ESD-TR-75-97

MTR-3006

A RANDOM WORD GENERATOR FOR PRONOUNCEABLE PASSWORDS

NOVEMBER 1975

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Bedford, Massachusetts

Project No. 522N

Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts

Contract No. F19628-75-C-0001

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

F. WAH LEONG, Captain, USAF
Project Officer
Air Force Data Services Center

ROGER R. SCHELL, Major, USAF
Project Engineer

FOR THE COMMANDER

FRANK J. EMMA, Colonel, USAF
Director, Information Systems
Technology Applications Office
Deputy for Command and Management Systems

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-75-97	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A RANDOM WORD GENERATOR FOR PRONOUNCEABLE PASSWORDS		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER MTR-3006
7. AUTHOR(s) M. Gasser		8. CONTRACT OR GRANT NUMBER(s) F19628-75-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS The MITRE Corporation Box 208 Bedford, MA 01730		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Project No. 522N
11. CONTROLLING OFFICE NAME AND ADDRESS Deputy for Command and Management Systems Electronic Systems Division, AFSC Hanscom Air Force Base, Bedford, MA 01731		12. REPORT DATE November 1975
		13. NUMBER OF PAGES 181
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

COMPUTER SECURITY
MULTICS
PASSWORDS

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The random word generator is a PL/I program designed to run on Honeywell's Multics system that generates random pronounceable words suitable for use as passwords for Multics users.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ACKNOWLEDGEMENT

Special acknowledgement is extended to Lt. Col. Robert Park and 1Lt. Brian Woodruff for their expert guidance in the preparation of Section IV of this report on the statistical analysis of the random word generator.

TABLE OF CONTENTS

TABLE OF CONTENTS (concluded)

LIST OF ILLUSTRATIONS

SECTION I

BACKGROUND

The random word generator is a PL/I program designed to run on Honeywell's Multiplexed Information and Computer System (Multics), a large timesharing system. The purpose of the program is to generate passwords that serve to authenticate the identities of users of Multics.

Users of the standard Multics system authenticate themselves at each login (start of a terminal session) by typing in a password known only to the user and the system. Usually the user selects a password for himself at initial login and can change this password at any subsequent login. The ability to change a password is useful when a user suspects that someone else may have guessed his password.

A password is the key to user identification and protection. From previous experience, however, it has become apparent that user-selected passwords are frequently fairly easy to guess. For example, passwords are often the user's own first name, a name of a family member, or his telephone number. [1] Some Multics installations, such as the Air Force Data Services Center (AFDSC), are used to process classified information, and installations like the AFDSC cannot take a chance that one of their user's passwords will be guessed. The solution to the problem at the AFDSC was a decision to assign passwords to users, rather than to allow users to pick their own. [2]

The administrative overhead of assigning a random password manually whenever a user changes his password is generally too much of a burden -- especially when one considers that only a select few individuals may have access to other users' passwords. Instead, the possibility of providing computer generated passwords that are printed out at the user's console on each password change was investigated. A user's request to change his password would then become a call to a system password generator program. This password generator has been given the more general and descriptive name of "random word generator" in this report.

Section II of this paper discusses the goals and methods used by the random word generator. Section III contains implementation details and discusses the word generating algorithm. Section IV contains an analysis of the algorithm and presents certain statistics. A sample of random words generated can be found in Appendix IV.

Most of this report describes the random word generator that is being made available for users on Multics. In order to satisfy a more stringent criterion of "randomness" a modified version of the random word generator has also been prepared that generates random words that are all equally probable. This modified version is discussed at the end of Section IV.

SECTION II

METHODOLOGY

REQUIREMENTS AND GOALS

The need for the Multics random word generator program can be satisfied by fulfilling the following two requirements:

generate easily remembered words, and
make the words difficult to guess.

The first requirement is very important because users might be inclined to write down passwords that are difficult to remember, thereby increasing the chances of a password compromise. Also, if too many users forget their passwords, the administrative overhead of getting the users logged in again could be greater than that of distributing easier-to-remember passwords manually. The need for the second requirement is apparent.

Both of these requirements are of course far too subjective for direct implementation as a computer program. It is necessary to restate these requirements in a much more concrete manner so that meaningful algorithms can be designed.

The requirement of "making the words difficult to guess" is most easily satisfied by giving the program the ability to generate a very large set of possible words, and the ability to generate these words in a random manner. Both these capabilities can easily be achieved, and thus we will be more concerned at this point with the first requirement.

Consider that the random word generator either needs a large data base of words to choose from -- an impractical approach that has been discarded -- or has to form words out of sequences of letters it creates through some algorithm. The requirement of rememberability can then be fulfilled if these sequences of letters are of one or more of the following types:

1. Sequences of letters that can be easily visualized, such as "aabbaa" or "xyxyxy".

2. Sequences of letters that form real English words.

3. Sequences of letters that form pronounceable "words", but are not necessarily real words.

Of these three choices, methods 1 and 2 suffer from the difficulty of specifying a practical algorithm for such sequences. Alternatively, method 1 could be implemented using rules that yield an arbitrarily defined subset of all possible easy-to-visualize sequences, but this subset is likely to be small for a reasonable number of rules. There is no alternative for method 2 other than storing a vast data base of real words.

The third method -- that of using pronounceable sequences -- is the selected approach. The data base required for this method is relatively small, the rules can be fairly well-defined, and the set of words that can be generated is quite large. Realizing that the more "English" a word looks the easier it usually is to remember, an attempt was made to restrict the set of words generated to those which obeyed some kinds of rules of English pronunciation. This attempt was restructured as an attempt to make it theoretically possible for the word generator to form most English words.

Because of this goal of making the generated words look like English, the word "pronounceable" in this paper refers not just to structures that can be phonetically vocalized, but to a set of more restrictive and English-looking structures. For example, "tsip" is easily pronounceable, but is "un-English" because of the "ts" at the beginning of the word. A different type of example is the "gh" combination. The word "cough" is pronounceable because "gh" in this context can be pronounced like "f", but "ghrom" is not pronounceable as "from" because "gh" never sounds like "f" at the beginning of a word.

PRONOUNCEABLE?

One may wonder how a goal of "pronounceability" can be attained with well-defined rules, considering how undefined and exception-laden the rules of English pronunciation are. The answer is simple: the program does not care <u>how</u> a given word is to be pronounced -- it only needs to make sure that the word <u>can</u> be pronounced. For example, "tophat" could be pronounced "top-hat" or "to-fat", depending on the reader's preference. On the other hand, "tophsat" is only pronounceable as "tof-sat", not "top-hsat". Also, the vowel "o" in this word might be pronounced in one of several ways.

Everyone knows that a given letter or sequence of letters could be pronounced differently in different contexts, but the program is usually not required to distinguish between the different contexts or pronunciations. Unlike the "rules of pronunciation", the "rules of pronounceability" can be made fairly precise.[1]

However precise, the rules and method used to generate pronounceable words have been arrived at in a "refined" ad hoc manner and based on the author's intuition. The method is not described in any published source. Hence, the words generated may be considered pronounceable only by the author. Others may find some of these words very difficult to pronounce, as when trying to pronounce a foreign word with a strange combination of letters. Because of this possible bias, the program was designed to incorporate as few global rules as possible within its text. An external data base, a table, is used to contain most of the subjectively determined rules. These external rules consist of "yes" or "no" answers to various questions asked by the program. The answers to the questions can easily be modified to suit the user's preference. "New" rules -- those asking new kinds of questions -- cannot be added without modifying the program.

THE POOR APPROACH

Letters are poor sources on which to base rules of pronounceability. Not only do individual letters sound different in different words, but pairs or triplets of letters often form single sounds that may be unlike any of the component letters. Determining whether a letter is pronounceable or "legal" in a given word often involves knowing how the letter is to be pronounced, which is in turn dependent on such things as its position in the word or syllable and adjacent letters. The large number of details that have to be checked for each letter makes determination of pronounceability very complex.

THE IDEAL APPROACH

The ideal approach that will always generate good pronounceable words would be to relieve the program of any notion of letters and use

---

[1]Phonological theory is a well developed science that in part attempts to describe the phonetic structure of English (and other languages) in a complete and consistent manner. The totality of rules and theorems used in such a description form far too complex a system for the scope of the application discussed in this report. Creation of a smaller subset of this system -- one that might be small enough to implement and would still give reasonable results -- appeared to be too vast an undertaking. Thus, standard phonological theory was not considered in this work.

"phonemes"[2] instead. A phoneme is an "element" of pronunciation -- a unit of sound that cannot be usefully broken down into smaller sounds. For example, the pair "sh" as pronounced in English can always be represented as a single phoneme; the vowel "a" can be represented as one of several phonemes depending on its context. If the rules could be defined, it should be possible to put together random phonemes to form a pronounceable phoneme-word.

Unfortunately, though this method yields good pronounceable sequences, the translation from phonemes to letters is very difficult and very un-algorithmic. An example of this difficulty is the phoneme representing the sound of "k". This phoneme can be translated into "c", "k", or "ck". Which one should be used? At the end of a word, usually any one of these will work, and another randomization factor has to be included to make the choice. At the beginning of a word, "k" is always legal, "ck" is never legal, and "c" is legal only if the following letter is not "e", "i", or "y" (in which case "c" would have been pronounced like "s"). Then, to determine whether "c" is a candidate the following phoneme must first be translated into letters, which may in turn depend on other adjacent phonemes. One can fix the translation so that the "k" sound is always translated to the letter "k", but then the goal of being able to generate most English words would be far from satisfied (not to mention that the letter "c" would never be used).

THE COMPROMISE APPROACH

One may notice that with the phoneme method the program "knows" how the generated word is pronounced -- specifically not a requirement as mentioned earlier. A compromise approach was chosen that uses simple "units", instead of phonemes, that consist of a single letter or a two-letter pair. A given unit is considered by the program as having only one "sound" in all its usages, although in reality that unit may be pronounced in many different ways.

Rules can be determined for each unit, without regard to how that unit is pronounced in context, by merely stating a rule that includes all usages of that unit. This composite rule is usually simpler than all of the individual rules for the different pronunciations of that unit. For example, the letter "g" can be treated as a unit, and the rule for this unit at the beginning of a word says "this unit may only be followed by a, e, i, l, o, r, u or y." In some of these cases "g" is pronounced soft and in others hard -- in fact there is no simple rule for how "g" is pronounced (e.g., "gigantic" and "giggle"),

---

[2]also called "phonetic segments" in phonology.

but the program doesn´t need to make any distinctions.

The 34 units presently used are listed below. These units are stored in a table and are input to the random word generator. A larger or smaller set can be defined if experience indicates these to be unsatisfactory.

a	f	k	p	v	ch	th
b	g	l	r	w	gh	wh
c	h	m	s	x	ph	qu
d	i	n	t	y	rh	ck
e	j	o	u	z	sh	

Note that the letter "q" is the only letter not appearing as a one-letter unit because English usage makes it more convenient to treat "qu" as a unit. Many two-letter vowel combinations, such as "ea", "ie", "ai", etc., that should be considered separate units are not included because little loss of generality occurs (i.e., the set of words that can be generated is nearly the same whether these vowels are separate units or not). Also, double letter pairs like "ll", "rr" and "tt" need not be included for similar reasons. On the other hand, the pair "sh" is needed because words such as "shrink" and "wash" are not pronounceable when "s" and "h" are treated as separate units.

SYLLABLES

Besides the rules used by the program, there is a primary assumption that governs the formation of words: if pronounceable syllables are concatenated (subject to some minor restrictions), they will form a pronounceable word. Thus, the task of the random word generator is to form pronounceable syllables.

This task requires precise definition of "syllable"; thus the following definition is made at this point: a syllable is an arbitrary series of units that contains exactly one or two consecutive vowel units. Vowel units are "a", "e", "i", "o", "u", and "y". For example, the following are legal syllables (where "v" represents a vowel unit and "c" represents a non-vowel unit, or consonant unit):

ccv   cvvccc   cv   v   vv   vc

and the following are illegal syllables:

cc	(no vowels)
vvv	(more than 2 consecutive vowels)
vccvc	(all vowels not consecutive)
vcv	(all vowels not consecutive)

Note that each of the last two examples can possibly be split into two syllables, such as "vc-cvc" and "v-cv".

The above definition of "syllable" seems to work in English except for one common case: the silent "e" at the end of words or sometimes syllables often forms a syllable containing two non-contiguous vowels.[3] Of course, that is because English usually only requires a vowel <u>sound</u> in a syllable, and in the case of silent "e" the "e" should not be considered a vowel. The program, however, has no way of telling whether the "e" is silent. To make matters worse, there are words, such as "subtle", "bugle", "little" that <u>do</u> have a final syllable whose only vowel is the final (silent) "e". These are common enough cases in English to warrant special consideration in the word generator.

JUXTAPOSITION

The random word generator forms syllables from left to right, by combining random units one at a time. For each new unit the program determines whether that unit can legally be appended to the units already in the syllable. If it cannot, the unit is discarded and another random unit is tried.

In English the legality of a unit is usually determined by checking immediately adjacent units. Units separated from each other frequently affect each other's pronunciation but only occasionally determine whether the construction is legal or not. The random word generator uses rules of juxtaposition as the bases for creating pronounceable syllables.

Each time the program gets a new random unit, it forms a pair consisting of that unit and the previous unit. This unit-pair is looked up in a table and bits of information are extracted that specify what can be done with that pair. For example, the unit-pair "rt" will have bits specifying that the pair may not begin a syllable and that a vowel must precede this pair if it is entirely contained within a syllable. The table may sometimes specify that a unit-pair must always be split between two syllables (for example, the pair "kp"), which is one way in which a new syllable can be started. Some pairs, such as "hh", can never appear together, even if split between syllables. The different types of rules that can be specified in the table are discussed in the next section.

---

[3]The vowel pair "ue" in "baroque" and "catalogue" is another exception, though much less common.

MISCELLANEOUS ASSUMPTIONS

Several more assumptions have to be made before syllables can be generated properly. Again, these assumptions were arrived at intuitively and no claim is made for their completeness. The assumptions discussed below are those that have been incorporated into the program structure, as opposed to those that are specified in external tables. They are presented in order of importance.

Consecutive Vowels

A rule, in part already stated, involving consecutive vowels, says that a maximum of two consecutive vowel units is permitted. This rule pertains to all consecutive vowel units even across syllables. The reason for this extension across syllables is that sequences such as "aiea" look "funny" and are sometimes difficult to pronounce, even though there can be a syllable split in the middle. The English language itself "admits" of this difficulty between consecutive words by trying to correct it in two common cases: the use of "an" instead of "a", and the alternate pronunciation of "the", when the following word begins with a vowel sound. There are few English examples of more than two consecutive vowels (the "eau" combination is one of them). Note that the word "queen" is legal according to random word generator rules because "qu" is considered to be a consonant unit.

A difficulty with this assumption involves the unit "y". For purposes of syllabification, "y" must be treated as a vowel (i.e., a syllable can contain the single vowel "y"), but for the above assumption "y" should not be treated as a vowel. Three-vowel sequences involving "y" are very common: "eye" and "you" being two examples. Thus the requirement of at most two consecutive vowel units must be waived if one of the vowels is "y".

The Vowel "y"

In order to solve the consecutive vowel problem above it sufficed to treat "y" always as a consonant. However, it should also be legal for "y" to be the only vowel in a syllable. Therefore, for the purposes of syllabification only, the random word generator treats "y" as a vowel only if the "y" is not immediately preceded by a vowel within that syllable. The sequence "vowel-y-vowel" would thus have to be split between two syllables, but "y-vowel-vowel" would not. The additional rule about silent "e" below allows a "vowel-y-e" sequence to end a word.

The Silent "e"

The special case of final "e" has previously been mentioned. The final "e" in a word in English is almost never pronounced and therefore cannot be used as the only vowel in the last syllable. There is no problem taking care of such exceptions in a uniform way. However, there is a very large set of exceptions to this final "e" rule: words such as "meddle", "nestle", "double" -- all ending in "le" -- are legal words in English, yet no vowel is pronounced in the last syllable. The rules used by the word generator do not allow final syllables of "ble" and "tle" and therefore such words cannot be generated. This class of words appears to be the largest that cannot be handled by the word generator. In order to solve this deficiency it would be necessary to first include "le" as a unit in the table, and then make special kinds of tests to determine whether this unit is legal in a given context. It is not possible, without creating new rules specific to this "le" unit, to specify the necessary restrictions. Creating new rules for this case was considered feasible, but appeared to be too awkward and so was left out.

The Initial "y"

The unit "y" may not be the only vowel in the first syllable of a word if the word begins with "y". Only strange words like "yclept" violate this rule. This is a minor point but must be taken care of explicitly. Otherwise, many strange words are generated.

Three Identical Units

There is nothing in the rules so far stated that prohibits three or more identical consecutive consonants. This condition may possibly be legal, provided that no more than two consecutive consonants occur in the same syllable. Instead of trying to force a syllable split between such groups, the decision was made to merely limit the number of consecutive identical units to two. Note that this restriction is not a pronounceability problem, but a case of an un-English-looking construction.

SUMMARY

The goal of the random word generator is to generate easily remembered words that are difficult enough to guess to be suitable for passwords. This goal has been translated into requirements of pronounceability and randomness. An attempt was made to include almost all English words in the set of words that can be generated, and to exclude constructions that are never found in English words.

The random word 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. The rules are used to determine whether a given unit is illegal or legal, 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. In a few cases checks do extend outside the current syllable. These case are:

1. Three identical consecutive units
2. Three consecutive vowel units
3. Silent "e" at the end of a word
4. "y" beginning a word
5. Certain illegal pairs of units

SECTION III

IMPLEMENTATION DETAILS

The random word generator is organized as a main procedure that references two tables and an external procedure. The user supplies the two tables: a "unit" table that defines the units (such as those listed on page 7) and specifies rules about each unit, and a "digram" table that specifies rules about all possible pairs of such units. The random_unit subroutine, which returns a random unit when called by random_word_, must also be provided by the user. The method used by this subroutine to generate the random units may be any method desired and based on any distribution. Such a distribution might, for example, be based on the frequency of use of the individual units in English.

SPECIFICATION OF RULES

As mentioned in Section II, the random word generator uses two types of rules: those that are fixed and embodied in the program structure and those that are variable and embodied in external tables. The fixed rules are general in that they are not specific to any one letter or unit. The tables specify rules pertaining to individual units or the juxtaposition of units. The tables will be discussed first, followed by specification of the internal rules.

The Digram Table

This table contains one entry for every possible pair of units (digram), whether that pair is allowed or not. Thus, with 34 different units, there would be 1156 entries. The entry for each pair consists of eight bits of information that together form the "rules" for that particular digram. Each bit is a yes or no answer to a specific question asked by the program. The name of each of these bits and the questions answered are as follows:

1. must_begin — Must this pair begin a syllable?
2. not_begin — Is this pair prohibited from beginning a syllable?
3. break — Is this pair illegal within a syllable (i.e. must it be split between two syllables)?

4. prefix	Must this pair be preceded by a vowel unit if it does not begin a syllable?
5. suffix	Must this pair be followed by a vowel unit if it does not end a syllable?
6. end	Must this pair end a syllable?
7. not_end	Is this pair prohibited from ending a syllable?
8. illegal_pair	Is this pair illegal (even if split between syllables)?

Obviously all eight bits are inherently non-independent. There are actually far fewer combinations of these eight bits that can be specified. Out of these, less than sixteen combinations are ever used in practice due to the structure of the English language. Thus, four bits yielding 16 combinations would be enough. The actual internal representation of these bits only affects speed and storage space, however, and is not of importance in this discussion. In addition, some other application of the random word generator (perhaps with a different language) may use more combinations. Appendix I contains the digram table currently in use for the 34 units defined on page 7.

An example will best illustrate the usage of these bits. Consider the digram table entry for the pair of units "f" and "l" as shown in Appendix I. The bits that are set for "fl" are:

must_begin
suffix
not_end

The must_begin bit says that if an "fl" is encountered in a syllable, it must begin that syllable. The suffix bit says that the unit following "fl" must be a vowel if "fl" is not the last pair in a syllable. The not_end bit says that "fl" may not be the last pair in a syllable. The specification of the digram "fl", thus, restricts its use within a syllable as the first pair in one of the following six contexts:

fla..., fle..., fli..., flo..., flu..., fly...

where "..." signifies additional units within the syllable. Of course, if there are any further restrictions on the use of the pairs "la", "le", etc. that prevent them from appearing after the "f", these restrictions must be taken into account. Note that none of the eight digram bits except illegal_pair apply when the pair is split between two syllables. If "fl" is split, the "l" becomes the first unit of the next syllable, and rules for pairs beginning with "l" must be examined. A quick glance at the digram table shows that all pairs beginning with "l" have the not_begin bit set, except the six pairs:

la, le, li, lo, lu, and ly,

and processing can continue with this information.

The random word generator makes sure that at all times the rules specified in the digram table are satisfied for every two consecutive units in the word being formed.

The Unit Table

In addition to rules for unit pairs, there is a table containing four bits of information pertaining to the individual units. For each unit, the four bits are as follows.

1. not_begin_syllable

   This bit indicates that this unit may not begin a syllable. This bit is redundant in that the digram table can specify that all possible pairs beginning with this unit may not begin a syllable. The purpose for using this bit is for efficiency -- when generating the first unit of a new syllable, the program would otherwise have to search through all possible digrams beginning with this unit in order to determine whether this unit is legal. This bit is currently set for the units "x" and "ck". A small number of words in English do begin with "x", but they are mostly technical or scientific terms.

2. no_final_split

   This bit indicates that this unit, when appearing at the end of a word, must not be the only vowel in the final syllable. This bit is only set when the "vowel" bit is set, and is currently set only for the unit "e".

3. vowel

   This bit is set for vowel units. It is currently set for the units: a, e, i, o, u, but may also be set for any units consisting of vowel pairs or that are to be treated as vowels that one might add to the table at some future time.

4. alternate_vowel

This bit indicates that this unit is to be treated as either a vowel or a consonant, depending on context as discussed in Section II of this report on page 9. This bit is set only for the unit "y".

Admittedly these four bits are highly specialized and at least bits 2 and 4 could just as easily be incorporated into the program logic as tests for specific units. However, the program actually works with numbers representing units, rather than the units themselves, and the assignment of a particular number to a particular unit is arbitrary. By using a bit in the unit table for all special cases, all references to specific letters or units are removed from the program. Refer to Appendix I for the unit table currently in use.

Random Units

As stated earlier, the random word generator requires the user to supply the subroutine random_unit. This routine is called by the word generator each time a random unit is needed. The random units are generated based on some predetermined distribution. Of course, not all units thus generated will be acceptable to the word generator in every position of the word: random_unit will be repeatedly called until an acceptable unit is returned. 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 and the digram and unit tables. The random_unit subroutine itself makes no tests for legal units, but merely uses its fixed distribution each time it is called.

The distribution of units that is currently in use along with the digram and unit tables discussed earlier is shown in Appendix VI in the description of the random_unit_ subroutine. There is another entry point in random_unit called random_vowel, which is called by the word generator for efficiency in cases when it is known that only a vowel unit will be acceptable. The distribution of vowels returned by this second entry is also shown.

The Algorithm

The digram table, the unit table, and the random_unit subroutine are considered user-supplied in that they may be modified without affecting the word generator program logic. The external rules were specified in the two tables. The algorithm used to generate random words based on these external rules defines the fixed internal rules. The internal rules cannot be modified without changing the logic of the algorithm. The complete algorithm is shown in Appendix II, written in a PL/I-like language, and a high level flowchart is shown in figure 1. Appendix III contains the source program listing of random_word which implements this algorithm.

The function of the main body of the algorithm is to determine whether a given unit, generated by random_unit, can be appended to the end of the partial word formed so far. If illegal, the unit is discarded and random_unit 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. A unit previously generated and accepted can never be discarded.

The flowchart in figure 1 shows generally how a word is built up. The names in all capitals (INDEX, SYLLABLE_LENGTH, etc.) are references to variables initialized within the flowchart. Names in quotes (e.g., "syllable_length") refer to the bits in the digram table or unit table for the last pair of units or the current unit. The array UNIT holds the units of the word as they are generated, where UNIT(INDEX) is the current unit.

Beginning at the top of the flowchart, the first unit of the word is selected at random by random_unit and inserted into UNIT(1). If this unit is legal, according to rules in the unit table, the second unit of the word is selected and loaded into UNIT(2). This time the rules must be satisfied for both the unit table entry for UNIT(2) and the digram table entry for the pair [UNIT(1),UNIT(2)]. If a given unit is not acceptable, another is tried in its place. When the end of the word is reached (as determined by the number of letters desired by the caller of random_word_), additional checks are made before the algorithm can terminate.

If the digram table is consistent, there should always be some unit that will be legal for any legal state of the algorithm.[4] However, self-consistency checks on the digram table are extremely difficult to make. Therefore, an arbitrary limit of 100 tries is placed on generating any particular unit. If 100 calls to random_unit fail to yield a legal unit, the whole word is discarded and the program starts over. This 100 tries limit is not explicitly shown in the flowchart but is contained in the program text (see Appendix III).

Another observation concerning the 100 limit is that, because the program is dealing with random events, it is theoretically possible for 100 tries to fail to yield a legal unit even though there is a

---

[4] A "state" here is defined by the values of the state variables used in the algorithm as given in Appendix II, and includes the units already accepted as part of the word being formed.

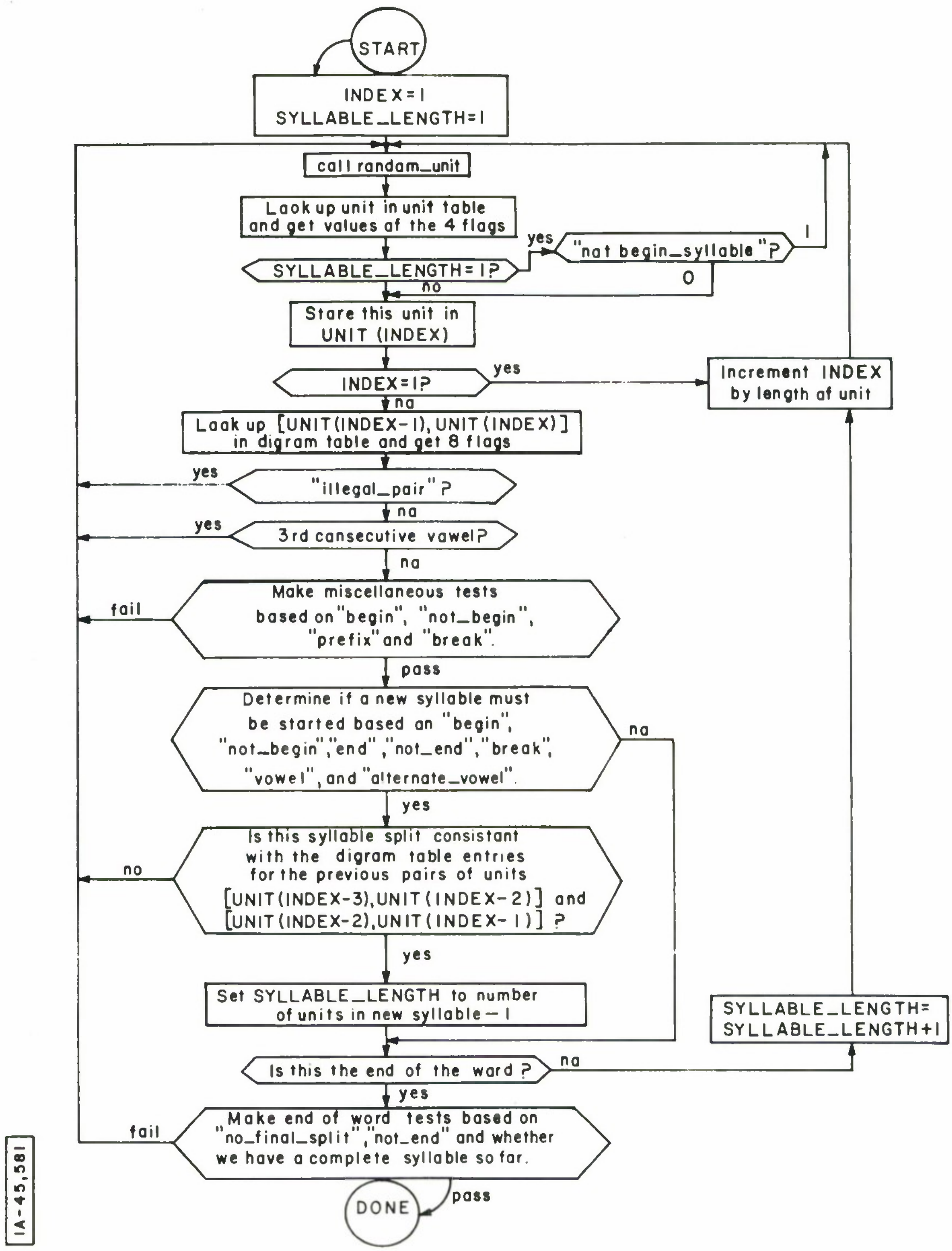


Figure 1. Random_Word_ Flowchart

unit that is legal. Thus, in order to prevent excessively long loops, it is useful to place a limit on the number of tries, even though the digram table may be consistent.

RESULTS

Appendix IV contains a printout of 2000 random words of five to eight letters. The length of five to eight letters was chosen for this run because such words are more pronounceable than longer words, and fewer than five letters causes too many duplicates to be generated, thus making the words unsuitable for use as passwords. Actually the random word generator has a capability of generating words of any length. The words in the printout have been sorted alphabetically merely as an aid to checking certain constructs. The possible relationships between successively generated words depend on the random number generator in use, which is outside the scope of this discussion.

Notice in the printout that alongside each word is the same word divided into syllables (hyphenated). An interesting by-product of the algorithm is the ability to determine syllable divisions in the word generated. In certain cases the syllable split can not be precisely defined. For example, the word "without" can be split as "with-out" or "wi-thout" according to the rules. In such cases the random word generator makes an arbitrary (but predetermined) choice of where to split the syllables.

It may also be that certain hyphenations are not the most logical as an aid to pronunciation. An example can again be found in "without", which would be hyphenated as "wit-hout" if the "t" and "h" were generated as individual units instead of as a "th" unit. The decisions about hyphenation made by the program are built into the algorithm and are based on what the author considers the most likely to be acceptable in the general case.

SECTION IV

ANALYSIS

Of the two requirements of the random word generator stated on page 3, the requirement of making the words "difficult to guess" was stated as being easy to achieve by giving the word generator the capability of generating a very large set of words. The more difficult requirement of pronounceability guided the design, and it was intuitively assumed that a large enough set of words to satisfy some criterion of randomness would automatically result.

In this section an attempt is made to present some quantitative measurements and statistics that may allow one to determine whether the word generator is actually "random enough" in some sense. With a tool as crucial to the security of the system as a password generator, it must be assured that the passwords really are "difficult to guess".

There is no one quantity to be calculated that will provide a meaningful description of the random word generator's effectiveness in all its possible applications. For example, in an application where the random words are used to create identifiers of individuals, the total number of possible words and the probability of duplication are of interest. In the application as a Multics password generator, duplicates are not as important as the probability that a given user's password will be guessed by another user.[5] For other applications the probability distribution of the words might be required.

It is hoped that enough statistical data is provided in this section so that, with sufficient further analysis, most quantities of interest can be calculated. A complete statistical analysis is beyond the scope of this discussion. However, attention will be focused on areas of interest to users of the random word generator as a password generator for Multics.

---

[5]In Multics, it is of little value to know a password without knowing the name of the user to whom it belongs; i.e., one cannot login to the system merely by typing a password and thereby impersonating whoever that password happened to belong to. Other systems may actually use the password to identify rather than verify.

The following four topics have been chosen for consideration:

total number of different pronounceable words,
probability of a given word being generated,
most probable word, and
distribution of word probabilities.

Some quantitative measure of each of the above has been obtained, but through empirical analysis of the random word generator's output rather than through an analysis of the algorithm. Analysis of the algorithm would of course yield the most precise statistics, but the complexity of the algorithm and its states, and the large amount of data in the supporting tables (which might be subject to change by anyone), make such an analysis extremely difficult and somewhat limited in applicability. Instead, minor modifications to the random word generator and some additional programs were incorporated to supply the data necessary for this analysis. If a change is made in one of the supporting tables, new data can be obtained merely by re-running these additional programs.

It should be noted that all of the statistics and numerical figures presented in this section apply only when the tables are set up as in Appendix I. The methods used to obtain the results, however, apply to any tables the user may supply.

NUMBER OF WORDS

The number of possible random words, though extremely difficult to determine by analyzing the algorithm, can be established to any degree of accuracy in a fairly simple manner.

Consider all possible words of a given length L that can be formed from the 26 letters of the alphabet, without regard to pronounceability. If N is the number of such words, then

$$N = 26^L. \quad (1)$$

Out of these, a certain fraction f are "pronounceable" according to random word generator rules. The value of f may, of course, depend on L. If we can determine f, we can calculate the number of pronounceable words n of a given length simply by

$$n = fN. \quad (2)$$

An estimate for the value of f can be obtained by picking a random subset of size m out of the N words, and finding out what fraction of this subset is pronounceable. The larger the value of m, the smaller the probable error we will have in our estimated value of f. Actually the accuracy of our estimate can be expressed in terms of a probability that its absolute error is less than a certain amount.

Generating a random subset of N words of length L is easy with a uniform random number generator. With a small modification[6] the random word generator can be given a particular word, and will "run through its rules" to determine whether the word is legal (i.e., pronounceable). A sample run of 100,000 words of eight letters was made. The length of eight letters was chosen for this run because that is the maximum length of a user's password acceptable to Multics. There were 2653 acceptable words out of this run of 100,000, yielding an estimate for f of .02653. For eight letters,

$$N = 26^8 = 208,827,064,576 \tag{3}$$

and the estimated value for n is

$$.02653N = 5.540 \times 10^9. \tag{4}$$

The accuracy of the estimate for f as determined above can be calculated as a confidence interval for f. This confidence interval is written approximately, for large m, in the form

$$\left[k/m \pm z\sqrt{\frac{k/m(1-k/m)}{m}} \right] \tag{5}$$

where k is the number of acceptable words out of the sample of m, and z is an appropriate percentage point of the standard normal distribution. For example, we might be interested in a 95% confidence interval, which corresponds to a value of z = 1.96. In the sample of 100,000 above, this yields

$$[.02653 \pm .00099] \tag{6}$$

For other confidence regions, and for sample runs of words of different lengths, see figure 2.

