is implementation dependent. NIST replaced the Unix random number routine in the original version of the program with the "DES Randomizer" and "Generate Random Key" function. The DES randomizer accepts an old password and a pseudorandom key created in accordance with Appendix C of ANSI X9.17 (FIPSPUB 171) and generates a random number. This number is used by the Random Word Generator to develop a password. As the password is being generated each group of letters is subjected to tests of grammar and semantics to determine if an acceptable word has been created. If all tests are passed, the new password is output to the PC.

In the NIST implementation, the values for **minlen** and **maxlen**, which define the minimun and maximum size of the password, were set at 5 and 8 respectively. A user needing a fixed length password word could set these variables to a specific value.

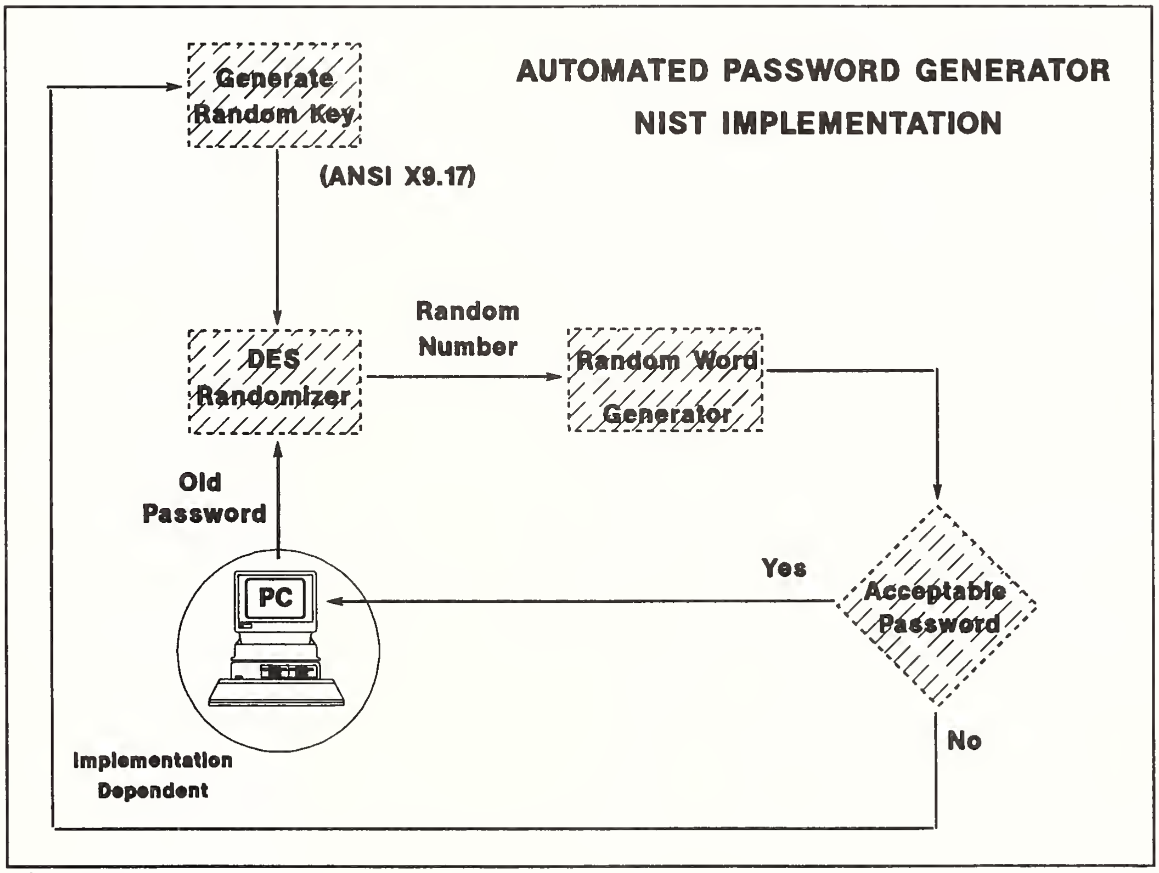

Figure 1

Appendix A

The following is a listing of the source code referenced in the Automated Password Generator Standard.

```
/*
 *   randomword (word, hyphenated_word, minlen, maxlen, restrict, seed)
 */



#include <stdio.h>
#include <sys/types.h>
#include <time.h>

#define RAN_DEBUG
#define B1

#define TRUE                 1
#define FALSE                0

#define RULE_SIZE              (sizeof(rules)/sizeof(struct unit))
#define ALLOWED(flag)  (digram[units_in_syllable[current_unit - 1]][unit] & (flag))

#define MAX_UNACCEPTABLE      20
#define MAX_RETRIES              (4 * (int) pwlen + RULE_SIZE)

#define NOT_BEGIN_SYLLABLE    010
#define NO_FINAL_SPLIT         04
#define VOWEL                02
#define ALTERNATE_VOWEL       01
#define NO_SPECIAL_RULE        0

#define BEGIN                0200
#define NOT_BEGIN              0100
#define BREAK                 040
#define PREFIX               020
#define ILLEGAL_PAIR           010
#define SUFFIX               04
#define END                  02
#define NOT_END               01
#define ANY_COMBINATION        0

typedef unsigned int uint;
typedef int    boolean;

static int get_word();
static boolean have_initial_y();
static boolean illegal_placement();
static boolean improper_word();
static boolean have_final_split();
static char    *get_syllable();
static unsigned short int  random_unit();
static unsigned int  randint();
static unsigned short int  get_random();
static void set_seed();

extern char    *calloc ();
extern char    *malloc ();
extern char    *strcpy ();
extern char    *strcat ();
extern long time ();
extern long atol ();
extern double drand48();
extern int fscanf();
extern int fprintf();
```

```
struct unit
{
    char     unit_code[5];
    unsigned short int  flags;
};

static struct unit  rules[] =
{
    "a", VOWEL,
    "b", NO_SPECIAL_RULE,
    "c", NO_SPECIAL_RULE,
    "d", NO_SPECIAL_RULE,
    "e", NO_FINAL_SPLIT | VOWEL,
    "f", NO_SPECIAL_RULE,
    "g", NO_SPECIAL_RULE,
    "h", NO_SPECIAL_RULE,
    "i", VOWEL,
    "j", NO_SPECIAL_RULE,
    "k", NO_SPECIAL_RULE,
    "l", NO_SPECIAL_RULE,
    "m", NO_SPECIAL_RULE,
    "n", NO_SPECIAL_RULE,
    "o", VOWEL,
    "p", NO_SPECIAL_RULE,
    "r", NO_SPECIAL_RULE,
    "s", NO_SPECIAL_RULE,
    "t", NO_SPECIAL_RULE,
    "u", VOWEL,
    "v", NO_SPECIAL_RULE,
    "w", NO_SPECIAL_RULE,
    "x", NOT_BEGIN_SYLLABLE,
    "y", ALTERNATE_VOWEL | VOWEL,
    "z", NO_SPECIAL_RULE,
    "ch", NO_SPECIAL_RULE,
    "gh", NO_SPECIAL_RULE,
    "ph", NO_SPECIAL_RULE,
    "rh", NO_SPECIAL_RULE,
    "sh", NO_SPECIAL_RULE,
    "th", NO_SPECIAL_RULE,
    "wh", NO_SPECIAL_RULE,
    "qu", NO_SPECIAL_RULE,
    "ck", NOT_BEGIN_SYLLABLE
};

static int  digram[][RULE_SIZE] =
{
     /* aa */ ILLEGAL_PAIR,
     /* ab */ ANY_COMBINATION,
     /* ac */ ANY_COMBINATION,
     /* ad */ ANY_COMBINATION,
     /* ae */ ILLEGAL_PAIR,
     /* af */ ANY_COMBINATION,
     /* ag */ ANY_COMBINATION,
     /* ah */ NOT_BEGIN | BREAK | NOT_END,
     /* ai */ ANY_COMBINATION,
     /* aj */ ANY_COMBINATION,
     /* ak */ ANY_COMBINATION,
     /* al */ ANY_COMBINATION,
     /* am */ ANY_COMBINATION,
     /* an */ ANY_COMBINATION,
     /* ao */ ILLEGAL_PAIR,
     /* ap */ ANY_COMBINATION,
     /* ar */ ANY_COMBINATION,
     /* as */ ANY_COMBINATION,
     /* at */ ANY_COMBINATION,
     /* au */ ANY_COMBINATION,
     /* av */ ANY_COMBINATION,
     /* aw */ ANY_COMBINATION,
     /* ax */ ANY_COMBINATION,
     /* ay */ ANY_COMBINATION,
    /* az */ ANY_COMBINATION,
    /* ach */ ANY_COMBINATION,
    /* agh */ ILLEGAL_PAIR,
    /* aph */ ANY_COMBINATION,
    /* arh */ ILLEGAL_PAIR,
    /* ash */ ANY_COMBINATION,
    /* ath */ ANY_COMBINATION,
    /* awh */ ILLEGAL_PAIR,
    /* aqu */ BREAK | NOT_END,
    /* ack */ ANY_COMBINATION,
    /* ba */ ANY_COMBINATION,
    /* bb */ NOT_BEGIN | BREAK | NOT_END,
    /* bc */ NOT_BEGIN | BREAK | NOT_END,
    /* bd */ NOT_BEGIN | BREAK | NOT_END,
    /* be */ ANY_COMBINATION,
    /* bf */ NOT_BEGIN | BREAK | NOT_END,
    /* bg */ NOT_BEGIN | BREAK | NOT_END,
    /* bh */ NOT_BEGIN | BREAK | NOT_END,
    /* bi */ ANY_COMBINATION,
    /* bj */ NOT_BEGIN | BREAK | NOT_END,
    /* bk */ NOT_BEGIN | BREAK | NOT_END,
    /* bl */ BEGIN | SUFFIX | NOT_END,
    /* bm */ NOT_BEGIN | BREAK | NOT_END,
    /* bn */ NOT_BEGIN | BREAK | NOT_END,
    /* bo */ ANY_COMBINATION,
    /* bp */ NOT_BEGIN | BREAK | NOT_END,
    /* br */ BEGIN | END,
    /* bs */ NOT_BEGIN,
    /* bt */ NOT_BEGIN | BREAK | NOT_END,
    /* bu */ ANY_COMBINATION,
    /* bv */ NOT_BEGIN | BREAK | NOT_END,
    /* bw */ NOT_BEGIN | BREAK | NOT_END,
    /* bx */ ILLEGAL_PAIR,
    /* by */ ANY_COMBINATION,
    /* bz */ NOT_BEGIN | BREAK | NOT_END,
    /* bch */ NOT_BEGIN | BREAK | NOT_END,
    /* bgh */ ILLEGAL_PAIR,
    /* bph */ NOT_BEGIN | BREAK | NOT_END,
    /* brh */ ILLEGAL_PAIR,
    /* bsh */ NOT_BEGIN | BREAK | NOT_END,
    /* bth */ NOT_BEGIN | BREAK | NOT_END,
    /* bwh */ ILLEGAL_PAIR,
    /* bqu */ NOT_BEGIN | BREAK | NOT_END,
    /* bck */ ILLEGAL_PAIR,
    /* ca */ ANY_COMBINATION,
    /* cb */ NOT_BEGIN | BREAK | NOT_END,
    /* cc */ NOT_BEGIN | BREAK | NOT_END,
    /* cd */ NOT_BEGIN | BREAK | NOT_END,
    /* ce */ ANY_COMBINATION,
    /* cf */ NOT_BEGIN | BREAK | NOT_END,
    /* cg */ NOT_BEGIN | BREAK | NOT_END,
    /* ch */ NOT_BEGIN | BREAK | NOT_END,
    /* ci */ ANY_COMBINATION,
    /* cj */ NOT_BEGIN | BREAK | NOT_END,
    /* ck */ NOT_BEGIN | BREAK | NOT_END,
    /* cl */ SUFFIX | NOT_END,
    /* cm */ NOT_BEGIN | BREAK | NOT_END,
    /* cn */ NOT_BEGIN | BREAK | NOT_END,
    /* co */ ANY_COMBINATION,
    /* cp */ NOT_BEGIN | BREAK | NOT_END,
    /* cr */ NOT_END,
    /* cs */ NOT_BEGIN | END,
    /* ct */ NOT_BEGIN | PREFIX,
    /* cu */ ANY_COMBINATION,
    /* cv */ NOT_BEGIN | BREAK | NOT_END,
    /* cw */ NOT_BEGIN | BREAK | NOT_END,
    /* cx */ ILLEGAL_PAIR,
    /* cy */ ANY_COMBINATION,
    /* cz */ NOT_BEGIN | BREAK | NOT_END,
    /* cch */ ILLEGAL_PAIR,
    /* cgh */ ILLEGAL_PAIR,
    /* cph */ NOT_BEGIN | BREAK | NOT_END,
/* crh */ ILLEGAL_PAIR,
/* csh */ NOT_BEGIN | BREAK | NOT_END,
/* cth */ NOT_BEGIN | BREAK | NOT_END,
/* cwh */ ILLEGAL_PAIR,
/* cqu */ NOT_BEGIN | SUFFIX | NOT_END,
/* cck */ ILLEGAL_PAIR,
/* da */ ANY_COMBINATION,
/* db */ NOT_BEGIN | BREAK | NOT_END,
/* dc */ NOT_BEGIN | BREAK | NOT_END,
/* dd */ NOT_BEGIN,
/* de */ ANY_COMBINATION,
/* df */ NOT_BEGIN | BREAK | NOT_END,
/* dg */ NOT_BEGIN | BREAK | NOT_END,
/* dh */ NOT_BEGIN | BREAK | NOT_END,
/* di */ ANY_COMBINATION,
/* dj */ NOT_BEGIN | BREAK | NOT_END,
/* dk */ NOT_BEGIN | BREAK | NOT_END,
/* dl */ NOT_BEGIN | BREAK | NOT_END,
/* dm */ NOT_BEGIN | BREAK | NOT_END,
/* dn */ NOT_BEGIN | BREAK | NOT_END,
/* do */ ANY_COMBINATION,
/* dp */ NOT_BEGIN | BREAK | NOT_END,
/* dr */ BEGIN | NOT_END,
/* ds */ NOT_BEGIN | END,
/* dt */ NOT_BEGIN | BREAK | NOT_END,
/* du */ ANY_COMBINATION,
/* dv */ NOT_BEGIN | BREAK | NOT_END,
/* dw */ NOT_BEGIN | BREAK | NOT_END,
/* dx */ ILLEGAL_PAIR,
/* dy */ ANY_COMBINATION,
/* dz */ NOT_BEGIN | BREAK | NOT_END,
/* dch */ NOT_BEGIN | BREAK | NOT_END,
/* dgh */ NOT_BEGIN | BREAK | NOT_END,
/* dph */ NOT_BEGIN | BREAK | NOT_END,
/* drh */ ILLEGAL_PAIR,
/* dsh */ NOT_BEGIN | NOT_END,
/* dth */ NOT_BEGIN | PREFIX,
/* dwh */ ILLEGAL_PAIR,
/* dqu */ NOT_BEGIN | BREAK | NOT_END,
/* dck */ ILLEGAL_PAIR,
/* ea */ ANY_COMBINATION,
/* eb */ ANY_COMBINATION,
/* ec */ ANY_COMBINATION,
/* ed */ ANY_COMBINATION,
/* ee */ ANY_COMBINATION,
/* ef */ ANY_COMBINATION,
/* eg */ ANY_COMBINATION,
/* eh */ NOT_BEGIN | BREAK | NOT_END,
/* ei */ NOT_END,
/* ej */ ANY_COMBINATION,
/* ek */ ANY_COMBINATION,
/* el */ ANY_COMBINATION,
/* em */ ANY_COMBINATION,
/* en */ ANY_COMBINATION,
/* eo */ BREAK,
/* ep */ ANY_COMBINATION,
/* er */ ANY_COMBINATION,
/* es */ ANY_COMBINATION,
/* et */ ANY_COMBINATION,
/* eu */ ANY_COMBINATION,
/* ev */ ANY_COMBINATION,
/* ew */ ANY_COMBINATION,
/* ex */ ANY_COMBINATION,
/* ey */ ANY_COMBINATION,
/* ez */ ANY_COMBINATION,
/* ech */ ANY_COMBINATION,
/* egh */ NOT_BEGIN | BREAK | NOT_END,
/* eph */ ANY_COMBINATION,
/* erh */ ILLEGAL_PAIR,
/* esh */ ANY_COMBINATION,
/* eth */ ANY_COMBINATION,
/* ewh */ ILLEGAL_PAIR,
/* equ */ BREAK | NOT_END,
/* eck */ ANY_COMBINATION,
/* fa */ ANY_COMBINATION,
/* fb */ NOT_BEGIN | BREAK | NOT_END,
/* fc */ NOT_BEGIN | BREAK | NOT_END,
/* fd */ NOT_BEGIN | BREAK | NOT_END,
/* fe */ ANY_COMBINATION,
/* ff */ NOT_BEGIN,
/* fg */ NOT_BEGIN | BREAK | NOT_END,
/* fh */ NOT_BEGIN | BREAK | NOT_END,
/* fi */ ANY_COMBINATION,