---

[6]The only change is to use a special version of the random_unit subroutine (which is user-supplied) that supplies units of a known word rather than random units.

word length	confidence range	number of words minimum	maximum
6 letters	99.9%	1.745×10^7	1.896×10^7
	99%	1.761×10^7	1.880×10^7
	95%	1.770×10^7	1.867×10^7
	90%	1.782×10^7	1.858×10^7
8 letters	99.9%	5.191×10^9	5.889×10^9
	99%	5.269×10^9	5.812×10^9
	95%	5.331×10^9	5.787×10^9
	90%	5.363×10^9	5.714×10^9
10 letters	99.9%	1.464×10^{12}	1.794×10^{12}
	99%	1.499×10^{12}	1.759×10^{12}
	95%	1.535×10^{12}	1.735×10^{12}
	90%	1.546×10^{12}	1.712×10^{12}

Figure 2. Number of Words of 6, 8 and 10 Letters

PROBABILITY OF A WORD

The words produced by the random word generator are not all equally probable for two reasons. First, different units have different probabilities of being generated by random_unit. Second, not all units thus generated are always acceptable. The probability of a given word must be calculated by examining the conditional probabilities of the individual units in that word.

Since random words are created left to right, at a given point during the creation of a word the units accepted so far determine which units may follow. Thus the probability of a particular unit appearing in a particular position of a word is the ratio of that unit's probability (of being returned by random_unit) to the total probability of all the units that are legal in that position. This calculation can be made for each unit based only on the units that precede it. In order to calculate the probability of a particular word, the probabilities of the individual units in that word are determined in this manner and then multiplied together.[7]

---

[7] The 100-try limit discussed on page 15 may cause entire words to be rejected even though some units were accepted. However, test runs have shown that the 100-try limit is almost never reached.

The method described above works because, for each position of the word, the random word generator keeps trying random units until a legal unit is found. The unacceptable units play no part in the probability that a particular legal unit will appear. For example, suppose in a given position of a particular word the only legal units are "e" and "a". If it is known that the probability of "e" appearing at random is .057, and the probability of "a" is .047, then the probability that the unit will be an "e" is .057/(.057+.047).

Since the random word generator does not throw out a unit once it has been accepted, it is merely necessary to multiply the individual conditional unit probabilities together to arrive at the probability of the word. Note that this probability only applies to words of a given length (i.e., the length of the word whose probability is calculated). The random word generator does not pick a length, but is asked to generate a word of a specified length. If random lengths are supplied to the random word generator, the distribution of these random lengths must be figured into the probability of the word calculated.

The special program described in the previous subsection that "gives" the random word generator known words was modified to calculate the probability of the known word in the manner described. The answer is exact in most cases,[8] and the method will work regardless of the definition of the units, the tables, or the nature of the algorithm. The only restriction is that the word generator not discard units that have already been accepted in a given position of a word, and that the distribution of the units returned by random_unit remain constant during the formation of the word.

MOST PROBABLE WORD

The "most probable" word (or words) and its probability as determined in the above manner is meaningful to those interested in the difficulty of guessing random words. In the password application, for

---

[8]Some words can be divided into units in one of two ways. For example, "w-i-t-h-o-u-t" and "w-i-th-o-u-t" are two ways of specifying the units of "without", both of which are legal. An exact calculation of the probability of this word would require adding the probabilities of both forms. In general, however, the probability of the version that contains more units (i.e., "w-i-t-h-o-u-t") is much lower because of the extra unit, and thus makes little difference in the total probability of the word. In calculating probabilities of words containing two-letter units that may possibly be split into two one-letter units the calculation is based on the word with the two-letter units.

example, it does not matter if there are one billion random words if the most probable word has a probability as high as 50% (even though the probability of all other words may be small). A systematic method for guessing a particular user's password would be to first try the most probable word and work down from there. If the first word tried has a high probability, a large set of legal words is of little value.

As important as this statistic is, it appears that only an extremely complex analysis will yield the most probable word. The obvious method of selecting only the most probable units to form a probable word does not work. For example, two of the most probable units are "e" and "t". One might expect that the most probable six letter word is something like "teetee". Actually, a word like "heehee" is almost twice as probable. A simple calculation can show that the first two units of a word are much more likely to be "he" than "te". Even though "t" has twice the probability of being first, the set of legal units following "t" is greater than the set of units following "h". It turns out that with the tables in use there are only six units that may follow "h", whereas there are eight that may follow "t". The probability that one of those six will be "e" is fairly high. The low probability of "h" mutiplied by the high probability of "e" yields a value greater than the probability of the pair "te".[9]

An empirical approach to arriving at the most probable word might be to generate a large number of random words and to calculate their probabilities. Unfortunately, even the most probable word may have a probability sufficiently low so that millions of words might be generated before the most probable word appears. Moreover, one would have no assurance that any particular word really is the most probable.

Once more, intuition and a "feeling" of the rules and restrictions of the algorithm were relied upon. The utility programs previously described made it easy to try many expected high-probability words manually. In this way, a guess of the highest probability words of 6, 7, 8 and 10 letters has been made. The words are listed below, along with their probabilities. There may actually be several words of each length with the same probability. The results below only apply if the specific digram and unit tables listed in Appendix I are used, and if the distribution of the units is as listed in that appendix.

---

[9]One should also consider that the use of two-letter units increases the probability of certain words. The six-letter word "quequo" is an order of magnitude more probable that "teetee", because it actually only contains four units (qu-e-qu-o), even though the probability of "qu" coming from the random_unit is very low.

word	p	1/p
quethe	2.45×10^{-6}	408,000
squequo	1.64×10^{-7}	6,098,000
queshquo	2.19×10^{-8}	45,662,000
queshquesh	1.81×10^{-10}	5,525,000,000

The probabilities above only apply to words of the specific length shown. For example, if the word generator is asked to generate words of a random length of 6, 7 or 8 letters, and each length is equally likely, then the probabilities above are multiplied by 1/3. Of course, the probability of the six letter word is so high that the other two words are of little interest if six letter words are allowed.

DISTRIBUTION OF PROBABILITIES

The ability to calculate the probability of a given word, and the total number of words allows us to arrive at an approximate distribution of the probabilities of the pronounceable words, from most probable to least probable. This distribution yields a kind of profile of the word generator that may be the best overall measure of the word generator's effectiveness. One method of arriving at such a distribution is outlined below. As with the number of words, the accuracy of the distribution curve depends on the size of the sample of random words used.

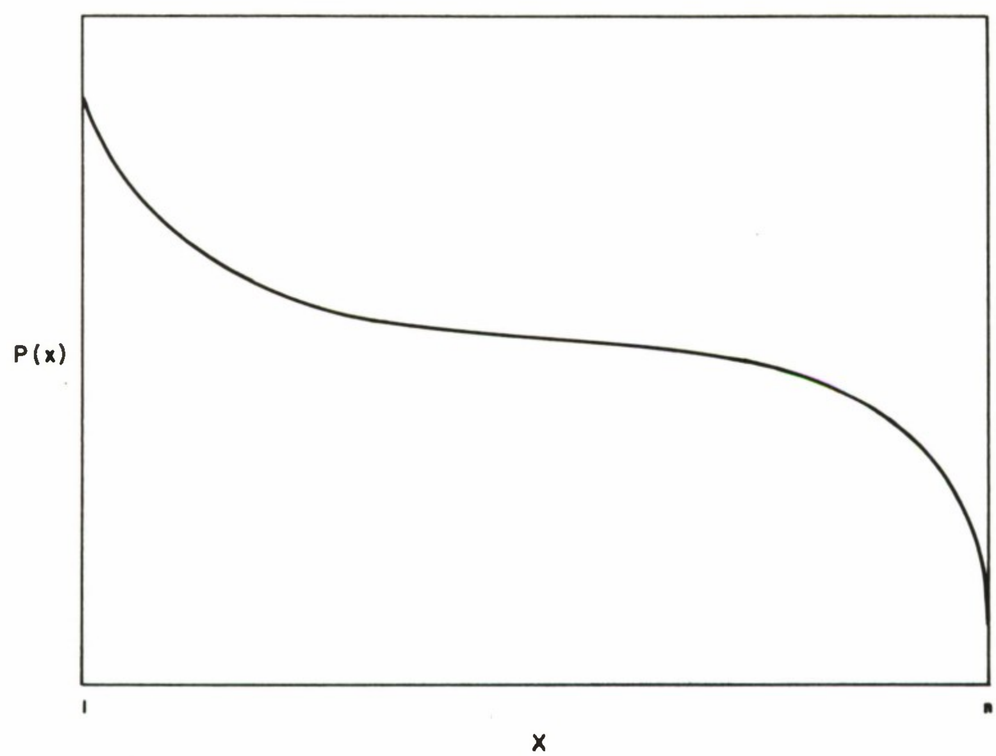
Assume that all n pronounceable words of length L are listed in order of probability, and that a "word number" x,[10] running from 1 to n, is assigned to each word, where x = 1 for the most probable word. Let p(x) be the probability of word x. If we had all n words, we could plot x against p(x) as in figure 3 to obtain a series of points. The distribution p(x) will be loosely referred to as a "curve" although strictly it is not a continuous function. Of course, p(x) is monotonically non-increasing by definition. The area under

[10]The letter "x" has been chosen for the word number instead of the more obvious choice of "i" to represent an integer in order to be more consistent with the notation generally used for some of the calculations in the following pages that treat x as a continuous variable.

the curve is unity, or more precisely

$$\sum_{x=1}^{n} p(x) = 1. \qquad (7)$$

Once the curve is obtained, quantities like the total probability of the m most probable words, the probability of duplicates within a certain number of tries, etc., can be calculated or measured from a graph of the curve.



IA-45,671

Figure 3. Distribution of Probabilities of Random Words

Determination of Distribution

If all n pronounceable words (and probabilities) were available, producing p(x) exactly would be no problem. In reality, we can only obtain a certain fraction of the n words. If we could in some way select every millionth word in the ordered list of n words, we could still estimate a curve of p(x) by merely plotting every millionth point in figure 3 and interpolating to get the values in between. The accuracy of such a plot will depend on the "smoothness" of p(x) in some sense (and of course on the method used to make the interpolation).

There is no direct way to arrive at every millionth word in the list. We can generate k random words but we have no way of knowing what their positions are in the list (i.e., their values of x). In fact, if we generate k random words, their values of x will not be evenly distributed in the interval [1, n], but will be weighted towards the lower end since the words of higher probability are more likely to appear at random. It is possible, however, to pick a random subset of the n words that is evenly distributed in the interval.

The uniform random word generator discussed near the top of page 21 can be used to provide a large enough set of equally likely random words so that the desired number k of these will be pronounceable. The k words thus obtained can be assumed to be equally spaced in the interval [1, n] because they were arrived at in a manner totally independent of their probabilities. That is to say, the least probable of the k words has just as high a probability of appearing (using the uniform generator) as does the most probable word.

An approximate graph of p(x) was obtained by taking the 2653 pronounceable words used to estimate the value of n in (4) and ordering them according to probability.. Each word was assigned an index i,

$$i = 1, 2, \ldots, k, \qquad (8)$$

where i = 1 for the most probable of these words. For each word, the position on the x-axis was determined by

$$x(i) = \frac{in}{k + 1}. \qquad (9)$$

A plot of the probabilities of the 2653 words is shown in figure 4.

Application of the Distribution

Figure 4 is a complete profile of the word generator and it can be used to measure various quantities. For example, the total probability of the m most probable words is simply the area under the curve from x = 0 to x = m. The number of words that make up any given fraction of the population can also easily be measured.

Remember that in figure 4 the value of x is actually the "word

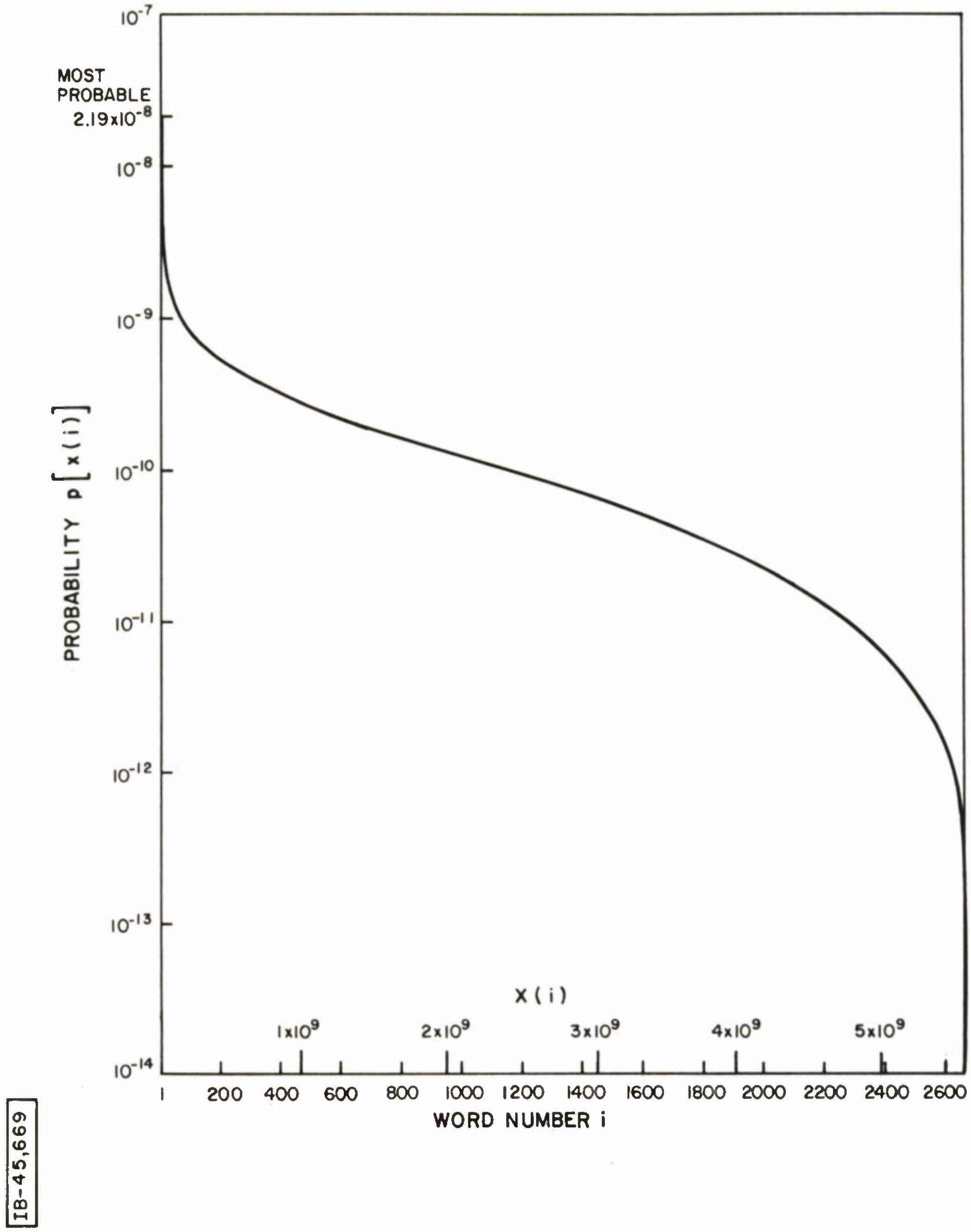


Figure 4. Distribution of Probabilities of 2653 Eight Letter Words

number" where x = 1 for the most probable word and x = n for the least probable. The value of x(i) for i = 1 in our sample of 2653 pronounceable words has a value of approximately 2,000,000 as calculated by (9). The most probable eight letter word out of the entire population is of course at x = 1. If we can believe for a moment that figure 4 is an exact representation, we can enlarge the extreme leftmost end of the curve where x is small as in figure 5, and extrapolate to the left of the point at x = 2,000,000 to double check the determination of the most probable word on page 25. Of course this extrapolation is not mathematically valid since there is no sound basis for assuming that the curve continues in any specific pattern. However, it does appear that extrapolation yields a value of the most probable word very close to that obtained by trial and error.

Another check on the distribution curve can be made by measuring the area under the curve. In order to approximately calculate this area, Simpson's rule was used where the first point (at x = 1) was assumed to be the most probable word as previously determined, and successive points are at intervals of n/2654. The area thus calculated came out to 1.006, only 0.6% off the expected value of 1.000.

Figures 4 and 5 apply only to a specific sample run for eight letter words. Appendix V presents similar data for six and ten letter words as a comparison. Of course, a different digram table or unit table would greatly change these distributions.

AN ALTERNATIVE METHOD

The main difficulty in the analysis of the random word generator lies in the complexity of the algorithm. The nature of the algorithm is such that a highly asymmetric distribution of probabilities of words results, with some words being many orders of magnitude more probable than others. The goal of the preceding analysis was to provide information as to the shape of the probability distribution curve so that the word generator's suitability for any particular application could be examined.

In its application as a Multics password generator, the results of figure 4 may indicate that the word generator is not suitable for passwords due to the high probability of the words at the leftmost end of the curve. Some installations may need passwords that have a probability less than 2.19×10^{-8}. It is possible to improve this probability by changing the digram and unit tables and the distribution of the units returned by the random_unit_ subroutine, but it is very difficult to anticipate the effect of any particular change on the probability distribution. Once the change is made in the tables, there is no easy way to determine what the most probable word actually is.

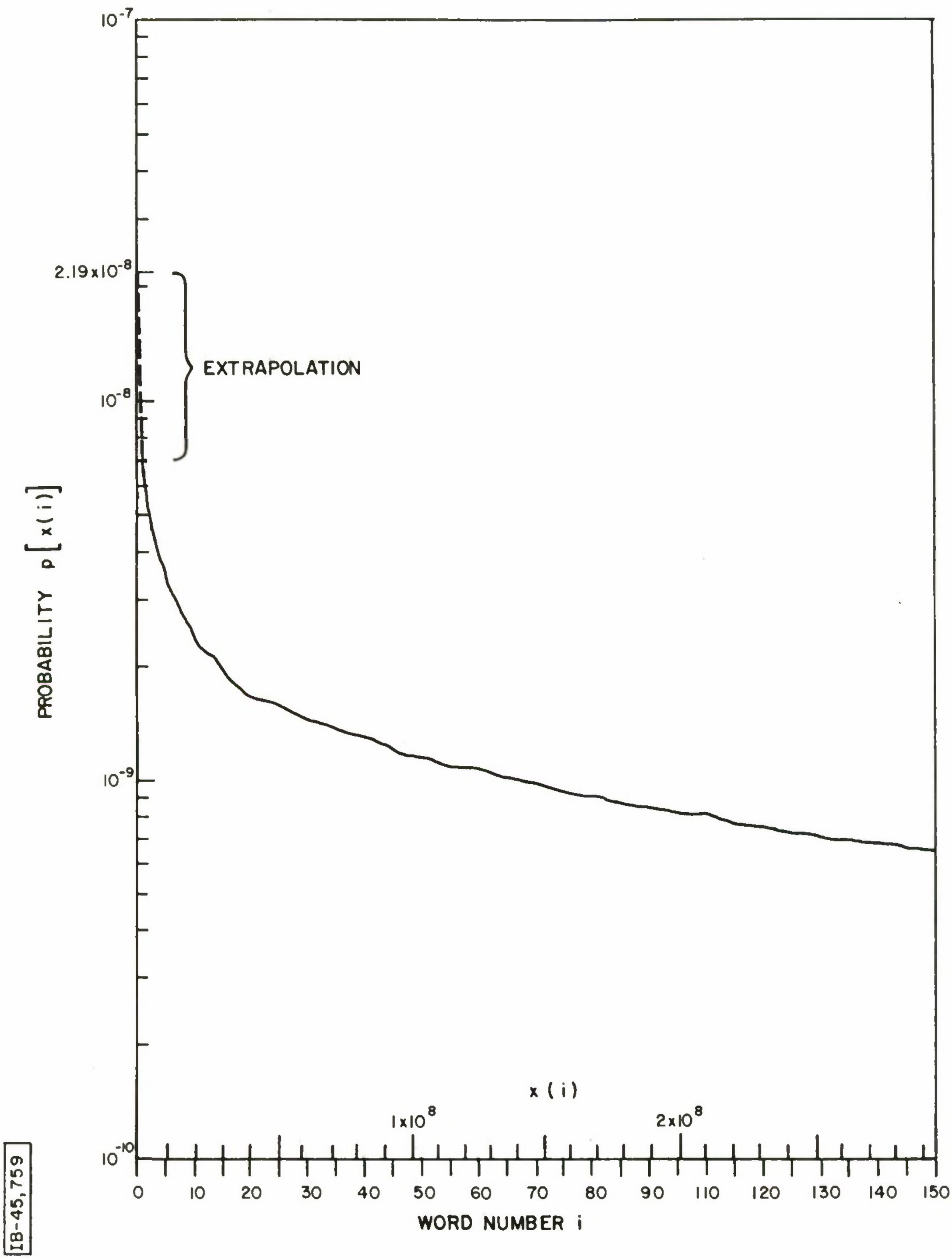


Figure 5. Enlarged Left Edge of Distribution

There is an alternative method, however, that can be employed without changing any of the tables, that yields a distribution that is much easier to determine.

In order to illustrate this alternative method, assume that we wish to improve the word generator's distribution so that no word is more probable than a word composed of six random letters. This example was chosen because the six random letter criterion for passwords is applied to several systems in use today. This criterion translates to a maximum probability of

$$\frac{1}{26^6} = 3.24 \times 10^{-9} \tag{10}$$

for any word. The most probable eight letter word shown on page 25 has a probability of 2.19×10^{-8} -- too high by a factor of seven, and Multics does not allow (nor would it be desirable considering the rememberability requirement) passwords longer than eight characters. Note, however, that the total number of pronounceable eight letter words from equation (4) is much greater than the total number of random six letter words. Thus, if there were some way to force the word generator to generate all n pronounceable words with equal probability, then the probability of any particular word would be 1/n and the analysis would be trivial.

By utilizing the random word generator in a slightly different way, at an additional cost in overhead, it is possible to force the probabilities of all words to be equal without changing the total number of words. Consider the method used to obtain the estimated value of n in equation (4). This value was obtained by generating words at random such that all eight letter words are equally likely, and computing the fraction of those that were pronounceable. A method for generating equally probable pronounceable words, then, is to generate equally likely random words and test them for pronounceability until an acceptable word turns up. All acceptable words thus generated are equally probable, and the randomness criterion is satisfied. Appendix VII contains the source code and the documentation for the two program modules that have been altered in order to optionally produce uniformly distributed random words. In addition, for those interested, a listing of the Multics encipher_ subroutine is included. This is the subroutine used to generate random numbers.

The question that arises is whether this "sampling" method is feasible considering the possible additional computer time required for testing and rejecting words. The answer depends on the fraction of words that are accepted, and whether it is more or less expensive to test a word for pronounceability rather than to generate it.

The fraction of pronounceable words was determined on page 21 for eight letter words using specific digram and unit tables. The value .02653 is an acceptance ratio of approximately 1 in 37. Thus we would expect, on the average, to make 37 tries before getting an acceptable word.

The time required to generate pronounceable eight letter words with the word generator is somewhat greater than the time required to generate a word at random and to test its pronounceability. This is because the word generator does not accept every unit supplied to it by random_unit in the word being formed, whereas all units of a pronounceable word to be tested are immediately accepted. The time required to test an unpronounceable word is usually less than the time required to test a pronounceable word because the whole word is rejected as soon as an illegal unit is encountered. If we expect, on the average, to make 37 tries to find a pronounceable word, we would expect the time required to do this to be no more 37 times that required to generate one pronounceable word. This is born out by empirical evidence indicating that the average time required to find a pronounceable eight letter word by trial and error is about 10 times that required to generate one pronounceable word. In Multics computer time, the figures are about .10 second and .01 second per word[11] respectively.

Since the number of pronounceable words is much greater (by a factor of 18) than that required to fulfill the six random letter criterion, it may be possible to significantly increase the pronounceability of words generated by modifying the tables to yield a smaller set of possible words. Consider the possible results if, for example, one could delete 17 out of every 18 words in the list in Appendix IV and save only the most pronounceable ones. Of course, by restricting the rules further, the "acceptance ratio" of 1 in 37 is decreased, thereby resulting in additional time spent finding a pronounceable word. A decrease by a factor of 18 will increase the average time of .10 seconds for finding a pronounceable eight letter word to several seconds. This may not be tolerable for some installations.

Thus, particularly when employing the alternative sampling method for generating pronounceable words, one must weigh the advantages of

---

[11]The time of .10 second for the sampling method was obtained by using a modified version of the random word generator that discards a partial word and starts over any time a unit is rejected. In this way extra (possibly unused) units are not generated as in the case when a whole random word is first created before testing.

pronounceability against overhead in computer time in determining what modifications to make to the tables. The advantage of using the sampling method is that the only statistical quantity to be determined, the total number of words, is fairly easily estimated. The effect of a change in the digram table on this estimate can be quickly examined.

SECTION V

CONCLUSION

EVALUATION

The random word generator described in this report has been successfully implemented and demonstrated on the Multics system. The list of 2000 random words contained in Appendix IV was shown to various people, and it became apparent that the degree to which words are considered pronounceable varies a great deal among individuals. Some people had many more complaints than others about particular words. In most cases, complaints were about words containing certain combinations of units that the individual did not consider to be "legal".

For the most part, combinations considered illegal could be directly eliminated using the rules of the digram table. In other much less frequent cases, eliminating the offending construct was either impossible or would result in eliminating many more constructs that should be legal. As mentioned earlier, there were many fewer complaints about the shorter words of four to six letters than about the longer ones.

The statistics discussed in Section IV and Appendix V, if presenting an unsatisfactory performance for a particular application, can be improved by modifying the digram table and unit table or by altering the manner in which the random word generator is utilized as discussed near the end of Section IV. For example, by eliminating all double-letter units, the most probable word will have a much lower probability than that indicated on page 25. This is done, however, at the cost of a reduction in the total number of different words. Or, at an increase in overhead, the alternative method for generating words discussed on page 29 can be employed. By modifying the rules in the digram table, particular statistical properties can be adjusted, but one must be aware of the interrelationships between the various properties and performance features before changing the tables to achieve a given result.

In conclusion it appears that the tables that are input to the word generator and the manner in which it is used could be tailored for almost any application, whether the main interest is in pronounceability or performance.

OTHER APPLICATIONS

Pronounceability and randomness were the primary goals of the word generator in its use for generating random words. However, the support program discussed in Section IV that gives the word generator a word to be tried, combined with the word generator's ability to divide a word into syllables, partially supports the facility of a general purpose word hyphenator for text processing. The random word generator can be given any word for hyphenation. If the word is accepted, the word will be returned hyphenated into syllables just as if it had been randomly generated. If rejected, the word is illegal according to random word generator rules and the tables. Of course, the hyphenation will not always be the "right" one according to the dictionary, but exceptions are apt to be few.[12] Perhaps some pre- or post-processing, combined with a list or dictionary of exceptional constructs, can yield a word hyphenator of general utility.

---

[12] An example of a common exception is found in words containing "tion", which is hyphenated "ti-on" as two syllables. This and many other problems could be avoided by defining new kinds of double letter units and changing the tables. Such modifications may make the word generator unsuitable for generating random words because many unpronounceable words may be considered legal. However, when used as a hyphenator the legality of a word is of no concern.

APPENDIX I

TABLES

The following pages list the unit table and digram table as described in Section III. The unit table appears first. Each entry in the unit table has the following format:

```
n cc wxyz
```

where:

n	is a unit number (1 to 34).
cc	is the unit (1 or 2 letters).
w	is 0 or 1, representing the value of "not_begin_syllable".
x	is the value of "no_final_split".
y	is "vowel".
z	is "alternate_vowel".

The digram table follows, with each entry of the following format:

```
abd-cc+ef
```

where:

a	is 0 or 1, which is the value of "begin".
b	is "not_begin".
d	is "break".
-	appears if "prefix" is set, otherwise blank.
cc	is a pair of units (2, 3, or 4 letters).
+	appears if "illegal_pair" is set. If it is "-", that means "suffix" is set. Otherwise it is blank.
e	is "end".
f	is "not_end".

The Unit Table

1	a	0010	10	j	0000	19	t	0000	28	ph	0000
2	b	0000	11	k	0000	20	u	0010	29	rh	0000
3	c	0000	12	l	0000	21	v	0000	30	sh	0000
4	d	0000	13	m	0000	22	w	0000	31	th	0000
5	e	0110	14	n	0000	23	x	1000	32	wh	0000
6	f	0000	15	o	0010	24	y	0001	33	qu	0000
7	g	0000	16	p	0000	25	z	0000	34	ck	1000
8	h	0000	17	r	0000	26	ch	0000			
9	i	0010	18	s	0000	27	gh	0000			

The Digram Table

000	aa	+00	011	bm	01	011	cz	01	000	ec	00	000	fo	00
000	ab	00	011	bn	01	000	cch	+00	000	ed	00	011	fp	01
000	ac	00	000	bo	00	000	cgh	+00	000	ee	00	100	fr	01
000	ad	00	011	bp	01	011	cph	01	000	ef	00	010	fs	00
000	ae	+00	100	br	01	000	crh	+00	000	eg	00	010	ft	00
000	af	00	010	bs	00	011	csh	01	011	eh	01	000	fu	00
000	ag	00	011	bt	01	011	cth	01	000	ei	01	011	fv	01
011	ah	01	000	bu	00	000	cwh	+00	000	ej	00	011	fw	01
000	ai	00	011	bv	01	010	cqu	-01	000	ek	00	000	fx	+00
000	aj	00	011	bw	01	000	cck	+00	000	el	00	010	fy	00
000	ak	00	000	bx	+00	000	da	00	000	em	00	011	fz	01
000	al	00	000	by	00	011	db	01	000	en	00	011	fch	01
000	am	00	011	bz	01	011	dc	01	001	eo	00	011	fgh	01
000	an	00	011	bch	01	010	dd	00	000	ep	00	011	fph	01
000	ao	+00	000	bgh	+00	000	de	00	000	er	00	000	frh	+00
000	ap	00	011	bph	01	011	df	01	000	es	00	011	fsh	01
000	ar	00	000	brh	+00	011	dg	01	000	et	00	011	fth	01
000	as	00	011	bsh	01	011	dh	01	000	eu	00	000	fwh	+00
000	at	00	011	bth	01	000	di	00	000	ev	00	011	fqu	01
000	au	00	000	bwh	+00	011	dj	01	000	ew	00	000	fck	+00
000	av	00	011	bqu	01	011	dk	01	000	ex	00	000	ga	00
000	aw	00	000	bck	+00	011	dl	01	000	ey	00	011	gb	01
000	ax	00	000	ca	00	011	dm	01	000	ez	00	011	gc	01
000	ay	00	011	cb	01	011	dn	01	000	ech	00	011	gd	01
000	az	00	011	cc	01	000	do	00	011	egh	01	000	ge	00
000	ach	00	011	cd	01	011	dp	01	000	eph	00	011	gf	01
000	agh	+00	000	ce	00	100	dr	01	000	erh	+00	010	gg	00
000	aph	00	011	cf	01	010	ds	10	000	esh	00	011	gh	01
000	arh	+00	011	cg	01	011	dt	01	000	eth	00	000	gi	00
000	ash	00	011	ch	01	000	du	00	000	ewh	+00	011	gj	01
000	ath	00	000	ci	00	011	dv	01	001	equ	01	000	gk	+00
000	awh	+00	011	cj	01	011	dw	01	000	eck	00	100	gl	-01
001	aqu	01	011	ck	01	000	dx	+00	000	fa	00	011	gm	01
000	ack	00	000	cl	-01	000	dy	00	011	fb	01	011	gn	01
000	ba	00	011	cm	01	011	dz	01	011	fc	01	000	go	00
011	bb	01	011	cn	01	011	dch	01	011	fd	01	011	gp	01
011	bc	01	000	co	00	011	dgh	01	000	fe	00	100	gr	01
011	bd	01	011	cp	01	011	dph	01	010	ff	00	010	gs	10
000	be	00	000	cr	01	000	drh	+00	011	fg	01	011	gt	01
011	bf	01	010	cs	10	010	dsh	01	011	fh	01	000	gu	00
011	bg	01	010-	ct	00	010-	dth	00	000	fi	00	011	gv	01
011	bh	01	000	cu	00	000	dwh	+00	011	fj	01	011	gw	01
000	bi	00	011	cv	01	011	dqu	01	011	fk	01	000	gx	+00
011	bj	01	011	cw	01	000	dck	+00	100	fl	-01	010	gy	00
011	bk	01	000	cx	+00	000	ea	00	011	fm	01	011	gz	01
100	bl	-01	000	cy	00	000	eb	00	011	fn	01	011	gch	01

000	ggh	+00	000	ig	00	011	jv	01	000	la	00	000	mo	00
011	gph	01	011	ih	01	011	jw	01	010-	lb	00	010	mp	00
000	grh	+00	000	ii	+00	000	jx	+00	011	lc	01	011	mr	01
010	gsh	00	000	ij	00	010	jy	00	010-	ld	00	010	ms	00
010	gth	00	000	ik	00	011	jz	01	000	le	00	010	mt	00
000	gwh	+00	000	il	00	011	jch	01	010-	lf	00	000	mu	00
011	gqu	01	000	im	00	011	jgh	01	010-	lg	00	011	mv	01
000	gck	+00	000	in	00	011	jph	01	011	lh	01	011	mw	01
000	ha	00	001	io	00	000	jrh	+00	000	li	00	000	mx	+00
011	hb	01	000	ip	00	011	jsh	01	010-	lj	00	000	my	00
011	hc	01	000	ir	00	011	jth	01	010-	lk	00	011	mz	01
011	hd	01	000	is	00	000	jwh	+00	010-	ll	00	010-	mch	00
000	he	00	000	it	00	011	jqu	01	010-	lm	00	011	mgh	01
011	hf	01	011	iu	00	000	jck	+00	011	ln	01	010	mph	00
011	hg	01	000	iv	00	000	ka	00	000	lo	00	000	mrh	+00
000	hh	+00	011	iw	01	011	kb	01	010-	lp	00	010	msh	00
000	hi	00	000	ix	00	011	kc	01	011	lr	01	010	mth	00
011	hj	01	011	iy	01	011	kd	01	010	ls	00	000	mwh	+00
011	hk	01	000	iz	00	000	ke	00	010-	lt	00	011	mqu	01
011	hl	01	000	ich	00	011	kf	01	000	lu	00	000	mck	+00
011	hm	01	010	igh	00	011	kg	01	010-	lv	00	000	na	00
011	hn	01	000	iph	00	011	kh	01	011	lw	01	011	nb	01
000	ho	00	000	irh	+00	000	ki	00	000	lx	+00	011	nc	01
011	hp	01	000	ish	00	011	kj	01	000	ly	00	010	nd	00
011	hr	01	000	ith	00	011	kk	01	011	lz	01	000	ne	00
011	hs	01	000	iwh	+00	000	kl	-01	010-	lch	00	011	nf	01
011	ht	01	001	iqu	01	011	km	01	011	lgh	01	010-	ng	00
000	hu	00	000	ick	00	100	kn	-01	010-	lph	00	011	nh	01
011	hv	01	000	ja	00	000	ko	00	000	lrh	+00	000	ni	00
011	hw	01	011	jb	01	011	kp	01	010-	lsh	00	011	nj	01
000	hx	+00	011	jc	01	000	kr	-01	010-	lth	00	010-	nk	00
000	hy	00	011	jd	01	010	ks	10	000	lwh	+00	011	nl	01
011	hz	01	000	je	00	011	kt	01	011	lqu	01	011	nm	01
011	hch	01	011	jf	01	000	ku	00	000	lck	+00	010	nn	00
011	hgh	01	000	jg	+00	011	kv	01	000	ma	00	000	no	00
011	hph	01	011	jh	01	011	kw	01	011	mb	01	011	np	01
000	hrh	+00	000	ji	00	000	kx	+00	011	mc	01	011	nr	01
011	hsh	01	000	jj	+00	010	ky	00	011	md	01	010	ns	00
011	hth	01	011	jk	01	011	kz	01	000	me	00	010	nt	00
000	hwh	+00	011	jl	01	011	kch	01	011	mf	01	000	nu	00
011	hqu	01	011	jm	01	011	kgh	01	011	mg	01	011	nv	01
000	hck	+00	011	jn	01	010-	kph	00	011	mh	01	011	nw	01
011	ia	00	000	jo	00	000	krh	+00	000	mi	00	000	nx	+00
000	ib	00	011	jp	01	010	ksh	00	011	mj	01	010	ny	00
000	ic	00	011	jr	01	011	kth	01	011	mk	01	011	nz	01
000	id	00	011	js	01	000	kwh	+00	011	ml	01	010-	nch	00
010	ie	00	011	jt	01	011	kqu	01	010	mm	00	011	ngh	01
000	if	00	000	ju	00	000	kck	+00	011	mn	01	010-	nph	00

000	nrh	+00	000	pi	00	000	rx	+00	011	tc	01	000	ur	00
010	nsh	00	011	pj	01	000	ry	00	011	td	01	000	us	00
010	nth	00	011	pk	01	010-	rz	00	000	te	00	000	ut	00
000	nwh	+00	000	pl	-01	010-	rch	00	011	tf	01	000	uu	+00
011	nqu	01	011	pm	01	011	rgh	01	011	tg	01	000	uv	00
010-	nck	00	011	pn	01	010-	rph	00	011	th	01	011	uw	01
000	oa	00	000	po	00	000	rrh	+00	000	ti	00	000	ux	00
000	ob	00	010-	pp	00	010-	rsh	00	011	tj	01	011	uy	01
000	oc	00	000	pr	01	010-	rth	00	011	tk	01	000	uz	00
000	od	00	010	ps	10	000	rwh	+00	011	tl	01	000	uch	00
000	oe	+00	010	pt	10	010-	rqu	01	011	tm	01	010-	ugh	00
000	of	00	000	pu	00	010-	rck	00	011	tn	01	000	uph	00
000	og	00	011	pv	01	000	sa	00	000	to	00	000	urh	+00
011	oh	01	011	pw	01	011	sb	01	011	tp	01	000	ush	00
000	oi	00	000	px	+00	000	sc	01	000	tr	01	000	uth	00
000	oj	00	000	py	00	011	sd	01	010	ts	10	000	uwh	+00
000	ok	00	011	pz	01	000	se	00	010-	tt	00	001	uqu	01
000	ol	00	011	pch	01	011	sf	01	000	tu	00	000	uck	00
000	om	00	011	pgh	01	011	sg	01	011	tv	01	000	va	00
000	on	00	011	pph	01	011	sh	01	100	tw	-01	011	vb	01
000	oo	00	000	prh	+00	000	si	00	000	tx	+00	011	vc	01
000	op	00	011	psh	01	011	sj	01	000	ty	00	011	vd	01
000	or	00	011	pth	01	000	sk	00	011	tz	01	000	ve	00
000	os	00	000	pwh	+00	100	sl	-01	010	tch	00	011	vf	01
000	ot	00	011	pqu	01	000	sm	-01	011	tgh	01	011	vg	01
000	ou	00	000	pck	+00	000-	sn	-01	010	tph	10	011	vh	01
000	ov	00	000	ra	00	000	so	00	000	trh	+00	000	vi	00
000	ow	00	010-	rb	00	000	sp	00	010	tsh	10	011	vj	01
000	ox	00	010-	rc	00	010	sr	01	011	tth	01	011	vk	01
000	oy	00	010-	rd	00	010-	ss	00	000	twh	+00	011	vl	01
000	oz	00	000	re	00	000	st	00	011	tqu	01	011	vm	01
000	och	00	010-	rf	00	000	su	00	000	tck	+00	011	vn	01
010	ogh	00	010-	rg	00	011	sv	01	011	ua	01	000	vo	00
000	oph	00	011	rh	01	100	sw	-01	000	ub	00	011	vp	01
000	orh	+00	000	ri	00	000	sx	+00	000	uc	00	011	vr	01
000	osh	00	010-	rj	00	000	sy	00	000	ud	00	011	vs	01
000	oth	00	010-	rk	00	011	sz	01	010	ue	00	011	vt	01
000	owh	+00	010-	rl	00	100	sch	-01	000	uf	00	000	vu	00
001	oqu	01	010-	rm	00	011	sgh	01	000	ug	00	011	vv	01
000	ock	00	010-	rn	00	011	sph	01	011	uh	01	011	vw	01
000	pa	00	000	ro	00	000	srh	+00	011	ui	01	000	vx	+00
011	pb	01	010-	rp	00	011	ssh	01	000	uj	00	010	vy	00
011	pc	01	010-	rr	00	011	sth	01	000	uk	00	011	vz	01
011	pd	01	010-	rs	00	000	swh	+00	000	ul	00	011	vch	01
000	pe	00	010-	rt	00	000	squ	-01	000	um	00	011	vgh	01
011	pf	01	000	ru	00	010	sck	00	000	un	00	011	vph	01
011	pg	01	010-	rv	00	000	ta	00	011	uo	00	000	vrh	+00
011	ph	01	011	rw	01	011	tb	01	000	up	00	011	vsh	01

011	vth	01	011	xk	01	010	yz	00	000	che	00	010-	ght	00
000	vwh	+00	011	xl	01	011	ych	01	011	chf	01	011-	ghu	01
011	vqu	01	011	xm	01	011	ygh	01	011	chg	01	011-	ghv	01
000	vck	+00	011	xn	01	011	yph	01	011	chh	01	011-	ghw	01
000	wa	00	010	xo	00	000	yrh	+00	000	chi	00	000	ghx	+00
010-	wb	00	011	xp	01	011	ysh	01	011	chj	01	011-	ghy	01
011	wc	01	011	xr	01	011	yth	01	011	chk	01	011-	ghz	01
010-	wd	10	011	xs	01	000	ywh	+00	011	chl	01	011-	ghch	01
000	we	00	011	xt	01	011	yqu	01	011	chm	01	000	ghgh	+00
010-	wf	00	010	xu	00	000	yck	+00	011	chn	01	011-	ghph	01
010-	wg	10	011	xv	01	000	za	00	000	cho	00	000	ghrh	+00
011	wh	01	011	xw	01	011	zb	01	011	chp	01	011-	ghsh	01
000	wi	00	000	xx	+00	011	zc	01	000	chr	01	011-	ghth	01
011	wj	01	010	xy	00	011	zd	01	011	chs	01	000	ghwh	+00
010-	wk	00	011	xz	01	000	ze	00	011	cht	01	011-	ghqu	01
010-	wl	-00	011	xch	01	011	zf	01	000	chu	00	000	ghck	+00
010-	wm	00	011	xgh	01	011	zg	01	011	chv	01	000	pha	00
010-	wn	00	011	xph	01	011	zh	01	010	chw	01	011	phb	01
000	wo	00	000	xrh	+00	000	zi	00	000	chx	+00	011	phc	01
010-	wp	00	011	xsh	01	011	zj	01	000	chy	00	011	phd	01
100	wr	-01	011	xth	01	011	zk	01	011	chz	01	000	phe	00
010-	ws	00	000	xwh	+00	011	zl	01	000	chch	+00	011	phf	01
010-	wt	00	011	xqu	01	011	zm	01	011	chgh	01	011	phg	01
000	wu	00	000	xck	+00	011	zn	01	011	chph	01	011	phh	01
010-	wv	00	000	ya	00	000	zo	00	000	chrh	+00	000	phi	00
011	ww	01	010	yb	00	011	zp	01	011	chsh	01	011	phj	01
010-	wx	00	010	yc	01	010	zr	01	011	chth	01	011	phk	01
000	wy	00	010	yd	00	011	zs	01	000	chwh	+00	100	phl	-01
010-	wz	00	000	ye	00	010	zt	00	011	chqu	01	011	phm	01
010	wch	00	010	yf	01	000	zu	00	000	chck	+00	011	phn	01
011	wgh	01	010	yg	00	011	zv	01	000	gha	00	000	pho	00
010	wph	00	011	yh	01	000	zw	-01	011-	ghb	01	011	php	01
000	wrh	+00	100	yi	01	000	zx	+00	011-	ghc	01	000	phr	01
010	wsh	00	010	yj	01	000	zy	00	011-	ghd	01	010	phs	00
010	wth	00	010	yk	00	010	zz	00	000	ghe	00	010	pht	00
000	wwh	+00	010	yl	01	011	zch	01	011-	ghf	01	000	phu	00
011	wqu	01	010	ym	00	011	zgh	01	011-	ghg	01	010	phv	01
010	wck	00	010	yn	00	011	zph	01	011-	ghh	01	010	phw	01
010	xa	00	000	yo	00	000	zrh	+00	100	ghi	01	000	phx	+00
011	xb	01	010	yp	00	011	zsh	01	011-	ghj	01	010	phy	00
011	xc	01	011	yr	01	011	zth	01	011-	ghk	01	011	phz	01
011	xd	01	010	ys	00	000	zwh	+00	011-	ghl	01	011	phch	01
010	xe	00	010	yt	00	011	zqu	01	011-	ghm	01	011	phgh	01
011	xf	01	000	yu	00	000	zck	+00	011-	ghn	01	000	phph	+00
011	xg	01	010	yv	01	000	cha	00	100	gho	01	000	phrh	+00
011	xh	01	011	yw	01	011	chb	01	011	ghp	01	011	phsh	01
010	xi	00	010	yx	00	011	chc	01	011-	ghr	01	011	phth	01
011	xj	01	000	yy	+00	011	chd	01	010-	ghs	00	000	phwh	+00

011	phqu	01
000	phck	+00
100	rha	01
000	rhb	+00
000	rhc	+00
000	rhd	+00
100	rhe	01
000	rhf	+00
000	rhg	+00
000	rhh	+00
100	rhi	01
000	rhj	+00
000	rhk	+00
000	rhl	+00
000	rhm	+00
000	rhn	+00
100	rho	01
000	rhp	+00
000	rhr	+00
000	rhs	+00
000	rht	+00
100	rhu	01
000	rhv	+00
000	rhw	+00
000	rhx	+00
100	rhy	00
000	rhz	+00
000	rhch	+00
000	rhgh	+00
000	rhph	+00
000	rhrh	+00
000	rhsh	+00
000	rhth	+00
000	rhwh	+00
000	rhqu	+00
000	rhck	+00
000	sha	00
011	shb	01
011	shc	01
011	shd	01
000	she	00
011	shf	01
011	shg	01
000	shh	+00
000	shi	00
011	shj	01
010	shk	00
100	shl	-01
100	shm	-01
100	shn	-01
000	sho	00
010	shp	00
100	shr	-01
011	shs	01
000	sht	-00
000	shu	00
011	shv	01
000	shw	-01
000	shx	+00
000	shy	00
011	shz	01
011	shch	01
011	shgh	01
011	shph	01
000	shrh	+00
000	shsh	+00
011	shth	01
000	shwh	+00
011	shqu	01
000	shck	+00
000	tha	00
011	thb	01
011	thc	01
011	thd	01
000	the	00
011	thf	01
011	thg	01
011	thh	01
000	thi	00
011	thj	01
011	thk	01
011	thl	01
011	thm	01
011	thn	01
000	tho	00
011	thp	01
000	thr	01
010	ths	10
011	tht	01
000	thu	00
011	thv	01
000	thw	-01
000	thx	+00
000	thy	00
011	thz	01
011	thch	01
011	thgh	01
011	thph	01
000	thrh	+00
011	thsh	01
000	thth	+00
000	thwh	+00
011	thqu	01
000	thck	+00
100	wha	01
000	whb	+00
000	whc	+00
000	whd	+00
100	whe	01
000	whf	+00
000	whg	+00
000	whh	+00
100	whi	01
000	whj	+00
000	whk	+00
000	whl	+00
000	whm	+00
000	whn	+00
100	who	01
000	whp	+00
000	whr	+00
000	whs	+00
000	wht	+00
000	whu	+00
000	whv	+00
000	whw	+00
000	whx	+00
100	why	00
000	whz	+00
000	whch	+00
000	whgh	+00
000	whph	+00
000	whrh	+00
000	whsh	+00
000	whth	+00
000	whwh	+00
000	whqu	+00
000	whck	+00
000	qua	00
000	qub	+00
000	quc	+00
000	qud	+00
000	que	00
000	quf	+00
000	qug	+00
000	quh	+00
000	qui	00
000	quj	+00
000	quk	+00
000	qul	+00
000	qum	+00
000	qun	+00
000	quo	00
000	qup	+00
000	qur	+00
000	qus	+00
000	qut	+00
000	quu	+00
000	quv	+00
000	quw	+00
000	qux	+00
000	quy	+00
000	quz	+00
000	quch	+00
000	qugh	+00
000	quph	+00
000	qurh	+00
000	qush	+00
000	quth	+00
000	quwh	+00
000	ququ	+00
000	quck	+00
011	cka	01
011	ckb	01
011	ckc	01
011	ckd	01
011	cke	01
011	ckf	01
011	ckg	01
011	ckh	01
011	cki	01
011	ckj	01
011	ckk	01
011	ckl	01
011	ckm	01
011	ckn	01
011	cko	01
011	ckp	01
011	ckr	01
010	cks	00
011	ckt	01
011	cku	01
011	ckv	01
011	ckw	01
000	ckx	+00
010	cky	00
011	ckz	01
011	ckch	01
011	ckgh	01
011	ckph	01
000	ckrh	+00
011	cksh	01
011	ckth	01
000	ckwh	+00
011	ckqu	01
000	ckck	+00

APPENDIX II

RANDOM WORD ALGORITHM

This appendix lists the algorithm described on page 15. Below is a description of the notation and the variables used to describe a state.

State Variables

Each loop through the algorithm produces new values of several state variables and possibly adds a unit to the random word being formed. The variables used to describe the state are defined as follows. "Binary" variables may have the value "true" or "false"; "decimal" variables have a number as their value.

vowel_found	Set when a vowel is found in a syllable (binary).
last_vowel_found	Value of vowel_found for previous unit in the random word (binary).
syllable_length	Number of units in syllable so far (decimal, initially 1).
index	Number of units in word so far (decimal, initially 1).
cons_count	Number of consecutive consonants (decimal).
nchars	Number of letters in word to be generated. Initially this is set to the length of the word (in letters) desired. This value is decremented each time a two-letter unit is generated so that the number of units (index) can be compared to nchars to determine if the end of the word has been reached.
unit(1), unit(2), ... unit(index)	Unit(i) represents the i'th unit in the word. Unit(index) is the current unit.

In addition to the state variables, two variables are defined for use only internal to the algorithm. They are used to simplify the notation.

v A binary variable which is set when the unit just generated is a vowel (or an alternate_vowel to be treated as a vowel).

b A binary variable which gets set when a "break" pair (as defined on page 12) is encountered, or when the previous pair was a "suffix" pair and the current unit is not a vowel.

Notation

The following names are used in the algorithm for the eight flags in the digram table:

begin,
not_begin,
end,
not_end,
break,
prefix,
suffix,
and illegal_pair.

If one of these names appears with a value in parentheses immediately following it, as "break(i)", the reference is to the "break" flag for the pair of units [unit(i-1), unit(i)]. If no value appears, the reference is to the pair [unit(index-1), unit(index)] -- that is, the reference is to the last two units.

The following names are used for the flags in the unit table:

no_final_split,
not_begin_syllable,
vowel,
alternate_vowel,
and double_letter.

The "double_letter" flag was not explicitly mentioned in the discussion earlier. It is set for units consisting of more than one letter. A value in parentheses following the name, as in vowel(i), refers to the vowel flag for unit(i). If no value appears the reference is to the flag for unit(index), or the current unit.

Three procedures are referred to: "random_unit" is a user-supplied procedure to generate a random unit; "random_vowel" is a user supplied procedure to generate a random vowel unit; and "done" is an internal procedure appearing near the end of the algorithm.

Algorithm

The algorithm is shown in the following pages. The text of the algorithm is essentially the same as the main body of the random_word_ subroutine appearing in Appendix III. The algorithm is shown here only for completeness -- it can stand alone and does not depend on any support subroutines. Since the random_word_ subroutine as shown in Appendix III is well documented and commented, no comments are supplied below. A correspondence between this algorithm and the random_word_ subroutine can easily be made.

The RANDOM_WORD procedure has an internal procedure DONE listed near the end. The extents of the if-then-else clauses are indicated by indentation.

RANDOM WORD ALGORITHM

```
BEGIN procedure RANDOM_WORD

retry:
  if syllable_length = 1
  then
    if index = nchars
    then call random_vowel
    else call random_unit
    if (index ≠ 1 & illegal_pair)
    then go to retry
    syllable_length = 2
    if vowel | alternate_vowel
    then cons_count = 0
    else cons_count = 1
    last_vowel_found = 0
    if double_letter
    then
      if index = nchars | (index = nchars-1 & ^vowel)
      then go to retry
    else nchars = nchars - 1
  else
    if (syllable_length = 2 & ^vowel_found & index = nchars) |
       (^vowel_found | not_end(index-1)) & suffix(index-1)
    then call random_vowel
    else call random_unit
    if illegal_pair |
       (unit(index)=unit(index-1)=unit(index-2) & index>2)
    then go to retry
    if double_letter & index = nchars
    then goto retry
    else nchars = nchars - 1
    if vowel | (alternate_vowel & ^vowel(index-1))
    then v = 1
    else v = 0
    if syllable_length > 2 & suffix(index-1) & ^v
    then b = 1
    else b = break
    if syllable_length = 2 & not_begin
    then go to no_good
    if vowel_found
    then
      if cons_count ≠ 0
      then
        if begin
        then
```

```
          if syllable_length ≠ 3 & not_end(index-2)
          then
            if not_end(index-1)
            then go to no_good
            else call done(v,2)
          else call done(v,3)
        else
          if not_begin
          then
            if b
            then
              if not_end(index-1)
              then go to no_good
              else call done(v,2)
            else
              if v
              then
                if not_end(index-1) ¦ not_begin_syllable
                then go to no_good
                else call done(1,2)
              else call done(^end,end)
          else
            if v
            then
              if not_end(index-2) ¦ syllable_length ^= 3
              then
                if not_end(index-1)
                then
                  if cons_count > 1
                  then
                    if not_end(index-3) ¦
                       not_begin(index-1)
                    then go to no_good
                    else call done(1,4)
                  else go to no_good
                else call done(1,2)
              else call done(1,3)
            else call done(1,0)
      else
        if v & vowel(index-2) & index > 2
        then go to no_good
        else
          if end
          then call done(0,1)
          else
```

```
          if begin
          then
            if last_vowel_found
            then
              if v
              then
                if syllable_length = 3
                then
                  if alternate_vowel(index-2)
                  then go to no_good
                  else call done(1,3)
                else
                  if not_end(index-2)
                  then go to no_good
                  else call done(1,3)
              else
                if syllable_length = 3
                then
                  if alternate_vowel(index-2)
                  then call done(1,3)
                  else go to no_good
                else
                  if not_end(index-2)
                  then
                    if not_end(index-1)
                    then go to no_good
                    else call done(0,2)
                  else call done(1,3)
            else
              if not_end(index-1) & syllable_length > 2
              then go to no_good
              else call done(v,2)
          else
            if b
            then
              if not_end(index-1) & syllable_length > 2
              then go to no_good
              else call done(v,2)
            else call done(1,0)
  else
    if b
    then go to no_good
    else
      if end
      then
```

```
        if v
        then call done(0,1)
        else go to no_good
      else
        if v
        then
          if begin & syllable_length > 2
          then go to no_good
          else call done(1,0)
        else
          if begin
          then
            if syllable_length > 2
            then go to no_good
            else call done(0,3)
          else call done(0,0)
  go to retry
no_good:
  if double_letter then nchars = nchars + 1
  go to retry


BEGIN procedure DONE:
called with 2 arguments:  call done(vf,sl)

if sl ≠ 2 & syllable_length ≠ 2 & prefix & ^vowel(index-2)
then
  if vowel_found
  then
    if not_end(index-1)
    then go to no_good
    else
      call done(0,2)
      return
  else go to no_good
else
  if sl ≠ 1 & index = nchars & (not_end | vf = 0)
  then go to no_good
if index = nchars & no_final_split & sl ≠ 1 & ^vowel(index-1)
then
  if ^vowel_found | not_end(index-1) | syllable_length < 3
  then go to no_good
  else sl = 0
if v | sl = 1
then cons_count = 0
```

```
else
  if sl = 0
  then cons_count = cons_count + 1
  else cons_count = min(syllable_length-1, cons_count+1)
if sl = 0
then syllable_length = syllable_length + 1
else syllable_length = sl
if syllable_length > 3
then last_vowel_found = vowel_found
else last_vowel_found = 0
vowel_found = vf
return

END procedure DONE
END procedure RANDOM_WORD
```

APPENDIX III

SOURCE CODE

The following pages contain source program listings of every command and subroutine applicable to the random word generator. The listings are in alphabetical order by program name. Below is a list of the programs with a brief description of the function of each. Complete documentation of the usage of each of these may be found in Appendix VI.

In Multics there is a distinction between commands and subroutines. Commands are callable by the user from his terminal, while subroutines can only be called by other subroutines or commands. The naming convention for programs specifies that subroutine names end with a trailing underscore, while command names should not. Generally the commands described below are user interfaces to specific subroutines.

columns	Prints lists of random words in columns.
convert_word_	Subroutine that converts an array of unit numbers to characters. Used by generate_word_.
convert_word_char_	Subroutine that inserts hyphens into a given word and formats the word for printing. Used by hyphen_test.
digram_table_compiler	Compiler for digram table.
generate_word_	Standard interface for user-written programs to generate a random word. Used by generate_words.
generate_words	Command to generate a list of random words.
get_line_length	Command and subroutine to return the line length of the output medium for the purposes of formatting output. Used by columns and digram_table_compiler. Also called get_line_length_.
hyphen_test	Command to hyphenate a supplied word using the rules of the random word generator. Also used to calculate the probability of a given word.

hyphenate_	Subroutine interface to perform same function as hyphen_test. Used by hyphen_test.
random_unit_	Standard subroutine for generating a random unit. Referenced by generate_word_ and hyphenate_ and called by random_word_.
random_unit_stat_	Assembly language program containing definitions of external static variables used by random_unit_.
random_word_	Subroutine implementing the word generator. Called by generate_word_ and hyphenate_.
read_table_	Subroutine that compiles the digram table. This is an internal interface only called by digram_table_compiler.