/* fj */ NOT_BEGIN | BREAK | NOT_END,
/* fk */ NOT_BEGIN | BREAK | NOT_END,
/* fl */ BEGIN | SUFFIX | NOT_END,
/* fm */ NOT_BEGIN | BREAK | NOT_END,
/* fn */ NOT_BEGIN | BREAK | NOT_END,
/* fo */ ANY_COMBINATION,
/* fp */ NOT_BEGIN | BREAK | NOT_END,
/* fr */ BEGIN | NOT_END,
/* fs */ NOT_BEGIN,
/* ft */ NOT_BEGIN,
/* fu */ ANY_COMBINATION,
/* fv */ NOT_BEGIN | BREAK | NOT_END,
/* fw */ NOT_BEGIN | BREAK | NOT_END,
/* fx */ ILLEGAL_PAIR,
/* fy */ NOT_BEGIN,
/* fz */ NOT_BEGIN | BREAK | NOT_END,
/* fch */ NOT_BEGIN | BREAK | NOT_END,
/* fgh */ NOT_BEGIN | BREAK | NOT_END,
/* fph */ NOT_BEGIN | BREAK | NOT_END,
/* frh */ ILLEGAL_PAIR,
/* fsh */ NOT_BEGIN | BREAK | NOT_END,
/* fth */ NOT_BEGIN | BREAK | NOT_END,
/* fwh */ ILLEGAL_PAIR,
/* fqu */ NOT_BEGIN | BREAK | NOT_END,
/* fck */ ILLEGAL_PAIR,
/* ga */ ANY_COMBINATION,
/* gb */ NOT_BEGIN | BREAK | NOT_END,
/* gc */ NOT_BEGIN | BREAK | NOT_END,
/* gd */ NOT_BEGIN | BREAK | NOT_END,
/* ge */ ANY_COMBINATION,
/* gf */ NOT_BEGIN | BREAK | NOT_END,
/* gg */ NOT_BEGIN,
/* gh */ NOT_BEGIN | BREAK | NOT_END,
/* gi */ ANY_COMBINATION,
/* gj */ NOT_BEGIN | BREAK | NOT_END,
/* gk */ ILLEGAL_PAIR,
/* gl */ BEGIN | SUFFIX | NOT_END,
/* gm */ NOT_BEGIN | BREAK | NOT_END,
/* gn */ NOT_BEGIN | BREAK | NOT_END,
/* go */ ANY_COMBINATION,
/* gp */ NOT_BEGIN | BREAK | NOT_END,
/* gr */ BEGIN | NOT_END,
/* gs */ NOT_BEGIN | END,
/* gt */ NOT_BEGIN | BREAK | NOT_END,
/* gu */ ANY_COMBINATION,
/* gv */ NOT_BEGIN | BREAK | NOT_END,
/* gw */ NOT_BEGIN | BREAK | NOT_END,
/* gx */ ILLEGAL_PAIR,
/* gy */ NOT_BEGIN,
/* gz */ NOT_BEGIN | BREAK | NOT_END,
/* gch */ NOT_BEGIN | BREAK | NOT_END,
/* ggh */ ILLEGAL_PAIR,
/* gph */ NOT_BEGIN | BREAK | NOT_END,
/* grh */ ILLEGAL_PAIR,
/* gsh */ NOT_BEGIN,
/* gth */ NOT_BEGIN,
/* gwh */ ILLEGAL_PAIR,
/* gqu */ NOT_BEGIN | BREAK | NOT_END,
/* gck */ ILLEGAL_PAIR,
/* ha */ ANY_COMBINATION,
/* hb */ NOT_BEGIN | BREAK | NOT_END,
/* hc */ NOT_BEGIN | BREAK | NOT_END,
/* hd */ NOT_BEGIN | BREAK | NOT_END,
/* he */ ANY_COMBINATION,
/* hf */ NOT_BEGIN | BREAK | NOT_END,
/* hg */ NOT_BEGIN | BREAK | NOT_END,
/* hh */ ILLEGAL_PAIR,
/* hi */ ANY_COMBINATION,
/* hj */ NOT_BEGIN | BREAK | NOT_END,
/* hk */ NOT_BEGIN | BREAK | NOT_END,
/* hl */ NOT_BEGIN | BREAK | NOT_END,
/* hm */ NOT_BEGIN | BREAK | NOT_END,
/* hn */ NOT_BEGIN | BREAK | NOT_END,
/* ho */ ANY_COMBINATION,
/* hp */ NOT_BEGIN | BREAK | NOT_END,
/* hr */ NOT_BEGIN | BREAK | NOT_END,
/* hs */ NOT_BEGIN | BREAK | NOT_END,
/* ht */ NOT_BEGIN | BREAK | NOT_END,
/* hu */ ANY_COMBINATION,
/* hv */ NOT_BEGIN | BREAK | NOT_END,
/* hw */ NOT_BEGIN | BREAK | NOT_END,
/* hx */ ILLEGAL_PAIR,
/* hy */ ANY_COMBINATION,
/* hz */ NOT_BEGIN | BREAK | NOT_END,
/* hch */ NOT_BEGIN | BREAK | NOT_END,
/* hgh */ NOT_BEGIN | BREAK | NOT_END,
/* hph */ NOT_BEGIN | BREAK | NOT_END,
/* hrh */ ILLEGAL_PAIR,
/* hsh */ NOT_BEGIN | BREAK | NOT_END,
/* hth */ NOT_BEGIN | BREAK | NOT_END,
/* hwh */ ILLEGAL_PAIR,
/* hqu */ NOT_BEGIN | BREAK | NOT_END,
/* hck */ ILLEGAL_PAIR,
/* ia */ ANY_COMBINATION,
/* ib */ ANY_COMBINATION,
/* ic */ ANY_COMBINATION,
/* id */ ANY_COMBINATION,
/* ie */ NOT_BEGIN,
/* if */ ANY_COMBINATION,
/* ig */ ANY_COMBINATION,
/* ih */ NOT_BEGIN | BREAK | NOT_END,
/* ii */ ILLEGAL_PAIR,
/* ij */ ANY_COMBINATION,
/* ik */ ANY_COMBINATION,
/* il */ ANY_COMBINATION,
/* im */ ANY_COMBINATION,
/* in */ ANY_COMBINATION,
/* io */ BREAK,
/* ip */ ANY_COMBINATION,
/* ir */ ANY_COMBINATION,
/* is */ ANY_COMBINATION,
/* it */ ANY_COMBINATION,
/* iu */ NOT_BEGIN | BREAK | NOT_END,
/* iv */ ANY_COMBINATION,
/* iw */ NOT_BEGIN | BREAK | NOT_END,
/* ix */ ANY_COMBINATION,
/* iy */ NOT_BEGIN | BREAK | NOT_END,
/* iz */ ANY_COMBINATION,
/* ich */ ANY_COMBINATION,
/* igh */ NOT_BEGIN,
/* iph */ ANY_COMBINATION,
/* irh */ ILLEGAL_PAIR,
/* ish */ ANY_COMBINATION,
/* ith */ ANY_COMBINATION,
/* iwh */ ILLEGAL_PAIR,
/* iqu */ BREAK | NOT_END,
/* ick */ ANY_COMBINATION,
/* ja */ ANY_COMBINATION,
/* jb */ NOT_BEGIN | BREAK | NOT_END,
/* jc */ NOT_BEGIN | BREAK | NOT_END,
/* jd */ NOT_BEGIN | BREAK | NOT_END,
/* je */ ANY_COMBINATION,
/* jf */ NOT_BEGIN | BREAK | NOT_END,
/* jg */ ILLEGAL_PAIR,
/* jh */ NOT_BEGIN | BREAK | NOT_END,
/* ji */ ANY_COMBINATION,
/* jj */ ILLEGAL_PAIR,
/* jk */ NOT_BEGIN | BREAK | NOT_END,
/* jl */ NOT_BEGIN | BREAK | NOT_END,
/* jm */ NOT_BEGIN | BREAK | NOT_END,
/* jn */ NOT_BEGIN | BREAK | NOT_END,
/* jo */ ANY_COMBINATION,
/* jp */ NOT_BEGIN | BREAK | NOT_END,
/* jr */ NOT_BEGIN | BREAK | NOT_END,
/* js */ NOT_BEGIN | BREAK | NOT_END,
/* jt */ NOT_BEGIN | BREAK | NOT_END,
/* ju */ ANY_COMBINATION,
/* jv */ NOT_BEGIN | BREAK | NOT_END,
/* jw */ NOT_BEGIN | BREAK | NOT_END,
/* jx */ ILLEGAL_PAIR,
/* jy */ NOT_BEGIN,
/* jz */ NOT_BEGIN | BREAK | NOT_END,
/* jch */ NOT_BEGIN | BREAK | NOT_END,
/* jgh */ NOT_BEGIN | BREAK | NOT_END,
/* jph */ NOT_BEGIN | BREAK | NOT_END,
/* jrh */ ILLEGAL_PAIR,
/* jsh */ NOT_BEGIN | BREAK | NOT_END,
/* jth */ NOT_BEGIN | BREAK | NOT_END,
/* jwh */ ILLEGAL_PAIR,
/* jqu */ NOT_BEGIN | BREAK | NOT_END,
/* jck */ ILLEGAL_PAIR,
/* ka */ ANY_COMBINATION,
/* kb */ NOT_BEGIN | BREAK | NOT_END,
/* kc */ NOT_BEGIN | BREAK | NOT_END,
/* kd */ NOT_BEGIN | BREAK | NOT_END,
/* ke */ ANY_COMBINATION,
/* kf */ NOT_BEGIN | BREAK | NOT_END,
/* kg */ NOT_BEGIN | BREAK | NOT_END,
/* kh */ NOT_BEGIN | BREAK | NOT_END,
/* ki */ ANY_COMBINATION,
/* kj */ NOT_BEGIN | BREAK | NOT_END,
/* kk */ NOT_BEGIN | BREAK | NOT_END,
/* kl */ SUFFIX | NOT_END,
/* km */ NOT_BEGIN | BREAK | NOT_END,
/* kn */ BEGIN | SUFFIX | NOT_END,
/* ko */ ANY_COMBINATION,
/* kp */ NOT_BEGIN | BREAK | NOT_END,
/* kr */ SUFFIX | NOT_END,
/* ks */ NOT_BEGIN | END,
/* kt */ NOT_BEGIN | BREAK | NOT_END,
/* ku */ ANY_COMBINATION,
/* kv */ NOT_BEGIN | BREAK | NOT_END,
/* kw */ NOT_BEGIN | BREAK | NOT_END,
/* kx */ ILLEGAL_PAIR,
/* ky */ NOT_BEGIN,
/* kz */ NOT_BEGIN | BREAK | NOT_END,
/* kch */ NOT_BEGIN | BREAK | NOT_END,
/* kgh */ NOT_BEGIN | BREAK | NOT_END,
/* kph */ NOT_BEGIN | PREFIX,
/* krh */ ILLEGAL_PAIR,
/* ksh */ NOT_BEGIN,
/* kth */ NOT_BEGIN | BREAK | NOT_END,
/* kwh */ ILLEGAL_PAIR,
/* kqu */ NOT_BEGIN | BREAK | NOT_END,
/* kck */ ILLEGAL_PAIR,
/* la */ ANY_COMBINATION,
/* lb */ NOT_BEGIN | PREFIX,
/* lc */ NOT_BEGIN | BREAK | NOT_END,
/* ld */ NOT_BEGIN | PREFIX,
/* le */ ANY_COMBINATION,
/* lf */ NOT_BEGIN | PREFIX,
/* lg */ NOT_BEGIN | PREFIX,
/* lh */ NOT_BEGIN | BREAK | NOT_END,
/* li */ ANY_COMBINATION,
/* lj */ NOT_BEGIN | PREFIX,
/* lk */ NOT_BEGIN | PREFIX,
/* ll */ NOT_BEGIN | PREFIX,
/* lm */ NOT_BEGIN | PREFIX,
/* ln */ NOT_BEGIN | BREAK | NOT_END,
/* lo */ ANY_COMBINATION,
/* lp */ NOT_BEGIN | PREFIX,
/* lr */ NOT_BEGIN | BREAK | NOT_END,
/* ls */ NOT_BEGIN,
/* lt */ NOT_BEGIN | PREFIX,
/* lu */ ANY_COMBINATION,
/* lv */ NOT_BEGIN | PREFIX,
/* lw */ NOT_BEGIN | BREAK | NOT_END,
/* lx */ ILLEGAL_PAIR,
/* ly */ ANY_COMBINATION,
/* lz */ NOT_BEGIN | BREAK | NOT_END,
/* lch */ NOT_BEGIN | PREFIX,
/* lgh */ NOT_BEGIN | BREAK | NOT_END,
/* lph */ NOT_BEGIN | PREFIX,
/* lrh */ ILLEGAL_PAIR,
/* lsh */ NOT_BEGIN | PREFIX,
/* lth */ NOT_BEGIN | PREFIX,
/* lwh */ ILLEGAL_PAIR,
/* lqu */ NOT_BEGIN | BREAK | NOT_END,
/* lck */ ILLEGAL_PAIR,
/* ma */ ANY_COMBINATION,
/* mb */ NOT_BEGIN | BREAK | NOT_END,
/* mc */ NOT_BEGIN | BREAK | NOT_END,
/* md */ NOT_BEGIN | BREAK | NOT_END,
/* me */ ANY_COMBINATION,
/* mf */ NOT_BEGIN | BREAK | NOT_END,
/* mg */ NOT_BEGIN | BREAK | NOT_END,
/* mh */ NOT_BEGIN | BREAK | NOT_END,
/* mi */ ANY_COMBINATION,
/* mj */ NOT_BEGIN | BREAK | NOT_END,
/* mk */ NOT_BEGIN | BREAK | NOT_END,
/* ml */ NOT_BEGIN | BREAK | NOT_END,
/* mm */ NOT_BEGIN,
/* mn */ NOT_BEGIN | BREAK | NOT_END,
/* mo */ ANY_COMBINATION,
/* mp */ NOT_BEGIN,
/* mr */ NOT_BEGIN | BREAK | NOT_END,
/* ms */ NOT_BEGIN,
/* mt */ NOT_BEGIN,
/* mu */ ANY_COMBINATION,
/* mv */ NOT_BEGIN | BREAK | NOT_END,
/* mw */ NOT_BEGIN | BREAK | NOT_END,
/* mx */ ILLEGAL_PAIR,
/* my */ ANY_COMBINATION,
/* mz */ NOT_BEGIN | BREAK | NOT_END,
/* mch */ NOT_BEGIN | PREFIX,
/* mgh */ NOT_BEGIN | BREAK | NOT_END,
/* mph */ NOT_BEGIN,
/* mrh */ ILLEGAL_PAIR,
/* msh */ NOT_BEGIN,
/* mth */ NOT_BEGIN,
/* mwh */ ILLEGAL_PAIR,
/* mqu */ NOT_BEGIN | BREAK | NOT_END,
/* mck */ ILLEGAL_PAIR,
/* na */ ANY_COMBINATION,
/* nb */ NOT_BEGIN | BREAK | NOT_END,
/* nc */ NOT_BEGIN | BREAK | NOT_END,
/* nd */ NOT_BEGIN,
/* ne */ ANY_COMBINATION,
/* nf */ NOT_BEGIN | BREAK | NOT_END,
/* ng */ NOT_BEGIN | PREFIX,
/* nh */ NOT_BEGIN | BREAK | NOT_END,
/* ni */ ANY_COMBINATION,
/* nj */ NOT_BEGIN | BREAK | NOT_END,
/* nk */ NOT_BEGIN | PREFIX,
/* nl */ NOT_BEGIN | BREAK | NOT_END,
/* nm */ NOT_BEGIN | BREAK | NOT_END,
/* nn */ NOT_BEGIN,
/* no */ ANY_COMBINATION,
/* np */ NOT_BEGIN | BREAK | NOT_END,
/* nr */ NOT_BEGIN | BREAK | NOT_END,
/* ns */ NOT_BEGIN,
/* nt */ NOT_BEGIN,
/* nu */ ANY_COMBINATION,
/* nv */ NOT_BEGIN | BREAK | NOT_END,
/* nw */ NOT_BEGIN | BREAK | NOT_END,
/* nx */ ILLEGAL_PAIR,
/* ny */ NOT_BEGIN,
/* nz */ NOT_BEGIN | BREAK | NOT_END,
/* nch */ NOT_BEGIN | PREFIX,
/* ngh */ NOT_BEGIN | BREAK | NOT_END,
/* nph */ NOT_BEGIN | PREFIX,
/* nrh */ ILLEGAL_PAIR,
/* nsh */ NOT_BEGIN,
/* nth */ NOT_BEGIN,
/* nwh */ ILLEGAL_PAIR,
/* nqu */ NOT_BEGIN | BREAK | NOT_END,
/* nck */ NOT_BEGIN | PREFIX,
/* oa */ ANY_COMBINATION,
/* ob */ ANY_COMBINATION,
/* oc */ ANY_COMBINATION,
/* od */ ANY_COMBINATION,
/* oe */ ILLEGAL_PAIR,
/* of */ ANY_COMBINATION,
/* og */ ANY_COMBINATION,
/* oh */ NOT_BEGIN | BREAK | NOT_END,
/* oi */ ANY_COMBINATION,
/* oj */ ANY_COMBINATION,
/* ok */ ANY_COMBINATION,
/* ol */ ANY_COMBINATION,
/* om */ ANY_COMBINATION,
/* on */ ANY_COMBINATION,
/* oo */ ANY_COMBINATION,
/* op */ ANY_COMBINATION,
/* or */ ANY_COMBINATION,
/* os */ ANY_COMBINATION,
/* ot */ ANY_COMBINATION,
/* ou */ ANY_COMBINATION,
/* ov */ ANY_COMBINATION,
/* ow */ ANY_COMBINATION,
/* ox */ ANY_COMBINATION,
/* oy */ ANY_COMBINATION,
/* oz */ ANY_COMBINATION,
/* och */ ANY_COMBINATION,
/* ogh */ NOT_BEGIN,
/* oph */ ANY_COMBINATION,
/* orh */ ILLEGAL_PAIR,
/* osh */ ANY_COMBINATION,
/* oth */ ANY_COMBINATION,
/* owh */ ILLEGAL_PAIR,
/* oqu */ BREAK | NOT_END,
/* ock */ ANY_COMBINATION,
/* pa */ ANY_COMBINATION,
/* pb */ NOT_BEGIN | BREAK | NOT_END,
/* pc */ NOT_BEGIN | BREAK | NOT_END,
/* pd */ NOT_BEGIN | BREAK | NOT_END,
/* pe */ ANY_COMBINATION,
/* pf */ NOT_BEGIN | BREAK | NOT_END,
/* pg */ NOT_BEGIN | BREAK | NOT_END,
/* ph */ NOT_BEGIN | BREAK | NOT_END,
/* pi */ ANY_COMBINATION,
/* pj */ NOT_BEGIN | BREAK | NOT_END,
/* pk */ NOT_BEGIN | BREAK | NOT_END,
/* pl */ SUFFIX | NOT_END,
/* pm */ NOT_BEGIN | BREAK | NOT_END,
/* pn */ NOT_BEGIN | BREAK | NOT_END,
/* po */ ANY_COMBINATION,
/* pp */ NOT_BEGIN | PREFIX,
/* pr */ NOT_END,
/* ps */ NOT_BEGIN | END,
/* pt */ NOT_BEGIN | END,
/* pu */ NOT_BEGIN | END,
/* pv */ NOT_BEGIN | BREAK | NOT_END,
/* pw */ NOT_BEGIN | BREAK | NOT_END,
/* px */ ILLEGAL_PAIR,
/* py */ ANY_COMBINATION,
/* pz */ NOT_BEGIN | BREAK | NOT_END,
/* pch */ NOT_BEGIN | BREAK | NOT_END,
/* pgh */ NOT_BEGIN | BREAK | NOT_END,
/* pph */ NOT_BEGIN | BREAK | NOT_END,
/* prh */ ILLEGAL_PAIR,
/* psh */ NOT_BEGIN | BREAK | NOT_END,
/* pth */ NOT_BEGIN | BREAK | NOT_END,
/* pwh */ ILLEGAL_PAIR,
/* pqu */ NOT_BEGIN | BREAK | NOT_END,
/* pck */ ILLEGAL_PAIR,
/* ra */ ANY_COMBINATION,
/* rb */ NOT_BEGIN | PREFIX,
/* rc */ NOT_BEGIN | PREFIX,
/* rd */ NOT_BEGIN | PREFIX,
/* re */ ANY_COMBINATION,
/* rf */ NOT_BEGIN | PREFIX,
/* rg */ NOT_BEGIN | PREFIX,
/* rh */ NOT_BEGIN | BREAK | NOT_END,
/* ri */ ANY_COMBINATION,
/* rj */ NOT_BEGIN | PREFIX,
/* rk */ NOT_BEGIN | PREFIX,
/* rl */ NOT_BEGIN | PREFIX,
/* rm */ NOT_BEGIN | PREFIX,
/* rn */ NOT_BEGIN | PREFIX,
/* ro */ ANY_COMBINATION,
/* rp */ NOT_BEGIN | PREFIX,
/* rr */ NOT_BEGIN | PREFIX,
/* rs */ NOT_BEGIN | PREFIX,
/* rt */ NOT_BEGIN | PREFIX,
/* ru */ ANY_COMBINATION,
/* rv */ NOT_BEGIN | PREFIX,
/* rw */ NOT_BEGIN | BREAK | NOT_END,
/* rx */ ILLEGAL_PAIR,
/* ry */ ANY_COMBINATION,
/* rz */ NOT_BEGIN | PREFIX,
/* rch */ NOT_BEGIN | PREFIX,
/* rgh */ NOT_BEGIN | BREAK | NOT_END,
/* rph */ NOT_BEGIN | PREFIX,
/* rrh */ ILLEGAL_PAIR,
/* rsh */ NOT_BEGIN | PREFIX,
/* rth */ NOT_BEGIN | PREFIX,
/* rwh */ ILLEGAL_PAIR,
/* rqu */ NOT_BEGIN | PREFIX | NOT_END,
/* rck */ NOT_BEGIN | PREFIX,
/* sa */ ANY_COMBINATION,
/* sb */ NOT_BEGIN | BREAK | NOT_END,
/* sc */ NOT_END,
/* sd */ NOT_BEGIN | BREAK | NOT_END,
/* se */ ANY_COMBINATION,
/* sf */ NOT_BEGIN | BREAK | NOT_END,
/* sg */ NOT_BEGIN | BREAK | NOT_END,
/* sh */ NOT_BEGIN | BREAK | NOT_END,
/* si */ ANY_COMBINATION,
/* sj */ NOT_BEGIN | BREAK | NOT_END,
/* sk */ ANY_COMBINATION,
/* sl */ BEGIN | SUFFIX | NOT_END,
/* sm */ SUFFIX | NOT_END,
/* sn */ PREFIX | SUFFIX | NOT_END,
/* so */ ANY_COMBINATION,
/* sp */ ANY_COMBINATION,
/* sr */ NOT_BEGIN | NOT_END,
/* ss */ NOT_BEGIN | PREFIX,
/* st */ ANY_COMBINATION,
/* su */ ANY_COMBINATION,
/* sv */ NOT_BEGIN | BREAK | NOT_END,
/* sw */ BEGIN | SUFFIX | NOT_END,
    /* sx */ ILLEGAL_PAIR,
    /* sy */ ANY_COMBINATION,
    /* sz */ NOT_BEGIN | BREAK | NOT_END,
    /* sch */ BEGIN | SUFFIX | NOT_END,
    /* sgh */ NOT_BEGIN | BREAK | NOT_END,
    /* sph */ NOT_BEGIN | BREAK | NOT_END,
    /* srh */ ILLEGAL_PAIR,
    /* ssh */ NOT_BEGIN | BREAK | NOT_END,
    /* sth */ NOT_BEGIN | BREAK | NOT_END,
    /* swh */ ILLEGAL_PAIR,
    /* squ */ SUFFIX | NOT_END,
    /* sck */ NOT_BEGIN,
    /* ta */ ANY_COMBINATION,
    /* tb */ NOT_BEGIN | BREAK | NOT_END,
    /* tc */ NOT_BEGIN | BREAK | NOT_END,
    /* td */ NOT_BEGIN | BREAK | NOT_END,
    /* te */ ANY_COMBINATION,
    /* tf */ NOT_BEGIN | BREAK | NOT_END,
    /* tg */ NOT_BEGIN | BREAK | NOT_END,
    /* th */ NOT_BEGIN | BREAK | NOT_END,
    /* ti */ ANY_COMBINATION,
    /* tj */ NOT_BEGIN | BREAK | NOT_END,
    /* tk */ NOT_BEGIN | BREAK | NOT_END,
    /* tl */ NOT_BEGIN | BREAK | NOT_END,
    /* tm */ NOT_BEGIN | BREAK | NOT_END,
    /* tn */ NOT_BEGIN | BREAK | NOT_END,
    /* to */ ANY_COMBINATION,
    /* tp */ NOT_BEGIN | BREAK | NOT_END,
    /* tr */ NOT_END,
    /* ts */ NOT_BEGIN | END,
    /* tt */ NOT_BEGIN | PREFIX,
    /* tu */ ANY_COMBINATION,
    /* tv */ NOT_BEGIN | BREAK | NOT_END,
    /* tw */ BEGIN | SUFFIX | NOT_END,
    /* tx */ ILLEGAL_PAIR,
    /* ty */ ANY_COMBINATION,
    /* tz */ NOT_BEGIN | BREAK | NOT_END,
    /* tch */ NOT_BEGIN,
    /* tgh */ NOT_BEGIN | BREAK | NOT_END,
    /* tph */ NOT_BEGIN | END,
    /* trh */ ILLEGAL_PAIR,
    /* tsh */ NOT_BEGIN | END,
    /* tth */ NOT_BEGIN | BREAK | NOT_END,
    /* twh */ ILLEGAL_PAIR,
    /* tqu */ NOT_BEGIN | BREAK | NOT_END,
    /* tck */ ILLEGAL_PAIR,
    /* ua */ NOT_BEGIN | BREAK | NOT_END,
    /* ub */ ANY_COMBINATION,
    /* uc */ ANY_COMBINATION,
    /* ud */ ANY_COMBINATION,
    /* ue */ NOT_BEGIN,
    /* uf */ ANY_COMBINATION,
    /* ug */ ANY_COMBINATION,
    /* uh */ NOT_BEGIN | BREAK | NOT_END,
    /* ui */ NOT_BEGIN | BREAK | NOT_END,
    /* uj */ ANY_COMBINATION,
    /* uk */ ANY_COMBINATION,
    /* ul */ ANY_COMBINATION,
    /* um */ ANY_COMBINATION,
    /* un */ ANY_COMBINATION,
    /* uo */ NOT_BEGIN | BREAK,
    /* up */ ANY_COMBINATION,
    /* ur */ ANY_COMBINATION,
    /* us */ ANY_COMBINATION,
    /* ut */ ANY_COMBINATION,
    /* uu */ ILLEGAL_PAIR,
    /* uv */ ANY_COMBINATION,
    /* uw */ NOT_BEGIN | BREAK | NOT_END,
    /* ux */ ANY_COMBINATION,
    /* uy */ NOT_BEGIN | BREAK | NOT_END,
    /* uz */ ANY_COMBINATION,
    /* uch */ ANY_COMBINATION,
/* ugh */ NOT_BEGIN | PREFIX,
/* uph */ ANY_COMBINATION,
/* urh */ ILLEGAL_PAIR,
/* ush */ ANY_COMBINATION,
/* uth */ ANY_COMBINATION,
/* uwh */ ILLEGAL_PAIR,
/* uqu */ BREAK | NOT_END,
/* uck */ ANY_COMBINATION,
/* va */ ANY_COMBINATION,
/* vb */ NOT_BEGIN | BREAK | NOT_END,
/* vc */ NOT_BEGIN | BREAK | NOT_END,
/* vd */ NOT_BEGIN | BREAK | NOT_END,
/* ve */ ANY_COMBINATION,
/* vf */ NOT_BEGIN | BREAK | NOT_END,
/* vg */ NOT_BEGIN | BREAK | NOT_END,
/* vh */ NOT_BEGIN | BREAK | NOT_END,
/* vi */ ANY_COMBINATION,
/* vj */ NOT_BEGIN | BREAK | NOT_END,
/* vk */ NOT_BEGIN | BREAK | NOT_END,
/* vl */ NOT_BEGIN | BREAK | NOT_END,
/* vm */ NOT_BEGIN | BREAK | NOT_END,
/* vn */ NOT_BEGIN | BREAK | NOT_END,
/* vo */ ANY_COMBINATION,
/* vp */ NOT_BEGIN | BREAK | NOT_END,
/* vr */ NOT_BEGIN | BREAK | NOT_END,
/* vs */ NOT_BEGIN | BREAK | NOT_END,
/* vt */ NOT_BEGIN | BREAK | NOT_END,
/* vu */ ANY_COMBINATION,
/* vv */ NOT_BEGIN | BREAK | NOT_END,
/* vw */ NOT_BEGIN | BREAK | NOT_END,
/* vx */ ILLEGAL_PAIR,
/* vy */ NOT_BEGIN,
/* vz */ NOT_BEGIN | BREAK | NOT_END,
/* vch */ NOT_BEGIN | BREAK | NOT_END,
/* vgh */ NOT_BEGIN | BREAK | NOT_END,
/* vph */ NOT_BEGIN | BREAK | NOT_END,
/* vrh */ ILLEGAL_PAIR,
/* vsh */ NOT_BEGIN | BREAK | NOT_END,
/* vth */ NOT_BEGIN | BREAK | NOT_END,
/* vwh */ ILLEGAL_PAIR,
/* vqu */ NOT_BEGIN | BREAK | NOT_END,
/* vck */ ILLEGAL_PAIR,
/* wa */ ANY_COMBINATION,
/* wb */ NOT_BEGIN | PREFIX,
/* wc */ NOT_BEGIN | BREAK | NOT_END,
/* wd */ NOT_BEGIN | PREFIX | END,
/* we */ ANY_COMBINATION,
/* wf */ NOT_BEGIN | PREFIX,
/* wg */ NOT_BEGIN | PREFIX | END,
/* wh */ NOT_BEGIN | BREAK | NOT_END,
/* wi */ ANY_COMBINATION,
/* wj */ NOT_BEGIN | BREAK | NOT_END,
/* wk */ NOT_BEGIN | PREFIX,
/* wl */ NOT_BEGIN | PREFIX | SUFFIX,
/* wm */ NOT_BEGIN | PREFIX,
/* wn */ NOT_BEGIN | PREFIX,
/* wo */ ANY_COMBINATION,
/* wp */ NOT_BEGIN | PREFIX,
/* wr */ BEGIN | SUFFIX | NOT_END,
/* ws */ NOT_BEGIN | PREFIX,
/* wt */ NOT_BEGIN | PREFIX,
/* wu */ ANY_COMBINATION,
/* wv */ NOT_BEGIN | PREFIX,
/* ww */ NOT_BEGIN | BREAK | NOT_END,
/* wx */ NOT_BEGIN | PREFIX,
/* wy */ ANY_COMBINATION,
/* wz */ NOT_BEGIN | PREFIX,
/* wch */ NOT_BEGIN,
/* wgh */ NOT_BEGIN | BREAK | NOT_END,
/* wph */ NOT_BEGIN,
/* wrh */ ILLEGAL_PAIR,
/* wsh */ NOT_BEGIN,
/* wth */ NOT_BEGIN,
/* wwh */ ILLEGAL_PAIR,
/* wqu */ NOT_BEGIN | BREAK | NOT_END,
/* wck */ NOT_BEGIN,
/* xa */ NOT_BEGIN,
/* xb */ NOT_BEGIN | BREAK | NOT_END,
/* xc */ NOT_BEGIN | BREAK | NOT_END,
/* xd */ NOT_BEGIN | BREAK | NOT_END,
/* xe */ NOT_BEGIN,
/* xf */ NOT_BEGIN | BREAK | NOT_END,
/* xg */ NOT_BEGIN | BREAK | NOT_END,
/* xh */ NOT_BEGIN | BREAK | NOT_END,
/* xi */ NOT_BEGIN,
/* xj */ NOT_BEGIN | BREAK | NOT_END,
/* xk */ NOT_BEGIN | BREAK | NOT_END,
/* xl */ NOT_BEGIN | BREAK | NOT_END,
/* xm */ NOT_BEGIN | BREAK | NOT_END,
/* xn */ NOT_BEGIN | BREAK | NOT_END,
/* xo */ NOT_BEGIN,
/* xp */ NOT_BEGIN | BREAK | NOT_END,
/* xr */ NOT_BEGIN | BREAK | NOT_END,
/* xs */ NOT_BEGIN | BREAK | NOT_END,
/* xt */ NOT_BEGIN | BREAK | NOT_END,
/* xu */ NOT_BEGIN,
/* xv */ NOT_BEGIN | BREAK | NOT_END,
/* xw */ NOT_BEGIN | BREAK | NOT_END,
/* xx */ ILLEGAL_PAIR,
/* xy */ NOT_BEGIN,
/* xz */ NOT_BEGIN | BREAK | NOT_END,
/* xch */ NOT_BEGIN | BREAK | NOT_END,
/* xgh */ NOT_BEGIN | BREAK | NOT_END,
/* xph */ NOT_BEGIN | BREAK | NOT_END,
/* xrh */ ILLEGAL_PAIR,
/* xsh */ NOT_BEGIN | BREAK | NOT_END,
/* xth */ NOT_BEGIN | BREAK | NOT_END,
/* xwh */ ILLEGAL_PAIR,
/* xqu */ NOT_BEGIN | BREAK | NOT_END,