COMPILATION LISTING OF SEGMENT columns
Compiled by: Experimental PL/I Compiler of Tuesday, March 25, 1975 at 14:19
Compiled on: 04/01/75 1720.7 edt Tue
Options: check source

```
/* This program takes a segment consisting of ASCII lines
   and rearranges them so that they may be printed out in
   columns on a page, reading top to bottom in each column.

   To call, enter command:

   columns seg -option-

1) seg        is the pathname of the segment to reformat.

2) option is one or more of the following

    -sm, -segment         specifies that output will be put in the segment
                          named seg.columns and can later be dprinted.
                          Default sends output to terminal.

   -line_length nn        is a number from 1 to 132 that specifies the length of
   -ll nn                 the output line.  If missing, the defaults are:

                          output to terminal -- length of terminal line
                          output to segment  -- 132
                          output via file_output -- 132

    -no_pagination, -npgn specifies that the page length is assumed
                          to be infinite, and no page skips occur.
                          In this case each columns reads all the way
                          down to the bottom of the printout.

    -page_length nn       page length to use, rather than the default of 60.
    -pl nn

    -adjust, -ad          fill in extra blank space in between columns.
*/

col: columns: procedure;
dcl cv_dec_check_ entry(char(*), fixed bin) returns(fixed bin(35));
dcl return_code fixed bin;
dcl total_length fixed bin; /* length of each line in output segment, which may include padding at end of last column */
dcl length_wanted fixed bin; /*output segment line length minus padding after newlines */
dcl last_line fixed bin; /* line number in output of last line on current page */
dcl get_line_length_ entry returns (fixed bin);
dcl status bit(72) aligned;
dcl gmode char(128) init((128)" ");
dcl max_length fixed bin init(0); /* max real input line length plus padding if adjust is on */
dcl real_max_length fixed bin; /* max input line length not counting padding but counting newlines. */
dcl last fixed bin; /* number of output lines to put on current output page */
dcl charcnt fixed bin(35) init(1);
dcl colcnt fixed bin;
dcl linecnt fixed bin;
dcl newline char(1) static init("
");
dcl newline_tab char(2) static init("
        ");
dcl string char(132) varying;
dcl real_length fixed bin;
```

```
dcl get_wdir_ entry returns(char(168) aligned);
dcl ll fixed bin init(0);
dcl (p, input_seg_ptr, scratch_seg_ptr init(null), output_seg_ptr) ptr;
dcl total_output_lines fixed bin;
dcl total_lines fixed bin; /* number of lines in input segment */
dcl stop_switch bit(1) static init("0"b);
dcl (com_err_, ioa_$ioa_stream) entry options(variable);
dcl ios_$write_ptr entry (ptr, fixed bin, fixed bin);
dcl cu_$arg_ptr entry(fixed,ptr,fixed,fixed bin(35));
dcl segment_switch bit(1) init("0"b);
dcl nlines fixed;
dcl page_length fixed init (60);
dcl interval fixed bin; /* number of output lines per page of output */
dcl arg_count fixed bin;
dcl nchars fixed(35);
dcl (bitcount, bc) fixed bin(24);
dcl ncolumns fixed bin; /* number of columns on output */
dcl null builtin;
dcl arg_length fixed bin;
dcl nn fixed bin;
dcl i fixed bin;
dcl j fixed bin;
dcl l fixed bin;
dcl arg char(arg_length) based(p) unaligned;
dcl code fixed bin(35) init(0);
dcl (error_table_$badopt, error_table_$inconsistent,
     error_table_$zero_length_seg) ext fixed bin(35);
dcl hcs_$make_seg entry (char(*), char(*), char(*), fixed bin(5), ptr,
                  fixed bin(35));
dcl hcs_$truncate_seg entry (ptr, fixed bin, fixed bin(35));
dcl hcs_$set_bc_seg entry (ptr, fixed bin(24), fixed bin(35));
dcl term_$seg_ptr entry (ptr, fixed bin(35));
dcl expand_path_ entry (ptr, fixed bin, ptr, ptr, fixed bin(35));
dcl dirname char(168) aligned;
dcl ename char(32) aligned;
dcl hcs_$initiate_count entry (char(*), char(*), char(*), fixed bin(24),
            fixed bin(2), ptr, fixed bin(35));
dcl hcs_$delentry_seg entry (ptr, fixed bin(35));
dcl adjust bit(1) aligned init("0"b);
dcl lines char(nchars) based(input_seg_ptr);
dcl output_file(total_output_lines,ncolumns) char(max_length) based (scratch_seg_ptr);
dcl whole_thing char(1048575) based (output_seg_ptr);
dcl output_lines(total_output_lines) char(total_length) based(scratch_seg_ptr);
dcl newlines char(63) static init((63)"
");

/* get pointer to segment to read */

    call cu_$arg_ptr (1, p, arg_length, code);  /* 1st argument is segment name */
    if code ^= 0 then do;
      call com_err_ (code, "columns",
"Usage is: columns path -args-
args may be: -segment -line_length n -no_pagination -page_length n -adjust");
      return;
      end;
    call expand_path_ (p, arg_length, addr(dirname), addr(ename), code);
    if code ^= 0 then call ugly (code, arg);
    call hcs_$initiate_count ((dirname), (ename), "", bitcount, 0, input_seg_ptr, code);
```

```
if input_seg_ptr = null then call ugly (code, before(dirname," ") || ">" || ename);
if bitcount = 0 then call ugly (error_table_$zero_length_seg, arg);
nchars = ceil(bitcount/9); /* total number of characters in segment */

/* get rest of arguments */

code = 0;
do arg_count = 2 by 1 while(code=0);
 call cu_$arg_ptr(arg_count, p, arg_length, code);    /* get argument */
 if code = 0
 then
   if arg = "-segment" | arg = "-sm"
   then segment_switch = "1"b;
   else
    if arg = "-ll" | arg = "-line_length" then do;
     arg_count = arg_count + 1;
     call cu_$arg_ptr (arg_count, p, arg_length, code);
     if code ^= 0 then call ugly (code, "");
     i = (cv_dec_check_(arg, return_code));
     if return_code ^= 0 | i <= 0 | i > 132
     then call ugly (0, "bad line length. " || arg);
     if ll ^= 0
     then call ugly (error_table_$inconsistent, arg);
     ll = i;
     end;
    else
     if arg = "-npgn" | arg = "-no_pagination"
     then page_length = 0;
     else
       if arg = "-page_length" | arg = "-pl" then do;
         arg_count = arg_count + 1;
         call cu_$arg_ptr (arg_count, p, arg_length, code);
         if code ^= 0 then call ugly (code, "Length of page.");
         page_length = cv_dec_check_ (arg, return_code);
         if return_code ^= 0 | page_length <= 0
         then call ugly (0, "Bad page length. " || arg);
         end;
       else
         if arg = "-adjust" | arg = "-ad" then do;
            adjust = "1"b;
            end;
         else call ugly (error_table_$badopt, arg);
end;

if ll = 0       /* line length not specified, get from user_output */
then
 if segment_switch
 then ll = 132;
 else ll = get_line_length_();

call hcs_$make_seg ("", "columns_temp", "", 01010b, scratch_seg_ptr, code); /* output will go here */
if scratch_seg_ptr = null then call ugly (code, "columns_temp in process directory");

/* Now that we have pointer to temporary segment, this stuff can be set */

if substr(lines, nchars, 1) ^= newline /* This may cause problems if not checked */
then call ugly (0, "Last line in segment does not end in new line character");
```

```
/* find number of lines in segment and longest line */

do nlines = 1 by 1 while(charcnt<nchars);

 /* we expect that most lines won't contain tabs in them, therefore
    the code below avoids calling tab_expander if no tabs are in the line.
    It takes advantage of the fact that searching for one of two characters
    is no more expensive than searching for one. */
 i = search (substr (lines, charcnt), newline_tab); /* look for newline or tab */
 if substr (lines, charcnt+i-1, 1) = newline then do; /* was it a newline? */
    real_length = i;   /* it was a newline, we have real length of line */
    end;
 else do;              /* the line must have contained a tab */
    i = index (substr (lines, charcnt), newline); /* find newline */
    real_length = length (tab_expander (substr (lines, charcnt, i)));
    end;
 if real_length > ll    /* new line not found within ll characters */
 then
  do;
   if max_length < ll     /* is this the first time we found a line too big? */
   then call ioa_$ioa_stream ("error_output", "Line in segment is longer than ^d characters", ll);
   call ioa_$ioa_stream ("error_output", "line ^d: " || tab_expander(substr(lines, charcnt, i - 1)), nlines);
  end;
 charcnt = charcnt + i;
 max_length = max(max_length, real_length);
end;

if max_length > ll then call ugly (0, "error in input segment");

/* set some initial variables */

ncolumns = (ll+1)/max_length; /* maximum number of columns that will fit */
real_max_length = max_length;
if adjust then do; /* try to put in extra padding */
    /* max_length now becomes length of line in each columns plus padding
       but the last line in each column has no padding, so its length is real_max_length */
    max_length = max_length + (ll + 1 - max_length*ncolumns)/(ncolumns-1);
    end;
linecnt = 1;
total_lines = nlines - 1;
total_output_lines = (total_lines + ncolumns - 1)/ncolumns; /* number of lines in output segment */
nlines, charcnt = 1;
if page_length = 0 then interval = total_output_lines;
                   else interval = page_length;

/* We are working on the output segment one page at a time.  The variable i gets incremented
   by the number of input lines (interval) to be put on each output page.  The input lines
   are processed in order -- we start at the top of the leftmost column, inserting input lines,
   and proceding down that column.  When reching the bottom, we go to the top of the next
   column.  The very last column on the last page may have blanks ending it. */

do i = 1 to total_output_lines by interval;
 if page_length = 0
 then last = total_output_lines;      /* number of lines on page */
 else last = min(page_length, total_output_lines - i + 1);
 last_line = i + last - 1; /* This forces columns on last page to be shorter so that minimum output lines are used */
 do colcnt = 1 to ncolumns;          /* work on one column at a time */
  do linecnt = i to last_line;       /* go down each column, inserting lines from input segment */
```

```
    if nlines > total_lines
    then output_file(linecnt,colcnt) = "";   /* all blanks if no more input lines */
    else do;
      j = search (substr (lines, charcnt), newline_tab);
      if substr (lines, charcnt + j - 1, 1) = newline
      then output_file(linecnt, colcnt) = substr (lines, charcnt, j - 1);
      else output_file(linecnt, colcnt) = tab_expander (substr (lines, charcnt, index (substr (lines, charcnt), newline) - 1));
      charcnt = charcnt + index (substr (lines, charcnt), newline);
      nlines = nlines + 1;
    end;
    if colcnt = ncolumns
    then substr(output_file(linecnt,colcnt),real_max_length) = newline;
  end;
 end;
end;

/* we can get rid of input segment */

call term_$seg_ptr (input_seg_ptr, code);

/* if output is to segment, move to segment and set bit count */

total_length = max_length*ncolumns;
length_wanted = total_length - (max_length - real_max_length);
if segment_switch then do;
  call hcs_$make_seg (get_wdir_(), before(ename, " ") || ".columns", "", 01010b, output_seg_ptr, code);
  if output_seg_ptr = null then goto output_seg_ptr_error;
  call hcs_$truncate_seg (output_seg_ptr, 0, code);
  if code ^= 0 then goto output_seg_ptr_error;
  nn = 1; /* move segment, page_length lines at a time, with formfeeds in between */
  do i = 1 by interval to total_output_lines;
    do linecnt = i to min (i + interval - 1, total_output_lines);
      /* if adjust is on, this next line may truncate extra spaces at end of each line (but the newline is in the right place) */
      substr (whole_thing, nn, length_wanted) = output_lines (linecnt);
      nn = nn + length_wanted;
    end;
    if linecnt < total_output_lines then do;
      substr (whole_thing, nn, 1) = "
```

```
"; /* insert formfeed */
          nn = nn + 1;
          end;
      end;
      call hcs_$set_bc_seg (output_seg_ptr, (nn-1)*9, code);
      if code ^= 0 then goto output_seg_ptr_error;
      call term_$seg_ptr (output_seg_ptr, code);
      if code ^= 0
      then

output_seg_ptr_error:

       call ugly (code, before(ename," ") || ".columns");
       end;
    else do;
      if page_length ^= 0
      then call ios_$write_ptr (addr(newlines), 0, 3);
      do i = 1 to total_output_lines;
        call ios_$write_ptr (scratch_seg_ptr, total_length*(i-1), length_wanted); /* write out the line */
        if page_length ^= 0 then if mod(i,page_length) = 0
        then call ios_$write_ptr (addr(newlines), 0, 66 - page_length);
      end;
      if page_length = 0
      then call ios_$write_ptr (addr(newlines), 0, 1);
      else call ios_$write_ptr (addr(newlines), 0, 63 - mod(i, page_length));
     end;

    /* enter here to clean up when finished */

terminate:
    if scratch_seg_ptr ^= null then call hcs_$delentry_seg (scratch_seg_ptr, code);

/* routine to turn tabs into spaces */

tab_expander:proc(string) returns (char(*)) reducible;
  dcl tab char(1) init("      ") static;
  dcl string char(*);
  dcl tab_position fixed bin;

  tab_position = index (string, tab);
  if tab_position ^= 0
  then
   return (tab_expander (
              substr(string, 1, tab_position - 1) ||
              substr((10)" ", mod(tab_position - 1, 10) + 1) ||
              substr(string, tab_position + 1)));
  else do;
     return(string);
     end;
end;

/* print an error message and terminate */

ugly: proc (code, message);
 dcl code fixed bin(35);
 dcl message char(*);
 call com_err_ (code, "columns", message);
 goto terminate;    /* nonlocal goto to finish up */
```

```
end;

end;
```

```
COMPILATION LISTING OF SEGMENT convert_word_
Compiled by: Experimental PL/I Compiler of Tuesday, March 25, 1975 at 14:19
Compiled on: 04/01/75  1721.0 edt Tue
    Options: check source

convert_word_: proc (word, hyphens, word_length, expanded_word, hyphenated_word);
 dcl word(0:*) fixed bin;
 dcl hyphens(0:*) bit(1) aligned;
 dcl word_length fixed bin;
 dcl expanded_word char(*);
 dcl hyphenated_word char(*);
 dcl i fixed bin;
 dcl no_hyphens bit(1) aligned;
 dcl word_index fixed bin init(1);
 dcl hyphenated_index fixed bin init(1);


/********** include file digram_structure.incl.pl1 *********/

dcl digrams$digrams external;
dcl digrams$n_units fixed bin external;
dcl digrams$letters external;
dcl digrams$rules external;

/* This array contains information about all possible pairs of units */

dcl 1 digrams(n_units, n_units) based (addr(digrams$digrams)),
      2 begin bit(1),       /* on if this pair must begin syllable */
      2 not_begin bit(1),   /* on if this pair must not begin */
      2 end bit(1),         /* on if this pair must end syllable */
      2 not_end bit(1),     /* on if this pair must not end */
      2 break bit(1),       /* on if this pair is a break pair */
      2 prefix bit(1),      /* on if vowel must precede this pair in same syllable */
      2 suffix bit(1),      /* on if vowel must follow this pair in same syllable */
      2 illegal_pair bit(1),/* on if this pair may not appear */
      2 pad bit(1);         /* this makes 9 bits/entry */

/* This array contains left justified 1 or 2-letter pairs representing each unit */

dcl letters(0:n_units) char(2) aligned based (addr(digrams$letters));

/* This is the same as letters, but allows reference to individual characters */

dcl 1 letters_split(0:n_units) based (addr(digrams$letters)),
      2 first char(1),
      2 second char(1),
      2 pad char(2);

/* This array has rules for each unit */

dcl 1 rules(n_units) aligned based (addr(digrams$rules)),
      2 no_final_split bit(1),    /* can't be the only vowel in last syllable */
      2 not_begin_syllable bit(1),/* can't begin a syllable */
      2 vowel bit(1),             /* this is a vowel */
      2 alternate_vowel bit(1);   /* this is an alternate vowel, (i.e., "y") */

dcl n_units defined digrams$n_units fixed bin;

/********** end include file digram_structure.incl.pl1 *************/
```

```

 no_hyphens = ""b;

convert_word:
 do i = 1 to word_length;
  if substr(letters(word(i)),2,1) = " "
  then
   do;
    substr(expanded_word, word_index, 1) = substr(letters(word(i)),1,1);
    if ^no_hyphens then
     do;
      substr(hyphenated_word, hyphenated_index, 1) = substr(letters(word(i)),1,1);
      hyphenated_index = hyphenated_index + 1;
     end;
    word_index = word_index + 1;
   end;
  else
   do;
    substr(expanded_word, word_index, 2) = letters(word(i));
    if ^no_hyphens then
     do;
      substr(hyphenated_word, hyphenated_index, 2) = letters(word(i));
      hyphenated_index = hyphenated_index + 2;
     end;
    word_index = word_index + 2;
   end;
  if ^no_hyphens
  then
   if hyphens(i)
   then
    do;
     substr(hyphenated_word, hyphenated_index, 1) = "-";
     hyphenated_index = hyphenated_index + 1;
    end;
 end;

 if ^no_hyphens then if hyphenated_index <= length(hyphenated_word) then substr(hyphenated_word, hyphenated_index) = "";
 if word_index <= length(expanded_word) then substr(expanded_word, word_index) = "";    /* fill out with spaces */
 return;

convert_word_$no_hyphens: entry (word, word_length, expanded_word);
 no_hyphens = "1"b;
 goto convert_word;

end;
```

```
COMPILATION LISTING OF SEGMENT convert_word_char_
Compiled by: Experimental PL/I Compiler of Tuesday, March 25, 1975 at 14:19
Compiled on: 04/01/75  1721.1 edt Tue
    Options: check source

convert_word_char_: proc (word, hyphens, last, result);
 dcl i fixed bin;
 dcl result char(*) varying;
 dcl word char(*);
 dcl hyphens(*) bit(1) aligned;
 dcl last fixed bin;
 if last < 0
 then
  do;
   result = word || "**";
   return;
  end;
 result = "";
 do i = 0 to length(word);
  if i ^= 0
  then
   do;
    result = result || substr(word,i,1);
    if hyphens(i) then result = result || "-";
   end;
  if last > 0 & last = i+1
  then result = result || "*";
 end;
end;
```

```
COMPILATION LISTING OF SEGMENT digram_table_compiler
Compiled by: Experimental PL/I Compiler of Tuesday, March 25, 1975 at 14:19
Compiled on: 04/01/75  1721.1 edt Tue
    Options: check source

/* This command compiles a source segment containing digrams for
the word generator and puts the compiled output in a segment
named "digrams".

Usage: digram_table_compiler pathname -option-

Where: option may be one of the following:

  -ls, -list  Lists the output on the terminal after compilation.
  -ls, n, -list n  Lists as above, but in n columns.

Usage: print_digram_table -n-

  n    Lists the output in n columns.  Allow 14 positions for each column.
       This call assuumes that the digrams segment already exists
       and has been compiled correctly.
*/

digram_table_compiler: procedure;
dcl (start, size) fixed bin;
dcl nrows fixed bin;
dcl cu_$arg_ptr entry(fixed bin,ptr,fixed bin,fixed bin(35));
dcl code fixed bin(35);
dcl codex fixed bin;
dcl cv_dec_check_ entry (char(*), fixed bin) returns (fixed bin(35));
dcl hcs_$truncate_seg entry (ptr, fixed bin, fixed bin(35));
dcl hcs_$terminate_name entry (char(*), fixed bin(35));
dcl com_err_ entry options(variable);
dcl (error_table_$noarg, error_table_$badopt) external fixed bin(35);
dcl get_line_length_ entry returns (fixed bin);
dcl read_table_ entry (ptr, fixed bin(24)) returns(bit(1));
dcl compile bit(1);
dcl who char(25) varying;
dcl list bit(1);
dcl segptr ptr static init(null);
dcl dirname char(168) aligned;
dcl ename char(32) aligned;
dcl ename_length fixed bin;
dcl null builtin;
dcl arg char(length) based (pp);
dcl hcs_$initiate_count entry (char(*), char(*), char(*),
     fixed bin(24), fixed bin(2), ptr, fixed bin(35));
dcl hcs_$terminate_noname entry (ptr, fixed bin(35));
dcl expand_path_ entry (ptr, fixed bin, ptr, ptr, fixed bin(35));
dcl bc fixed bin(24);
dcl i fixed;

/********** include file digram_structure.incl.pl1 ********/

dcl digrams$digrams external;
dcl digrams$n_units fixed bin external;
dcl digrams$letters external;
dcl digrams$rules external;

/* This array contains information about all possible pairs of units */
```

```

dcl 1 digrams(n_units, n_units) based (addr(digrams$digrams)),
      2 begin bit(1),         /* on if this pair must begin syllable */
      2 not_begin bit(1),     /* on if this pair must not begin */
      2 end bit(1),           /* on if this pair must end syllable */
      2 not_end bit(1),       /* on if this pair must not end */
      2 break bit(1),         /* on if this pair is a break pair */
      2 prefix bit(1),        /* on if vowel must precede this pair in same syllable */
      2 suffix bit(1),        /* on if vowel must follow this pair in same syllable */
      2 illegal_pair bit(1),/* on if this pair may not appear */
      2 pad bit(1);           /* this makes 9 bits/entry */

/* This array contains left justified 1 or 2-letter pairs representing each unit */

dcl letters(0:n_units) char(2) aligned based (addr(digrams$letters));

/* This is the same as letters, but allows reference to individual characters */

dcl 1 letters_split(0:n_units) based (addr(digrams$letters)),
      2 first char(1),
      2 second char(1),
      2 pad char(2);

/* This array has rules for each unit */

dcl 1 rules(n_units) aligned based (addr(digrams$rules)),
      2 no_final_split bit(1),    /* can't be the only vowel in last syllable */
      2 not_begin_syllable bit(1),/* can't begin a syllable */
      2 vowel bit(1),             /* this is a vowel */
      2 alternate_vowel bit(1);   /* this is an alternate vowel, (i.e., "y") */

dcl n_units defined digrams$n_units fixed bin;

/********** end include file digram_structure.incl.pl1 *************/

dcl pp ptr;
dcl (j, k) fixed;
dcl max fixed;
dcl length fixed bin;
dcl ioa_$nnl entry options(variable);
dcl argno fixed bin;
dcl (diff, last, ncolumns init(0), remainder, middle, first) fixed;
dcl ioa_ entry options(variable);

who = "digram_table_compiler";
goto start1;

dtc: entry;
   who = "dtc";

start1:

compile = "1"b;   /* set switch to compile */
call cu_$arg_ptr (1, pp, length, code);
argno = 1;
if code ^= 0 then goto argerr;
call expand_path_ (pp, length, addr(dirname), addr(ename), code);
if code ^= 0 then goto argerr;
```

```
ename_length = index(ename, " ");
if ename_length = 0
then ename_length = 32;
else ename_length = ename_length - 1;
if ename_length >= 4
then
 if substr(ename, ename_length - 3, 4) = ".dtc"
 then ename_length = ename_length - 4;

argno = 2;
call cu_$arg_ptr (2, pp, length, code);  /* get option */
if code ^= 0
then list = "0"b;    /* no listing desired */
else
  if arg = "-ls" | arg = "-list"
  then do;
     list = "1"b;
     argno = 3;
     end;
  else do;
     code = error_table_$badopt;
     goto argerr;
     end;
goto get_ncolumns;

pdt: entry;
   who = "pdt";
   goto start2;

print_digram_table: entry;
   who = "print_digram_table";

start2:

   list = "1"b;
   argno = 1;
   compile = "0"b;

get_ncolumns:
   call cu_$arg_ptr (argno, pp, length, code);
   if code ^= 0
   then ncolumns = get_line_length_()/14;
   else do;
      ncolumns = cv_dec_check_ (arg, codex);
      if codex ^= 0
      then do;
        code = error_table_$badopt;
        goto argerr;
        end;
      end;

    if ^compile then goto dont_compile;

/* now initiate the source segment */

    call hcs_$initiate_count ((dirname), substr(ename, 1, ename_length) ||
         ".dtc", "", bc, 0, segptr, code);
    if segptr = null
```

```
   then do;
      call com_err_ (code, who, "^a>^a.dtc", dirname, substr(ename, 1, ename_length));
      return;
      end;

/* compile the segment */

   call hcs_$terminate_name ("digrams", code); /* terminate previous copies */
   if read_table_(segptr, bc) /* any error? */
   then
    do;
     call com_err_(0, who, "Error in source segment.");
     return;
    end;

  /* terminate the source now */

  call hcs_$terminate_noname (segptr, code);
  if ^list then return;    /* if no listing wanted, leave now */

dont_compile:
  if compile then call ioa_ ("^/^/^/^/");
  nrows = (n_units-1)/ncolumns + 1; /* This is the first reference to the digrams segment */
  if ncolumns^=0
  then
   do;
    do i = 1 to nrows;
     do j = i by nrows while (j <= n_units);
      call ioa_$nnl("    ^2d ^2a ^1b^1b^1b^1b",j,letters(j),rules(j).not_begin_syllable,
         rules(j).no_final_split,rules(j).vowel,rules(j).alternate_vowel);
     end;
     call ioa_("");
    end;
    call ioa_("");
    do start = 1, ncolumns*(59-nrows) + 1 by ncolumns*60
                 while (start<n_units**2);
     if start = 1
     then size = min(n_units*n_units, ncolumns*(59-nrows));
     else size = min(n_units*n_units-start+1, ncolumns*60);
     diff = size/ncolumns;
     remainder = size - diff*ncolumns;
     last = (size + ncolumns - 1)/ncolumns + start - 1;
     do first = start to last;
      middle = first + remainder*(diff + 1);
      if first = last & middle ^= first
      then max = middle - (diff+1);
      else max = middle + (ncolumns - remainder - 1)*diff;
      do i = first to middle by diff+1 while(i<=max), middle+diff to max by diff;
       j = (i-1)/n_units + 1;
       k = i - (j-1)*n_units;
       call ioa_$nnl ("   ^1b^1b^1b" || charac() || characc(j) || letters(k) || chara() || "^1b^1b",
        digrams(j,k).begin, digrams(j,k).not_begin,
        digrams(j,k).break, digrams(j,k).end, digrams(j,k).not_end);
      end;
      call ioa_ ("");
     end;
     if start = 1
     then call ioa_$nnl(copy("^/",66-first-nrows));
```

```
      else call ioa_$nnl(copy("^/",start+66-first));
     end;
    end;
return;

charac: proc returns(char(1));
  if digrams(j,k).prefix then return("-"); else return(" ");
end;

chara: proc returns (char(1));
 if digrams(j,k).illegal_pair
 then return("+");
 else
  if digrams(j,k).suffix
  then return("-");
  else return(" ");
end;

characc: proc(c) returns(char(2));
 dcl c fixed;
 if letters_split(c).second = " "
 then return(" " || letters_split(c).first);
 else return(letters(c));
end;

argerr:
    if code = error_table_$noarg
    then call com_err_ (code, who);
    else
     do;
      call cu_$arg_ptr (argno, pp, length, 0);
      call com_err_ (code, who, arg);
     end;

end;
```

```
COMPILATION LISTING OF SEGMENT generate_word_
Compiled by: Experimental PL/I Compiler of Tuesday, March 25, 1975 at 14:19
Compiled on: 04/01/75  1721.2 edt Tue
    Options: check source

/* This procedure is the subroutine interface to generate random words.
   It is called when the standard distributi?(n of random units (as returned by
   random_unit_) is desired.  The clock value is used as the starting seed unless
   generate_word_$init_seed is called.
*/
generate_word_: procedure (word, hyphenated_word, min, max);
 dcl word char(*);
 dcl hyphenated_word char(*);
 dcl min fixed bin;
 dcl max fixed bin;
 dcl (random_unit_, random_unit_$random_vowel) entry (fixed bin);
 dcl convert_word_ entry ((0:*) fixed bin, (0:*) bit(1) aligned,
     fixed bin, char(*), char(*));
 dcl random_word_ entry ((0:*) fixed bin, (0:*) bit(1) aligned,
     fixed bin, fixed bin, entry, entry);
 dcl hyphens (0:20) bit(1) aligned;
 dcl random_word (0:20) fixed bin;
 dcl length_in_units fixed bin;
 dcl random_length fixed bin;
 dcl unique_bits_ entry returns (bit(70));
 dcl encipher_ entry (fixed bin(71), (*) fixed bin (71), (*) fixed bin(71), fixed bin);
 dcl random_unit_stat_$seed(1) fixed bin(71) external;
 dcl first_call bit(1) static aligned init("1"b);

 /* On the very first call to this procedure in a process (if the
    init_seed entry was not called), use unique_bits to get a
    random number to initialize the random seed. */

 if first_call then do;
      random_unit_stat_$seed(1) = fixed (unique_bits_ ());
      first_call = "0"b;
      end;

 /* Get the length of the word desired.  We use the old value
    of the seed to determine this length so that the length of the word
    will not in some way be correlated with the word itself.
    We calculate this to be a uniformly distributed random number between
    min and max. */

 random_length = mod (abs (fixed (random_unit_stat_$seed(1), 17)), (max - min + 1)) + min;

 /* encipher the seed to get a random number and the next value of the seed */

 call encipher_ (random_unit_stat_$seed(1), random_unit_stat_$seed, random_unit_stat_$seed, 1);

 /* Get the random word and convert it to characters */

 call random_word_ (random_word, hyphens, random_length, length_in_units, random_unit_, random_unit_$random_vowel);
 call convert_word_ (random_word, hyphens, length_in_units, word, hyphenated_word);
 return;

/* This entry allows the user to set the seed.  If the seed argument is zero, we
   go back to using the clock value.
*/
generate_word_$init_seed: entry (seed);
```

```
dcl seed fixed bin(35);

if seed = 0 then first_call = "1"b;
else do;
 random_unit_stat_$seed(1) = seed;
 first_call = "0"b;
 end;
return;
end;
```

```
COMPILATION LISTING OF SEGMENT generate_words
Compiled by: Experimental PL/I Compiler of Tuesday, March 25, 1975 at 14:19
Compiled on: 04/01/75  1721.3 edt Tue
    Options: check source

generate_words: gw: procedure;
dcl cu_$arg_ptr entry (fixed,ptr,fixed,fixed bin(35));
dcl cu_$arg_ptr_rel entry (fixed bin, ptr, fixed bin, fixed bin(35), ptr);
dcl cu_$arg_list_ptr entry (ptr);
dcl argno fixed;
dcl new_line char(1) init("
");
dcl error_table_$badopt external fixed bin(35);
dcl arglen fixed bin;
dcl generate_word_ entry (char(*), char(*), fixed bin, fixed bin);
dcl generate_word_$init_seed entry (fixed bin(35));
dcl ios_$write_ptr entry (ptr, fixed bin, fixed bin);
dcl argptr ptr;
dcl hyphenate bit(1) init("0"b);
dcl cv_dec_check_ entry (char(*), fixed bin) returns (fixed bin(35));
dcl maximum_length fixed bin init(-1); /* set to maximum length of words */
dcl minimum_length fixed bin init(-1); /* minimum length of words */
dcl seed_value fixed bin(35) init(-1);     /* value of seed typed by user */
dcl com_err_ entry options(variable);
dcl i fixed, code fixed bin(35) init(0);
dcl unique_bits_ entry returns (fixed bin(70));
dcl result fixed bin;
dcl nwords fixed init(0);
dcl max_words fixed init(0);
dcl arg char(arglen) based (argptr) unaligned;
dcl maximum_hyphenated fixed bin;
dcl area char(56); /* where output line goes */
dcl output_line_length fixed bin; /* length of the output line in area */
dcl unhyphenated_word char (maximum_length) based (addr(area));
dcl hyphenated_word char (maximum_hyphenated) based (hph_ptr);
dcl hph_ptr ptr; /* pointer to position in area where hyphenated word goes */

dcl arglistptr ptr;

call cu_$arg_list_ptr (arglistptr);
do argno = 1 by 1 while(code = 0);
 call cu_$arg_ptr (argno,argptr,arglen,code);
 if code = 0
 then
  if arg = "-hph" | arg = "-hyphenate"
  then hyphenate = "1"b;
  else
   if arg = "-max"
   then maximum_length = value("maximum");
   else
     if arg = "-min"
     then minimum_length = value("minimum");
     else
        if arg = "-length" | arg = "-ln"
        then do;
          maximum_length = value("length");
          minimum_length = maximum_length;
          end;
        else
          if arg = "-seed" then do;
```

```
          seed_value = value("seed");
          call generate_word_$init_seed (seed_value);
          end;
         else do;
           nwords = cv_dec_check_ (arg, result); /* look for number of words */
           if result = 0 & nwords > 0
           then max_words = nwords;
           else call ugly (error_table_$badopt, arg);
         end;
end;

/* Below we decide whether minimum, maximum, both, or none have been specified,
   and set their default values accordingly. */

if nwords = 0 then max_words = 1;
if minimum_length = -1
then if maximum_length = -1
     then do;
         minimum_length = 6;
         maximum_length = 8;
         end;
     else minimum_length = 4;
else if maximum_length = -1
     then maximum_length = 20;
if minimum_length < 4 | minimum_length > maximum_length |
  maximum_length > 20 then
      call ugly (0, "Bad value of lengths: 3<min<max<21 required.");

maximum_hyphenated = maximum_length + 2*maximum_length/3; /* maximum length of hyphenated word */

hph_ptr = addr (substr (area, maximum_length + 2)); /* where hyphenated word is put */
                                       /* even if we're not printing it out, it needs a place to go */
if hyphenate /* for efficiency, put newline character in expected place in output string */
then do;
   substr (unhyphenated_word, maximum_length + 1, 1) = " ";
   substr (hyphenated_word, maximum_hyphenated + 1, 1) = new_line;
   output_line_length = maximum_length + maximum_hyphenated + 2;
   end;
else do;
   substr (unhyphenated_word, maximum_length + 1, 1) = new_line;
   output_line_length = maximum_length + 1;
   end;

/* generate max_words and write them all out */

do i = 1 to max_words;
   call generate_word_ (unhyphenated_word, hyphenated_word, minimum_length, maximum_length);
   call ios_$write_ptr (addr(area), 0, output_line_length);
end;


ugly: procedure (codex, message);
  dcl (code, codex) fixed bin(35);
  dcl message char(*);
  call com_err_ (codex, "generate_words", message);
  goto return;
end;
```

```
value: procedure (name) returns (fixed bin(35));
   dcl number fixed bin(35);
   dcl name char (*);
   argno = argno + 1;
   call cu_$arg_ptr_rel (argno, argptr, arglen, code, arglistptr);
   if code ^= 0 then call ugly (code, "Value of " || name);
   number = cv_dec_check_ (arg, result);
   if result ^= 0 | number < 0
   then call ugly (0, "Bad " || name || " value. " || arg);
   return(number);
end;

return:
end;
```

```
COMPILATION LISTING OF SEGMENT get_line_length
Compiled by: Experimental PL/I Compiler of Tuesday, March 25, 1975 at 14:19
Compiled on: 04/01/75  1721.4 edt Tue
    Options: check source

get_line_length: gll: proc;
 dcl com_err_ entry options(variable);
 dcl ioa_ entry options(variable);
 dcl active_fnc_err_ entry options(variable);
 dcl cu_$af_return_arg entry (fixed bin, ptr, fixed bin, fixed bin(35));
 dcl cu_$arg_count entry (fixed bin);
 dcl return_string char(max_length) varying based (return_string_ptr);
 dcl code fixed bin(35);
 dcl (error_table_$not_act_fnc, error_table_$wrong_no_of_args) fixed bin(35) external;
 dcl error_table_$nodescr fixed bin(35) external;
 dcl max_length fixed bin;
 dcl return_string_ptr ptr;
 dcl gmode char(168);
 dcl nargs fixed bin;
 dcl active_fnc fixed bin;
 dcl error_message(2) entry options(variable) variable init (com_err_, active_fnc_err_);

 /* entry on active function or command call */

 call cu_$af_return_arg (nargs, return_string_ptr, max_length, code);
 if code = error_table_$not_act_fnc
 then     /* not called as active function */
  do;
   active_fnc = 0;    /* set flag */
   call cu_$arg_count (nargs);  /* if called as a command, see if any arguments are present */
  end;
 else active_fnc = 1; /* this was an active function call */
 if code = error_table_$nodescr /* if no descriptors, assume we were called as a command */
 then do; nargs = 0; active_fnc = 0; end;
 if nargs ^= 0
 then
  do;                 /* arguments not allowed */
   call error_message(active_fnc + 1) (error_table_$wrong_no_of_args, "get_line_length");
   return;
  end;
 if active_fnc = 0
 then do;
   dcl temp char(3) ; /* bug in compiler requires this statement */
   temp = line_length();
   call ioa_ (temp);
   end;
 else return_string = line_length();
 return;

 /* entry on subroutine call */

get_line_length_: entry returns(fixed bin);
 return (fixed(line_length(), 17));

 /* internal procedure to get length of line */

line_length: procedure returns (char(*));
 dcl ios_$changemode entry (char(*), char(*), char(*), bit(72) aligned);
 dcl status bit(72) aligned;
 dcl 1 expanded_status based (addr(status)),
```

```
      2 bits bit(36),
      2 status_code fixed bin(35);
dcl numbers_index fixed bin;
dcl ll_index fixed bin;

 call ios_$changemode ("user_output", "", gmode, status);
if status_code ^= 0
then return ("132");

ll_index = index(gmode, "ll");   /* look for "ll" in returned modes */
if ll_index = 0
then return ("132");             /* "ll" not found */
else
 if ll_index ^= 1                /* if "ll" is not at beginning of string */
 then                            /* then it should be preceeded by "," */
  do;
   ll_index = index(gmode, ",ll");
   if ll_index = 0
   then return ("132");
   ll_index = ll_index + 1;
  end;
numbers_index = verify (substr (gmode, ll_index + 2), "0123456789"); /* find end of number after "ll" */
if numbers_index <= 1 | numbers_index > 4
then return ("132");             /* this number must be 1-3 digits */
else return (substr(gmode, ll_index + 2, numbers_index - 1));
end;

end;
```

```
COMPILATION LISTING OF SEGMENT hyphen_test
Compiled by: Experimental PL/I Compiler of Tuesday, March 25, 1975 at 14:19
Compiled on: 04/01/75  1721.5 edt Tue
    Options: check source

hyphen_test: ht: proc;
 dcl cu_$arg_ptr entry(fixed,ptr,fixed,fixed bin(35));
 dcl length fixed bin;
 dcl j fixed bin;
 dcl status fixed bin;
 dcl hyphenate_ entry (char(*), (*) bit(1) aligned, fixed bin);
 dcl hyphenate_$probability entry(char(*), (*) bit(1) aligned, fixed bin, float bin);
 dcl probability float bin;
 dcl hyphens(20) bit(1) aligned;
 dcl ioa_ entry options(variable);
 dcl arg char(length) based(argptr);
 dcl argptr ptr;
 dcl code fixed bin(35);
 dcl i fixed bin;
 dcl convert_word_char_ entry(char(*), (*) bit(1) aligned, fixed bin, char(*) varying);
 dcl result char(30) varying;
 dcl calculate bit(1) aligned init("0"b);

 do i = 1 by 1;
  call cu_$arg_ptr (i, argptr, length, code);
  if code ^= 0 then return;
  if arg = "-probability" | arg = "-pb" then calculate = "1"b;
  else do;
    if calculate
    then call hyphenate_$probability(arg,hyphens,status,probability);
    else call hyphenate_ (arg, hyphens, status);
    call convert_word_char_ (arg, hyphens, status, result);
    if calculate
    then call ioa_ ("^a ^f", result, probability);
    else call ioa_ (result);
    end;
 end;
end;
```

```
COMPILATION LISTING OF SEGMENT hyphenate_
Compiled by: Experimental PL/I Compiler of Tuesday, March 25, 1975 at 14:19
Compiled on: 04/01/75  1721.5 edt Tue
    Options: check source

/* This procedure tries to hyphenate a word supplied by the caller.

     dcl hyphenate_ entry(char(*), (*) bit(1) aligned, fixed bin);

     call hyphenate_ (word, returned_hyphens, code);

1) word       A word consisting of ASCII letters to be hyphenated.
              The first character may be uppercase or lowercase; the other
              characters may be lowercase only.

3) returned_hyphens
              A one bit in this array means that the corresponding
              character in word is to have a hyphen after it.

3) code       Status code: 0  word has been successfully hyphenated.
                          -1 word contains illegal characters.
                          -2 word was more than 20 or less than 3 characters.
              Any positive value of code means the word couldn't
              be hyphenated.  In this case code is the position of the
              first character that was not accepted.

The word is hyphenated by using random_word_ and whatever existing digram
table is in use by random_word_ to determine the syllabification and pronounceability
of the word supplied.  The standard random_unit_ routine is not used,
except that random_unit_$probabilities is called by hyphenate_$probability.
*/

hyphenate_: procedure(word, returned_hyphens, code);
 dcl word char(*);
 dcl code fixed bin;
 dcl debug static bit(1) init("0"b);
 dcl ioa_$nnl entry options(variable);
 dcl word_array(20) fixed bin static; /* word spread out into units */
 dcl hyphenated_word(0:20) bit(1) aligned; /* returned hyphens from random_word_ */
 dcl returned_hyphens(*) bit(1) aligned; /* hyphens to be returned to caller */
 dcl split_point fixed bin;  /* set on internal call at 2-letter unit to be split */
 dcl word_length_in_chars fixed bin static; /* length of word in characters */
 dcl word_length fixed bin static; /* length of word_array in units */
 dcl i fixed bin;
 dcl j fixed bin;
 dcl chars char(2);
 dcl char char(1);
 dcl word_index fixed bin static; /* index into word_array */
 dcl returned_word(0:20) fixed bin; /* word returned by random_word_ */
 dcl vowel_flag bit(1);      /* 1 when random_vowel is called */
 dcl last_good_unit static fixed bin; /* word_index of last good unit */
 dcl new_unit fixed bin;
 dcl random_word_ entry ((0:*)fixed bin, (0:*) bit(1) aligned,
         fixed bin, fixed bin, entry, entry);
 dcl random_unit_$probabilities entry ((*) float bin, (*) float bin);
 dcl probability float bin; /* value of p returned to user */
 dcl calculate bit(1) static; /* says we're calculating the probability of a word */
 dcl p float bin static; /* accumulated product of probability for the word */
 dcl total_p_this_unit float bin; /* total sum of probabilities of units that could be accepted in this position */
 dcl returned_length fixed bin;
```

```

/* probabilities of generating each unit at random */
/* obtained from a call to random_unit_$probabilities */

dcl (unit_probabilities based (u_p_ptr), vowel_probabilities based (v_p_ptr)) dim (n_units) float bin;
dcl (u_p_ptr, v_p_ptr) static ptr init(null());
dcl first_call static bit(1) init("1"b);

/********* include file digram_structure.incl.pl1 ********/

dcl digrams$digrams external;
dcl digrams$n_units fixed bin external;
dcl digrams$letters external;
dcl digrams$rules external;

/* This array contains information about all possible pairs of units */

dcl 1 digrams(n_units, n_units) based (addr(digrams$digrams)),
      2 begin bit(1),       /* on if this pair must begin syllable */
      2 not_begin bit(1),   /* on if this pair must not begin */
      2 end bit(1),         /* on if this pair must end syllable */
      2 not_end bit(1),     /* on if this pair must not end */
      2 break bit(1),       /* on if this pair is a break pair */
      2 prefix bit(1),      /* on if vowel must precede this pair in same syllable */
      2 suffix bit(1),      /* on if vowel must follow this pair in same syllable */
      2 illegal_pair bit(1),/* on if this pair may not appear */
      2 pad bit(1);         /* this makes 9 bits/entry */

/* This array contains left justified 1 or 2-letter pairs representing each unit */

dcl letters(0:n_units) char(2) aligned based (addr(digrams$letters));

/* This is the same as letters, but allows reference to individual characters */

dcl 1 letters_split(0:n_units) based (addr(digrams$letters)),
      2 first char(1),
      2 second char(1),
      2 pad char(2);

/* This array has rules for each unit */

dcl 1 rules(n_units) aligned based (addr(digrams$rules)),
      2 no_final_split bit(1),    /* can't be the only vowel in last syllable */
      2 not_begin_syllable bit(1),/* can't begin a syllable */
      2 vowel bit(1),             /* this is a vowel */
      2 alternate_vowel bit(1);   /* this is an alternate vowel, (i.e., "y") */

dcl n_units defined digrams$n_units fixed bin;

/********* end include file digram_structure.incl.pl1 ************/

 split_point = 0;
 calculate = "0"b;  /* we aren't calculating probabilities, just hyphenating */
 goto continue;
/*
```

```
*/
/* This entry is the same as hyphenate_, except that an additional value returned
   is the probability that the word would have been generated by random_word_
   using the current digram table and random_unit_ subroutine.  On the first call
   to this entry, random_unit_$probabilities is called to obtain the probabilities
   of all units.  If these change within a process, hyphenate_$reset must be called
   before hyphenate_$probability is called again.
*/

hyphenate_$probability: entry (word, returned_hyphens, code, probability);
 split_point = 0;
 p = 1;
 calculate = "1"b;
 if first_call then do;
     allocate unit_probabilities, vowel_probabilities;
     call random_unit_$probabilities (unit_probabilities, vowel_probabilities);
     first_call = "0"b;
     end;
 goto continue;

/* This entry is used to reset the probability arrays in case a new
   version of random_unit_ (with different probabilities) is used.
   Note that if a new version of digrams is also supplied, the old
   digrams must be terminated. */

hyphenate_$reset: entry; first_call = "1"b;
   if v_p_ptr ^= null() then free unit_probabilities, vowel_probabilities;
   return;

/* This entry point is called internally as a recursive call to hyphenate_.
   It is referenced when random_word_ did not accept the word because a 2-letter unit
   was illegal.  In this case we call this entry and tell hyphenate_ to split the 2-letter
   unit into 2 separate units.  The splitpoint argument specifies which one to do this with. */

hyphenate_$split: entry (word, returned_hyphens, code, splitpoint);
 dcl splitpoint fixed bin;
 split_point = splitpoint;

continue:
 word_length_in_chars = length(word);
 if word_length_in_chars > 20 | word_length_in_chars < 3
 then
  do;
   code = -2;
   if calculate then probability = 0;
   return;
  end;

/* Now that we have the word we want to hyphenate, we try to divide it up ino units as defined
   in the digram table.  We start with the first two letters in the word, and see if they are equal to any
   of the 2-letter units.  If they are, we store the index of that unit in the word_array, and increment
   our word_index by 2.  If they are not, we see if the first letter is equal to any of the 1-letter units.
   If it is, we store that unit and increment the word_index by 1.  If still not found, the character is
   not defined as a unit in the digram table and the word is illegal.  Of course, the word may still not be
   "legal" according to random_word_ rules of pronunciation and the digram table, but we'll find that out
   later.
*/
```

```
word_index = 1;
do i = 1 to word_length_in_chars;
  chars = substr(word, i, min(2, word_length_in_chars - i + 1));
  if i = 1 then substr(chars,1,1) = translate(substr(chars,1,1),"abcdefghijklmnopqrstuvwxyz","ABCDEFGHIJKLMNOPQRSTUVWXYZ");
  j = 1;
  do j = 1 to n_units while(chars^=letters(j)); /* look for 2-letter unit match */
  end;
  if j <= n_units & word_index ^= split_point
  then              /* match found */
    do;
      word_array(word_index) = j;       /* store 2-letter unit index */
      word_index = word_index + 1;
      i = i + 1;   /* skip over next unit */
    end;
  else
    do;             /* two-letter unit not found, search for 1-letter unit */
      char = substr(chars,1,1);
      if i = 1
      then char = translate(char,"abcdefghijklmnopqrstuvwxyz","ABCDEFGHIJKLMNOPQRSTUVWXYZ");
      char = substr(char,1,1);
      j = 1;
      do j = 1 to n_units while(char^=letters(j));
      end;
      if j <= n_units
      then       /* match found */
        do;
          word_array(word_index) = j;   /* store 1-letter unit index */
          word_index = word_index + 1;
        end;
      else
        do;       /* not found, unit is illegal */
          code = -1;
          if calculate then probability = 0;
          return;
        end;
    end;
end;
word_length = word_index - 1;
word_index = 0;

/* Now call random_word_, trying to get the word hyphenated.  Special versions of random_unit and
   random_vowel are supplied that return units of the word we are trying to hyphenate rather than
   random units.
*/

call random_word_ (returned_word, hyphenated_word, word_length_in_chars, returned_length, random_unit, random_vowel);