/* xck */ ILLEGAL_PAIR,
/* ya */ ANY_COMBINATION,
/* yb */ NOT_BEGIN,
/* yc */ NOT_BEGIN | NOT_END,
/* yd */ NOT_BEGIN,
/* ye */ ANY_COMBINATION,
/* yf */ NOT_BEGIN | NOT_END,
/* yg */ NOT_BEGIN,
/* yh */ NOT_BEGIN | BREAK | NOT_END,
/* yi */ BEGIN | NOT_END,
/* yj */ NOT_BEGIN | NOT_END,
/* yk */ NOT_BEGIN,
/* yl */ NOT_BEGIN | NOT_END,
/* ym */ NOT_BEGIN,
/* yn */ NOT_BEGIN,
/* yo */ ANY_COMBINATION,
/* yp */ NOT_BEGIN,
/* yr */ NOT_BEGIN | BREAK | NOT_END,
/* ys */ NOT_BEGIN,
/* yt */ NOT_BEGIN,
/* yu */ ANY_COMBINATION,
/* yv */ NOT_BEGIN | NOT_END,
/* yw */ NOT_BEGIN | BREAK | NOT_END,
/* yx */ NOT_BEGIN,
/* yy */ ILLEGAL_PAIR,
/* yz */ NOT_BEGIN,
/* ych */ NOT_BEGIN | BREAK | NOT_END,
/* ygh */ NOT_BEGIN | BREAK | NOT_END,
/* yph */ NOT_BEGIN | BREAK | NOT_END,
/* yrh */ ILLEGAL_PAIR,
/* ysh */ NOT_BEGIN | BREAK | NOT_END,
/* yth */ NOT_BEGIN | BREAK | NOT_END,
/* ywh */ ILLEGAL_PAIR,
/* yqu */ NOT_BEGIN | BREAK | NOT_END,
/* yck */ ILLEGAL_PAIR,
/* za */ ANY_COMBINATION,
/* zb */ NOT_BEGIN | BREAK | NOT_END,
/* zc */ NOT_BEGIN | BREAK | NOT_END,
/* zd */ NOT_BEGIN | BREAK | NOT_END,
/* ze */ ANY_COMBINATION,
/* zf */ NOT_BEGIN | BREAK | NOT_END,
/* zg */ NOT_BEGIN | BREAK | NOT_END,
/* zh */ NOT_BEGIN | BREAK | NOT_END,
/* zi */ ANY_COMBINATION,
/* zj */ NOT_BEGIN | BREAK | NOT_END,
/* zk */ NOT_BEGIN | BREAK | NOT_END,
/* zl */ NOT_BEGIN | BREAK | NOT_END,
/* zm */ NOT_BEGIN | BREAK | NOT_END,
/* zn */ NOT_BEGIN | BREAK | NOT_END,
/* zo */ ANY_COMBINATION,
/* zp */ NOT_BEGIN | BREAK | NOT_END,
/* zr */ NOT_BEGIN | NOT_END,
/* zs */ NOT_BEGIN | BREAK | NOT_END,
/* zt */ NOT_BEGIN,
/* zu */ ANY_COMBINATION,
/* zv */ NOT_BEGIN | BREAK | NOT_END,
/* zw */ SUFFIX | NOT_END,
/* zx */ ILLEGAL_PAIR,
/* zy */ ANY_COMBINATION,
/* zz */ NOT_BEGIN,
/* zch */ NOT_BEGIN | BREAK | NOT_END,
/* zgh */ NOT_BEGIN | BREAK | NOT_END,
/* zph */ NOT_BEGIN | BREAK | NOT_END,
/* zrh */ ILLEGAL_PAIR,
/* zsh */ NOT_BEGIN | BREAK | NOT_END,
/* zth */ NOT_BEGIN | BREAK | NOT_END,
/* zwh */ ILLEGAL_PAIR,
/* zqu */ NOT_BEGIN | BREAK | NOT_END,
/* zck */ ILLEGAL_PAIR,
/* cha */ ANY_COMBINATION,
/* chb */ NOT_BEGIN | BREAK | NOT_END,
/* chc */ NOT_BEGIN | BREAK | NOT_END,
/* chd */ NOT_BEGIN | BREAK | NOT_END,
/* che */ ANY_COMBINATION,
/* chf */ NOT_BEGIN | BREAK | NOT_END,
/* chg */ NOT_BEGIN | BREAK | NOT_END,
/* chh */ NOT_BEGIN | BREAK | NOT_END,
/* chi */ ANY_COMBINATION,
/* chj */ NOT_BEGIN | BREAK | NOT_END,
/* chk */ NOT_BEGIN | BREAK | NOT_END,
/* chl */ NOT_BEGIN | BREAK | NOT_END,
/* chm */ NOT_BEGIN | BREAK | NOT_END,
/* chn */ NOT_BEGIN | BREAK | NOT_END,
/* cho */ ANY_COMBINATION,
/* chp */ NOT_BEGIN | BREAK | NOT_END,
/* chr */ NOT_END,
/* chs */ NOT_BEGIN | BREAK | NOT_END,
/* cht */ NOT_BEGIN | BREAK | NOT_END,
/* chu */ ANY_COMBINATION,
/* chv */ NOT_BEGIN | BREAK | NOT_END,
/* chw */ NOT_BEGIN | NOT_END,
/* chx */ ILLEGAL_PAIR,
/* chy */ ANY_COMBINATION,
/* chz */ NOT_BEGIN | BREAK | NOT_END,
/* chch */ ILLEGAL_PAIR,
/* chgh */ NOT_BEGIN | BREAK | NOT_END,
/* chph */ NOT_BEGIN | BREAK | NOT_END,
/* chrh */ ILLEGAL_PAIR,
/* chsh */ NOT_BEGIN | BREAK | NOT_END,
/* chth */ NOT_BEGIN | BREAK | NOT_END,
/* chwh */ ILLEGAL_PAIR,
/* chqu */ NOT_BEGIN | BREAK | NOT_END,
/* chck */ ILLEGAL_PAIR,
/* gha */ ANY_COMBINATION,
/* ghb */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghc */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghd */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghe */ ANY_COMBINATION,
/* ghf */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghg */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghh */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghi */ BEGIN | NOT_END,
/* ghj */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghk */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghl */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghm */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghn */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* gho */ BEGIN | NOT_END,
/* ghp */ NOT_BEGIN | BREAK | NOT_END,
/* ghr */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghs */ NOT_BEGIN | PREFIX,
/* ght */ NOT_BEGIN | PREFIX,
/* ghu */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghv */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghw */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghx */ ILLEGAL_PAIR,
/* ghy */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghz */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghch */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghgh */ ILLEGAL_PAIR,
/* ghph */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghrh */ ILLEGAL_PAIR,
/* ghsh */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghth */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghwh */ ILLEGAL_PAIR,
/* ghqu */ NOT_BEGIN | BREAK | PREFIX | NOT_END,
/* ghck */ ILLEGAL_PAIR,
/* pha */ ANY_COMBINATION,
/* phb */ NOT_BEGIN | BREAK | NOT_END,
/* phc */ NOT_BEGIN | BREAK | NOT_END,
/* phd */ NOT_BEGIN | BREAK | NOT_END,
/* phe */ ANY_COMBINATION,
/* phf */ NOT_BEGIN | BREAK | NOT_END,
/* phg */ NOT_BEGIN | BREAK | NOT_END,
/* phh */ NOT_BEGIN | BREAK | NOT_END,
/* phi */ ANY_COMBINATION,
/* phj */ NOT_BEGIN | BREAK | NOT_END,
/* phk */ NOT_BEGIN | BREAK | NOT_END,
/* phl */ BEGIN | SUFFIX | NOT_END,
/* phm */ NOT_BEGIN | BREAK | NOT_END,
/* phn */ NOT_BEGIN | BREAK | NOT_END,
/* pho */ ANY_COMBINATION,
/* php */ NOT_BEGIN | BREAK | NOT_END,
/* phr */ NOT_END,
/* phs */ NOT_BEGIN,
/* pht */ NOT_BEGIN,
/* phu */ ANY_COMBINATION,
/* phv */ NOT_BEGIN | NOT_END,
/* phw */ NOT_BEGIN | NOT_END,
/* phx */ ILLEGAL_PAIR,
/* phy */ NOT_BEGIN,
/* phz */ NOT_BEGIN | BREAK | NOT_END,
/* phch */ NOT_BEGIN | BREAK | NOT_END,
/* phgh */ NOT_BEGIN | BREAK | NOT_END,
/* phph */ ILLEGAL_PAIR,
/* phrh */ ILLEGAL_PAIR,
/* phsh */ NOT_BEGIN | BREAK | NOT_END,
/* phth */ NOT_BEGIN | BREAK | NOT_END,
/* phwh */ ILLEGAL_PAIR,
/* phqu */ NOT_BEGIN | BREAK | NOT_END,
/* phck */ ILLEGAL_PAIR,
/* rha */ BEGIN | NOT_END,
/* rhb */ ILLEGAL_PAIR,
/* rhc */ ILLEGAL_PAIR,
/* rhd */ ILLEGAL_PAIR,
/* rhe */ BEGIN | NOT_END,
/* rhf */ ILLEGAL_PAIR,
/* rhg */ ILLEGAL_PAIR,
/* rhh */ ILLEGAL_PAIR,
/* rhi */ BEGIN | NOT_END,
/* rhj */ ILLEGAL_PAIR,
/* rhk */ ILLEGAL_PAIR,
/* rhl */ ILLEGAL_PAIR,
/* rhm */ ILLEGAL_PAIR,
/* rhn */ ILLEGAL_PAIR,
/* rho */ BEGIN | NOT_END,
/* rhp */ ILLEGAL_PAIR,
/* rhr */ ILLEGAL_PAIR,
/* rhs */ ILLEGAL_PAIR,
/* rht */ ILLEGAL_PAIR,
/* rhu */ BEGIN | NOT_END,
/* rhv */ ILLEGAL_PAIR,
/* rhw */ ILLEGAL_PAIR,
/* rhx */ ILLEGAL_PAIR,
/* rhy */ BEGIN | NOT_END,
/* rhz */ ILLEGAL_PAIR,
/* rhch */ ILLEGAL_PAIR,
/* rhgh */ ILLEGAL_PAIR,
/* rhph */ ILLEGAL_PAIR,
/* rhrh */ ILLEGAL_PAIR,
/* rhsh */ ILLEGAL_PAIR,
/* rhth */ ILLEGAL_PAIR,
/* rhwh */ ILLEGAL_PAIR,
/* rhqu */ ILLEGAL_PAIR,
/* rhck */ ILLEGAL_PAIR,
/* sha */ ANY_COMBINATION,
/* shb */ NOT_BEGIN | BREAK | NOT_END,
/* shc */ NOT_BEGIN | BREAK | NOT_END,
/* shd */ NOT_BEGIN | BREAK | NOT_END,
/* she */ ANY_COMBINATION,
/* shf */ NOT_BEGIN | BREAK | NOT_END,
/* shg */ NOT_BEGIN | BREAK | NOT_END,
/* shh */ ILLEGAL_PAIR,
/* shi */ ANY_COMBINATION,
/* shj */ NOT_BEGIN | BREAK | NOT_END,
/* shk */ NOT_BEGIN,
/* shl */ BEGIN | SUFFIX | NOT_END,
/* shm */ BEGIN | SUFFIX | NOT_END,
/* shn */ BEGIN | SUFFIX | NOT_END,
/* sho */ ANY_COMBINATION,
/* shp */ NOT_BEGIN,
/* shr */ BEGIN | SUFFIX | NOT_END,
/* shs */ NOT_BEGIN | BREAK | NOT_END,
/* sht */ SUFFIX,
/* shu */ ANY_COMBINATION,
/* shv */ NOT_BEGIN | BREAK | NOT_END,
/* shw */ SUFFIX | NOT_END,
/* shx */ ILLEGAL_PAIR,
/* shy */ ANY_COMBINATION,
/* shz */ NOT_BEGIN | BREAK | NOT_END,
/* shch */ NOT_BEGIN | BREAK | NOT_END,
/* shgh */ NOT_BEGIN | BREAK | NOT_END,
/* shph */ NOT_BEGIN | BREAK | NOT_END,
/* shrh */ ILLEGAL_PAIR,
/* shsh */ ILLEGAL_PAIR,
/* shth */ NOT_BEGIN | BREAK | NOT_END,
/* shwh */ ILLEGAL_PAIR,
/* shqu */ NOT_BEGIN | BREAK | NOT_END,
/* shck */ ILLEGAL_PAIR,
/* tha */ ANY_COMBINATION,
/* thb */ NOT_BEGIN | BREAK | NOT_END,
/* thc */ NOT_BEGIN | BREAK | NOT_END,
/* thd */ NOT_BEGIN | BREAK | NOT_END,
/* the */ ANY_COMBINATION,
/* thf */ NOT_BEGIN | BREAK | NOT_END,
/* thg */ NOT_BEGIN | BREAK | NOT_END,
/* thh */ NOT_BEGIN | BREAK | NOT_END,
/* thi */ ANY_COMBINATION,
/* thj */ NOT_BEGIN | BREAK | NOT_END,
/* thk */ NOT_BEGIN | BREAK | NOT_END,
/* thl */ NOT_BEGIN | BREAK | NOT_END,
/* thm */ NOT_BEGIN | BREAK | NOT_END,
/* thn */ NOT_BEGIN | BREAK | NOT_END,
/* tho */ ANY_COMBINATION,
/* thp */ NOT_BEGIN | BREAK | NOT_END,
/* thr */ NOT_END,
/* ths */ NOT_BEGIN | END,
/* tht */ NOT_BEGIN | BREAK | NOT_END,
/* thu */ ANY_COMBINATION,
/* thv */ NOT_BEGIN | BREAK | NOT_END,
/* thw */ SUFFIX | NOT_END,
/* thx */ ILLEGAL_PAIR,
/* thy */ ANY_COMBINATION,
/* thz */ NOT_BEGIN | BREAK | NOT_END,
/* thch */ NOT_BEGIN | BREAK | NOT_END,
/* thgh */ NOT_BEGIN | BREAK | NOT_END,
/* thph */ NOT_BEGIN | BREAK | NOT_END,
/* thrh */ ILLEGAL_PAIR,
/* thsh */ NOT_BEGIN | BREAK | NOT_END,
/* thth */ ILLEGAL_PAIR,
/* thwh */ ILLEGAL_PAIR,
/* thqu */ NOT_BEGIN | BREAK | NOT_END,
/* thck */ ILLEGAL_PAIR,
/* wha */ BEGIN | NOT_END,
/* whb */ ILLEGAL_PAIR,
/* whc */ ILLEGAL_PAIR,
/* whd */ ILLEGAL_PAIR,
/* whe */ BEGIN | NOT_END,
/* whf */ ILLEGAL_PAIR,
/* whg */ ILLEGAL_PAIR,
/* whh */ ILLEGAL_PAIR,
/* whi */ BEGIN | NOT_END,
/* whj */ ILLEGAL_PAIR,
/* whk */ ILLEGAL_PAIR,
/* whl */ ILLEGAL_PAIR,
/* whm */ ILLEGAL_PAIR,
/* whn */ ILLEGAL_PAIR,
/* who */ BEGIN | NOT_END,