goto accepted;

/* If random_unit ever finds that random_word_ did not accept a unit from the word to be hyphenated,
   a nonlocal goto directly to this label (which pops random_word_ off the stack) is made, and we
   abort the whole operation.  If the last unit tried (i.e. the one not accepted) was a 2-letter unit,
   we might be able to make the word legal by splitting that unit up into two 1-letter units and
   starting all over.  Unfortunately, this is a lot of code and complication for a relatively rare case.
*/

not_accepted: word_index = word_index - 1; /* index of last unit accepted */
              p = 0;     /* zero probability if word was not accepted */
```

```
accepted: if debug then if calculate then call ioa_$nnl ("^/");
j = 1;
returned_hyphens = "0"b;
do i = 1 to word_length;
 if i > word_index & word_index < word_length /* we never got done with the word */
 then
  do;
   code = j;    /* word was not accepted */
   if letters_split(word_array(i)).second ^= " "   /* was it not accepted because of an illegal */
       & split_point = 0
   then do;                              /* 2-letter unit? */
    p = 1;
    call hyphenate_$split (word, returned_hyphens, code, i); /* try again with split pair */
    end;  /* Note: in even rarer cases, the unit that might be split to make this word legal is not the
          unit that was rejected, but a previous unit.  It's too hard to deal with this case, so we'll refuse the word,
          even though it might be legal.  As an example, using the standard digram table, "preeg-hu-o" is a legal word.
          However, our first attempt was to supply p-r-e-e-gh-u-o units.  Random_word_ rejects the
          "u" because it may not follow a "gh" unit in this context.  Since "u" is not a 2-letter
          unit, we can't try to split it up, so the word is thrown out.  However, p-r-e-e-g-h-u-o
          would have been acceptable to random_word_.  This is the only case where a
          word that could have been produced by random_word_ will be rejected by hyphenate_. */
   if calculate then probability = p;
   return; /* otherwise, return */
  end;

   /* set returned_hyphens bits corresponding to character in word.  Note that
      hyphens returned from random_word_ (hyphenated_word array) point to units,
      not characters. */

 if letters_split(word_array(i)).second ^= " "
 then j = j + 2;
 else j = j + 1;
 returned_hyphens(j-1) = hyphenated_word(i);
end;
code = 0;
if calculate then probability = p;
return;
/*
```

```
*/
/* The internal procedures random_unit and random_vowel keep track of the acceptance or rejectance of
   units they are supplying to random_word_.  Most of the code in the first part is to calculate probabilities
   when hyphenate_$probability is called.
*/
random_vowel: proc (returned_unit);
 dcl returned_unit fixed bin;
 vowel_flag = "1"b;
 goto generate;

random_unit: entry (returned_unit);
 vowel_flag = "0"b;

generate:

/* at this point, we either calculate probabilities or just go for another unit */

/* If probabilities are being calculated, we proceed as follows:
     In every position of the word, we send off to random_word_ all possible units except the one
     that is actually in the word.  We send these as negative numbers so that random_word_ will not actually use
     them, but will tell us whether they are legal.  Since we know the probabilies of all units, the
     total of the probabilities of the acceptable units can be calculated and normalized to 1 in order
     to determine the probability of the unit we are actually trying.  For example, if "e" is the only legal
     unit in a given position of the word, then its probability of appearing in that position is 1, since
     random_word_ will not accept anything else.

     When all units but the actual unit have been tried, we send off the actual unit with a positive sign.
     It should be accepted by random_word_ if the word is legal, and the ratio of its probability
     to the total probability of the legal units is the probability of the unit being in this word position.
     This multiplied by the product of these probabilities of the previous units gives us a "running product"
     that will eventually yield the probability of the whole word.
*/

if calculate then do;          /* we are calculating */
 if debug then
     if returned_unit < 0 then
          if returned_unit ^= -new_unit then
               call ioa_$nnl ("^a,", letters(-returned_unit));
 if returned_unit = 0 & word_index = 0 then do;   /* this is the first unit of the word */
   total_p_this_unit = 0;               /* initialize probabilities */
   word_index = 1;
   end;
 else if returned_unit = 0 & word_index ^= 0 then goto not_accepted; /* it tried to start a word all over on us */
 new_unit = word_array(word_index);     /* get the current unit from the word */
 if returoed_unit > 0 then do;          /* was the last unit accepted */
   if returned_unit = new_unit then do; /* yes, was it the one from this word position? */
     total_p_this_unit = 0;   /* initialize for next word position */
     word_index = word_index + 1;
     new_unit = word_array(word_index);/* get next unit from word, which now becomes current unit */
     returned_unit = 0;
     end;
   else do;         /* unit just accepted was not the one at this word position */
     if vowel_flag /* add its probability to total for this position and keep trying more units */
     then total_p_this_unit = total_p_this_unit + vowel_probabilities(returned_unit);
     else total_p_this_unit = total_p_this_unit + unit_probabilities(returned_unit);
     end;
   end;
 if -returned_unit = new_unit then goto not_accepted; /* current unit was not accepted */
```

```
skip_unit:
 returned_unit = abs(returned_unit) + 1; /* try next unit in unit table */
 if returned_unit = new_unit then returned_unit = returned_unit + 1; /* but skip the current one */
 if returned_unit > n_units
 then do; /* we've tried all the other units, try the current one now */
     /* If we are trying the current unit for real, we can calculate the probability of
        of this unit appearing at this position, assuming it will be accepted.
        Ratio of probability of this unit to total
        probability for the units accepted at this position gives the probability of this unit
        having legally been generated at this position */
     if vowel_flag
     then p = p * vowel_probabilities(new_unit)/(vowel_probabilities(new_unit) + total_p_this_unit);
     else p = p * unit_probabilities(new_unit)/(unit_probabilities(new_unit) + total_p_this_unit);
     returned_unit = new_unit;
     end;
 else returned_unit = -returned_unit; /* if not the current one, make it negative so it won't be used */
 if vowel_flag /* if vowel was wanted and this isn't one, it can't be used */
  then if ^rules.vowel(abs(returned_unit))
      then if ^rules.alternate_vowel(abs(returned_unit))
           then
             if returned_unit < 0   /* if we didn't care to keep it anyway, just ignore */
             then goto skip_unit;
             else goto not_accepted; /* if we wanted to keep it, the word is illegal */
 if debug then
         if returned_unit > 0 then call ioa_$nnl ("^a^a_; ", letters(returned_unit), ");
 end;

/* This section of code just supplies the next unit of the word */

else do;
   if returned_unit < 0 | (returned_unit = 0 & word_index ^= 0)
   then goto not_accepted; /* if last unit was not accepted */
   word_index = word_index + 1;
   new_unit = word_array(word_index);              /* get next unit from word */
   if vowel_flag                                   /* if random_word_ wanted a vowel, and this next unit is not one, */
   then if ^rules.vowel(new_unit)                  /* then we have to give up */
        then if ^rules.alternate_vowel(new_unit)  /* I wouldn't dare give random_word_ a non-vowel when it expects a vowel */
             then goto not_accepted;
   returned_unit = new_unit;
   return;
  end;
end;

debug_on: entry; debug = "1"b; return;

debug_off:entry; debug = "0"b; return;
end;
```

```
COMPILATION LISTING OF SEGMENT random_unit_
Compiled by: Experimental PL/I Compiler of Tuesday, March 25, 1975 at 14:19
Compiled on: 04/01/75  1721.7 edt Tue
    Options: check source

/* This is the standard random unit generating routine for random_word_.
   It is specified in the call to random_word_ by generate_word_.
   It does not reference the digram table, but assumes that it contains
   34 units in a certain 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 as returned from encipher_)
   is used to do a table lookup into an array containing unit indexes.
   There are 211 entries in the array for the random_unit_ entry point.
   The probablity 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.
*/

random_unit_: procedure (number);
 dcl numbers (0:210) fixed static init ((10)1,(8)2,(12)3,(12)4,(12)5,(8)6,
    (8)7,(6)8,(10)9,(8)10,(8)11,(6)12,(6)13,(10)14,(10)15,(6)16,
    (10)17,(8)18,(10)19,(6)20,(8)21,(8)22,23,(8)24,25,
    26,27,28,29,(2)30,(2)31,32,33,34);
 dcl vowel_numbers(0:11) fixed static init(1,1,5,5,5,9,9,15,15,20,20,24);
 dcl encipher_ entry (fixed bin(71), (*) fixed bin(71), (*) fixed bin(71), fixed bin);
 dcl random_unit_stat_$seed(1) external fixed bin(71);
 dcl number fixed bin;

 call encipher_ (random_unit_stat_$seed(1), random_unit_stat_$seed, random_unit_stat_$seed, 1);
 number = numbers (mod (abs (fixed (random_unit_stat_$seed(1), 17)), 211));
 return;
random_vowel:  entry (number);
 call encipher_ (random_unit_stat_$seed(1), random_unit_stat_$seed, random_unit_stat_$seed, 1);
 number = vowel_numbers (mod (abs (fixed (random_unit_stat_$seed(1), 17)), 12));
 return;

/* This entry returns the probabilities of the 34 units in two arrays.
   The first array contains the probabilities of all units assuming
   the random_unit_ entry was called.  The second array contains the
   probabilities of all units assuming random_vowel was called.
   Of course, there will be a lot of zeros in this second array, since
   most units aren't vowels.

   This entry is used by hyphenate_$probability to find out what the
   probabilities of the different units are.  Hyphenate_ does not know
   how many units there are or what their probabilities are.  It also
   makes no assumption about the unit index - to - letter correspondence
   of the units.  Thus this program can be replaced without changing
   anything in hyphenate_.
*/

probabilities: entry (unit_probs, vowel_probs);
```

```
  dcl unit_probs (34) float bin;
  dcl vowel_probs (34) float bin;
  dcl i fixed bin;

  unit_probs, vowel_probs = 0;

  /* These probabilities are calculated merely by adding up the number of
     occurances of each of the unit indexes in the numbers array and the
     vowel_numbers array. */

  do i = 0 to 210;
    unit_probs (numbers(i)) = unit_probs(numbers(i)) + 1;
    if i < 12
    then vowel_probs (vowel_numbers(i)) = vowel_probs (vowel_numbers(i)) + 1;
  end;

  unit_probs = unit_probs/211;  /* Normalize these values so they add up to 1.0 */
  vowel_probs = vowel_probs/12;
  return;
end;
```

```
ASSEMBLY LISTING OF SEGMENT >udd>hd>pg>word_gen>random_unit_stat_.alm
ASSEMBLED ON:       04/01/75  1725.6 edt Tue
OPTIONS USED:       ls  symbols  new_call  new_object
ASSEMBLED BY:       ALM Version 4.5, September 1974
ASSEMBLER CREATED:  02/24/75  1625.7 edt Mon

    000000                          1            name  random_unit_stat_
                                    2            use   linkc
                                    3            join  /link/linkc
                                    4
    000010              000010      5            segdef seed
                                    6            even
    000010  aa   012345 676543      7   seed:    oct    012345676543,123456765432
    000011  aa   123456 765432
                                    8            end

NO LITERALS
```

```
COMPILATION LISTING OF SEGMENT random_word_
Compiled by: Experimental PL/I Compiler of Tuesday, March 25, 1975 at 14:19
Compiled on: 04/01/75  1721.7 edt Tue
    Options: check source

/* This procedure generates a pronounceable random word of
  caller specified length and returns the
  word and the hyphenated (divided into syllables) form of the word.

  dcl random_word_ entry ((0:*) fixed, (0:*) bit(1) aligned, fixed, fixed, entry, entry);

  call random_word_ (word, hyphens, length, n, random_unit, random_vowel);

      word          random word, 1 unit per array element. (Output)

      hyphens       position of hyphens, bit on indicates hyphen appears after
                    corresponding unit in "word". (Input)

      length        length of word to be generated in letters. (Input)

      n             actual length of word in units. (Output)

      random_unit   routine to be called to generate a random unit. (Input)

      random_vowel  routine to be called to generate a random vowel. (Input)

*/

random_word_: procedure(password,hyphenated_word,length,word_length,random_unit,random_vowel);


/********** include file digram_structure.incl.pl1 ********/

dcl digrams$digrams external;
dcl digrams$n_units fixed bin external;
dcl digrams$letters external;
dcl digrams$rules external;

/* This array contains information about all possible pairs of units */

dcl 1 digrams(n_units, n_units) based (addr(digrams$digrams)),
      2 begin bit(1),          /* on if this pair must begin syllable */
      2 not_begin bit(1),      /* on if this pair must not begin */
      2 end bit(1),            /* on if this pair must end syllable */
      2 not_end bit(1),        /* on if this pair must not end */
      2 break bit(1),          /* on if this pair is a break pair */
      2 prefix bit(1),         /* on if vowel must precede this pair in same syllable */
      2 suffix bit(1),         /* on if vowel must follow this pair in same syllable */
      2 illegal_pair bit(1),/* on if this pair may not appear */
      2 pad bit(1);            /* this makes 9 bits/entry */

/* This array contains left justified 1 or 2-letter pairs representing each unit */

dcl letters(0:n_units) char(2) aligned based (addr(digrams$letters));

/* This is the same as letters, but allows reference to individual characters */

dcl 1 letters_split(0:n_units) based (addr(digrams$letters)),
      2 first char(1),
      2 second char(1),
```

```
         2 pad char(2);

/* This array has rules for each unit */

dcl 1 rules(n_units) aligned based (addr(digrams$rules)),
         2 no_final_split bit(1),     /* can't be the only vowel in last syllable */
         2 not_begin_syllable bit(1),/* can't begin a syllable */
         2 vowel bit(1),              /* this is a vowel */
         2 alternate_vowel bit(1);    /* this is an alternate vowel, (i.e., "y") */

dcl n_units defined digrams$n_units fixed bin;

/********* end include file digram_structure.incl.pl1 ************/

  dcl debug bit(1) aligned static init("0"b); /* set for printout of words that can't be generated */
   dcl password(0:*) fixed bin;
   dcl hyphenated_word(0:*) bit(1) aligned;
   dcl length fixed bin;
   dcl word_length fixed bin;
   dcl number float bin(27);
   dcl nchars fixed; /* number of characters in password */
   dcl index fixed init(1);  /* index of current unit in password */
   dcl i fixed;
   dcl syllable_length fixed init(1);  /* 1 when next unit is 1st in syllable, 2 if 2nd, etc. */
   dcl cons_count fixed init(0); /* count of consecutive consonants in syllable preceeding current unit */
   dcl vowel_found aligned bit(1);  /* 1 if vowel was found in syllable before this unit */
   dcl last_vowel_found aligned bit(1); /* same for previous unit in this syllable */
   dcl (first, second) fixed init(1); /* index into digram table for current pair */
   dcl (random_unit, random_vowel) entry (fixed);
   dcl unit fixed bin;
   dcl ioa_$nnl entry options(variable);

do i = 0 to length;
 password(i) = 0;
 hyphenated_word(i) = "0"b;
end;
nchars = length;

   /* get rest of units in password */

   unit = 0;
   do index = 1 by 1 while(index <= nchars);
      if syllable_length = 1
      then
         do;  /* on first unit of a syllable, use any unit */
keep_trying: unit = abs(unit);   /* last unit was accepted (or first in word), make positive */
           goto first_time;
retry:     unit = -abs(unit); /* last unit was not accepted, make negative */
first_time:
           if index = nchars /* if last unit of word must be a syllable, it must be a vowel */
           then call random_vowel(unit);
           else call random_unit(unit);
           password(index) = abs(unit); /* put actual unit in word */
           if index^=1 then if digrams(password(index-1),password(index)).illegal_pair
           then goto retry; /* this pair is illegal */
           if rules(password(index)).not_begin_syllable then goto retry;
           if letters_split.second(password(index)) ^= " "
           then
```

```
            if index = nchars
            then goto retry;
            else
             if index = nchars-1 & ^rules(password(index)).vowel & ^rules(password(index)).alternate_vowel
             then goto retry;  /* last unit was a double-letter unit and not a vowel */
             else if unit < 0
                  then goto keep_trying;
                  else nchars = nchars - 1;
           else if unit < 0 then goto keep_trying;
           syllable_length = 2;
           if rules(password(index)).vowel | rules(password(index)).alternate_vowel
           then
            do;
             cons_count = 0;
             vowel_found = "1"b;
            end;
           else
            do;
             cons_count = 1;
             vowel_found = "0"b;
            end;
           last_vowel_found = "0"b;
         end;
      else
         do;
           call generate_unit;
           if second = 0 then goto all_done; /* we have word already */
         end;
   end;

/* enter here at end of word */

all_done:
word_length = index - 1;
return;

/* various other entries */

debug_on: entry;
  debug = "1"b;
  return;

debug_off: entry;
  debug = "0"b;
  return;
/*
```

```
*/
/* PROCEDURE GENERATE_UNIT */

/* 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 syllable
     or previous 2 units are "not_end".
  c. "begin" pair generated but no vowel in syllable preceeding
     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 & third units of syllable must begin, and first unit is "alternate_vowel".

  The done routine checks for required prefix vowels & end of word conditions.*/

generate_unit: procedure;
 dcl 1 x aligned like digrams;
 dcl try_for_vowel bit(1) aligned;
 dcl unit_count fixed init(1); /* count of tries to generate this unit */
 dcl v bit(1) aligned;
 dcl i fixed;

 first = password(index-1);

/* on last unit of word and no vowel yet in syllable, or if previous pair
  requires a vowel and no vowel in syllable, then try only for a vowel */

  if syllable_length = 2 /* this is the second unit of syllable */
  then try_for_vowel = ^vowel_found & index=nchars; /* last unit of word and no vowel yet, try for vowel */
  else /* this is at least the third unit of syllable */
     if ^vowel_found | digrams(password(index-2),first).not_end
     then try_for_vowel = digrams(password(index-2),first).suffix;
     else try_for_vowel = "0"b;
  goto keep_trying; /* on first try of a unit, don't make the tests below */

/* come here to try another unit when previous one was not accepted */

try_more:
  unit = -abs(unit); /* last unit was not accepted, set sign negative */
  if unit_count = 100
  then
   do;
```

```
    if debug
    then
     do;
      call ioa_$nnl("100 tries failed to generate unit.^/ password so far is: ");
      do i = 1 to index;
        call ioa_$nnl("^a", letters(password(i)));
      end;
      call ioa_$nnl("^/");
     end;
    call random_word_(password, hyphenated_word, length, index, random_unit, random_vowel);
    second = 0;
    return;
   end;

/* come here to try another unit whether last one was accepted or not */

keep_trying:
if try_for_vowel
 then call random_vowel(unit);
 else call random_unit(unit);
 second = abs(unit); /* save real value of unit number */
 if unit > 0 then unit_count = unit_count + 1;    /* count number of tries */

 /* check if this pair is legal */

 if digrams(first,second).illegal_pair
 then goto try_more;
 else
  if first = second           /* if legal, throw out 3 in a row */
  then
   if index >2
   then
    if password(index-2) = first
    then goto try_more;
 if letters_split(second).second ^= " " /* check if this is 2 letters */
 then
  if index = nchars     /* then if this is the last unit of word */
  then goto try_more;   /* then a two-letter unit is illegal */
  else nchars = nchars - 1; /* otherwise decrement number of characters */
 password(index) = second;
 if rules(second).alternate_vowel
 then v = ^rules(first).vowel;
 else  v = rules(second).vowel;
  x.begin = digrams(first,second).begin;
  x.not_begin = digrams(first,second).not_begin;
  x.end = digrams(first,second).end;
  x.not_end = digrams(first,second).not_end;
  x.break = digrams(first,second).break;
  x.prefix = digrams(first,second).prefix;
  x.suffix = digrams(first,second).suffix;
  x.illegal_pair = digrams(first,second).illegal_pair;
 if syllable_length > 2   /* force break if last pair must be followed by a */
 then          /* vowel and this unit is not a vowel */
  if digrams(password(index-2),first).suffix
  then
   if ^v then break = "1"b; /*(if last pair was not_end, new_unit gave us a vowel)*/

/* In the notation to the right, the series of letters and dots stands
```

```
  for the last n units in this syllable, to be interpreted as follows:
   v stands for a vowel (including alternate_vowel)
   c stands for a consonant
   x stands for any unit
  the dots are interpreted as follows (c is used as example)
   c...c  one or more consecutive consonants
   c..c   zero or more consecutive consonants
   ...c   one or more consecutive consonants from beginning of syllable
   ..c    zero or more consecutive consonants from beginning of syllable
  the vertical line ¦ marks a syllable break.
  The group of symbols indicates what units there are in current
  syllable.  The last symbol is always the current unit.
  The first symbol is not necessarily the first unit in the
  syllable, unless preceeded by dots.  Thus, "vcc..cv" should be
  interpreted as "..xvcc..cv" (i.e., add "..x" to the beginning of all
  syllables unless dots begin the syllable.). */

if syllable_length = 2 & not_begin  /* pair may not begin syllable */
then goto loop;                     /* rule 3e. */
if vowel_found
then
 if cons_count ^= 0
 then
  if begin                          /* vc...cx */
  then
   if syllable_length ^= 3 & not_end_(3) /* vc...cx begin */
   then                             /* can we break at vc..c¦cx */
    if not_end_(2)                  /* no, try a break at vc...c¦x */
    then goto loop;                 /* rule 3c. */
    else call done(v,2);            /* vc...c¦x begin, treat as break */
   else call done(v,3);             /* vc..c¦cx begin */
  else
   if not_begin                     /* vc...cx ^begin */
   then
    if break                        /* vc...cx not_begin */
    then
     if not_end_(2)                 /* vc...c¦x break */
     then goto loop;                /* rule 3b, can't break */
     else call done(v,2);           /* vc...c¦x break */
    else
     if v                           /* vc...cx ^break not_begin */
     then                           /* vc...cv ^break not_begin */
      if not_end_(2)                /* try break at vc...c¦v */
      then goto loop;               /* rule 3f, break no good */
      else call done("1"b,2);       /* vc...c¦v treat as break */
     else
      if end                        /* vc...cc ^break not_begin */
      then call done("0"b,1);       /* vc...cc¦ end */
      else call done("1"b,0);       /* vc...cc ^break ^end not_begin */
   else
    if v                            /* vc...cx ^begin ^not_begin */
    then
      if not_end_(3) & syllable_length ^= 3 /* vc...cv rule 2a says we must break somewhere */
      then
       if not_end_(2)               /* vc..c¦cv doesn't work */
       then
        if cons_count > 1           /* vc...c¦v doesn't work */
        then                        /* vc...ccv */
```

```
          if not_end_(4)                /* try vc..c|ccv */
             | digrams(password(index-2),first).not_begin
          then goto loop;               /* rule 3f */
          else call done("1"b,4);       /* vc...c|ccv */
         else goto loop;                /* vc...c|v and vc..c|cv are no good */
        else call done("1"b,3);         /* vc...c|v treat as break */
       else call done("1"b,3);          /* vc..c|cv treat as break */
     else call done("1"b,0);            /* vc...cc ^begin ^not_begin */
  else /* vowel found and last unit is not consonant => last unit is vowel */
   if v & rules.vowel(password(index-2)) & index > 2
   then goto loop;                      /* rule 3a, 3 consecutive vowels non-y */
   else
    if end                              /* vx */
    then call done("0"b,1);             /* vx end */
    else
     if begin                           /* vx ^end */
     then
      if last_vowel_found               /* vx begin */
      then
       if v                             /* v...vvx begin */
       then
        if syllable_length = 3          /* v...vvv begin */
        then
         if rules(password((index-2))).alternate_vowel /* |vvv begin */
         then goto loop;                /* rule 3g, |"y"|vv is no good */
         else call done("1"b,3);        /* |v|vv begin */
        else
         if not_end_(3)                 /* v...vvv begin */
         then goto loop;                /* rule 3c, v...v|vv no good */
         else call done("1"b,3);        /* v...v|vv begin */
       else
        if syllable_length = 3          /* v...vvc begin */
        then
         if rules.alternate_vowel(password(index-2)) /* |vvc begin */
         then goto loop;                /* rule 3g, |"y"|vc is no good */
         else
          if rules.vowel(password(index-2)) /* |x|vc begin */
          then call done("1"b,3);       /* |v|vc begin */
          else goto loop;               /* |c|vc begin is illegal */
        else
         if not_end_(3)                 /* v...vvc begin */
         then                           /* v...vvc begin try to split pair */
          if not_end_(2)                /* v...vvc begin */
          then goto loop;               /* v...vv|c no good */
          else call done("0"b,2);       /* v...vv|c */
         else call done("1"b,3);        /* v...v|vc begin */
      else /* try splitting begin pair */
       if syllable_length > 2 /* ..cvx begin */
       then
        if not_end_(2)                  /* ...cvx begin */
        then goto loop;                 /* rule 3c, ...cv|x no good */
        else call done(v,2);            /* ...cv|x begin */
       else call done("1"b,0);          /* |vx begin */
     else
      if break                          /* ..xvx ^begin ^end */
      then
       if not_end_(2) & syllable_length > 2 /* ..xvx break */
       then goto loop;                  /* rule 3b, ..xv|x is no good */
```

```
      else call done(v,2);             /* ..v|x break */
     else call done("1"b,0);           /* ..vx ^end ^begin ^break */
 else
  if break                             /* ...cx */
  then goto loop;                      /* rule 3b, ...c|x break no good */
  else
   if end                              /* ...cx ^break */
   then
    if v                               /* ...cx end */
    then call done("0"b,1);            /* ...cv| end (new syllable) */
    else goto loop;                    /* rule 3b, ...cc| end no good */
   else
    if v                               /* ...cx ^end ^break */
    then
     if begin & syllable_length > 2    /* ...cv ^end ^break */
     then goto loop;                   /* c...c|cv ^end ^break begin, rule 3c */
     else call done("1"b,0);           /* ...cv ^end ^break ^begin */
    else
     if begin                          /* ...cc ^break ^end */
     then
      if syllable_length > 2           /* ..ccc begin */
      then goto loop;                  /* rule 3c, ...ccc begin */
      else call done("0"b,3);          /* |cc begin */
     else call done("0"b,0);           /* ..xcc ^end ^break ^begin */

/********** return here when unit generated has been accepted ********/

return;

/********** enter here when unit generated was good, but we don't want to use it because
           it was supplied as a negative number by random_unit or random_vowel ***********/

accepted_but_keep_trying: if letters_split(second).second ^= " "
                          then nchars = nchars + 1;   /* pretend unit was no good */
                          unit = -unit;   /* make positive to say that it would have been accepted */
                          goto keep_trying;
/********** enter here when unit generated is no good *********/

loop: if letters_split(second).second ^= " " then nchars = nchars + 1;
      goto try_more;
/*
```

```
*/
/* PROCEDURE DONE */

/* this routine is internal to generate_unit because it can return to loop */
/* call done when new unit is generated and determined to be
  legal. Arguments are new values of:
          vf  vowel_found
          mb  syllable_length (number of units in syllable. 0 means increment for this unit)
*/

done: procedure (vf, sl);
 dcl vf bit(1) aligned;
 dcl sl fixed;

 /* if we are not within first 2 units of syllable, check if
     vowel must precede this pair */

 if sl^=2 then if syllable_length^=2 then if prefix then if ^rules.vowel(password(index-2))
 then /* vowel must precede pair but no vowel precedes pair */
  if vowel_found   /* if there is a vowel in this syllable, */
  then             /* we may be able to break this pair     */
   if not_end_(2) /* check if this pair may be treated as break */
   then goto loop; /* no, previous 2 units can't end */
   else            /* yes, break can be forced */
    do;
     call done("0"b,2); /* ...cxx or ...cvx */
     return;
    end;
  else goto loop;     /* no vowel in syllable */

/* Check end of word conditions.  If end of word is reached, then
   1. We must have a vowel in current syllable, and
   2. This pair must be allowed to end syllable
*/

 if sl ^= 1
 then
  if index = nchars
  then
   if not_end
   then goto loop;
   else
     if vf = "0"b
    then goto loop;

/* A final "e" may not be the only vowel in the last syllable. */

 if index = nchars
 then
  if rules(second).no_final_split       /* this bit is on for "e" */
  then
   if sl ^= 1
   then
    if rules.vowel(first)               /* e preceded by vowel is ok, however */
    then;
    else
     if ^vowel_found|syllable_length<3  /* otherwise previous 2 letters must be*/
     then goto loop;                    /* able to end the syllable */
```

```
    else
     if unit < 0
     then goto accepted_but_keep_trying;
     else sl = 0;
if unit < 0 then goto accepted_but_keep_trying;
if v | sl = 1
then cons_count = 0;                   /* this unit is a vowel or new syllable is to begin */
else
 if sl = 0
 then cons_count = cons_count + 1;     /* this was a consonant, increment count */
else /* a new syllable was started some letters back, cons_count gets */
 cons_count = min(sl-1,cons_count+1); /* incremented, but no more than number of units in syllable */
if sl = 0
then syllable_length = syllable_length + 1;
else syllable_length = sl;
if syllable_length > 3
then last_vowel_found = vowel_found;
else last_vowel_found = "0"b;
vowel_found = vf;
if index - syllable_length + 1 ^= nchars
then hyphenated_word(index - syllable_length + 1) = "1"b;

end done;

end generate_unit;
/*
```

```
*/
/* PROCEDURE NOT_END_ */
/* not_end_(i) returns "1"b when ( password(index-i), password(index-i+1) )
   may not end a syllable, or when password(index-i+2) may not begin a syllable */

not_end_: procedure(i) returns(bit(1));
 dcl i fixed;
 if i = index
 then return(^rules.vowel(password(1)));
 if i ^= 1
 then
  if rules.not_begin_syllable(password(index-i+2)) then return("1"b);
 return(digrams(password(index-i),password(index-i+1)).not_end);
end;

end;
```

```
COMPILATION LISTING OF SEGMENT read_table_
Compiled by: Experimental PL/I Compiler of Tuesday, March 25, 1975 at 14:19
Compiled on: 04/01/75  1721.9 edt Tue
    Options: check source

/* This subroutine compiles the digram table, given a pointer to the
   segment containing the source.  It returns a flag if compiling was
   unsuccessful */

read_table_: procedure (source_table_ptr, bc) returns(bit(1));

dcl source_table char(1048575) based (source_table_ptr);
dcl source_table_ptr ptr;

/********* include file digram_structure.incl.pl1 ********/

dcl digrams$digrams external;
dcl digrams$n_units fixed bin external;
dcl digrams$letters external;
dcl digrams$rules external;

/* This array contains information about all possible pairs of units */

dcl 1 digrams(n_units, n_units) based (addr(digrams$digrams)),
      2 begin bit(1),        /* on if this pair must begin syllable */
      2 not_begin bit(1),    /* on if this pair must not begin */
      2 end bit(1),          /* on if this pair must end syllable */
      2 not_end bit(1),      /* on if this pair must not end */
      2 break bit(1),        /* on if this pair is a break pair */
      2 prefix bit(1),       /* on if vowel must precede this pair in same syllable */
      2 suffix bit(1),       /* on if vowel must follow this pair in same syllable */
      2 illegal_pair bit(1),/* on if this pair may not appear */
      2 pad bit(1);          /* this makes 9 bits/entry */

/* This array contains left justified 1 or 2-letter pairs representing each unit */

dcl letters(0:n_units) char(2) aligned based (addr(digrams$letters));

/* This is the same as letters, but allows reference to individual characters */

dcl 1 letters_split(0:n_units) based (addr(digrams$letters)),
      2 first char(1),
      2 second char(1),
      2 pad char(2);

/* This array has rules for each unit */

dcl 1 rules(n_units) aligned based (addr(digrams$rules)),
      2 no_final_split bit(1),    /* can't be the only vowel in last syllable */
      2 not_begin_syllable bit(1),/* can't begin a syllable */
      2 vowel bit(1),             /* this is a vowel */
      2 alternate_vowel bit(1);   /* this is an alternate vowel, (i.e., "y") */

dcl n_units defined digrams$n_units fixed bin;

/********* end include file digram_structure.incl.pl1 ************/

dcl (i, j, k, l) fixed bin;
dcl errflag bit(1) init("0"b);
dcl fatal_flag bit(1) init("0"b);
```

```
dcl neither_is_vowel bit(1);
dcl p ptr;
dcl 1 x based (p) like digrams;
dcl letters_(0:90) aligned char(2); /* storage for letters until we know how many units there are */
dcl 1 rules_(90) aligned like rules; /* ditto for rules */
dcl code fixed bin (35);
dcl flag bit(1);
dcl char char(1) init(" ");
dcl bc fixed bin(24);
dcl bitcount fixed bin(24);
dcl cleanup condition;
dcl term_$seg_ptr entry (ptr, fixed bin(35));
dcl get_group_id_ entry returns(char(32) aligned);
dcl hcs_$delentry_seg entry (ptr, fixed bin(35));
dcl hcs_$make_seg entry (char(*), char(*), char(*), fixed bin(5), ptr, fixed bin(35));
dcl (hcs_$add_acl_entries, hcs_$delete_acl_entries) entry
      (char(*), char(*), ptr, fixed bin, fixed bin(35));
dcl 1 acl aligned,
      2 user_name char(32),
      2 modes bit(36),
      2 pad bit(36),
      2 code fixed bin(35);
dcl null builtin;
dcl loc fixed init(1);
dcl end bit(1);
dcl new_line char(1) init ("
");
dcl com_err_$suppress_name entry options(variable);
dcl hcs_$set_bc_seg entry(ptr, fixed bin(24), fixed bin(35));
dcl get_pdir_ entry returns(char(168) aligned);
dcl get_wdir_ entry returns(char(168) aligned);
dcl alm entry options(variable);
dcl alm_prog based(prog_ptr) char(262144);
dcl prog_ptr ptr static init(null);
dcl seg_index init(1) fixed bin;

/* This procedure creates an ALM program containing empty blocks of storage.
   After finding out how many units there are, the size of each of these
   blocks can be determined.  The ALM program is then assembled, and
   segdef's are thus created which point to the beginning of each of
   these blocks.

   The first statement of the ALM program will be:

      equ n,xxxxx

   where xxxxx will be the number of units determined.  The rest of
   the statements are below:  */

dcl alm_statements(9) char(30) varying init (
   "segdef digrams",
   "segdef n_units",
   "segdef letters",
   "segdef rules",
   "bss n_units,1",           /* n_units fixed bin */
   "bss digrams,(n*n+3)/4",   /* digrams(n_units,n_units) bit(9) */
   "bss letters,n+1",         /* letters(0:n_units) char(2) aligned */
   "bss rules,4*n",           /* 1 rules(n_units) aligned, 2 (b1,b2,b3,b4) bit(1) */
```

```
   "end");




dcl ioa_ entry options (variable);
dcl ioa_$nnl entry options (variable);

/* check if a dollar sign ends segment */

if substr(source_table, bc/9 - 1, 1) ^= "$"
then goto dollar_error;

/* first read all the different letters or pairs to be defined */

do i = 1 to 90 while(char ^= ";"); /* read until semicolon */
 char = substr(source_table,loc,1);
 loc = loc + 1;
 if char < "a" | char > "z"
 then
  do;
   call ioa_$nnl ("alpha character expected");
fatal_error:
   fatal_flag = "1"b;
   goto err;
  end;
 substr(letters_(i),1,1) = char;
 char = substr(source_table,loc,1);
 loc = loc + 1;
 substr(letters_(i),2,1) = " ";
 if char < "a" | char > "z"
 then                            /* second character is not alphabetic */

try_bits:

  if char = "1" | char = "0" | char = " "
  then                           /* second character is a bit */
   do;
    rules_(i).not_begin_syllable = char="1";
    char = substr(source_table,loc,1);
    loc = loc + 1;
    if char = "1" | char = "0" | char = " "
    then                         /* another "rules" bit */
     do;
      rules_(i).no_final_split = char="1";
      char = substr(source_table,loc,1);
      loc = loc + 1;
     end;
    else                         /* no second "rules" bit */
     do;
      rules_(i).no_final_split = "0"b;
     end;
   end;
  else                           /* second character is not a bit and not alphabetic */
   rules_(i).not_begin_syllable, rules_(i).no_final_split = "0"b;
 else                            /* second character is alphabetic */
  do;
   substr(letters_(i),2,1) = char;
```

```
  char = substr(source_table,loc,1);
  loc = loc + 1;
  goto try_bits;
 end;

/* check character following for comma, new_line, or semicolon */

if char ^= ";" & char ^= "," & char ^= new_line
then
 do;
  call ioa_$nnl ("comma, blank, zero, one, or letter expected");
  goto fatal_error;
 end;

/* check if this unit is already defined */

if i ^= 1
then
   do j = 1 to i - 1;
      if letters_(j) = letters_(i)
      then do;
          call ioa_$nnl ("duplicate unit specification ""^a""", letters_(j));
          goto fatal_error;
          end;
   end;

/* set vowel flags */

rules_(i).vowel = letters_(i) = "a " | letters_(i) = "e " | letters_(i) = "i " | letters_(i) = "o " | letters_(i) = "u ";
rules_(i).alternate_vowel = letters_(i) = "y ";
end;

if i > 90
then
 do;
  call ioa_ ("Too many units defined"); /* more than 90 units */
  return ("1"b);
 end;

/* this is the on unit for aborted compilation
   It deletes the temporary segment containing the alm program, and
   deletes the acl entry of digrams that references this process's id. */

on condition(cleanup)
 begin;
  if prog_ptr ^= null
  then call hcs_$delentry_seg (prog_ptr, code);
  call hcs_$delete_acl_entries (get_wdir_(), "digrams", addr(acl), 1, code);
 end;

/* now that we know how many units, we can create the ALM program */

   /* first create the source segment in the process directory */

call hcs_$make_seg ("", "digrams.alm", "", 01010b, prog_ptr, code);
if prog_ptr = null
then do;
error_in_alm_prog:
```

```
   call com_err_$suppress_name (code, "digram_table_compiler", "digrams.alm in process directory");
   return("1"b);
   end;

call addline ("equ n," || substr(character(i-1), verify(character(i-1)," ")));  /* first line of ALM program */
do j = 1 to hbound(alm_statements,1); /* all the rest of the lines */
 call addline (alm_statements(j));
end;

    /* set the bit count of the source segment */

call hcs_$set_bc_seg (prog_ptr, (seg_index - 1)*9, code);
if code ^= 0 then goto error_in_alm_prog;

   /* assemble the ALM program */

call alm (before(get_pdir_()," ") || ">digrams");

/* Hopefully we got no errors.  If we did, we can't tell */
/* Delete the alm program, and set the acl of the object program
   to rw for this process */

call hcs_$delentry_seg (prog_ptr, code); /* ignore code */
prog_ptr = null(); /* just to be clean */
acl.user_name = get_group_id_();
acl.modes = "101"b;
acl.pad = ""b;
call hcs_$add_acl_entries (get_wdir_(), "digrams", addr(acl), 1, code);
if code ^= 0
then do;
   call com_err_$suppress_name (code, "digram_table_compiler", "digrams");
   return("1"b);
   end;

/* Store stuff into the object segment */

n_units = i - 1;  /* This is the first reference to the object segment */
letters(0) = "";
do i = 1 to n_units;
 letters(i) = letters_(i);
 rules(i) = rules_(i);
end;

/* digram table is compiled now */

do i = 1 to n_units;
  do j = 1 to n_units;
    p = addr(digrams(i,j));
    x.begin, x.not_begin, x.end, x.not_end, x.break, x.prefix, x.suffix = "0"b;
    char = substr(source_table,loc,1);
    do while (char = new_line);
     loc = loc + 1;
     char = substr(source_table,loc,1);
    end;
    if char="$" then do; call ioa_("illegal $ -- premature end"); return("1"b); end;
    if char = " " | char = "1"
    then
       do;
```

```
          x.begin = char="1";
          loc = loc + 1;
          call next_char_bit;
          x.not_begin = char="1";
          call next_char_bit;
          x.break = char="1";
          call next_char;
          x.prefix = char="-";
        end;
     call next_char;
     if char = " " | char = "-" then goto erra;
     if char ^= letters_split(i).first then goto errb;
     call next_letter(i);
     call next_char;
     if char = " " | char = "-" then goto erra;
     if char ^= letters_split(j).first
     then
      do;

      /* in case the second unit of a digram pair specification is illegal,
       this sequence attempts to get in sync again so that messages will not
       be printed indefinitely.  If the first lunit is illegal,
       no attempt is made to get in sync. */
       k = 1;
errb1: do k = max(k,1) to n_units while(char ^= letters_split(k).first);
       end; /* this takes care of skipping some units or duplicating the last unit */
       if k <= n_units
       then
        do;
         if letters_split(k).second ^= " "
         then
          do;
           char = substr(source_table,loc,1);
           if char ^= letters_split(k).second
           then
            do;
             k = k + 1;
             goto errb1;
            end;
          end;
         j = k + 1;
        end;
       else j = j + 1; /* if the unit can't be found, assume it's there but spelled wrong */
errb:  j = j - 1; /* if there is an extra digram that can't be found, we'll get another message */
       call ioa_$nnl("out of order or illegal letter"); goto err;
      end;
     call next_letter(j);
     char = substr(source_table,loc,1);
     loc = loc + 1;
     if char ^= "," & char ^= new_line & char ^= "$"
     then
        do;
          if char ^= " " & char ^= "-" & char ^= "+"
          then
erra:       do;
             call ioa_$nnl("alpha character expected");
             goto err;
            end;
```

```
          if char = "-"
          then x.suffix = "1"b;
          else
           if char = "+"
           then x.illegal_pair = "1"b;
          call next_bit;
          if end then goto loop;
          x.end = char="1";
          call next_bit;
          if end then goto loop;
          x.not_end = char="1";
          char = substr(source_table,loc,1);
          if char ^= new_line & char ^= ","
          then do; call ioa_$nnl("end of line expected"); goto err; end;
          loc = loc + 1;
        end;
loop:
   neither_is_vowel = ^rules.vowel(i) & ^rules.vowel(j) & ^rules.alternate_vowel(i) & ^rules.alternate_vowel(j);
   if (x.begin & (x.not_begin|(x.end & neither_is_vowel)|(^x.not_end & neither_is_vowel)|(x.break & ^rules.vowel(i)))) |
     (rules.not_begin_syllable(j) & x.break) |
      (x.end & (x.not_end|(^x.not_begin & neither_is_vowel)|(x.break & ^rules.vowel(j)))) |
      (x.break & (^x.not_begin & ^rules.vowel(i) | ^x.not_end & ^rules.vowel(j))) |
      (x.begin|x.not_begin|x.end|x.not_end|x.break|x.prefix|x.suffix)&x.illegal_pair
   then
      do;
         call ioa_$nnl ("consistency error");
err:     do k = 1 to loc-1 while (substr(source_table, loc-k, 1) ^= new_line);
         end;
         do l = 0 to bc/9-loc while (substr(source_table, loc+l, 1) ^= new_line);
         end;
         if ^errflag then
            call ioa_$nnl (" before * on following line");
         call ioa_ ("^/    " || substr(source_table,loc-k+1,k-1) ||
              "*" || substr(source_table,loc,1));
         if fatal_flag then return("1"b);  /* fatal error, can't continue */
         char = substr(source_table,loc-1,1);
         do loc = loc by 1 while (char ^= "," &
               char ^= new_line & char ^= "$");
          char = substr(source_table,loc,1);
         end;
         errflag = "1"b;
      end;
   end;
end;

call hcs_$delete_acl_entries (get_wdir_(), "digrams", addr(acl),1,code);

/* at end of table, make sure "$" follows and terminate segment */

if substr(source_table,loc,1)^="$"
then do;
dollar_error:
  call ioa_ ("$ not found at end of segment");
  return ("1"b);
end;
call term_$seg_ptr (source_table_ptr, code);
return(errflag);
```

```
/* get next letter, space, or "-" */

next_char: procedure;
   char = substr(source_table,loc,1);
   loc = loc + 1;
   if (char<"a" | char>"z") & char ^= " " & char ^= "-"
   then do; call ioa_$nnl("alpha character expected"); goto err; end;
end;

/* get next space or "1" */

next_char_bit: procedure;
   char = substr(source_table,loc,1);
   if char ^= " " & char ^= "1"
   then
    do;
     call ioa_$nnl ("space or 1 expected");
    goto err;
    end;
   loc = loc + 1;
end;

/* get next space, "1", ",", or new_line */

next_bit: procedure;
   char = substr(source_table,loc,1);
   end = "0"b;
   loc = loc + 1;
   if char ^= " "
   then
    if char = "," | char = new_line
    then end = "1"b;
    else
     if char ^= "1"
     then
      do;
       call ioa_$nnl("space, 1, comma, or new line expected");
       goto err;
      end;
end;

/* get next letter if this unit is a 2-letter unit */

next_letter: proc(i);
 dcl i fixed bin;
 if letters_split(i).second ^= " "
 then
  do;
   call next_char;
   if char ^= letters_split(i).second
   then
    do;
     call ioa_$nnl("""" || letters_split(i).second || """ expected");
     goto err;
    end;
  end;
end;
```

```
/* Add a line to ALM program */

addline: proc (string);
 dcl string char(30) varying;
 substr(alm_prog, seg_index, length(string) + 1) = string || "
";
 seg_index = seg_index + length(string) + 1;
end;

end;
```