/* whp */ ILLEGAL_PAIR,
/* whr */ ILLEGAL_PAIR,
/* whs */ ILLEGAL_PAIR,
/* wht */ ILLEGAL_PAIR,
/* whu */ ILLEGAL_PAIR,
/* whv */ ILLEGAL_PAIR,
/* whw */ ILLEGAL_PAIR,
/* whx */ ILLEGAL_PAIR,
/* why */ BEGIN | NOT_END,
/* whz */ ILLEGAL_PAIR,
/* whch */ ILLEGAL_PAIR,
/* whgh */ ILLEGAL_PAIR,
/* whph */ ILLEGAL_PAIR,
/* whrh */ ILLEGAL_PAIR,
/* whsh */ ILLEGAL_PAIR,
/* whth */ ILLEGAL_PAIR,
/* whwh */ ILLEGAL_PAIR,
/* whqu */ ILLEGAL_PAIR,
/* whck */ ILLEGAL_PAIR,
/* qua */ ANY_COMBINATION,
/* qub */ ILLEGAL_PAIR,
/* quc */ ILLEGAL_PAIR,
/* qud */ ILLEGAL_PAIR,
/* que */ ANY_COMBINATION,
/* quf */ ILLEGAL_PAIR,
/* qug */ ILLEGAL_PAIR,
/* quh */ ILLEGAL_PAIR,
/* qui */ ANY_COMBINATION,
/* quj */ ILLEGAL_PAIR,
/* quk */ ILLEGAL_PAIR,
/* qul */ ILLEGAL_PAIR,
/* qum */ ILLEGAL_PAIR,
/* qun */ ILLEGAL_PAIR,
/* quo */ ANY_COMBINATION,
/* qup */ ILLEGAL_PAIR,
    /* qur */ ILLEGAL_PAIR,
    /* qus */ ILLEGAL_PAIR,
    /* qut */ ILLEGAL_PAIR,
    /* quu */ ILLEGAL_PAIR,
    /* quv */ ILLEGAL_PAIR,
    /* quw */ ILLEGAL_PAIR,
    /* qux */ ILLEGAL_PAIR,
    /* quy */ ILLEGAL_PAIR,
    /* quz */ ILLEGAL_PAIR,
    /* quch */ ILLEGAL_PAIR,
    /* qugh */ ILLEGAL_PAIR,
    /* quph */ ILLEGAL_PAIR,
    /* qurh */ ILLEGAL_PAIR,
    /* qush */ ILLEGAL_PAIR,
    /* quth */ ILLEGAL_PAIR,
    /* quwh */ ILLEGAL_PAIR,
    /* ququ */ ILLEGAL_PAIR,
    /* quck */ ILLEGAL_PAIR,
    /* cka */ NOT_BEGIN | BREAK | NOT_END,
    /* ckb */ NOT_BEGIN | BREAK | NOT_END,
    /* ckc */ NOT_BEGIN | BREAK | NOT_END,
    /* ckd */ NOT_BEGIN | BREAK | NOT_END,
    /* cke */ NOT_BEGIN | BREAK | NOT_END,
    /* ckf */ NOT_BEGIN | BREAK | NOT_END,
    /* ckg */ NOT_BEGIN | BREAK | NOT_END,
    /* ckh */ NOT_BEGIN | BREAK | NOT_END,
    /* cki */ NOT_BEGIN | BREAK | NOT_END,
    /* ckj */ NOT_BEGIN | BREAK | NOT_END,
    /* ckk */ NOT_BEGIN | BREAK | NOT_END,
    /* ckl */ NOT_BEGIN | BREAK | NOT_END,
    /* ckm */ NOT_BEGIN | BREAK | NOT_END,
    /* ckn */ NOT_BEGIN | BREAK | NOT_END,
    /* cko */ NOT_BEGIN | BREAK | NOT_END,
    /* ckp */ NOT_BEGIN | BREAK | NOT_END,
    /* ckr */ NOT_BEGIN | BREAK | NOT_END,
    /* cks */ NOT_BEGIN,
    /* ckt */ NOT_BEGIN | BREAK | NOT_END,
    /* cku */ NOT_BEGIN | BREAK | NOT_END,
    /* ckv */ NOT_BEGIN | BREAK | NOT_END,
    /* ckw */ NOT_BEGIN | BREAK | NOT_END,
    /* ckx */ ILLEGAL_PAIR,
    /* cky */ NOT_BEGIN,
    /* ckz */ NOT_BEGIN | BREAK | NOT_END,
    /* ckch */ NOT_BEGIN | BREAK | NOT_END,
    /* ckgh */ NOT_BEGIN | BREAK | NOT_END,
    /* ckph */ NOT_BEGIN | BREAK | NOT_END,
    /* ckrh */ ILLEGAL_PAIR,
    /* cksh */ NOT_BEGIN | BREAK | NOT_END,
    /* ckth */ NOT_BEGIN | BREAK | NOT_END,
    /* ckwh */ ILLEGAL_PAIR,
    /* ckqu */ NOT_BEGIN | BREAK | NOT_END,
    /* ckck */ ILLEGAL_PAIR
};


#ifdef   RAN_DEBUG
main (argc, argv)
int      argc;
char     *argv[];
{
    register int     argno;
    register long     seed;
    register unsigned short int pwlen;
    register unsigned short int minimum;
    int      number_of_words;
    boolean no_legal_words;
    register char   *unhyphenated_word;
    register char   *hyphenated_word;
    time_t ltime;
```

```
#ifdef B1
    int algorithm = 0;
#endif

    number_of_words = 1;
    no_legal_words = FALSE;
    seed = 0L;
    pwlen = 8;
    minimum = 6;

    for (argno = 0; argno < argc; argno++)
    {
     if (argv[argno][0] == '-')
         switch (argv[argno][1])
         {
#ifdef B1
          case 'a':
              algorithm = atoi (&argv[argno][2]);
              break;
#endif
          case 's':
              seed = atol (&argv[argno][2]);
              if (seed == 0L)
               seed = 1L;
              set_seed(seed);
              break;
          case 'l':
              pwlen = abs (atoi (&argv[argno][2]));
              if (pwlen < 1)
               pwlen = 8;
              break;
          case 'm':
              minimum = abs (atoi (&argv[argno][2]));
              if (minimum < 1)
               minimum = 1;
              break;
          case 'n':
              no_legal_words = TRUE;
              break;
         }
     else
         number_of_words = atoi (argv[argno]);
     if (number_of_words < 1)
         number_of_words = 1;
    }

    /*
     * During debugging (RAN_DEBUG is set), we generate the seed from here
     * rather than the first entry to randomword() .
     */
    if (seed == 0L){
        time(&ltime);
        set_seed((long) ltime);
    }

    if (minimum > pwlen)
    {
     (void) fflush(stdout);
     (void) fprintf (stderr, "minimum (%u) new password length cannot exceed maximum (%u)\n", (uint) minimum, (uint) pwlen);
     (void) fflush(stderr);
     exit (1);
    }
    (void) fflush(stderr);
    (void) fprintf (stdout, "(New password will be between %u and %u characters long)\n", (uint) minimum, (uint) pwlen);
    (void) fflush (stdout);
    for (argno = 1; argno <= number_of_words; argno++)
    {
     unhyphenated_word = calloc (sizeof (char), pwlen + 1);
     hyphenated_word = calloc (sizeof (char), 2 * pwlen);
 #ifdef B1
     switch (algorithm)  {
         default:
            case 0:
             (void) randomword (unhyphenated_word, hyphenated_word, minimum, pwlen, no_legal_words, 0L);
             (void) fflush(stderr);
             (void) fprintf (stdout, "%s (%s)\n", unhyphenated_word, hyphenated_word);
             break;
            case 1:
             (void) randomchars (unhyphenated_word, minimum, pwlen, no_legal_words, 0L);
             (void) fflush(stderr);
             (void) fprintf (stdout, "%s\n", unhyphenated_word);
             break;
            case 2:
             (void) randomletters (unhyphenated_word, minimum, pwlen, no_legal_words, 0L);
             (void) fflush(stderr);
             (void) fprintf (stdout, "%s\n", unhyphenated_word);
             break;
        }
#else
        (void) randomword (unhyphenated_word, hyphenated_word, minimum, pwlen, no_legal_words, 0L);
        (void) fflush(stderr);
        (void) fprintf (stdout, "%s (%s)\n", unhyphenated_word, hyphenated_word);
#endif
        (void) fflush (stdout);
        free (unhyphenated_word);
        free (hyphenated_word);
       }
}
#endif


#ifdef B1
/*
 * Randomchars will generate a random string and place it in the
 * buffer word.  The word must be pre-allocated.  The words generated
 * will have sizes between minlen and maxlen.  If restrict is TRUE,
 * words will not be generated that appear as login names or as entries
 * in the on-line dictionary.  The seed is used on first use of the routine.
 * The length of the word is returned, or -1 if there were an error
 * (length settings are wrong or dictionary checking could not be done).
 * The seed is used on first use of the routine.
 */
int
randomchars(string, minlen, maxlen, restrict, seed)
      register char  *string;
      register unsigned short int minlen;
      register unsigned short int maxlen;
      register boolean restrict;
      long seed;
{
      register int loop_count;
      register unsigned short int string_size;
      register unsigned short int build;
      static  been_here_before = FALSE;

      /*
       * Execute this upon startup.  This initializes the
       * environment, including seed'ing the random number
       * generator and loading the on-line dictionary.
       */
      if (!been_here_before)
      {
            been_here_before = TRUE;

#ifndef RAN_DEBUG
            set_seed(seed);
#endif
      }
      /*
       * Check for minlen > maxlen.  This is an error.
       */
      if (minlen > maxlen)
            return (-1);
        loop_count = 0;
        string_size = get_random(minlen, maxlen);

        do  {
            for (build = 0; build < string_size; build++)  {
                string[build] = (char) get_random((unsigned short int) '!',
                                                  (unsigned short int) '~');
            }




        restrict = 0;

            loop_count ++;
        }
        while (restrict && (loop_count <= MAX_UNACCEPTABLE));

        string[string_size] = '\0';

        return string_size;
}


/*
 * Randomletters will generate a random string of letters and place it in the
 * buffer word.  The word must be pre-allocated.  The words generated
 * will have sizes between minlen and maxlen.  If restrict is TRUE,
 * words will not be generated that appear as login names or as entries
 * in the on-line dictionary.  The seed is used on first use of the routine.
 * The length of the word is returned, or -1 if there were an error
 * (length settings are wrong or dictionary checking could not be done).
 * The seed is used on first use of the routine.
 */
int
randomletters(string, minlen, maxlen, restrict, seed)
        register char  *string;
        register unsigned short int minlen;
        register unsigned short int maxlen;
        register boolean restrict;
        long seed;
{
        register int loop_count;
        register unsigned short int string_size;
        register unsigned short int build;
        static  been_here_before = FALSE;

        /*
         * Execute this upon startup.  This initializes the
         * environment, including seed'ing the random number
         * generator and loading the on-line dictionary.
         */
        if (!been_here_before)
        {
            been_here_before = TRUE;

#ifndef RAN_DEBUG
            set_seed(seed);
#endif
        }
        /*
         * Check for minlen > maxlen.  This is an error.