APPENDIX IV

2000 RANDOM WORDS

The 2000 random words listed on the following pages were generated in one particular sample run using the tables described in Appendix I. See page 18 for a description of this listing.

2000 RANDOM WORDS

acbra	ac-bra
accarsju	ac-cars-ju
acmico	ac-mi-co
acnaw	ac-naw
adakgem	a-dak-gem
addazov	ad-da-zov
addwus	add-wus
adeocro	a-de-oc-ro
adfarvra	ad-farv-ra
adfrobga	ad-frob-ga
adicoc	a-di-coc
adkrev	ad-krev
adlisa	ad-li-sa
adoif	a-doif
adtemruf	ad-tem-ruf
advuj	ad-vuj
afttwir	aft-twir
agniji	ag-ni-ji
agromjax	a-grom-jax
agrovca	a-grov-ca
aiboc	ai-boc
aicboaj	aic-boaj
aijsav	aij-sav
aintjee	aint-jee
ajdama	aj-da-ma
ajhery	aj-he-ry
ajkealv	aj-kealv
ajlytwa	aj-ly-twa
ajnek	aj-nek
akdil	ak-dil
akhec	ak-hec
akhibwos	ak-hib-wos
akjaruj	ak-ja-ruj
akklokto	ak-klok-to
akprujo	ak-pru-jo
aktadssu	ak-tads-su
alcho	al-cho
algofwee	al-gof-wee
alltomp	all-tomp
altkeye	alt-keye
alwafi	al-wa-fi
ambrigno	am-brig-no
amshy	am-shy
amvacs	am-vacs
amvuti	am-vu-ti
anafniv	a-naf-niv
anpedavi	an-pe-da-vi
anviv	an-viv
anwobaj	an-wo-baj
apcloy	ap-cloy
apdrase	ap-drase
apkudaci	ap-ku-da-ci
apnopku	ap-nop-ku
apwry	ap-wry
araco	a-ra-co
arcmawmo	arc-maw-mo
arego	a-re-go
arjhoi	arj-hoi
armvodru	arm-vo-dru
arobli	a-ro-bli
arsshu	ars-shu
aseld	a-seld
aseyjaha	a-sey-ja-ha
ashfa	ash-fa
ashuwirp	as-hu-wirp
asuwoj	a-su-woj
ateunnga	a-teunn-ga
athoigna	a-thoig-na
atkopyej	at-ko-pyej
atojshyn	a-toj-shyn
atshub	atsh-ub
atwej	a-twej
auhuva	au-hu-va
aupkahy	aup-ka-hy
auptcu	aupt-cu
avavy	a-vav-y
avcarmev	av-car-mev
avriss	av-riss
avthwy	av-thwy
avthyve	av-thyve
avutman	a-vut-man
awwecba	aw-wec-ba
aycleti	ay-cle-ti
ayjedsi	ay-jeds-i
aylow	ay-low
aymsfop	ayms-fop
ayootta	a-yoot-ta
aypvihy	ayp-vi-hy
aysig	ay-sig
babfelby	bab-fel-by
bacgebvo	bac-geb-vo
bafdacga	baf-dac-ga
baiddbyt	baidd-byt
bajoo	ba-joo
balhayo	bal-ha-yo
baliom	ba-li-om
baquon	ba-quon
basciwa	bas-ci-wa
basfag	bas-fag
becamnob	be-cam-nob
becgroha	bec-gro-ha
beckreo	bec-kre-o
becwyd	bec-wyd
bedfleey	bed-fleey
bedibhi	be-dib-hi
bejeg	be-jeg
benchdyn	bench-dyn
beofy	be-of-y
beokyo	be-ok-yo
berho	ber-ho
berhyveu	ber-hy-veu
betavi	be-ta-vi
betwey	be-twey
bevsnudd	bev-snudd
bevtu	bev-tu
biadbeng	bi-ad-beng
bibjav	bib-jav
binchrod	binch-rod
biphs	biphs
bipku	bip-ku
bippu	bip-pu
bivics	bi-vics
biyebryg	bi-ye-bryg
biyus	bi-yus
blaidcej	blaid-cej
bleahiya	blea-hi-ya
blipjove	blip-jove
blitfefe	blit-fefe
blofe	blofe
blyijnee	bly-ij-nee
blyvabs	bly-vabs
bocks	bocks
bocwa	boc-wa
bodjobli	bod-jo-bli
bogcet	bog-cet
bogvuswy	bog-vu-swy
bojiri	bo-ji-ri
booval	boo-val

(continued)

boshtpel	bosht-pel
boudcof	boud-cof
bowyt	bo-wyt
bradcur	brad-cur
brefep	bre-fep
brerthy	brer-thy
brighe	brighe
brikaw	bri-kaw
broctbar	broct-bar
brolwoi	brol-woi
bronjept	bron-jept
bruosta	bru-os-ta
bruvlufe	bruv-lufe
bryarne	bryarne
bryffoj	bryf-foj
bryse	bryse
bucushu	bu-cu-shu
buebwu	bueb-wu
bussdene	buss-dene
butoc	bu-toc
buyen	bu-yen
buyovna	bu-yov-na
bybguf	byb-guf
bybhihyp	byb-hi-hyp
bycij	by-cij
byhojoc	by-ho-joc
byipio	by-i-pi-o
byjoarsy	by-joar-sy
bykmeol	byk-me-ol
bylij	by-lij
bynri	byn-ri
bypvee	byp-vee
byruc	by-ruc
bysstoct	bys-stoct
bytatha	by-ta-tha
cacky	cack-y
caged	ca-ged
caibo	cai-bo
cajthett	caj-thett
calkny	cal-kny
carurla	ca-rur-la
catseco	cats-e-co
caugewd	cau-gewd
cawete	ca-wete
cawlgu	cawl-gu
ceakhen	ceak-hen
ceasjota	ceas-jo-ta
ceays	ceays
ceehat	cee-hat
ceekasom	cee-ka-som
ceemm	ceemm
ceethpun	ceeth-pun
ceetvif	ceet-vif
cegmowec	ceg-mo-wec
cegpu	ceg-pu
cegvu	ceg-vu
ceigi	cei-gi
ceindhy	ceind-hy
cejiogmo	ce-ji-og-mo
cemjatbu	cem-jat-bu
cenjo	cen-jo
cenved	cen-ved
ceoliho	ce-o-li-ho
ceowf	ce-owf
cerba	cer-ba
cestha	ces-tha
ceugcho	ceug-cho
ceuhau	ceu-hau
cewfu	cew-fu
cewphjo	cewph-jo
cezcy	cez-cy
chafja	chaf-ja
chaheog	cha-he-og
chatmum	chat-mum
chetki	chet-ki
chewchut	chew-chut
chimwed	chim-wed
chishrai	chi-shrai
chivi	chi-vi
choasdy	choas-dy
chrubfu	chrub-fu
chuspryn	chusp-ryn
chysnior	chys-ni-or
ciaki	ci-a-ki
cibbihi	cib-bi-hi
cickka	cick-ka
cienbafi	cien-ba-fi
cifdabgi	cif-dab-gi
cigzawm	cig-zawm
cijpuyon	cij-pu-yon
cilch	cilch
cingloo	cin-gloo
ciscreny	cis-cren-y
civbybab	civ-by-bab
civjece	civ-jece
cixdo	cix-do
clefno	clef-no
clehy	cle-hy
clislo	cli-slo
cliwa	cli-wa
clorgcy	clorg-cy
coasbebi	coas-be-bi
cocuevo	co-cue-vo
codfri	cod-fri
coftti	coft-ti
comtdoa	comt-doa
conclay	con-clay
conmeco	con-me-co
copsgha	cops-gha
copuac	co-pu-ac
corlibbu	cor-lib-bu
coshryg	co-shryg
cotkni	cot-kni
cottrufa	cott-ru-fa
couhile	cou-hile
couwukfu	cou-wuk-fu
coysfo	coys-fo
cozrer	coz-rer
crafnec	craf-nec
craktamu	crak-ta-mu
crasco	cras-co
cravpo	crav-po
crejorry	cre-jor-ry
crelerhi	cre-ler-hi
crerju	crer-ju
cribmac	crib-mac
crova	cro-va
crujapki	cru-jap-ki
crybtoi	cryb-toi
cryce	cryce
cryjawl	cry-jawl
cubyuct	cu-byuct
cucksja	cucks-ja
cucsmu	cucs-mu
cuitreb	cu-it-reb
cujmemy	cuj-me-my
cuoywri	cu-oy-wri
cupco	cup-co

curirai	cu-ri-rai
cuwaso	cu-wa-so
cuyisguc	cu-yis-guc
cuyovvif	cu-yov-vif
cyayo	cya-yo
cyceckeo	cy-ceck-e-o
cyciacki	cy-ci-ac-ki
cyeemto	cyeem-to
cyitma	cy-it-ma
cykku	cyk-ku
cylow	cy-low
cylydga	cy-lyd-ga
cymri	cym-ri
cymshday	cymsh-day
cyngai	cyn-gai
cyoath	cyoath
cyocyaf	cyo-cyaf
cyppio	cyp-pi-o
cysofbi	cy-sof-bi
cytgipe	cyt-gipe
cyvuej	cy-vuej
cywroft	cy-wroft
dabowt	da-bowt
dacoafa	da-coa-fa
dacoryun	da-co-ryun
dacra	dac-ra
dadpawa	dad-pa-wa
dafekoab	da-fe-koab
dagby	dag-by
daheeno	da-hee-no
daiyino	dai-yi-no
dakfi	dak-fi
darad	da-rad
darrgage	darr-gage
dashon	da-shon
dashy	da-shy
dasyce	da-syce
dasypfea	da-syp-fea
datlyka	dat-ly-ka
datrouwa	dat-rou-wa
davpy	dav-py
dawdi	dawd-i
dawjokhu	daw-jok-hu
dayru	day-ru
deasfis	deas-fis
debkav	deb-kav
decreab	dec-reab
dedycea	de-dy-cea
deerb	deerb
defusk	de-fusk
deipie	dei-pie
dejev	de-jev
delro	del-ro
depcans	dep-cans
derjogfo	der-jog-fo
desgaku	des-ga-ku
deshleon	de-shle-on
detha	de-tha
dethmewg	deth-mewg
detmek	det-mek
detsbo	dets-bo
detwynd	de-twynd
dexba	dex-ba
dicte	dicte
diddy	did-dy
dieje	dieje
diethi	die-thi
dieyedi	die-ye-di
diffos	dif-fos
digba	dig-ba
digisubs	di-gi-subs
digwy	dig-wy
dihixag	di-hix-ag
dihyswif	di-hy-swif
dijdaisk	dij-daisk
dijeyli	di-jey-li
dijhejpo	dij-hej-po
dikruind	dik-ru-ind
dilgwi	dilg-wi
dindna	dind-na
dirssa	dirs-sa
dithwuic	dith-wu-ic
ditvayps	dit-vayps
diufo	di-u-fo
diumi	di-u-mi
diutcic	di-ut-cic
divka	div-ka
diwroxa	di-wrox-a
diyondy	di-yon-dy
doacrie	doac-rie
dobheby	dob-he-by
docta	doc-ta
dodnirwa	dod-nir-wa
doglin	do-glin
dogsjugs	dogs-jugs
dokabe	do-kabe
dokmo	dok-mo
domoo	do-moo
donond	do-nond
doslyip	do-sly-ip
dotlu	dot-lu
dotog	do-tog
doukdywo	douk-dy-wo
dowigoco	do-wi-go-co
dowvays	dow-vays
doynn	doynn
draja	dra-ja
dralop	dra-lop
dreje	dreje
dremra	drem-ra
dreoms	dre-oms
dresyji	dre-sy-ji
drietoi	drie-toi
drite	drite
drobveje	drob-veje
drocja	droc-ja
drodvove	drod-vove
droiro	droi-ro
droswen	dro-swen
druaney	dru-a-ney
druche	druche
drykucco	dry-kuc-co
dryoporc	dryo-porc
duetjiso	duet-ji-so
duibda	du-ib-da
duneoy	du-ne-oy
dunzeshi	dun-ze-shi
durrdo	durr-do
dyackdy	dyack-dy
dydfro	dyd-fro
dyfew	dy-few
dyfow	dy-fow
dyklys	dyk-lys
dykso	dyks-o
dynuci	dy-nu-ci
dytco	dyt-co
dytsner	dyts-ner
dywra	dy-wra

(continued)

dywud	dy-wud
eabcet	eab-cet
eagodco	ea-god-co
eajco	eaj-co
ealgado	eal-ga-do
eapbetga	eap-bet-ga
eatvafea	eat-va-fea
eaybfoa	eayb-foa
ebeogi	e-be-o-gi
ebjiab	eb-ji-ab
eblis	e-blis
ebnat	eb-nat
ebveo	eb-ve-o
eceink	e-ceink
ecjasvu	ec-jas-vu
eckajhyn	eck-aj-hyn
ecmitt	ec-mitt
ecnajo	ec-na-jo
ecouvda	e-couv-da
ecsfipe	ecs-fipe
ecuna	e-cu-na
ecvansh	ec-vansh
ecywa	e-cy-wa
edcouj	ed-couj
eddyhi	ed-dy-hi
edfevu	ed-fe-vu
edgoj	ed-goj
edhucaw	ed-hu-caw
edmuir	ed-mu-ir
edonoi	e-do-noi
edweep	ed-weep
eedeky	ee-dek-y
eedneka	eed-ne-ka
eedpo	eed-po
eehow	ee-how
eejor	ee-jor
eekvusu	eek-vu-su
eemju	eem-ju
eemyscle	ee-myscle
eengo	een-go
eepous	ee-pous
eeswygs	ee-swygs
eetrazeb	eet-ra-zeb
efonry	e-fon-ry
efrud	e-frud
eftha	eft-ha
efwupdan	ef-wup-dan
egalo	e-ga-lo
egcachy	eg-ca-chy
egidgema	e-gid-ge-ma
egoute	e-goute
egumsh	e-gumsh
egyigtuf	eg-yig-tuf
eibeu	ei-beu
eikbra	eik-bra
eippu	eip-pu
ejcodfej	ej-cod-fej
ejeowg	e-je-owg
ejkehib	ej-ke-hib
ejtiarr	ej-ti-arr
ekcee	ek-cee
ekdedva	ek-ded-va
ekdeu	ek-deu
ekeroa	e-ke-roa
ekjapriu	ek-jap-ri-u
elcyvo	el-cy-vo
elipiecy	e-li-pie-cy
elojchod	e-loj-chod
elolveag	e-lol-veag
emdrepe	em-drepe
emtid	em-tid
encypi	en-cy-pi
enddy	end-dy
enkpalt	enk-palt
envoj	en-voj
enyew	en-yew
eojmyg	e-oj-myg
eonco	e-on-co
eopjow	e-op-jow
epmiga	ep-mi-ga
epodmi	e-pod-mi
epoldto	e-pold-to
epphew	ep-phew
erfki	erf-ki
ersunaj	er-su-naj
ertaho	er-ta-ho
eruri	e-ru-ri
eryesko	e-ryes-ko
eshlazy	e-shla-zy
eshro	e-shro
esroyeba	es-ro-ye-ba
essocove	es-so-cove
ethkeff	eth-keff
ethukkli	e-thuk-kli
etnoween	et-no-ween
etosi	e-to-si
etsva	ets-va
eubgons	eub-gons
euckus	euck-us
eudapgol	eu-dap-gol
eufzo	euf-zo
eurcea	eur-cea
evauya	e-vau-ya
evbue	ev-bue
evcliho	ev-cli-ho
evcrof	ev-crof
evimdelm	e-vim-delm
evluhek	ev-lu-hek
evrantan	ev-ran-tan
evsarro	ev-sar-ro
evurfswa	e-vurf-swa
evuvi	e-vu-vi
evvof	ev-vof
ewbknepu	ewb-kne-pu
ewdkaph	ewd-kaph
ewecy	e-we-cy
ewisanyu	e-wi-san-yu
ewreckab	e-wreck-ab
ewskaye	ews-kaye
eybemi	ey-be-mi
eycust	ey-cust
eykeosfu	ey-ke-os-fu
eysba	eys-ba
eytwi	ey-twi
ezkowhu	ez-kow-hu
facjacjo	fac-jac-jo
fafyevby	faf-yev-by
fahawd	fa-hawd
fahemfai	fa-hem-fai
faifmeef	faif-meef
fakpluer	fak-pluer
falka	fal-ka
falryds	fal-ryds
fariribs	fa-ri-ribs
farul	fa-rul
fathba	fath-ba
fawneg	faw-neg
fecdruba	fec-dru-ba

fedtyo	fed-tyo
feent	feent
fehukemo	fe-hu-ke-mo
feike	feike
fejwi	fej-wi
fejwo	fej-wo
fekblyba	fek-bly-ba
femliurp	fem-li-urp
fenbawjo	fen-baw-jo
ferhyts	fer-hyts
feroda	fe-ro-da
feushno	feu-shno
fewjoli	few-jo-li
feyel	fe-yel
feyso	fey-so
ficfu	fic-fu
fierbju	fierb-ju
fiewods	fie-wods
filhi	fil-hi
filhi	fil-hi
filiwuth	fi-li-wuth
fisabthu	fi-sab-thu
fisdu	fis-du
fitbat	fit-bat
fithvajo	fith-va-jo
fitwif	fi-twif
fiufrase	fi-u-frase
fleac	fleac
flecky	fleck-y
fleudfu	fleud-fu
flywond	fly-wond
fobdry	fob-dry
fociogu	fo-ci-o-gu
focvuso	foc-vu-so
fofidry	fo-fi-dry
fofyind	fof-yind
fogfli	fog-fli
foinlir	foin-lir
foivni	foiv-ni
fokcrif	fok-crif
fokwuv	fok-wuv
fonwuwri	fon-wu-wri
fooyd	fooyd
fotha	fot-ha
fraca	fra-ca
fraja	fra-ja
fralav	fra-lav
frepond	fre-pond
frertci	frert-ci
frerwru	frer-wru
freyt	freyt
fridetyp	fri-de-typ
frivik	fri-vik
froadwix	froad-wix
frokijo	fro-ki-jo
fromi	fro-mi
fryayst	fryayst
fryood	fryood
frypli	fryp-li
fuand	fu-and
fucfamme	fuc-famme
fuchroaw	fuch-roaw
fuehuega	fue-hue-ga
fugheyn	fug-heyn
fugjijo	fug-ji-jo
fuhuol	fu-hu-ol
fuifew	fu-i-few
fujdeyvi	fuj-dey-vi
funnga	funn-ga
futho	fu-tho
fuyet	fu-yet
gadmajbu	gad-maj-bu
gaibci	gaib-ci
gairgy	gairg-y
gaisuvi	gai-su-vi
galdno	gald-no
ganrym	gan-rym
garive	ga-rive
gatidfo	ga-tid-fo
gavfa	gav-fa
gavneg	gav-neg
geajdu	geaj-du
gecpoby	gec-po-by
gedad	ge-dad
geddni	gedd-ni
gefocfef	ge-foc-fef
gefpup	gef-pup
geicnand	geic-nand
gelikac	ge-li-kac
geljelo	gel-je-lo
gemda	gem-da
genaho	ge-na-ho
geokays	ge-o-kays
gesfi	ges-fi
getito	ge-ti-to
geunejka	geu-nej-ka
gevjarsk	gev-jarsk
ghassy	ghas-sy
ghawndik	ghawn-dik
ghebgeb	gheb-geb
ghecmarn	ghec-marn
gheebo	ghee-bo
ghefrap	ghe-frap
ghelim	ghe-lim
ghibgeks	ghib-geks
ghinwa	ghin-wa
ghits	ghits
ghormfu	ghorm-fu
giadya	gi-a-dya
giewob	gie-wob
gigblody	gig-blo-dy
gilfyuev	gilf-yuev
gimlak	gim-lak
ginzu	gin-zu
gipdacna	gip-dac-na
gipquoi	gip-quoi
gipromso	gip-rom-so
girtwu	gir-twu
giwarnn	gi-warnn
glatt	glatt
glecjibs	glec-jibs
gliafuj	gli-a-fuj
glicy	gli-cy
glodim	glo-dim
glojpyt	gloj-pyt
glyceilm	gly-ceilm
glymkon	glym-kon
goatgope	goat-gope
gobpo	gob-po
gocmi	goc-mi
gofya	gof-ya
gogruoco	go-gru-o-co
goiduco	goi-du-co
goiwop	goi-wop
gojosody	go-jo-so-dy
gomwu	gom-wu
gonpacyu	gon-pa-cyu
goohungy	goo-hung-y