         */
        if (minlen > maxlen)
            return (-1);


        loop_count = 0;
        string_size = get_random(minlen, maxlen);

        do  {
            for (build = 0; build < string_size; build++)  {
                string[build] = (char) get_random((unsigned short int) 'a', (unsigned short int) 'z');
            }




        restrict = 0;

            loop_count ++;
        }
        while (restrict && (loop_count <= MAX_UNACCEPTABLE));

        string[string_size] = '\0';

        return string_size;
}
#endif


/*
 * Randomword will generate a random word and place it in the
 * buffer word.  Also, the hyphenated word will be placed into
 * the buffer hyphenated_word.  Both word and hyphenated_word must
 * be pre-allocated.  The words generated will have sizes between
 * minlen and maxlen.  If restrict is TRUE, words will not be generated that
 * appear as login names or as entries in the on-line dictionary.
 * This algorithm was initially worded out by Morrie Gasser in 1975.
 * Any changes here are minimal so that as many word combinations
 * can be produced as possible (and thus keep the words random).
 * The seed is used on first use of the routine.
 * The length of the unhyphenated word is returned, or -1 if there
 * were an error (length settings are wrong or dictionary checking
 * could not be done.
 */
int
randomword (word, hyphenated_word, minlen, maxlen, restrict, seed)
register char   *word;
register char   *hyphenated_word;
register unsigned short int minlen;
register unsigned short int maxlen;
register boolean restrict;
long seed;
{
     register int        pwlen;
     register int        loop_count;
     static   been_here_before = FALSE;

  /*
   * Execute this upon startup.  This initializes the
   * environment, including seed'ing the random number
   * generator and loading the on-line dictionary.
   */
     if (!been_here_before)
     {
      been_here_before = TRUE;


#ifndef RAN_DEBUG
      set_seed(seed);
#endif
     }

  /*
   * Check for minlen>maxlen.  This is an error.
   * and a length of 0.
   */
     if (minlen > maxlen)
      return (-1);

  /*
   * Check for zero length words.  This is technically not an error,
 * so we take the short cut and return a null word and a length of 0.
 */
  if (maxlen == 0)
  {
   word[0] = '\0';
   hyphenated_word[0] = '\0';
   return (0);
  }

/*
 * Continue finding words until the criteria are satisfied.
 * The criteria are, if restrict is set, that if the word appears
 * as either a login name or as part of the on-line dictionary,
 * throw out the word and look for another.
 */
  loop_count = 0;

  do
  {
  /*
   * Get a random word.  Its length is a random quantity
   * from with the limits specified in the call to
   * randomword().
   */
   pwlen = get_word (word, hyphenated_word, get_random (minlen, maxlen));




   restrict = 0;

   loop_count++;
  }
  while (restrict && (loop_count <= MAX_UNACCEPTABLE));

  if (restrict)  {
   (void) fflush(stdout);
   (void) fprintf(stderr, "could not find acceptable random password\n");
   (void) fflush(stderr);
   exit(1);
  }

  return (pwlen);
}


/*
 * This is the routine that returns a random word -- as
 * yet unchecked against the passwd file or the dictionary.
 * It collects random syllables until a predetermined
 * word length is found.  If a retry threshold is reached,
 * another word is tried.  Given that the random number
 * generator is uniformly distributed, eventually a word
 * will be found if the retry limit is adequately large enough.
 */
static int
get_word (word, hyphenated_word, pwlen)
char    *word;
char    *hyphenated_word;
unsigned short int   pwlen;
{
     register unsigned short int word_length;
     register unsigned short int syllable_length;
     register char  *new_syllable;
     register unsigned short int *syllable_units;
     register unsigned short int word_size;
     register unsigned short int word_place;
     int unsigned short *word_units;
     int unsigned short  syllable_size;
     int unsigned      tries;
```

```
    /*
     * Keep count of retries.
     */
    tries = 0;

    /*
     * The length of the word in characters.
     */
    word_length = 0;

    /*
     * The length of the word in character units (each of which is one or
     * two characters long.
     */
    word_size = 0;

    /*
     * Initialize the array storing the word units.  Since we know the
     * length of the word, we only need one of that length.  This method is
     * preferable to a static array, since it allows us flexibility in
     * choosing arbitrarily long word lengths.  Since a word can contain one
     * syllable, we should make syllable_units, the array holding the
     * analogous units for an individual syllable, the same length.  No
     * explicit rule limits the length of syllables, but digram rules and
     * heuristics do so indirectly.
     */
    word_units =
     (unsigned short int *)
     calloc (sizeof (unsigned short int), pwlen);
    syllable_units =
     (unsigned short int *)
     calloc (sizeof (unsigned short int), pwlen);
    new_syllable =
     calloc (sizeof (unsigned short int), pwlen);

    /*
     * Find syllables until the entire word is constructed.
     */
    while (word_length < pwlen)
    {
     /*
      * Get the syllable and find its length.
      */
     (void) get_syllable (new_syllable, pwlen - word_length, syllable_units, &syllable_size);
     syllable_length = strlen (new_syllable);

     /*
      * Append the syllable units to the word units.
      */
     for (word_place = 0; word_place <= syllable_size;
word_place++)
         word_units[word_size + word_place] = syllable_units[word_place];
     word_size += syllable_size + 1;

     /*
      * If the word has been improperly formed, throw out
      * the syllable.  The checks performed here are those
      * that must be formed on a word basis.  The other
      * tests are performed entirely within the syllable.
      * Otherwise, append the syllable to the word and
      * append the syllable to the hyphenated version of
      * the word.
      */
     if (improper_word (word_units, word_size) ||
          ((word_length == 0) &&
               have_initial_y (syllable_units, syllable_size)) ||
          ((word_length + syllable_length == pwlen) &&
               have_final_split (syllable_units, syllable_size)))
         word_size -= syllable_size + 1;
     else
     {
         if (word_length == 0)
```

```
          {
           (void) strcpy (word, new_syllable);
           (void) strcpy (hyphenated_word, new_syllable);
          }
          else
          {
           (void) strcat (word, new_syllable);
           (void) strcat (hyphenated_word, "-");
           (void) strcat (hyphenated_word, new_syllable);
          }
          word_length += syllable_length;
      }

        /*
         * Keep track of the times we have tried to get
         * syllables.  If we have exceeded the threshold,
         * reinitialize the pwlen and word_size variables, clear
         * out the word arrays, and start from scratch.
         */
      tries++;
      if (tries > MAX_RETRIES)
      {
          word_length = 0;
          word_size = 0;
          tries = 0;
          (void) strcpy (word, "");
          (void) strcpy (hyphenated_word, "");
      }
     }

     /*
      * The units arrays and syllable storage are internal to this
      * routine.  Since the caller has no need for them, we
      * release the space.
      */
     free ((char *) new_syllable);
     free ((char *) syllable_units);
     free ((char *) word_units);

     return ((int) word_length);
}



/*
 * Check that the word does not contain illegal combinations
 * that may span syllables.  Specifically, these are:
 *    1. An illegal pair of units between syllables.
 *    2. Three consecutive vowel units.
 *    3. Three consecutive consonant units.
 * The checks are made against units (1 or 2 letters), not against
 * the individual letters, so three consecutive units can have
 * the length of 6 at most.
 */
static boolean
improper_word (units, word_size)
register unsigned short int *units;
register unsigned short int word_size;
{
     register unsigned short int unit_count;
     register     boolean failure;

     failure = FALSE;

     for (unit_count = 0; !failure && (unit_count < word_size);
          unit_count++)
     {
      /*
       * Check for ILLEGAL_PAIR.  This should have been caught
       * for units within a syllable, but in some cases it
       * would have gone unnoticed for units between syllables
       * (e.g., when saved_unit's in get_syllable() were not
         * used).
         */
        if ((unit_count != 0) &&
             (digram[units[unit_count - 1]][units[unit_count]] &
                 ILLEGAL_PAIR))
            failure = TRUE;

        /*
         * Check for consecutive vowels or consonants.  Because
         * the initial y of a syllable is treated as a consonant
         * rather than as a vowel, we exclude y from the first
         * vowel in the vowel test.  The only problem comes when
         * y ends a syllable and two other vowels start the next,
         * like fly-oint.  Since such words are still
         * pronounceable, we accept this.
         */
        if (!failure && (unit_count >= 2))
        {
            /*
             * Vowel check.
             */
            if ((((rules[units[unit_count - 2]].flags & VOWEL) &&
                          !(rules[units[unit_count - 2]].flags &
                           ALTERNATE_VOWEL)) &&
                  (rules[units[unit_count - 1]].flags & VOWEL) &&
                  (rules[units[unit_count]].flags & VOWEL)) ||
            /*
             * Consonant check.
             */
                 (!(rules[units[unit_count - 2]].flags & VOWEL) &&
                  !(rules[units[unit_count - 1]].flags & VOWEL) &&
                  !(rules[units[unit_count]].flags & VOWEL)))
             failure = TRUE;
         }
        }

    return (failure);
}


/*
 * Treating y as a vowel is sometimes a problem.  Some words
 * get formed that look irregular.  One special group is when
 * y starts a word and is the only vowel in the first syllable.
 * The word ycl is one example.  We discard words like these.
 */
static boolean
have_initial_y (units, unit_size)
register unsigned short int *units;
register unsigned short int unit_size;
{
    register unsigned short int unit_count;
    register unsigned short int vowel_count;
    register unsigned short int normal_vowel_count;

    vowel_count = 0;
    normal_vowel_count = 0;

    for (unit_count = 0; unit_count <= unit_size; unit_count++)
     /*
      * Count vowels.
      */
     if (rules[units[unit_count]].flags & VOWEL)
     {
         vowel_count++;

         /*
          * Count the vowels that are not: 1. y, 2. at the start of
          * the word.
          */
         if (!(rules[units[unit_count]].flags & ALTERNATE_VOWEL)
||
            (unit_count != 0))
          normal_vowel_count++;
     }

     return ((vowel_count <= 1) && (normal_vowel_count == 0));
}


/*
 * Besides the problem with the letter y, there is one with
 * a silent e at the end of words, like face or nice.  We
 * allow this silent e, but we do not allow it as the only
 * vowel at the end of the word or syllables like ble will
 * be generated.
 */
static boolean
have_final_split (units, unit_size)
register unsigned short int *units;
register unsigned short int unit_size;
{
     register unsigned short int unit_count;
     register unsigned short int vowel_count;

     vowel_count = 0;

     /*
      *    Count all the vowels in the word.
      */
     for (unit_count = 0; unit_count <= unit_size; unit_count++)
      if (rules[units[unit_count]].flags & VOWEL)
          vowel_count++;

     /*
      * Return TRUE iff the only vowel was e, found at the end if the
      * word.
      */
     return ((vowel_count == 1) &&
          (rules[units[unit_size]].flags & NO_FINAL_SPLIT));
}


/*
 * Generate next unit to password, making sure that it follows
 * these rules:
 *    1. Each syllable must contain exactly 1 or 2 consecutive
 *       vowels, where y is considered a vowel.
 *    2. Syllable end is determined as follows:
 *         a. Vowel is generated and previous unit is a
 *            consonant and syllable already has a vowel.  In
 *            this case, new syllable is started and already
 *            contains a vowel.
 *         b. A pair determined to be a "break" pair is encountered.
 *            In this case new syllable is started with second unit
 *            of this pair.
 *         c. End of password is encountered.
 *         d. "begin" pair is encountered legally.  New syllable is
 *            started with this pair.
 *         e. "end" pair is legally encountered.  New syllable has
 *            nothing yet.
 *    3. Try generating another unit if:
 *         a. third consecutive vowel and not y.
 *         b. "break" pair generated but no vowel yet in current
 *            or previous 2 units are "not_end".
 *         c. "begin" pair generated but no vowel in syllable
 *            preceding begin pair, or both previous 2 pairs are
 *           designated "not_end".
 *         d. "end" pair generated but no vowel in current syllable
 *            or in "end" pair.
 *         e. "not_begin" pair generated but new syllable must
 *            begin (because previous syllable ended as defined in
 *            2 above).
 *         f. vowel is generated and 2a is satisfied, but no syllable
 *             break is possible in previous 3 pairs.
 *          g. Second and third units of syllable must begin, and
 *             first unit is "alternate_vowel".
 */
static char *
get_syllable (syllable, pwlen, units_in_syllable, syllable_length)
char    *syllable;
unsigned short int  pwlen;
unsigned short int *units_in_syllable;
unsigned short int *syllable_length;
{
    register unsigned short int unit;
    register short int  current_unit;
    register unsigned short int vowel_count;
    register     boolean rule_broken;
    register     boolean want_vowel;
    register     boolean want_another_unit;
    int unsigned      tries;
    int unsigned short  last_unit;
    int short    length_left;
    unsigned short int  hold_saved_unit;
    static unsigned short int    saved_unit;
    static unsigned short int    saved_pair[2];

    /*
     * This is needed if the saved_unit is tries and the syllable then
     * discarded because of the retry limit. Since the saved_unit is OK and
     * fits in nicely with the preceding syllable, we will always use it.
     */
    hold_saved_unit = saved_unit;

    /*
     * Loop until valid syllable is found.
     */
    do
    {
     /*
      * Try for a new syllable.  Initialize all pertinent
      * syllable variables.
      */
     tries = 0;
     saved_unit = hold_saved_unit;
     (void) strcpy (syllable, "");
     vowel_count = 0;
     current_unit = 0;
     length_left = (short int) pwlen;
     want_another_unit = TRUE;

     /*
      * This loop finds all the units for the syllable.
      */
     do
     {
         want_vowel = FALSE;

         /*
          * This loop continues until a valid unit is found for the
          * current position within the syllable.
          */
         do
         {
          /*
           * If there are saved_unit's from the previous
           * syllable, use them up first.
           */
          if (saved_unit != 0)
          {
              /*
               * If there were two saved units, the first is
               * guaranteed (by checks performed in the previous
               * syllable) to be valid.  We ignore the checks
               * and place it in this syllable manually.
```

```
             */
            if (saved_unit == 2)
            {
             units_in_syllable[0] = saved_pair[1];
             if (rules[saved_pair[1]].flags & VOWEL)
                 vowel_count++;
             current_unit++;
             (void) strcpy (syllable, rules[saved_pair[1]].unit_code);
             length_left -= strlen (syllable);
            }

            /*
             * The unit becomes the last unit checked in the
             * previous syllable.
             */
            unit = saved_pair[0];

            /*
             * The saved units have been used.  Do not try to
             * reuse them in this syllable (unless this particular
             * syllable is rejected at which point we start to rebuild
             * it with these same saved units.
             */
            saved_unit = 0;
        }
        else
            /*
             * If we don't have to scoff the saved units,
             * we generate a random one.  If we know it has
             * to be a vowel, we get one rather than looping
             * through until one shows up.
             */
            if (want_vowel)
             unit = random_unit (VOWEL);
            else
             unit = random_unit (NO_SPECIAL_RULE);

        length_left -= (short int) strlen
(rules[unit].unit_code);

        /*
         * Prevent having a word longer than expected.
         */
        if (length_left < 0)
            rule_broken = TRUE;
        else
            rule_broken = FALSE;

        /*
         * First unit of syllable.  This is special because the
         * digram tests require 2 units and we don't have that yet.
         * Nevertheless, we can perform some checks.
         */
        if (current_unit == 0)
        {
            /*
             * If the shouldn't begin a syllable, don't
             * use it.
             */
            if (rules[unit].flags & NOT_BEGIN_SYLLABLE)
             rule_broken = TRUE;
            else
             /*
              * If this is the last unit of a word,
              * we have a one unit syllable.  Since each
              * syllable must have a vowel, we make sure
              * the unit is a vowel.  Otherwise, we
              * discard it.
              */
             if (length_left == 0)
                 if (rules[unit].flags & VOWEL)
                  want_another_unit = FALSE;
            else
             rule_broken = TRUE;
}
else
{
    /*
     * There are some digram tests that are
     * universally true.  We test them out.
     */

    /*
     * Reject ILLEGAL_PAIRS of units.
     */
    if ((ALLOWED (ILLEGAL_PAIR)) ||

    /*
     * Reject units that will be split between syllables
     * when the syllable has no vowels in it.
     */
         (ALLOWED (BREAK) && (vowel_count == 0)) ||

    /*
     * Reject a unit that will end a syllable when no
     * previous unit was a vowel and neither is this one.
     */
         (ALLOWED (END) && (vowel_count == 0) &&
          !(rules[unit].flags & VOWEL)))
     rule_broken = TRUE;

    if (current_unit == 1)
    {
     /*
      * Reject the unit if we are at te starting digram of
      * a syllable and it does not fit.
      */
     if (ALLOWED (NOT_BEGIN))
         rule_broken = TRUE;
    }
    else
    {
     /*
      * We are not at the start of a syllable.
      * Save the previous unit for later tests.
      */
     last_unit = units_in_syllable[current_unit - 1];

     /*
      * Do not allow syllables where the first letter is y
      * and the next pair can begin a syllable.  This may
      * lead to splits where y is left alone in a syllable.
      * Also, the combination does not sound to good even
      * if not split.
      */
     if (((current_unit == 2) &&
              (ALLOWED (BEGIN)) &&
              (rules[units_in_syllable[0]].flags &
               ALTERNATE_VOWEL)) ||

          /*
           * If this is the last unit of a word, we should
           * reject any digram that cannot end a syllable.
           */
          (ALLOWED (NOT_END) &&
              (length_left == 0)) ||

          /*
           * Reject the unit if the digram it forms wants
           * to break the syllable, but the resulting
           * digram that would end the syllable is not
           * allowed to end a syllable.
           */
          (ALLOWED (BREAK) &&
            (digram[units_in_syllable
                 [current_unit - 2]]
             [last_unit] &
             NOT_END)) ||

        /*
         * Reject the unit if the digram it forms
         * expects a vowel preceding it and there is
         * none.
         */
        (ALLOWED (PREFIX) &&
            !(rules[units_in_syllable
                 [current_unit - 2]].flags &
             VOWEL)))
       rule_broken = TRUE;

    /*
     * The following checks occur when the current unit
     * is a vowel and we are not looking at a word ending
     * with an e.
     */
    if (!rule_broken &&
        (rules[unit].flags & VOWEL) &&
        ((length_left > 0) ||
            !(rules[last_unit].flags &
             NO_FINAL_SPLIT)))

        /*
         * Don't allow 3 consecutive vowels in a
         * syllable.  Although some words formed like this
         * are OK, like beau, most are not.
         */
        if ((vowel_count > 1) &&
            (rules[last_unit].flags & VOWEL))
         rule_broken = TRUE;
        else
         /*
          * Check for the case of
          * vowels-consonants-vowel, which is only
          * legal if the last vowel is an e and we are
          * the end of the word (wich is not
          * happening here due to a previous check.
          */
         if ((vowel_count != 0) &&
              !(rules[last_unit].flags & VOWEL))
         {
             /*
              * Try to save the vowel for the next
              * syllable, but if the syllable left here
              * is not proper (i.e., the resulting last
              * digram cannot legally end it), just
              * discard it and try for another.
              */
             if (digram[units_in_syllable
                    [current_unit - 2]]
                   [last_unit] &
                   NOT_END)
              rule_broken = TRUE;
             else
             {
              saved_unit = 1;
              saved_pair[0] = unit;
              want_another_unit = FALSE;
             }
         }
    }

    /*
     * The unit picked and the digram formed are legal.
     * We now determine if we can end the syllable.  It may,
     * in some cases, mean the last unit(s) may be deferred to
     * the next syllable.  We also check here to see if the
```

```
       * digram formed expects a vowel to follow.
       */
      if (!rule_broken && want_another_unit)
      {
       /*
        * This word ends in a silent e.
        */
       if (((vowel_count != 0) &&
                 (rules[unit].flags & NO_FINAL_SPLIT) &&
                 (length_left == 0) &&
                 !(rules[last_unit].flags &
                  VOWEL)) ||

            /*
             * This syllable ends either because the digram
             * is an END pair or we would otherwise exceed
             * the length of the word.
             */
            (ALLOWED (END)) || (length_left == 0)))
           want_another_unit = FALSE;
       else
           /*
            * Since we have a vowel in the syllable
            * already, if the digram calls for the end of the
            * syllable, we can legally split it off. We also
            * make sure that we are not at the end of the
            * dangerous because that syllable may not have
            * vowels, or it may not be a legal syllable end,
            * and the retrying mechanism will loop infinitely
            * with the same digram.
            */
           if ((vowel_count != 0) && (length_left > 0))
           {
            /*
             * If we must begin a syllable, we do so if
             * the only vowel in THIS syllable is not part
             * of the digram we are pushing to the next
             * syllable.
             */
            if (ALLOWED (BEGIN) &&
                 (current_unit > 1) &&
                 !((vowel_count == 1) &&
                      (rules[last_unit].flags &
                       VOWEL)))
            {
                saved_unit = 2;
                saved_pair[0] = unit;
                saved_pair[1] = last_unit;
                want_another_unit = FALSE;
            }
            else
                if (ALLOWED (BREAK))
                {
                 saved_unit = 1;
                 saved_pair[0] = unit;
                 want_another_unit = FALSE;
                }
           }
           else
            if (ALLOWED (SUFFIX))
                want_vowel = TRUE;
      }
}

tries++;

/*
 * If this unit was illegal, redetermine the amount of
 * letters left to go in the word.
 */
if (rule_broken)
    length_left += (short int) strlen (rules[unit].unit_code);
```

```
        }
        while (rule_broken && (tries <= MAX_RETRIES));

        /*
         * The unit fit OK.
         */
        if (tries <= MAX_RETRIES)
        {
         /*
          * If the unit were a vowel, count it in.
          * However, if the unit were a y and appear
          * at the start of the syllable, treat it
          * like a constant (so that words like year can
          * appear and not conflict with the 3 consecutive
          * vowel rule.
          */
         if ((rules[unit].flags & VOWEL) &&
             ((current_unit > 0) ||
                 !(rules[unit].flags & ALTERNATE_VOWEL)))
            vowel_count++;

         /*
          * If a unit or units were to be saved, we must
          * adjust the syllable formed.  Otherwise, we
          * append the current unit to the syllable.
          */
         switch (saved_unit)
         {
             case 0:
              units_in_syllable[current_unit] = unit;
              (void) strcat (syllable, rules[unit].unit_code);
              break;
             case 1:
              current_unit--;
              break;
             case 2:
              (void) strcpy (&syllable[strlen (syllable) -
                      strlen (rules[last_unit].
                        unit_code)],
                   "");
              length_left += (short int) strlen (rules[last_unit].unit_code);
              current_unit -= 2;
              break;
          }
         }
         else
         /*
          * Whoops!  Too many tries.  We set rule_broken so we can
          * loop in the outer loop and try another syllable.
          */
          rule_broken = TRUE;

         /*
          * ...and the syllable length grows.
          */
         *syllable_length = current_unit;

         current_unit++;
      }
      while ((tries <= MAX_RETRIES) && want_another_unit);
     }
     while (rule_broken ||
           illegal_placement (units_in_syllable, *syllable_length));

     return (syllable);
}


/*
 * This routine goes through an individual syllable and checks
 * for illegal combinations of letters that go beyond looking
 * at digrams.  We look at things like 3 consecutive vowels or
 * consonants, or syllables with consonants between vowels (unless
 * one of them is the final silent e).
 */
static boolean
illegal_placement (units, pwlen)
register unsigned short int *units;
register unsigned short int pwlen;
{
    register unsigned short int vowel_count;
    register unsigned short int unit_count;
    register     boolean failure;

    vowel_count = 0;
    failure = FALSE;

    for (unit_count = 0; !failure && (unit_count <= pwlen);
         unit_count++)
    {
     if (unit_count >= 1)
     {
         /*
          * Don't allow vowels to be split with consonants in
          * a single syllable.  If we find such a combination
          * (except for the silent e) we have to discard the
          * syllable).
          */
         if ((!(rules[units[unit_count - 1]].flags & VOWEL) &&
               (rules[units[unit_count]].flags & VOWEL) &&
               !((rules[units[unit_count]].flags &
                    NO_FINAL_SPLIT) &&
                   (unit_count == pwlen)) &&
               (vowel_count != 0)) ||

         /*
          * Perform these checks when we have at least 3 units.
          */
              ((unit_count >= 2) &&

                  /*
                   * Disallow 3 consecutive consonants.
                   */
               ((!(rules[units[unit_count - 2]].flags & VOWEL) &&
                    !(rules[units[unit_count - 1]].flags &
                        VOWEL) &&
                    !(rules[units[unit_count]].flags &
                        VOWEL)) ||

                   /*
                    * Disallow 3 consecutive vowels, where the first is
                    * not a y.
                    */
                   (((rules[units[unit_count - 2]].flags &
                        VOWEL) &&
                        !((rules[units[0]].flags &
                            ALTERNATE_VOWEL) &&
                         (unit_count == 2))) &&
                    (rules[units[unit_count - 1]].flags &
                        VOWEL) &&
                    (rules[units[unit_count]].flags &
                        VOWEL)))))
          failure = TRUE;
     }

     /*
      * Count the vowels in the syllable.  As mentioned somewhere
      * above, exclude the initial y of a syllable.  Instead,
      * treat it as a consonant.
      */
     if ((rules[units[unit_count]].flags & VOWEL) &&
          !((rules[units[0]].flags & ALTERNATE_VOWEL) &&
              (unit_count == 0) && (pwlen != 0)))
         vowel_count++;
    }

    return (failure);
}


/*
 * This is the standard random unit generating routine for
 * get_syllable().  It does not reference the digrams, but
 * assumes that it contains 34 units in a particular order.
 * This routine attempts to return unit indexes with a distribution
 * approaching that of the distribution of the 34 units in
 * English.  In order to do this, a random number (supposedly
 * uniformly distributed) is used to do a table lookup into an
 * array containing unit indices.  There are 211 entries in
 * the array for the random_unit entry point.  The probability
 * of a particular unit being generated is equal to the
 * fraction of those 211 entries that contain that unit index.
 * For example, the letter 'a' is unit number 1.  Since unit
 * index 1 appears 10 times in the array, the probability of
 * selecting an 'a' is 10/211.
 *
 * Changes may be made to the digram table without affect to this
 * procedure providing the letter-to-number correspondence of
 * the units does not change.  Likewise, the distribution of the
 * 34 units may be altered (and the array size may be changed)
 * in this procedure without affecting the digram table or any other
 * programs using the random_word subroutine.
 */
static unsigned short int  numbers[] =
{
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    1, 1, 1, 1, 1, 1, 1, 1,
    2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
    3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
    5, 5, 5, 5, 5, 5, 5, 5,
    6, 6, 6, 6, 6, 6, 6, 6,
    7, 7, 7, 7, 7, 7,
    8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
    9, 9, 9, 9, 9, 9, 9, 9,
    10, 10, 10, 10, 10, 10, 10, 10,
    11, 11, 11, 11, 11, 11,
    12, 12, 12, 12, 12, 12,
    13, 13, 13, 13, 13, 13, 13, 13, 13, 13,
    14, 14, 14, 14, 14, 14, 14, 14, 14, 14,
    15, 15, 15, 15, 15, 15,
    16, 16, 16, 16, 16, 16, 16, 16, 16, 16,
    17, 17, 17, 17, 17, 17, 17, 17,
    18, 18, 18, 18, 18, 18, 18, 18, 18, 18,
    19, 19, 19, 19, 19, 19,
    20, 20, 20, 20, 20, 20, 20, 20,
    21, 21, 21, 21, 21, 21, 21, 21,
    22,
    23, 23, 23, 23, 23, 23, 23, 23,
    24,
    25,
    26,
    27,
    28,
    29, 29,
    30,
    31,
    32,
    33
  };


  /*
   * This structure has a typical English frequency of vowels.
   * The value of an entry is the vowel position (a=0, e=4, i=8,
 * o=14, u=19, y=23) in the rules array.  The number of times
 * the value appears is the frequency.  Thus, the letter "a"
 * is assumed to appear 2/12 = 1/6 of the time.  This array
 * may be altered if better data is obtained.  The routines that
 * use vowel_numbers will adjust to the size difference automatically.
 */
static unsigned short int  vowel_numbers[] =
{
    0, 0, 4, 4, 4, 8, 8, 14, 14, 19, 19, 23
};


/*
 * Select a unit (a letter or a consonant group).  If a vowel is
 * expected, use the vowel_numbers array rather than looping through
 * the numbers array until a vowel is found.
 */
static unsigned short int
random_unit (type)
register unsigned short int type;
{
    register unsigned short int number;

    /*
     * Sometimes, we are asked to explicitly get a vowel (i.e., if
     * a digram pair expects one following it).  This is a shortcut
     * to do that and avoid looping with rejected consonants.
     */
    if (type & VOWEL)
     number = vowel_numbers[get_random (0, sizeof (vowel_numbers) / sizeof (unsigned short int))];
    else
     /*
      * Get any letter according to the English distribution.
      */
     number = numbers[get_random (0, sizeof (numbers) / sizeof (unsigned short int))];
    return (number);
}


/*
 * This routine should return a uniformly distributed random number between
 * minlen and maxlen inclusive.  The Electronic Code Book form of DES is
 * used to produce the random number.  The inputs to DES are the old pass-
 * word and a pseudorandom key generated according to the procedure out-
 * lined in Appendix C of ANSI X9.17.
*/

static unsigned short int
get_random (minlen, maxlen)
register unsigned short int  minlen;
register unsigned short int  maxlen;
{
    return minlen + (unsigned short int) randint ((int) (maxlen - minlen + 1));
}


/*
 * Produces a random number from 0 to n-1 .
 */
static unsigned int
randint(n)
     int n;
{
     return ((unsigned int) (randfunc(n)));
}


/ * Set the seed.  This routine will only set the seed once, even if
 * called from multiple sources.