(continued)

gosha	gos-ha	hocfoyd	hoc-foyd	idikicu	i-di-ki-cu
gosieg	go-sieg	hocky	hock-y	idmybe	id-mybe
goyelts	go-yelts	hodvoi	hod-voi	idsidro	ids-i-dro
goyndnia	goynd-ni-a	hofcroy	hof-croy	idsod	ids-od
grabiwar	gra-bi-war	hogsh	hogsh	idtryted	id-try-ted
grafsgi	grafs-gi	hoheckni	ho-heck-ni	ifaihu	i-fai-hu
greccla	grec-cla	hoisu	hoi-su	ifanciry	i-fan-ci-ry
grinfict	grin-fict	hokdu	hok-du	ifcasihi	if-ca-si-hi
grodore	gro-dore	homeb	ho-meb	ifietkla	i-fiet-kla
grogeddo	gro-ged-do	hooll	hooll	ifretwye	i-fre-twye
gruije	gru-ije	hophli	ho-phli	ifrie	i-frie
grumum	gru-mum	hordeett	hor-deett	ifttan	ift-tan
grykpho	gryk-pho	howusilu	ho-wu-si-lu	ifwri	if-wri
grypvair	gryp-vair	hoycaisy	hoy-cai-sy	igekni	i-ge-kni
gulty	gul-ty	hucte	hucte	igjit	ig-jit
gurzijcy	gur-zij-cy	hupbiv	hup-biv	igmur	ig-mur
guskni	gus-kni	hycalo	hy-ca-lo	igviva	ig-vi-va
hadgha	had-gha	hygept	hy-gept	igwaur	ig-waur
hadghoce	had-ghoce	hyjane	hy-jane	igyeithe	ig-yeithe
hafbaj	haf-baj	hyktovo	hyk-to-vo	ijnaldo	ij-nal-do
hajcy	haj-cy	hyteuka	hy-teu-ka	ijnarca	ij-nar-ca
hapdiff	hap-diff	hyunasyg	hyu-na-syg	ijobi	i-jo-bi
hapfowdo	hap-fowd-o	hyvathad	hy-vat-had	ijvop	ij-vop
harisu	ha-ri-su	hyvock	hy-vock	ikibsa	i-kib-sa
haths	haths	ibdogin	ib-do-gin	iligmu	i-lig-mu
hatjec	hat-jec	ibiwo	i-bi-wo	ilrybna	il-ryb-na
hatse	hats-e	iblees	i-blees	iltjo	ilt-jo
heetad	hee-tad	ibolel	i-bo-lel	ilvnev	ilv-nev
heeycri	heey-cri	ibyat	i-byat	impfiuna	imp-fi-u-na
heilgheg	heil-gheg	ibyeliwi	i-bye-li-wi	inagwo	i-nag-wo
hemcying	hem-cy-ing	icbetcuk	ic-bet-cuk	incileo	in-ci-le-o
herewru	he-re-wru	icbryso	ic-bry-so	inkaiff	in-kaiff
hexwu	hex-wu	icdejcu	ic-dej-cu	inotho	i-no-tho
hicbesiv	hic-be-siv	icdesi	ic-de-si	inovinbo	i-no-vin-bo
hidfapo	hid-fa-po	ichdu	ich-du	inpak	in-pak
hieth	hieth	ichoad	i-choad	inthshor	inth-shor
hifdy	hif-dy	ichwrorl	ich-wrorl	iofiow	i-o-fi-ow
hijnan	hij-nan	iclelyd	ic-le-lyd	iofja	i-of-ja
hijraja	hij-ra-ja	icpeydmu	ic-peyd-mu	ipacnor	i-pac-nor
hirgu	hir-gu	icquej	ic-quej	ipcro	ip-cro
hirquav	hir-quav	icryp	ic-ryp	ipgha	ip-gha
hisibyru	hi-si-by-ru	icsdafo	ics-da-fo	ipgior	ip-gi-or
hispko	hisp-ko	icsroba	ics-ro-ba	iphgokvu	iph-gok-vu
hiuby	hi-u-by	icweacks	ic-weacks	iphreitu	iph-rei-tu
hoacved	hoac-ved	iddgo	idd-go	ipseg	ips-eg
hocbyn	hoc-byn	idignos	i-dig-nos	irfbick	irf-bick

2000 RANDOM WORDS

irinttuo	i-rint-tu-o
iseunk	i-seunk
ished	i-shed
isneept	is-neept
issfi	iss-fi
isshna	is-shna
isszeva	iss-ze-va
istcy	ist-cy
itagni	i-tag-ni
itban	it-ban
itevud	i-te-vud
ithfoj	ith-foj
ithgushe	ith-gushe
itofwaba	i-tof-wa-ba
itsdeg	its-deg
itsnes	its-nes
itthou	it-thou
ivbysu	iv-by-su
ivcrocwi	iv-croc-wi
iveit	i-veit
ivhee	iv-hee
ivmamts	iv-mamts
ivpirlo	iv-pir-lo
izoye	i-zoye
jabruxi	ja-brux-i
jabvel	jab-vel
jadfithi	jad-fit-hi
jadsbeho	jads-be-ho
jagnaveo	jag-na-ve-o
jahablad	ja-ha-blad
jahut	ja-hut
jaiwywu	jai-wy-wu
jakpyto	jak-py-to
jaumcef	jaum-cef
javdajbo	jav-daj-bo
javmi	jav-mi
jayrobyn	jay-ro-byn
jeanos	jea-nos
jebtro	jeb-tro
jedfotdy	jed-fot-dy
jeehiwa	jee-hi-wa
jefoulu	je-fou-lu
jeguape	je-gu-ape
jehilkti	je-hilk-ti
jeinri	jein-ri
jeinvo	jein-vo
jejlu	jej-lu
jektryit	jek-try-it
jekueccu	je-kuec-cu
jelewnhi	je-lewn-hi
jemafry	je-ma-fry
jenbi	jen-bi
jendd	jendd
jenho	jen-ho
jeobhyff	je-ob-hyff
jerctri	jerc-tri
jerhig	jer-hig
jeurimm	jeu-rimm
jewlys	jew-lys
jeybsk	jeybsk
jeycafa	jey-ca-fa
jeyees	je-yees
jiepcag	jiep-cag
jifkledd	jif-kledd
jijnapcy	jij-nap-cy
jikdeeb	jik-deeb
jimra	jim-ra
jindcro	jind-cro
jirsh	jirsh
jiscasdi	jis-cas-di
jitvadec	jit-va-dec
jivbo	jiv-bo
jivroan	jiv-roan
jivunuby	ji-vu-nu-by
jiwioc	ji-wi-oc
jiwoabca	ji-woab-ca
joafruo	joa-fru-o
jobci	job-ci
jocaforr	jo-ca-forr
jocci	joc-ci
jofsteiz	jofs-teiz
jojegdra	jo-jeg-dra
jolspe	jolspe
jomofe	jo-mofe
jonho	jon-ho
jonistsu	jo-nists-u
jopamnu	jo-pam-nu
jorka	jor-ka
jostpla	jost-pla
jotafcli	jo-taf-cli
jothsha	joth-sha
joucyka	jou-cy-ka
jouwi	jou-wi
jovcyk	jov-cyk
jowbe	jowbe
jubji	jub-ji
jufdyle	juf-dyle
juluwi	ju-lu-wi
jumcokdu	jum-cok-du
juonan	ju-o-nan
juvreg	juv-reg
juxlet	jux-let
kabisesu	ka-bi-se-su
kahabjuv	ka-hab-juv
kaisi	kai-si
kakti	kak-ti
kavdo	kav-do
kavuk	ka-vuk
kawuya	ka-wu-ya
keabcry	keab-cry
keatpana	keat-pa-na
kebinde	ke-binde
kecblen	kec-blen
kecca	kec-ca
keckiva	kec-ki-va
kefwept	kef-wept
keickdi	keick-di
kejbrema	kej-bre-ma
kejrenwa	kej-ren-wa
kekuo	ke-ku-o
keruve	ke-ruve
kewvo	kew-vo
keybaiya	key-bai-ya
kiaka	ki-a-ka
kiaswusp	ki-a-swusp
kicuir	ki-cu-ir
kidjaf	kid-jaf
kidubdu	ki-dub-du
kifnewfa	kif-new-fa
kisga	kis-ga
kivan	ki-van
klagge	klagge
klecerru	kle-cer-ru
klecwio	klec-wi-o
klowdno	klowd-no
klure	klure
klyhibe	kly-hibe
knavduew	knav-duew

(continued)

knect	knect
knenouke	kne-nouke
knesa	kne-sa
knifru	kni-fru
knige	knige
knoacy	knoa-cy
knojiti	kno-ji-ti
knoke	knoke
knuedcu	knued-cu
knuwa	knu-wa
knydglak	knyd-glak
knyings	kny-ings
knywo	kny-wo
kobvilwo	kob-vil-wo
kofeto	ko-fe-to
koinbi	koin-bi
kojrup	koj-rup
kojya	koj-ya
kokbrye	kok-brye
kokfuo	kok-fu-o
koleesce	ko-leesce
kolskja	kolsk-ja
kooche	kooche
kophja	koph-ja
kosujpa	ko-suj-pa
kotvads	kot-vads
kovshya	kov-shya
koybji	koyb-ji
koywo	koy-wo
kraincy	krain-cy
krapiep	kra-piep
krarracu	krar-ra-cu
krete	krete
krethuto	kre-thu-to
krivo	kri-vo
krochis	kroc-his
kromjo	krom-jo
krovew	kro-vew
kruofe	kru-ofe
kryov	kryov
kucvuko	kuc-vu-ko
kugwog	kug-wog
kulsh	kulsh
kumvicnu	kum-vic-nu
kuvhy	kuv-hy
lagiu	la-gi-u
lagmijy	lag-mij-y
lajtusu	laj-tu-su
lalfe	lalfe
lamlu	lam-lu
larre	larre
larro	lar-ro
larysk	la-rysk
lasawnt	la-sawnt
latalsvi	la-tals-vi
lathheow	lath-he-ow
lecpotdu	lec-pot-du
leilve	leilve
lemgure	lem-gure
lenhohay	len-ho-hay
leoneth	le-o-neth
lerface	ler-face
lesobkuo	le-sob-ku-o
levscyva	lev-scy-va
lezda	lez-da
lezycrec	le-zy-crec
liada	li-a-da
lidnogya	lid-nog-ya
lifgak	lif-gak
liloc	li-loc
lisjo	lis-jo
liulm	li-ulm
liunawyd	li-u-na-wyd
lofovu	lo-fo-vu
logsjo	logs-jo
lokosmin	lo-kos-min
lolhuf	lol-huf
lophyo	loph-yo
lopnute	lop-nute
lorpcier	lorp-cier
lorphohu	lor-pho-hu
ludnu	lud-nu
lutatkie	lu-tat-kie
lyavdyg	lyav-dyg
lycaff	ly-caff
lycel	ly-cel
lygefgi	ly-gef-gi
lyjeehok	ly-jee-hok
lykzybi	lyk-zy-bi
lyruvpi	ly-ruv-pi
lythowa	ly-tho-wa
lytis	ly-tis
lyuwag	lyu-wag
lyvas	ly-vas
mabka	mab-ka
macjay	mac-jay
madudi	ma-du-di
makiv	ma-kiv
malnu	mal-nu
malzata	mal-za-ta
marvici	mar-vi-ci
maswo	ma-swo
mavdodu	mav-do-du
mawfbeny	mawf-ben-y
mayrut	may-rut
meapafku	mea-paf-ku
medcahu	med-ca-hu
meeljilb	meel-jilb
mehuvo	me-hu-vo
meise	meise
memjel	mem-jel
memyn	me-myn
mencu	men-cu
mepdyo	mep-dyo
mepoohej	me-poo-hej
metupjob	me-tup-job
mevmewif	mev-me-wif
mexcu	mex-cu
miachaft	mi-ac-haft
mifjas	mif-jas
migshyje	mig-shyje
mihuty	mi-hu-ty
mikug	mi-kug
minri	min-ri
miosvia	mi-os-vi-a
misvago	mis-va-go
miukba	mi-uk-ba
mizpi	miz-pi
modheca	mod-he-ca
moithi	moi-thi
mojmo	moj-mo
momil	mo-mil
moothalp	moo-thalp
mowopy	mo-wo-py
mufiho	mu-fi-ho
muijcry	mu-ij-cry
muthoby	mu-tho-by
mutiplyk	mu-tip-lyk

myhati	my-ha-ti
myhow	my-how
myifeeg	my-i-feeg
mykna	my-kna
myrofed	my-ro-fed
mysgere	mys-gere
mytetvif	my-tet-vif
mythov	my-thov
nacgryo	nac-gryo
nadasthu	na-das-thu
nadvof	nad-vof
nafba	naf-ba
nafcybcy	naf-cyb-cy
nafohu	na-fo-hu
nagejtu	na-gej-tu
naibfon	naib-fon
nakro	nak-ro
nalblu	nal-blu
naldyd	nal-dyd
namme	namme
napom	na-pom
nashiwra	na-shi-wra
nastu	nas-tu
nattva	natt-va
natynhim	na-tyn-him
nauwaf	nau-waf
navfebda	nav-feb-da
nawflent	naw-flent
nayityha	na-yi-ty-ha
naykarci	nay-kar-ci
necprya	nec-prya
necra	nec-ra
necucucu	ne-cu-cu-cu
neeredeu	nee-re-deu
neergtec	neerg-tec
nefri	ne-fri
negagut	ne-ga-gut
negip	ne-gip
negot	ne-got
nehidpro	ne-hid-pro
nehiru	ne-hi-ru
neice	neice
neipvan	neip-van
nelrie	nel-rie
nelshjer	nelsh-jer
neltoag	nel-toag
nemaserr	ne-ma-serr
nenair	ne-nair
nenomdu	ne-nom-du
neokipi	ne-o-ki-pi
neppri	nepp-ri
nerga	ner-ga
neshghee	nesh-ghee
nesob	ne-sob
nethsknu	neths-knu
neudu	neu-du
newexam	ne-wex-am
newmlo	newm-lo
neyfedwy	ney-fed-wy
neywyep	ney-wyep
nicja	nic-ja
nidin	ni-din
nievdiv	niev-div
niezokyu	nie-zok-yu
nijwyd	nij-wyd
nikinkyo	ni-kink-yo
nilmku	nilm-ku
niness	ni-ness
ninmy	nin-my
nipcu	nip-cu
nipha	ni-pha
nishgryo	nish-gryo
nisna	nis-na
niwair	ni-wair
nocar	no-car
nocci	noc-ci
noceete	no-ceete
noclot	noc-lot
nocoss	no-coss
nodyo	no-dyo
noftha	nof-tha
noisspa	noiss-pa
nokroi	nok-roi
nolsht	nolsht
noogo	noo-go
nopykfa	no-pyk-fa
noudcen	noud-cen
nourycel	nou-ry-cel
novcias	nov-ci-as
noxna	nox-na
noyce	noyce
noydu	noy-du
nubhyev	nub-hyev
nuchso	nuch-so
nucwror	nuc-wror
nuishfi	nu-ish-fi
nulnot	nul-not
nuvnilt	nuv-nilt
nuvocida	nu-vo-ci-da
nuvoinzi	nu-voin-zi
nuwynip	nu-wy-nip
oabfu	oab-fu
oadedglu	oa-ded-glu
oakdydnu	oak-dyd-nu
oatokhot	oa-tok-hot
oawojion	oa-wo-ji-on
obsso	obs-so
obvipye	ob-vi-pye
oceppdif	o-cepp-dif
ockprad	ock-prad
ocnoy	oc-noy
octrye	oct-rye
octtin	oct-tin
odcrye	od-crye
odcuka	od-cu-ka
odhece	od-hece
odkos	od-kos
odvewju	od-vew-ju
odwreec	od-wreec
ofcai	of-cai
ofnaycy	of-nay-cy
ofnic	of-nic
ofway	of-way
ofyad	of-yad
ofyewn	of-yewn
ogbra	og-bra
ogfata	og-fa-ta
ogiwru	o-gi-wru
ogjipca	og-jip-ca
oiboay	oi-boay
oidvebs	oid-vebs
oighcabo	oigh-ca-bo
oighir	oig-hir
oimnib	oim-nib
oimut	oi-mut
oisckuma	oisck-u-ma
oiskasa	ois-ka-sa
oitha	oit-ha

(continued)

oitques	oit-ques
oitredi	oit-re-di
ojenty	o-jen-ty
ojheibu	oj-hei-bu
ojikgi	o-jik-gi
ojkojdy	oj-koj-dy
ojojoga	o-jo-jo-ga
ojphov	oj-phov
ojstro	oj-stro
ojugcry	o-jug-cry
okajo	o-ka-jo
okcewyve	ok-ce-wyve
oksibo	oks-i-bo
olactmam	o-lact-mam
olahit	o-la-hit
olshlely	ol-shle-ly
olsli	ol-sli
olswodpa	ol-swod-pa
ombomo	om-bo-mo
omreddwu	om-redd-wu
onbloth	on-bloth
onbro	on-bro
ondlunya	ond-lun-ya
ondpliak	ond-pli-ak
onoxe	o-noxe
onstju	onst-ju
ontdoi	ont-doi
onuokto	o-nu-ok-to
oocdet	ooc-det
oocre	oocre
oosnorgu	oos-nor-gu
ootdyhoa	oot-dy-hoa
openk	o-penk
ophthuig	oph-thu-ig
opmyti	op-my-ti
opyti	o-py-ti
orgami	or-ga-mi
oroaba	o-roa-ba
orquej	or-quej
ortta	ort-ta
osgroso	os-gro-so
oshgegtu	osh-geg-tu
oswev	o-swev
otgrafs	ot-grafs
othivdo	o-thiv-do
othnoak	oth-noak
otkria	ot-kri-a
otlif	ot-lif
otost	o-tost
otvayd	ot-vayd
otynro	o-tyn-ro
oudti	oud-ti
ourhogs	our-hogs
ourri	our-ri
ouryjau	ou-ry-jau
ovcyfen	ov-cy-fen
ovglyt	ov-glyt
ovgroo	ov-groo
ovitte	o-vitte
ovojetig	o-vo-je-tig
ovriso	ov-ri-so
ovsmiri	ov-smi-ri
ovteyo	ov-te-yo
ovvuc	ov-vuc
owjope	ow-jope
owpciovo	owp-ci-o-vo
oydjali	oyd-ja-li
oyouvers	o-you-vers
oytjefi	oyt-je-fi
ozuovgu	o-zu-ov-gu
padtra	pad-tra
pailp	pailp
pajles	paj-les
pakyith	pak-yith
pauvweu	pauv-weu
pawdedo	pawd-e-do
pawubi	pa-wu-bi
paywu	pay-wu
pedoghof	pe-do-ghof
pedti	ped-ti
peetieg	pee-tieg
pefcarr	pef-carr
pejvonga	pej-von-ga
pekcag	pek-cag
pesdray	pes-dray
peshegha	pes-he-gha
pesshmu	pes-shmu
pexgeuxa	pex-geux-a
phache	phache
phacwef	phac-wef
phagfi	phag-fi
phapvez	phap-vez
pheaglo	phea-glo
pheapomp	phea-pomp
phelti	phel-ti
phijca	phij-ca
phlegva	phleg-va
phless	phless
phlief	phlief
phubilst	phu-bilst
phunogso	phu-nogs-o
phuwo	phu-wo
pictten	pict-ten
pijted	pij-ted
pingsvu	pings-vu
piyiffe	pi-yiffe
plaught	plaught
plecpu	plec-pu
pleett	pleett
plivu	pli-vu
ployhat	ploy-hat
plydahoi	ply-da-hoi
pokjij	pok-jij
potheffi	po-thef-fi
povboula	pov-bou-la
pozcygti	poz-cyg-ti
pracfrob	prac-frob
pratardo	pra-tar-do
prorj	prorj
proucha	prou-cha
pryee	pryee
pryin	pry-in
pryitkek	pry-it-kek
pudaps	pu-daps
puits	pu-its
pujbiagy	puj-bi-ag-y
punipciv	pu-nip-civ
purvmyoj	purv-myoj
pusadeta	pu-sa-de-ta
pyadvu	pyad-vu
pydaft	py-daft
pydtejtu	pyd-tej-tu
pyjukra	py-juk-ra
pypyoco	py-pyo-co
pyrabmal	py-rab-mal
pyris	py-ris
pyuijhi	pyu-ij-hi
quagro	qua-gro

quavu	qua-vu	reyja	rey-ja	roudu	rou-du
quaytha	quay-tha	rhicwo	rhic-wo	rovovu	ro-vo-vu
quedyed	que-dyed	rhiemt	rhiemt	ruadpi	ru-ad-pi
queske	queske	rhilke	rhilke	rugnur	rug-nur
quiapho	qui-ap-ho	rhoinko	rhoin-ko	ruici	ru-i-ci
quibs	quibs	rhynute	rhy-nute	rujno	ruj-no
rabpi	rab-pi	rialqua	ri-al-qua	rumcawt	rum-cawt
racrinn	rac-rinn	riasa	ri-a-sa	runkzi	runk-zi
radpildd	rad-pildd	riccea	ric-cea	ruppbedd	rupp-bedd
raihusav	rai-hu-sav	ricda	ric-da	rutwece	ru-twece
raimmp	raimmp	ricio	ri-ci-o	ruwabkid	ru-wab-kid
rainegha	rai-ne-gha	ricki	ric-ki	ruwik	ru-wik
rajvow	raj-vow	riclea	ric-lea	ruyadpoy	ru-yad-poy
rakyac	rak-yac	ricnu	ric-nu	ruyucvi	ru-yuc-vi
ramdru	ram-dru	ricsi	rics-i	ryane	ryane
ranwye	ran-wye	ridatcee	ri-dat-cee	rychyg	ry-chyg
rarenbu	ra-ren-bu	rietvue	riet-vue	rycitwev	ry-ci-twev
rarjnu	rarj-nu	rijacas	ri-ja-cas	rycoabco	ry-coab-co
ratvuld	rat-vuld	rijreci	rij-re-ci	rycoye	ry-coye
ravef	ra-vef	rijviki	rij-vi-ki	rydilb	ry-dilb
ravek	ra-vek	rimavmaw	ri-mav-maw	ryfladgi	ry-flad-gi
rawej	ra-wej	rinen	ri-nen	ryjobo	ry-jo-bo
rayceoc	ray-ce-oc	rinnphi	rinn-phi	ryjupo	ry-ju-po
rebhet	reb-het	riphha	riph-ha	rykty	ryk-ty
rebku	reb-ku	ripni	rip-ni	rylynswy	ry-lyn-swy
rebujoca	re-bu-jo-ca	ristrewp	rist-rewp	rymro	rym-ro
reddy	red-dy	ritas	ri-tas	rynadoge	ry-na-doge
redjeyon	red-je-yon	rivgipav	riv-gi-pav	ryouv	ryouv
reedbi	reed-bi	rivoam	ri-voam	rytjior	ryt-ji-or
reedphot	reed-phot	rivva	riv-va	rytmodmu	ryt-mod-mu
reeygu	reey-gu	riwrida	ri-wri-da	ryuajir	ryu-a-jir
regcaib	reg-caib	riwyac	ri-wyac	ryuwa	ryu-wa
regsfa	regs-fa	roadswus	roads-wus	rywictfi	ry-wict-fi
reifiec	rei-fiec	roadtu	road-tu	sadjip	sad-jip
rejayoi	re-ja-yoi	rocwain	roc-wain	sadol	sa-dol
rejsnan	rej-snan	rodbla	rod-bla	sahoarvy	sa-hoarv-y
remwru	rem-wru	rokakny	ro-ka-kny	sahuv	sa-huv
renmo	ren-mo	rokkect	rok-kect	salnodu	sal-no-du
reproo	rep-roo	rolety	ro-le-ty	sanfo	san-fo
reraij	re-raij	ronso	ron-so	sanomt	sa-nomt
rerjby	rerj-by	roocwu	rooc-wu	satnan	sat-nan
rerjneav	rerj-neav	roombrac	room-brac	savki	sav-ki
reshbuf	resh-buf	ropflaty	rop-fla-ty	sawtoyad	saw-to-yad
rethe	rethe	rorjam	ror-jam	saysbu	says-bu
revkrey	rev-krey	rosfat	ros-fat	sceeltfu	sceelt-fu
reyff	reyff	rothko	roth-ko	schesy	sche-sy

(continued)

sciobo	sci-o-bo	sorifri	so-ri-fri	tethnan	teth-nan
screa	screa	spabu	spa-bu	tevevi	te-ve-vi
scuocu	scu-o-cu	speitoto	spei-to-to	teybwu	teyb-wu
sedvude	sed-vude	spibsmu	spibs-mu	thasachi	tha-sa-chi
seifrav	seif-rav	spipvigh	spip-vigh	thatbogs	that-bogs
seisap	sei-sap	spoijtoc	spoij-toc	thebaims	the-baims
sejtheb	sej-theb	spyfodd	spy-fodd	thefts	thefts
sewocle	se-wocle	spyun	spyun	theillpa	theill-pa
sheccu	shec-cu	stacso	stacs-o	thepcli	thep-cli
sheguogy	she-gu-og-y	stanod	sta-nod	thets	thets
shere	shere	stier	stier	thevu	the-vu
shikgevi	shik-ge-vi	strongha	stron-gha	thiarlfo	thi-arl-fo
shiwyt	shi-wyt	stuxa	stux-a	thojca	thoj-ca
shlabry	shla-bry	stycip	sty-cip	thona	tho-na
shmeithi	shmeit-hi	stylig	sty-lig	thradfi	thrad-fi
shnujdog	shnuj-dog	subfomi	sub-fo-mi	threzbo	threz-bo
shoawn	shoawn	swetknuj	swet-knuj	thridd	thridd
shrapdro	shrap-dro	swoghu	swog-hu	throaga	throa-ga
shtee	shtee	swoyfi	swoy-fi	throyott	thro-yott
shtiadeg	shti-a-deg	swurjuss	swur-juss	thrup	thrup
shtild	shtild	swurny	swurn-y	thuda	thu-da
shyib	shy-ib	syavo	sya-vo	thuisko	thu-is-ko
siasics	si-a-sics	sybdog	syb-dog	thwiwil	thwi-wil
sijty	sij-ty	syesjats	syes-jats	thyaym	thyaym
sklima	skli-ma	taccy	tac-cy	thylu	thy-lu
skrec	skrec	taceszy	ta-ces-zy	tiadd	ti-add
skymid	sky-mid	taiset	tai-set	tibhay	tib-hay
slamfod	slam-fod	tajsciko	taj-sci-ko	tifdol	tif-dol
slasu	sla-su	tajva	taj-va	tificbu	ti-fic-bu
slujna	sluj-na	tajyaryb	taj-ya-ryb	tihadna	ti-had-na
slyhy	sly-hy	talmptwu	talmpt-wu	tilocoy	ti-lo-coy
slyos	slyos	tarorg	ta-rorg	tionu	ti-o-nu
slyry	sly-ry	tawsdi	taws-di	tiowark	ti-o-wark
slyshra	sly-shra	tebes	te-bes	tipgi	tip-gi
smoyceko	smoy-ce-ko	tedoa	te-doa	tirpikdi	tir-pik-di
snemuews	sne-muews	teeji	tee-ji	titshcuk	titsh-cuk
snenbi	snen-bi	teerny	teern-y	titwes	ti-twes
snepvu	snep-vu	teivropa	teiv-ro-pa	tivned	tiv-ned
snoith	snoith	temblino	tem-bli-no	tiyifa	ti-yi-fa
snomwrul	snom-wrul	tenort	te-nort	toccuafi	toc-cu-a-fi
snotic	sno-tic	tenragwu	ten-rag-wu	tocon	to-con
snupdo	snup-do	teolub	te-o-lub	tokget	tok-get
snynju	snyn-ju	tequeaf	te-queaf	tokssu	toks-su
sodcy	sod-cy	terch	terch	tomfrys	tom-frys
sodnacti	sod-nac-ti	terir	te-rir	tosbi	tos-bi
sofbri	sof-bri	terynvoa	te-ryn-voa	trafciv	traf-civ

trebo	tre-bo
treormer	tre-or-mer
tretozwa	tre-toz-wa
tretrado	tret-ra-do
trickacy	trick-a-cy
trida	tri-da
trilmklu	trilm-klu
tripre	tripre
trodor	tro-dor
tronksya	tronks-ya
trosnayn	tros-nayn
tryhow	try-how
tudrymm	tu-drymm
tuduyen	tu-du-yen
tuesh	tuesh
tufhaf	tuf-haf
tugdu	tug-du
tugoyu	tu-go-yu
tuivfu	tu-iv-fu
tujestty	tu-jest-ty
tukici	tu-ki-ci
tuljtank	tulj-tank
tuownsmi	tu-owns-mi
tuthu	tu-thu
twejut	twe-jut
twelyte	twe-lyte
twepvou	twep-vou
twufoist	twu-foist
twyvaswo	twy-va-swo
tyciaqua	ty-ci-a-qua
tylywiv	ty-ly-wiv
tyocle	tyocle
typayryo	ty-pay-ryo
typeaf	ty-peaf
tytba	tyt-ba
ubcedu	ub-ce-du
ubpiji	ub-pi-ji
ubpru	ub-pru
ubwru	ub-wru
ubybfrex	u-byb-frex
ucdapcib	uc-dap-cib
ucecai	u-ce-cai
ucgij	uc-gij
uchiogid	u-chi-o-gid
udaufmi	u-dauf-mi
udodd	u-dodd

udweryho	ud-we-ry-ho
udygnedi	u-dyg-ne-di
ufdujmod	uf-duj-mod
ufjeaki	uf-jea-ki
ufkodbot	uf-kod-bot
ufwata	uf-wa-ta
ugazsca	u-gaz-sca
ugnea	ug-nea
ugveje	ug-veje
ujeshump	u-je-shump
ujhejda	uj-hej-da
ujvoba	uj-vo-ba
ukogchi	u-kog-chi
ukuckody	u-kuck-o-dy
ukyid	uk-yid
ultic	ul-tic
undau	un-dau
unraksom	un-raks-om
untduaci	unt-du-a-ci
unvawth	un-vawth
unvuecad	un-vue-cad
uphdelyk	uph-de-lyk
uphij	u-phij
upsha	up-sha
urhodri	ur-ho-dri
urshfo	ursh-fo
ushgat	ush-gat
usjuhy	us-ju-hy
uskcruk	usk-cruk
uskluds	usk-luds
usktyse	usk-tyse
usspt	usspt
usvuruij	us-vu-ru-ij
utcerd	ut-cerd
utewade	u-te-wade
utwub	u-twub
uvnoosod	uv-noo-sod
uvovveo	u-vov-ve-o
uvubsoi	u-vub-soi
vabju	vab-ju
vabriho	va-bri-ho
vacquic	vac-quic
vadfuo	vad-fu-o
vadwyhey	vad-wy-hey
vadyjo	va-dy-jo
vafjitdu	vaf-jit-du

vagyoo	vag-yoo
vaiwru	vai-wru
valco	val-co
vardvu	vard-vu
vargro	var-gro
vasce	vasce
vashcli	vash-cli
vassvu	vass-vu
veagfags	veag-fags
vebobo	ve-bo-bo
vedim	ve-dim
vedtwo	ved-two
veejplo	veej-plo
veemreur	veem-reur
vefhyo	vef-hyo
vegcopco	veg-cop-co
vegewal	ve-ge-wal
vegontha	ve-gon-tha
vegti	veg-ti
vegwa	veg-wa
vejlyen	vej-lyen
veocplu	ve-oc-plu
vetda	vet-da
vetdeif	vet-deif
vevma	vev-ma
viaja	vi-a-ja
viaklu	vi-ak-lu
vibdiju	vib-di-ju
viclup	vic-lup
viconow	vi-co-now
vijeicbo	vi-jeic-bo
vijku	vij-ku
villja	vill-ja
vinomjo	vi-nom-jo
vinuhici	vi-nu-hi-ci
viomlop	vi-om-lop
virte	virte
visjehy	vis-je-hy
visviens	vis-viens
viteyfu	vi-tey-fu
vobmocgi	vob-moc-gi
vociwen	vo-ci-wen
vogtho	vog-tho
voithiyo	voi-thi-yo
voitmu	voit-mu
vojcrigg	voj-crigg

(continued)

voldyu	vol-dyu
vopsbyk	vops-byk
vorgdra	vorg-dra
vortho	vor-tho
vorwajel	vor-wa-jel
voumdy	voum-dy
vowgta	vowg-ta
voysblo	voys-blo
vuaby	vu-a-by
vuccorer	vuc-co-rer
vudrawm	vu-drawm
vufgoaja	vuf-goa-ja
vulynri	vu-lyn-ri
vunlusho	vun-lu-sho
vupwoa	vup-woa
vuquatu	vu-qua-tu
vuyeevo	vu-yee-vo
vuyovitt	vu-yo-vitt
waban	wa-ban
wacase	wa-case
wadyt	wa-dyt
wagmy	wag-my
wandyb	wan-dyb
wapawwo	wa-paw-wo
warogi	wa-ro-gi
wavku	wav-ku
wavsnu	wav-snu
weday	we-day
weenipho	wee-ni-pho
wegoka	we-go-ka
wekbu	wek-bu
wemekchi	we-mek-chi
wepri	wep-ri
werho	wer-ho
wetukfi	we-tuk-fi
weutak	weu-tak
whacfida	whac-fi-da
whawjel	whaw-jel
whedcli	whed-cli
whetcrul	whet-crul
whokyohu	whok-yo-hu
whosdeb	whos-deb
whoyhyo	whoy-hyo
whyabdi	whyab-di
whyje	whyje
whyshiv	why-shiv
wiabko	wi-ab-ko
widgawa	wid-ga-wa
wieci	wie-ci
wiefu	wie-fu
wifadal	wi-fa-dal
wifvenod	wif-ve-nod
wigawoz	wi-ga-woz
wigdosim	wig-do-sim
wijwoa	wij-woa
wimac	wi-mac
wimosu	wi-mo-su
winect	wi-nect
wirro	wir-ro
witlugdu	wit-lug-du
wivtuje	wiv-tuje
wiyeiju	wi-yei-ju
wiyicyn	wi-yi-cyn
wiyijko	wi-yij-ko
wizpio	wiz-pi-o
wobloyky	wo-bloyk-y
woimnav	woim-nav
woiwue	woi-wue
wojyu	woj-yu
woknut	wo-knut
wonba	won-ba
wonwijmy	won-wij-my
wopho	wo-pho
wopku	wop-ku
worlegig	wor-le-gig
wovru	wov-ru
woyfum	woy-fum
woytaict	woy-taict
wrahiud	wra-hi-ud
wrahok	wra-hok
wrarjhu	wrarj-hu
wreasgo	wreas-go
wregy	wreg-y
wreng	wreng
wrerry	wrer-ry
wrijtrur	wrij-trur
wrocpish	wroc-pish
wroije	wroije
wruvpi	wruv-pi
wryfsru	wryfs-ru
wuajqui	wu-aj-qui
wuavsyo	wu-av-syo
wucuca	wu-cu-ca
wupkuamu	wup-ku-a-mu
wurctivi	wurc-ti-vi
wuyeawed	wu-yea-wed
wybylel	wy-by-lel
wyclinoc	wy-cli-noc
wygdrel	wyg-drel
wykya	wyk-ya
wyojyilm	wyoj-yilm
wyosevu	wyo-se-vu
wyrahu	wy-ra-hu
wyril	wy-ril
wyvotded	wy-vot-ded
yacrad	yac-rad
yafduhev	yaf-du-hev
yafrig	ya-frig
yaicjopo	yaic-jo-po
yaisi	yai-si
yaiyew	yai-yew
yajtuv	yaj-tuv
yakca	yak-ca
yakvacgo	yak-vac-go
yakyafa	yak-ya-fa
yamlovna	yam-lov-na
yamyen	ya-myen
yands	yands
yapegu	ya-pe-gu
yarjamju	yar-jam-ju
yasbut	yas-but
yashgo	yash-go
yasspka	yassp-ka
yayigca	ya-yig-ca
yayjint	yay-jint
yebjo	yeb-jo
yeceag	ye-ceag
yecwrur	yec-wrur
yefanaki	ye-fa-na-ki
yefeit	ye-feit
yefhy	yef-hy
yefleoka	ye-fle-o-ka
yegoce	ye-goce
yejut	ye-jut
yekcoghu	yek-cog-hu
yekcu	yek-cu
yekjiva	yek-ji-va
yenrit	yen-rit