 */
static void
set_seed(seed)
```

```
     long seed;
{
     int been_here_before = 0;

     if (!been_here_before)  {
          been_here_before = 1;
          srand(seed);
     }
}

/* takes in old password and calls program to generate random number by
   calling the DES function.  This function is called many times.  It
   asks for the old password the first time and then sends that password
   to the DES function on every successive call afterwards.  */

static int randfunc(n)
int n;
{
    int i, len;
    static char passwd[9];
    static boolean newpass=0;
    if(!newpass)
    {
         printf("Please enter old password or input string: ");
         gets(passwd);
            printf("string entered: %s\n", passwd);
            len = strlen(passwd);
            for (i=len; i<8; i++)
                 passwd[i] = '0';
            printf("string padded: %s\n", passwd);
         newpass=1;
    }
    return(descall(passwd, n));
}


/* descall calls the pseudorandom key generator, random, described in Appendix C of
ANSI 9.17 and puts the resulting value into the array key.  The arguments of random
indicate that odd parity should be generated and that the input string is eight bytes
in length.  The des routine, which uses key and the old password as arguments, is then
called.  The output from des, out, is then sent to the routine, answer, for processing */


descall(in, n)
unsigned char *in;
int n;
{
unsigned char key[8];
unsigned char out[8];
int i;

random(key, 1,);
setkey(0, 0, key);
des(in, out);
return (answer(out,n));
}

/* answer takes the array out and creates variable sum by adding
   certain values within the array together.  To get a number from 0 to
   n-1, it returns sum mod n (sum%n) */

int answer(out,n)
unsigned char *out;
int n;
{
unsigned int sum;
/* every time this function is called, it adds the first three positions of
   out to get sum.*/

    sum = out[0] + out[1] +out[2];
    return (sum%n);
}
```

```
/*******************************************************************************
 * DES.C                                              VERSION 4.00         *
 *-------------------------------------------------------------------------*
 * D A T A   E N C R Y P T I O N   S T A N D A R D                         *
 * FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION (FIPS PUB) 46-1
 *-------------------------------------------------------------------------*
 * This software was produced at the National Institute of Standards and Techology *
 * (NIST) as a part of research efforts and for demonstration purposes only.  Our  *
 * primary goals in its design did not include widespread use outside of          *
 * our own laboratories.  Acceptance of this software implies that you             *
 * agree to accept it as nonproprietary and unlicensed, not supported by           *
 * NIST, and not carrying any warranty, either expressed or implied, as to         *
 * its performance or fitness for any particular purpose.                          *
 *-------------------------------------------------------------------------*
 * Cryptographic devices and technical data regarding them are subject to          *
 * Federal Government export controls as specified in Title 22, Code of            *
 * Federal Regulations, Parts 121 through 128.  Cryptographic devices              *
 * implementing the Data Encryption Standard (DES) and technical data              *
 * regarding them must comply with these Federal regulations.                      *
 *-------------------------------------------------------------------------*
 *******************************************************************************/

#define BYTE unsigned char
#define INT unsigned int

/*******************************************************************************
 * SETKEY()  Generate key schedule for given key and type of cryption         *
 *******************************************************************************/

/* PERMUTED CHOICE 1 (PC1) */
INT PC1[] = {
        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,
};

/* Schedule of left shifts for C and D blocks */
unsigned short shifts[] = { 1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1 };

/* PERMUTED CHOICE 2 (PC2) */
INT PC2[] = {
        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,
};

/* Key schedule of 16 48-bit subkeys generated from 64-bit key */
BYTE KS[16][48];

setkey(sw1,sw2,pkey)
INT sw1;                /* parity: 0=ignore,1=check */
INT sw2;                /* type cryption: 0=encrypt,1=decrypt */
BYTE *pkey;             /* 64-bit key packed into 8 bytes */
{
        register INT i, j, k, t1, t2;
        static BYTE key[64];
        static BYTE CD[56];

        /* Double-check 'parity' parameter */
        if (sw1 != 0 && sw1 != 1) {
                printf("\007*** setkey: bad parity parameter (%d) ***\n", sw1);
                return(0);
        }

        /* Double-check 'type of cryption' parameter */
        if (sw2 != 0 && sw2 != 1) {
                printf("\007*** setkey: bad cryption parameter (%d) ***\n",sw2);
                return(0);
        }

        /* Unpack KEY from 8 bits/byte into 1 bit/byte */
        unpack8(pkey,key);

          /* Check for ODD key parity */
        if (sw1 == 1) {
                        for (i=0; i<64; i++) {
                                k = 1;
                                for (j=0; j<7; j++,i++) k = (k + key[i]) % 2;
                                if (key[i] != k) return(0);
                        }
          }

          /* Permute unpacked key with PC1 to generate C and D */
        for (i=0; i<56; i++) CD[i] = key[PC1[i]-1];

        /* Rotate and permute C and D to generate 16 subkeys */
        for (i=0; i<16; i++) {
                /* Rotate C and D */
                        for (j=0; j<shifts[i]; j++) {
                          t1 = CD[0];
                          t2 = CD[28];
                          for (k=0; k<27; k++) {
                                                CD[k] = CD[k+1];
                                  CD[k+28] = CD[k+29];
                                   }
                          CD[27] = t1;
                          CD[55] = t2;
                }
                /* Set order of subkeys for type of cryption */
                j = sw2 ? 15-i : i;
                /* Permute C and D with PC2 to generate KS[j] */
                for (k=0; k<48; k++) KS[j][k] = CD[PC2[k]-1];
        }
        return(1);
}


/********************************************************************************
 *  DES()                                                                       *
 ********************************************************************************/


/* INITIAL PERMUTATION (IP) */
INT 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,
};

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

/* E BIT-SELECTION TABLE */
INT E[] = {
        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,
};

/* PERMUTATION FUNCTION P */
INT 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 S-BOXES */
INT S[8][64] = {
        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,

        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,

        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,

         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,

         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,

        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,

         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,

        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,
};
```

```
des(in,out)
BYTE *in;                    /* packed 64-bit INPUT block */
BYTE *out;                   /* packed 64-bit OUTPUT block */
{
        register INT i, j, k, t;
        static BYTE block[64]; /* unpacked 64-bit input/output block */
        static BYTE LR[64], f[32], preS[48];

          /* Unpack the INPUT block */
          unpack8(in,block);

        /* Permute unpacked input block with IP to generate L and R */
        for (j=0; j<64; j++) LR[j] = block[IP[j]-1];

        /* Perform 16 rounds */
        for (i=0; i<16; i++) {
                /* Expand R to 48 bits with E and XOR with i-th subkey */
                for (j=0; j<48; j++) preS[j] = LR[E[j]+31] ^ KS[i][j];
                /* Map 8 6-bit blocks into 8 4-bit blocks using S-Boxes */
                for (j=0; j<8; j++) {
                                /* Compute index t into j-th S-box */
                        k = 6*j;
                        t = preS[k];
                        t = (t<<1) | preS[k+5];
                        t = (t<<1) | preS[k+1];
                        t = (t<<1) | preS[k+2];
                        t = (t<<1) | preS[k+3];
                        t = (t<<1) | preS[k+4];
                                /* Fetch t-th entry from j-th S-box */
                        t = S[j][t];
                                /* Generate 4-bit block from S-box entry */
                        k = 4*j;
                        f[k] = (t>>3) & 1;
                        f[k+1] = (t>>2) & 1;
                        f[k+2] = (t>>1) & 1;
                        f[k+3] = t & 1;
                }
                        for (j=0; j<32; j++) {
                        /* Copy R */
                        t = LR[j+32];
                        /* Permute f w/ P and XOR w/ L to generate new R */
                        LR[j+32] = LR[j] ^ f[P[j]-1];
                        /* Copy original R to new L */
                                LR[j] = t;
                        }
        }

        /* Permute L and R with reverse IP-1 to generate output block */
        for (j=0; j<64; j++) block[j] = LR[RFP[j]-1];

        /* Pack data into 8 bits per byte */
          pack8(out,block);
}


/******************************************************************************
 *  PACK8()  Pack 64 bytes at 1 bit/byte into 8 bytes at 8 bits/byte         *
 ******************************************************************************/

pack8(packed,binary)
BYTE *packed;                /* packed block ( 8 bytes at 8 bits/byte) */
BYTE *binary;                /* the unpacked block (64 bytes at 1 bit/byte) */
{
        register INT i, j, k;

        for (i=0; i<8; i++) {
                k = 0;
                for (j=0; j<8; j++) k = (k<<1) + *binary++;
                *packed++ = k;
        }
}
```

```
/******************************************************************************
 *  UNPACK8()  Unpack 8 bytes at 8 bits/byte into 64 bytes at 1 bit/byte       *
 ******************************************************************************/

unpack8(packed,binary)
BYTE *packed;                  /* packed block (8 bytes at 8 bits/byte) */
BYTE *binary;                  /* unpacked block (64 bytes at 1 bit/byte) */
{
        register INT i, j, k;

        for (i=0; i<8; i++) {
                k = *packed++;
                for (j=0; j<8; j++) *binary++ = (k>>(7-j)) & 01;
        }
}

#include <dos.h>

#define BYTE unsigned char
#define INT unsigned int

#define DECRYPT  1
#define ENCRYPT  0

#define FALSE  0
#define TRUE  1

#define SINGLE  1
#define PAIR  2

#define IGNORE  0
#define PKEYLEN  8

int krypt(int, int, int, BYTE *, int, BYTE *, BYTE *);
void random(BYTE *, int);
void set_parity(BYTE *, int);
void daytime(char *);
BYTE *bytncpy(BYTE *, BYTE *, int);
BYTE *bytnxor(BYTE *, BYTE *, BYTE *, int);

/******************************************************************************
 *  KRYPT()     Encrypt/decrypt key or key pair                                *
 ******************************************************************************/

int krypt(sw1,sw2,sw3,kek,sw4,ikey,okey)
int sw1;               /* ODD parity: 0=ignore, 1=check & report */
int sw2;               /* type of cryption: 0=encrypt, 1=decrypt */
int sw3;               /* length of kek: 1=single, 2=pair */
BYTE *kek;                 /* packed key-encrypting key */
int sw4;               /* length of key: 1=single, 2=pair */
BYTE *ikey;              /* packed input key */
BYTE *okey;                    /* packed output key */
{
        char tkey[PKEYLEN];

        /* DOUBLE-CHECK PARAMETERS */
        if (sw3!=SINGLE && sw3!=PAIR) {
            printf("krypt: bad kek length (%d)",sw3);
            exit(1);
        }
        if (sw4!=SINGLE && sw4!=PAIR) {
            printf("krypt: bad key length (%d)",sw4);
            exit(1);
        }
        if (sw3==SINGLE && sw4==PAIR)
            exit(1);

        if (!setkey(sw1,sw2,kek)) return(FALSE);
        des(ikey,okey);
        if (sw3==SINGLE && sw4==SINGLE) return(TRUE);  /* single by single */
        if (!setkey(sw1,sw2^01,kek+PKEYLEN)) return(FALSE);
        des(okey,tkey);
        (void) setkey(sw1,sw2,kek);
        des(tkey,okey);
        if (sw4==SINGLE) return(TRUE);  /* single by double */

        (void) krypt(sw1,sw2,PAIR,kek,SINGLE,ikey+PKEYLEN,okey+PKEYLEN);
        return(TRUE);  /* double by double */
}


/*****************************************************************************
 *  RANDOM()    Pseudorandom KEY and IV Generator.                          *
 *****************************************************************************/

/* Random Key */
static BYTE rndkey[PAIR*PKEYLEN] = {0xE0,0x9A,0xA8,0x0F,0xAB,0x72,0x1C,0x3D,
                                    0x8F,0x7D,0xC9,0x9E,0x8F,0x02,0xB6,0x2A};

/* Seed */
static BYTE seed[PKEYLEN] = {0xCF,0x65,0xAE,0x7F,0xB1,0x79,0xBB,0xE3};

void random(dest,odd)
BYTE *dest;   /* destination for random KEY or IV */
int odd;      /* generate ODD parity? 0=no, 1=yes */
{
        BYTE dt[PKEYLEN];           /* date/time vector */
        BYTE i[PKEYLEN];
        BYTE j[PKEYLEN];
        BYTE r[PKEYLEN];

        /* DOUBLE-CHECK PARAMETERS */
        if (odd!=FALSE && odd!=TRUE) {
            printf("random: bad generate parity option (%d)", odd);
            exit(1);
        }

        /* GET DATE/TIME VECTOR */
        daytime(dt);

        /* I = eRNDKEY(DT) */
        (void) krypt(IGNORE,ENCRYPT,PAIR,rndkey,SINGLE,dt,i);

        /* R = eRNDKEY(I + V) */
        (void) bytnxor(j,i,seed,PKEYLEN);
        (void) krypt(IGNORE,ENCRYPT,PAIR,rndkey,SINGLE,j,r);

        /* new seed = eRNDKEY(R + I) */
        (void) bytnxor(j,i,r,PKEYLEN);
        (void) krypt(IGNORE,ENCRYPT,PAIR,rndkey,SINGLE,j,seed);

        /* GENERATE ODD PARITY, IF NEEDED */
        if (odd) set_parity(r,SINGLE);

        (void) bytncpy(dest,r,PKEYLEN);
}

/*****************************************************************************
 *  SET_PARITY()   Set ODD parity                                          *
 *****************************************************************************/

void set_parity(key,len)
BYTE *key;    /* packed 64 or 128-bit key */
int len;      /* key length: 1=SINGLE, 2=PAIR */
{
        int i, j, parity, mask;

        /* DOUBLE-CHECK PARAMETER */
        if (len!=SINGLE && len!=PAIR) {
            printf("set_parity: bad len parameter (%d)", len);
            exit(1);
        }
        for (i=0; i<(len*PKEYLEN); i++) {
            parity = 1;
            mask = 2;
            for (j=0; j<7; j++) {
                parity = (parity + ((key[i] & mask) >> (j+1))) % 2;
                mask = 2 * mask;
            }
            if (parity==0) key[i] = key[i] & 0xfe;/* clear */
            if (parity==1) key[i] = key[i] | 0x01;/* set */
        }
}


void daytime(dt)
char *dt;               /* dt[8] = 64-bit block based on date & time */
{
            register i;
            int tt[8];
 /*   struct regval { int ax, bx, cx, dx, si, di, ds, es; };
    struct regval call_regs, ret_regs; */
    union REGS call_regs;
    union REGS ret_regs;

    call_regs.x.ax = 0x2a00;            /* GET DATE */
    intdos(&call_regs, &ret_regs);
    tt[0] = ret_regs.x.cx - 1900;       /* cx = year */
    tt[1] = (ret_regs.x.dx & 0xff00) >> 8;      /* dh = month */
    tt[2] = ret_regs.x.dx & 0x00ff;         /* dl = day */

    call_regs.x.ax = 0x2c00;            /* GET TIME */
    intdos(&call_regs, &ret_regs);
    tt[3] = (ret_regs.x.cx & 0xff00) >> 8;      /* ch = hours */
    tt[4] = ret_regs.x.cx & 0x00ff;         /* cl = minutes */
    tt[5] = (ret_regs.x.dx & 0xff00) >> 8;      /* dh = seconds */
            tt[6] = 0;
            tt[7] = 0;

            for (i=0;i<8;i++)
                        dt[i] = (char) tt[i];
}

/******************************************************************************
 *  BYTNCPY()   Copy block of packed BYTEs                                   *
 ******************************************************************************/

BYTE *bytncpy(dest,src,len)     /* return pointer to destination block */
BYTE *dest;             /* destination block */
BYTE *src;              /* source block */
int len;    /* number of bytes */
{
            while (len-- > 0) *dest++ = *src++;
            return(dest);
}

/******************************************************************************
 *  BYTNXOR()   XOR blocks of packed BYTEs                                   *
 ******************************************************************************/

BYTE *bytnxor(dest,src1,src2,len)           /* return ptr to destination block */
BYTE *dest;             /* destination block */
BYTE *src1;             /* source block 1 */
BYTE *src2;             /* source block 2 */
int len;    /* number of BYTEs */
{
            while (len-- > 0) *dest++ = *src1++ ^ *src2++;
            return(dest);
}
```