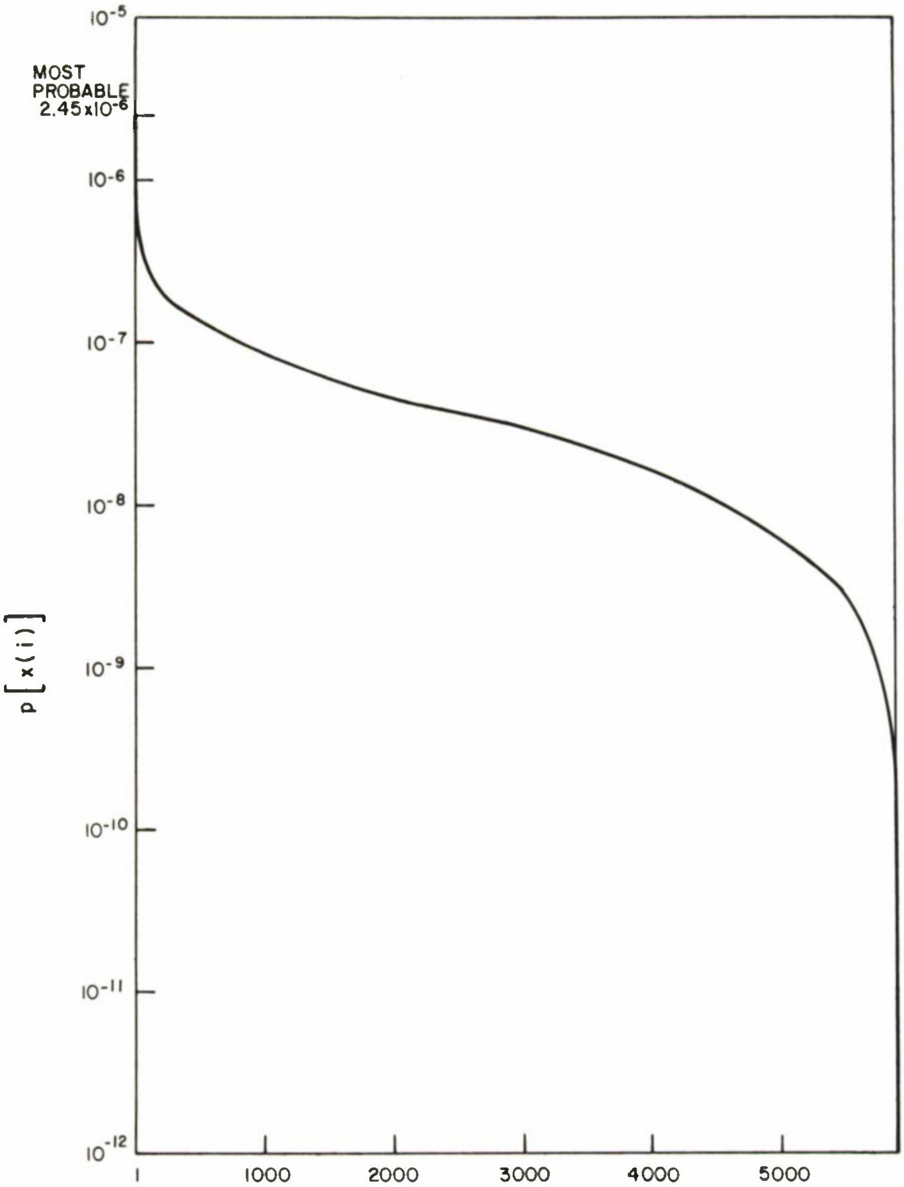
2000 RANDOM WORDS (concluded)

yepros	yep-ros
yerelifs	ye-re-lifs
yesfryac	yes-fryac
yesps	yesps
yetmawoi	yet-ma-woi
yevon	ye-von
yewlibol	yew-li-bol
yewnshlo	yewn-shlo
yexpli	yex-pli
yibfeem	yib-feem
yibluv	yib-luv
yibnen	yib-nen
yiccoha	yic-co-ha
yictryti	yict-ry-ti
yidbogdo	yid-bog-do
yidjetli	yid-jet-li
yiemtak	yiem-tak
yigfrale	yig-frale
yijwis	yij-wis
yikeej	yi-keej
yiklo	yik-lo
yilak	yi-lak
yilvdu	yilv-du
yipweksa	yip-weks-a
yitjeyef	yit-je-yef
yitnesju	yit-nes-ju
yivgadso	yiv-gads-o
yivos	yi-vos
yivvi	yiv-vi
yoarmoi	yoar-moi
yobcruro	yob-cru-ro
yocefkan	yo-cef-kan
yocgi	yoc-gi
yocuct	yo-cuct
yogieth	yo-gieth
yogway	yog-way
yombo	yom-bo
yomciub	yom-ci-ub
yonmeg	yon-meg
yopojvob	yo-poj-vob
yororcdi	yo-rorc-di
yotylica	yo-ty-li-ca
yovshmu	yov-shmu
yoyweufa	yoy-weu-fa
yuebkovu	yueb-ko-vu
yuetyo	yue-tyo
yufugha	yu-fu-gha
yugmopyz	yug-mo-pyz
yuhienfa	yu-hien-fa
yuitdib	yu-it-dib
yunny	yunn-y
yuondyd	yu-on-dyd
yupshty	yup-shty
yupwijat	yup-wi-jat
yureppri	yu-repp-ri
yurytha	yu-ry-tha
yuyitboc	yu-yit-boc
zaney	za-ney
zatshdu	zatsh-du
zatvel	zat-vel
zefphlie	zef-phlie
zeyel	ze-yel
zodagbo	zo-dag-bo
zujuce	zu-juce
zwiufe	zwi-ufe
zwudrul	zwu-drul
zyipunpa	zy-i-pun-pa
zylocwyt	zy-loc-wyt

APPENDIX V

STATISTICS

In Section IV complete data was presented for an analysis of random words of eight letters. The two figures in this appendix are the probability distribution curves for words of six and ten letters, similar to that shown in figure 4 for eight letter words.



IB-45,670

Figure 6. Distribution of Probabilities of 5893 Six Letter Words

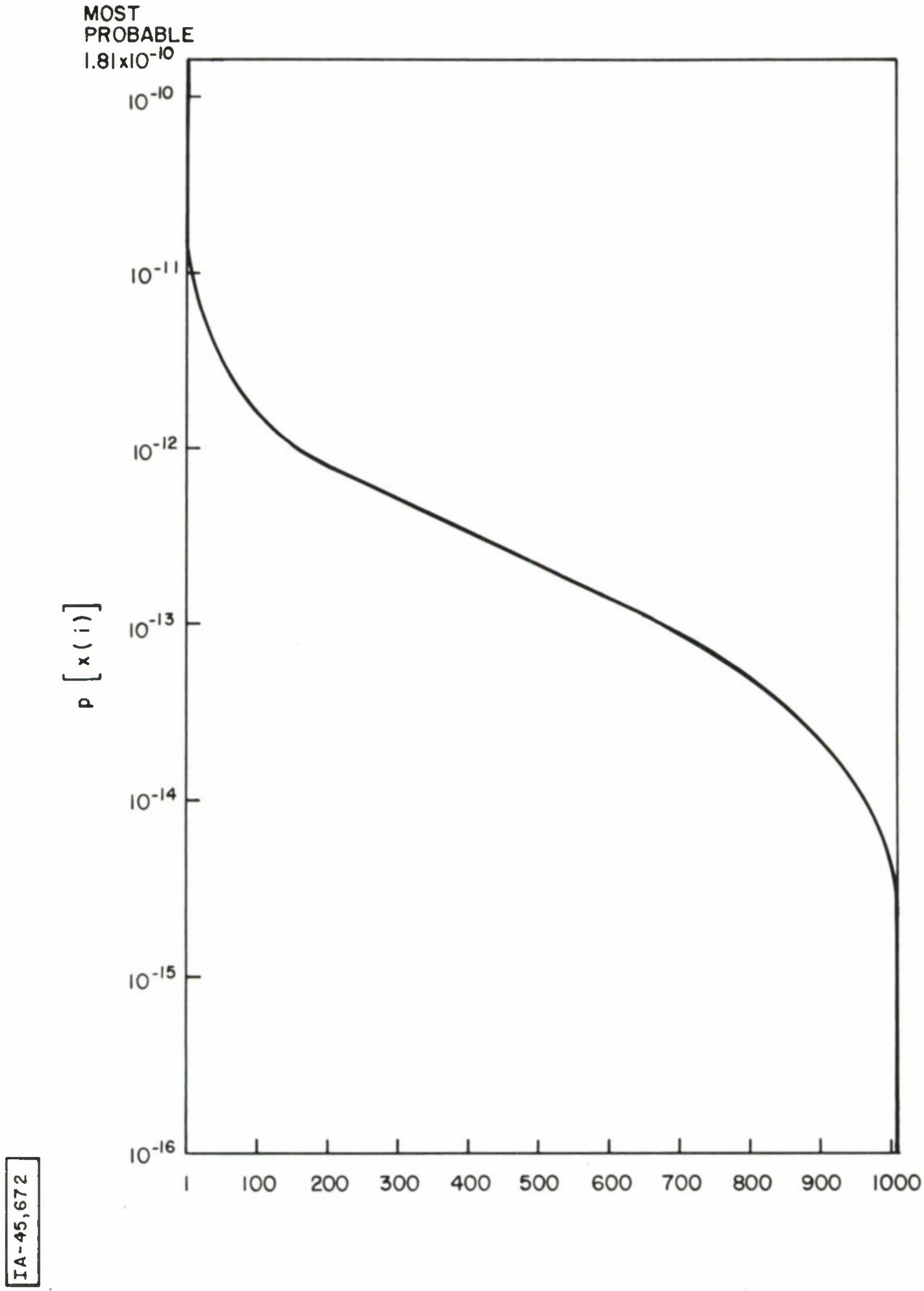


Figure 7. Distribution of Probabilities of 1039 Ten Letter Words

APPENDIX VI

DOCUMENTATION

The documentation on the programs contained in this appendix is in the form of Multics Programmers' Manual command and subroutine descriptions. [3] The individual write-ups are in alphabetical order.

Name: columns, col

The columns command will reformat a segment consisting of short lines into columns that read vertically down the page. The number of columns that will appear depends on the length of the longest line in the segment and the length of the output line. The reformatted segment is printed on the terminal with blank lines between pages, or can be stored for dprinting.

Usage

columns pathname -control_args-

1) pathname — Name of the segment to be reformatted. Lines in the segment may be of any length up to 132 characters, and all lines need not be the same length.

2) control_args are one or more of the following:

-segment, -sm — If present, this argument specifies that the reformatted segment will be stored in a segment called pathname.columns, in a format suitable for dprint. If this argument does not appear, output will be printed on the terminal.

-line_lenth nn, -ll nn
If present, nn is the length of line to be used for output. This must be a number in the range 1 to 132. If this argument is missing, the length of line used will be 132 for the -segment option, or the length of the terminal output line if -segment is not specified. If the user is absentee or file_output is being used, a length of 132 will be used.

-page_length nn, -pl nn
This argument sets the length of the page produced by columns. If output is to the terminal, nn lines will be printed on each page, and blank lines will be used to pad the end of each page up to a total of 66 lines per page. If output is to a segment, a page will be

ejected every nn lines. The default page length is 60 lines. This control argument is incompatible with the -no_pagination control argument.

-adjust, -ad
: specifies that the blank space between the columns is to be adjusted so that the maximum amount of white space appears between the columns. If this control argument does not appear, there will always be one blank space between the columns. Note that the number of columns to be printed is not affected by the use of this control argument. The only effect is to possibly add some blanks between the columns that would otherwise appear at the end of each line of output.

-no_pagination, -npgn
: This argument specifies that the output is not to be paged, i.e., the page length is assumed to be infinite. This argument is useful for terminal output when page breaks are not desired, and avoids extra blank lines at the end of the last page.

Notes

The command first determines how many columns can fit on a line by scanning the segment for the longest line, and using that length as the width of each column. If -adjust is not specified, there will be one blank space between columns, otherwise any extra space will be inserted between the columns. Lines from the input segment that are shorter than the longest line will be left justified in the columns.

When the -segment option is not specified (and -no_pagination is not specified), columns will put 60 lines per page, with 3 blank lines at the top and bottom of each page. When -segment is specified without -no_pagination, there will be 60 lines per page with no blank lines between pages (NEWPAGE characters are inserted into the output segment to eject a page when dprinting).

When dprinting the output segment, the -no_endpage option should be specified for the dprint command. This is necessary to avoid extra blank pages because columns formats its own pages.

Warning

This command expands tabs into spaces properly, but treats backspaces and all other control characters as single characters. Generally, if the segment contains any control characters (other than tabs and newlines) the columns on the output will not line up properly.

Name: convert_word_

This subroutine is used to convert the random word array returned by random_word_ to ASCII.

Usage

```
dcl convert_word_ entry ((0:*) fixed bin, (0:*) bit(1) aligned,
     fixed bin, char(*), char(*));

call convert_word_ (word, hyphenated_word, word_length,
     ascii_word, ascii_hyphenated_word);
```

1) word — Array of random units returned from a previous call to random_word_. (Input)

2) hyphenated_word — Array of bits indicating where hyphens are to be placed, returned from random_word_. (Input)

3) word_length — Number of units in word, returned from random_word_. (Input)

4) ascii_word — This string will contain the word, left justified, with trailing blanks. This string should be long enough to hold the longest word that may be returned. This is normally the value of "maximum" supplied to random_word_. (Output)

5) ascii_hyphenated_word — This string will contain the word, with hyphens between the syllables, left justified within the string. The length of this string should be at least 3*maximum/2+1 to guarantee that the hyphenated word will fit. (Output)

Entry: convert_word_$no_hyphens

This entry can be used to obtain the ASCII form of a random word without the hyphenated form.

Usage

```
dcl convert_word_$no_hyphens ((0:*) fixed bin, fixed bin,
     char(*));

call convert_word_$no_hyphens (word, word_length, ascii_word);
```

Arguments are the same as above.

Name: convert_word_

This subroutine facilitates printing of the hyphenated word returned from a call to hyphenate_.

Usage

```
dcl convert_word_char_ entry (char(*), (*) bit(1) aligned, fixed
     bin, char(*) varying);

call convert_word_char_ (word, hyphens, last, result);
```

1) word — This string is the word to be hyphenated. (Input)

2) hyphens — This is the array returned from a call to hyphenate_ that marks characters in word after which hyphens are to be inserted. (Input)

3) last — This is the status code returned from hyphenate_. If negative, the result will be the original word, unhyphenated, with ** following it. If positive, the word will be returned hyphenated, but with an asterisk preceding the last'th character. If zero, the word will be returned hyphenated without any asterisks. (Input)

4) result — This string contains the resultant hyphenated word. (Output)

Name: digram_table_compiler, dtc

This command compiles a source segment containing the digrams for the random word generator and produces an object segment with the name "digrams".

Usage

digram_table_compiler pathname -option-

1) pathname — is the pathname of the source segment. If the suffix ".dtc" does not appear, it will be assumed. Regardless of the name of the source segment, the output segment will always be given the name "digrams" and will be placed in the working directory.

2) -option- — may be the following:

-list, -ls — lists the compiled table on the terminal. The table will be printed in columns to fit the terminal line length. If file_output is being used, lines will be 132 characters long.

-list n, -ls n — lists the table as above, but uses n as the number of columns to print. Each column occupies 14 positions, thus a value of 5 will cause 5 columns to be printed, each line being 70 characters long. This option is useful when file_output is being used, so that the lines produced are not too long to fit on the terminal to be used to print the output file.

Notes

The compiler makes an attempt to detect inconsistent combinations of attributes, as well as syntax errors. If an error is encountered during compilation, processing of the source segment will continue if possible. The digrams segment in case of an error will be left in an undefined state.

During compilation, the ALM assembler is used. At that point the letters "ALM" will be printed on the terminal. If compilation was successful, no other messages should appear.

The listing produced by digram_table_compiler is in a format suitable for printing on the terminal -- not for dprinting. This is because blank lines are used for page breaks, instead of the "new page" character as recognized by dprint.

Syntax

The syntax of the source segment is specified below. Spaces are meaningful to this compiler and a space is only allowed where specified as <space>. The new line character is indicated as <new line>.

```
<digram table>::= <unit specs>;[<new line>]...<digram specs>$
<unit specs>::= <unit spec>[<delim><unit spec>]...
<digram specs>::= <digram spec>[<delim><digram spec>]...
<delim>::= ,[<new line>]|<new line>
<unit spec>::= <unit name>[<not begin syllable>[<no final split>]]
<digram spec>::= [<begin><not begin><break><prefix>]
                 <unit name><unit name>[<suffix>[<end>[<not end>]]]
<unit name>::= <letter>[<letter>]
<letter>::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<not begin syllable>::= <bit>
<no final split>::= <bit>
<begin>::= <bit>
<not begin>::= <bit>
<break>::= <bit>
<prefix>::= <space>|-
<suffix>::= <space>|-|+
<end>::= <bit>
<not end>::= <bit>
<bit>::= <space>|1
```

The first part of the <digram table> consists of definitions of the various units that are to be used and their attributes. The units are defined as one or two-letter pairs, and the order in which they are defined is unimportant. For each unit, the attributes <not begin syllable> and <no final split> may be specified. In addition, if <unit name> is a, e, i, o, or u, the "vowel" attribute is set. If the unit is y, the "alternate vowel" attribute is set. A <bit> is assumed to be zero if specified as <space>, or one if specified as 1.

The second part of <digram table> specifies all possible pairs of units and the attributes for each pair. The order in which these pairs must be specified depends on the order of the <unit specs> as follows:

Number the <unit spec>s from 1 to n in the order in which they appeared in <unit specs>. The first <digram spec> must consist of the pair of units numbered (1,1), the second <digram spec> is the pair (1,2), etc., and the last <digram spec> is the pair (n,n). All pairs must be specified, i.e., there must be n*n <digram spec>s. The <bit>s preceding or following each pair set the attributes for that pair as shown. The <prefix> and <suffix> indicators are set to 1 if specified as "-". If <suffix> is specified as "+", the "illegal pair" indicator will be set, and no other attributes may be specified for that <digram spec>.

Example

The following is a very short example of a <digram table>. Only four units are defined, "a", "b", "sh" and "e". The letter "e" is given the "no final split" attribute, the pair "aa" is given "illegal pair", the pair "ae" is given the "not begin", "break", and "not end" attributes, etc.

```
a,b,sh,e 1;
aa+,ab,ash, 11 ae  1
ba, 1  bb, 11 bsh  1,be
sha, 11 shb  1,shsh+,she,ea,eb,esh,ee
$
```

Assume the above segment was named "dt.dtc". Below is an example of the command used to compile and list the table produced for dt.

```
digram_table_compiler dt -ls
ALM

     1 a   0010      2 b   0000      3 sh 0000      4 e   0110

  000   aa +00   000   ba   00   000 sha   00   000   ea   00
  000   ab   00  010   bb   00   011 shb   01   000   eb   00
  000   ash 00   011   bsh 01    000 shsh+00    000   esh 00
  011   ae   01  000   be   00   000 she   00   000   ee   00
```

The first line of output lists the individual units. The number preceding the unit is the unit index. The four bits following the unit are respectively:

not begin syllable
no final split
vowel
alternate vowel

Following the unit specifications are the digram specifications. Preceding each digram are three bits and a space (or possibly a "-") with meanings corresponding to those specified in the source segment as follows:

begin
not begin
break
prefix (if "-" appears)

Immediately following each digram is a field which may be blank, "-", or "+". If "+", the "illegal pair" flag is set. Otherwise, the meaning of the "-" and following two bits are as follows:

suffix (if "-" appears)
end
not end

Name: print_digram_table, pdt

This entry merely prints the digram table on the terminal, assuming that it has already been compiled successfully. The segment "digrams" is assumed to be located in the working directory.

Usage

print_digram_table -n-

1) n is the number of columns in which to print the table. If not specified, the maximum number of columns that will fit in the terminal line will be used. Each column occupies 14 positions. If file_output is being used, the terminal line width is assumed to be 132.

Notes

This entry performs the same function as the -list option of digram_table_compiler.

Name: generate_word_

This subroutine returns a random pronounceable word as an ASCII character string. It also returns the same word split by hyphens into syllables as an aid to pronunciation.

Usage

```
declare generate_word_ entry (char(*), char(*), fixed bin, fixed
     bin);

call generate_word_ (word, hyphenated_word, min, max);
```

1) word — is the random word, padded on the right with blanks. This string must be long enough to hold the word (at least as long as max). (Output)

2) hyphenated_word — is the same word split into syllables. The length of this string must be greater than max to allow for the hyphens. A length of 3*max/2 + 1 will always be sufficient. (Output)

3) min — is the minimum length of the word to be generated. This value must be greater than 3 and less than 21. (Input)

4) max — is the maximum length of the word to be generated. The actual length of the word will be uniformly random between min and max. The value of max must be greater then or equal to min, and less than 21. (Input)

Note

Each call to generate_word_ should produce a different random word, regardless of when the call is made. However, as with any random generator, there is no guarantee that there will be no duplicates. The probability of duplication is greater with shorter words.

Entry: generate_word_$init_seed

This entry allows the user to specify a starting seed for generating random words. If a seed is specified, the exact same sequence of random words will always be generated on subsequent calls to generate_word_ providing the same values of min and max are specified. If this entry is not called in a process, the value of the clock is used as the initial seed on the first call to generate_word_, thereby "guaranteeing" different sequences of words in different processes.

Usage

```
declare generate_word_$init_seed entry (fixed bin(35));

call generate_word_$init_seed (seed);
```

1) seed — is the initial seed value. If zero, the system clock will be used as the seed. (Input)

Name: generate_words, gw

This command will print random pronounceable "words" on the user´s terminal.

Usage

generate_words -control_args-

1) control_args may be selected from the following:

nwords — is the number of words to print. If not specified, one word is printed.

-min n — specifies the minimum length, in characters, of the words to be generated.

-max n — specifies the maximum length of the words to be generated.

-length n, -ln n — specifies the length of the words to be generated. If this argument is specified, all words will be this length, and -min or -max may not be specified.

-hyphenate, -hph — causes the hyphenated form (divided into syllables) of each word to be printed alongside the original word.

-seed SEED — On the first call to generate_words in a process, the system clock is used to obtain a starting "seed" for generating random words. This seed is updated for every word generated, and subsequent values of the seed depend on previous values (in a rather complex way). If the -seed argument is specified, SEED must be a positive decimal integer. For a given value of SEED, the sequence of random words will always be the same providing the same length values are specified. When no -seed argument is specified, the last value of the updated seed from the previous call to generate_words will be used. To revert back to using the system clock as the seed, specify a zero value for SEED, i.e., -seed 0.

Notes

If neither -min, -max, nor -length are specified, the defaults are -min 6 and -max 8. In all other cases, the defaults are -min 4 and -max 20.

If -length is not specified, the lengths of the random words will be uniformly distributed between min and max. Words generated are printed one per line, with the hyphenated forms, if specified, lined up in a column alongside the original words.

Command/Active Function

Name: get_line_length, gll

This command or active function returns the current length of a line on the stream user_output. When used as a command, the line length is printed on the terminal. When used as an active function, the line length is returned as a character string.

Usage

```
get_line_length
[get_line_length]
```

Notes

The length of the line is obtained from the modes supplied to the tw_ IOSIM. If user_output is attached through some other interface module (such as file_ when using file_output) the line length may be undefined. If the line length can not be obtained, a default length of 132 will be returned.

Name: get_line_length_

This subroutine returns the current length of a line on the stream user_output.

Usage

```
dcl get_line_length_ entry returns (fixed bin);

ll = get_line_length_ ();
```

1) ll   is the length of the line.

Notes

The length of the line is obtained from the modes supplied to the tw_ IOSIM. If user_output is attached through some other interface module (such as file_ when using file_output) the line length may be undefined. If the line length can not be obtained, a default length of 132 will be returned.

Names: hyphen_test, ht

This command uses the random word generator (the same one used by generate_words) to divide words into syllables. Words are printed on the terminal with hyphens between the syllables.

Usage

hyphen_test -control_arg- -word1- ... -wordn-

1) control_arg — may be -probability (-pb), specifying that the probability of each of the words that follows be printed alongside the hyphenated word.

2) wordi — are one or more words to be hyphenated. A word may consist of three to twenty alphabetic characters, only the first of which may be uppercase.

Notes

The control argument may appear anywhere in the command line. However, it only applies to words that follow. Words preceding the option will be hyphenated but no probabilities will be calculated.

If a word contains any illegal characters, or is not of three to twenty characters in length, the word will be printed unhyphenated, followed by **.

If the word could not be completely hyphenated because it was considered unpronounceable, an asterisk (*) will be printed out in front of the first character that was not accepted. The part of the word before the asterisk will be properly hyphenated.

The calculated probability is the probability that the word would have been generated by generate_words, assuming generate_words was requested to generate a word of that length only. If a range of lengths is requested of generate_words, each length has equal probability. For example, if generate_words is called to generate words of 6, 7, or 8 characters, there is a 33% probability that a given word will have 8 characters. If hyphen_test is then asked to calculate the probability of a given 8 letter word, that probability should be divided by 3 to obtain the correct probability for the case of three possible lengths.

Name: hyphenate_

This subroutine attempts to hyphenate a word into syllables.

Usage

```
dcl hyphenate_ entry (char(*), (*) bit(1) aligned, fixed bin);

call hyphenate_ (word, hyphens, code);
```

1) word — This is a left justified ASCII string, 3 to 20 characters in length. This string must contain all lowercase alphabetic characters, except the first character may be uppercase. Trailing blanks are not permitted in this string. (Input)

2) hyphens — This array will contain a "1"b for every character in the word that is to have a hyphen following it. (Output)

3) code — This is a status code, as follows:

   0 word has been successfully hyphenated.
   -1 word contains illegal (non alphabetic or uppercase) characters.
   -2 word was not from three to twenty characters in length.

   Any positive value of code means that the word couldn't be completely hyphenated. In this case, code is the position of the first character in word that was not acceptable. The part of the word before code will be properly hyphenated. (Output)

Notes

This subroutine uses random_word_ to provide the hyphenation. It does this by calling random_word_$give_up and supplying its own version of random_unit and random_vowel that return specified units (of the particular word to be hyphenated) instead of random units.

The word supplied to hyphenate_ is first transformed into units by translating pairs of letters into single units if a 2-letter unit is defined for the pair, and then by translating the remaining single letters into units. See the write-ups of random_word_ and random_unit_ for a description of units. If any units of the word are refused by random_word_, hyphenate tries to determine if the refused unit was a 2-letter unit. If this is the case, then the 2-letter unit is broken into two 1-letter units and random_word_ is called again. In rare cases, hyphenate_ is not able to determine which 2-letter unit is at fault, and will return a status code indicating that the word is unpronounceable, when, in fact, it could have been properly divided by breaking up a 2-letter unit.

Entry: hyphenate_$probability

This entry returns information as above, but also supplies the probability of the word having been generated at random by generate_word_ or random_word_generator_. The assumption is made that generate_word_ or random_word_generator_ was asked to supply a word of exactly the same length as the word given to hyphenate_, rather than a range of lengths. If a range of lengths was asked of generate_word_, the probability must be divided by the number of different lengths (all lengths are equally probable).

Usage

```
dcl hyphenate_ entry (char(*), (*) bit(1) aligned, fixed bin,
     float bin);

call hyphenate_ (word, hyphens, code, probability);
```

1) to 3) are as above.

4) probability is the probability as defined above. (Output)

Notes

If the supplied word is illegal (i.e. code is not zero), the probability will be returned as zero.

Entry: hyphenate_$debug_on, hyphenate_$debug_off

These entries set and reset a switch that causes hyphenate_$probability to print, on user_output, all units (see random_word_ and random_unit_ for a description of units) that are illegal in a given position of the word. This entry is useful for debugging a digram table for random_word_.

Usage

```
dcl hyphenate_$debug_on entry;
dcl hyphenate_$debug_off entry;

call hyphenate_$debug_on;
call hyphenate_$debug_off;
```

Notes

An example of the output produced is as follows. The assumption is that hyphenate_$probability is invoked by the hyphen_test command using the -probability option.

```
hyphenate_$debug_on
hyphen_test -probability fish
x,ck,f; b,c,d,f,g,h,j,k,m,n,p,s,t,v,w,x,y,z,ch,gh,ph,
rh,sh,th,wh,qu,ck,i; i,rh,wh,qu,sh;
fish  6.04127576e-5
```

In the above example, the units x and ck are shown to have been illegal as the first unit of the word, and the unit f, (underlined) is the first unit of the word that was accepted. All other units that were not printed are legal as the first unit of the word. Following the semicolon after f are the units that are illegal in the second position of the word (assuming that f is the first unit). Then i is shown as the legal unit that is taken from the word "fish". This repeats for each position of the word, ending in the legal unit sh (note only one underline).

If the supplied word is illegal, the last underlined letter in the output is (usually) the letter that was not accepted. In cases where hyphenate_ has to split up a 2-letter unit, the word will be shown to start over from the beginning.

Name: random_unit_

This subroutine provides a random unit number for random_word_ based on a standard distribution of a given set of units. It is referenced by the generate_word_ subroutine as an entry value that is passed in the call to random_word_. This subroutine assumes that the digram table being used by random_word_ is a standard table. The digram table itself is not referenced by this subroutine.

Usage

```
declare random_unit_ entry (fixed bin);

call random_unit_ (unit);
```

1) unit is a number from 1 to 34 that corresponds to a particular unit as listed in Notes below. (Output)

Notes

The table below contains the units that are assumed specified in the digrams supplied to random_word_. Shown in the table are the unit number, the letter or letters that unit represents, and the probability of that unit number being generated.

1	a	.04739	8	h	.02844	15	o	.04739	22	w	.03792	29	rh	.00474
2	b	.03792	9	i	.04739	16	p	.02844	23	x	.00474	30	sh	.00948
3	c	.05687	10	j	.03792	17	r	.04739	24	y	.03792	31	th	.00948
4	d	.05687	11	k	.03792	18	s	.03792	25	z	.00474	32	wh	.00474
5	e	.05687	12	l	.02844	19	t	.04739	26	ch	.00474	33	qu	.00474
6	f	.03792	13	m	.02844	20	u	.02844	27	gh	.00474	34	ck	.00474
7	g	.03792	14	n	.04739	21	v	.03792	28	ph	.00474			

Entry: random_unit_$random_vowel

This entry returns a vowel unit number only.

Usage

```
declare random_unit_$random_vowel (fixed bin);
```

```
call random_unit_$random_vowel (unit);
```

1) unit   As above. (Output)

Notes

Below are listed the vowel units and their distributions.

1	a	.167
5	e	.250
9	i	.167
15	o	.167
20	u	.167
24	y	.083

Entry: random_unit_$probabilities

This entry returns arrays containing the probabilities of the units as listed in the table on the previous page. This entry is provided for hyphenate_$probability and any other program that might require this information. The probabilities must be computed when this entry is called, so it is suggested that the call be made only once per process and the values saved in internal static storage.

Usage

```
declare random_unit_$probabilities entry ((*) float bin,
     (*) float bin);

call random_unit_$probabilities (unit_probs, vowel_probs);
```

1) unit_probs   This array contains the probabilities of the individual units assuming the random_unit_ entry is called to generate the random units. The value of unit_probs(i) is the probability of unit(i). (Output)

2) vowel_probs   This array contains the probabilities of the units when random_vowel is called. Since there are only 6 vowels, most of these values will be zero. (Output)

Notes

A future version of random_unit_ may use different units with different probabilities. The size of the two arrays must be large enough to hold the maximum number of values that may be returned by random_unit_ (which is currently 34). Programs should not depend on the unit_index-to-letter correspondence as shown in the table. This information can be obtained by using the include file digram_structure.incl.pl1.

Name: random_word_

This routine returns a single random pronounceable word of specified length. It is called by generate_word_, and allows the caller to specify the particular subroutines to be used to generate random units. For users desiring random words with an English-like distribution of letters, generate_word_ should be used.

Usage

```
dcl random_word_ entry ((0:*) fixed, (0:*) bit(1) aligned, fixed,
     fixed, entry, entry);

call random_word_(word, hyphens, char_length, unit_length,
     random_unit, random_vowel);
```

1) word — The random word will be stored in this array starting at word(1) (word(0) will always be 0). The numbers stored will correspond to a "unit index" as described in Notes below. This array must have a length at least equal to the value of "char_length". Unused positions in this array, up to word(char_length), will be set to zero. (Output)

2) hyphens — This array must be of length at least "char_length". A bit on in a position of this array indicates that the corresponding unit in "word" (including the very last unit) is the last unit of a syllable. (Output)

3) char_length — Length of the word to be generated, in characters. (Input)

4) unit_length — This is the length of the generated random word in units, i.e., the index of the last non-zero entry in the "word" array. The actual length of the word in equivalent characters will be the value of char_length. (Output)

5) random_unit — This is the routine that will be called by random_word_ each time a random_unit is needed. The random_unit

routine is declared as follows:

```
dcl random_unit entry (fixed bin);
```

where the value returned is a unit index between 1 and n_units. If an English-like distribution of letters is desired, the "random_unit_" subroutine may be specified here. See Notes below. (Input)

6) random_vowel
This is the routine called by random_word_ when a vowel unit is required. This routine must return the index of a unit whose "vowel" or "alternate_vowel" bits are on. See Notes below. This routine is declared as follows:

```
dcl random_vowel entry (fixed bin);
```

If desired, the subroutine "random_unit_$random_vowel" may be specified in this place. (Input)

Notes

The word array can be converted into characters by calling convert_word_.

In order to use random_word, a digram table, contained in a segment named "digrams", must be available in the search path. This table can be created by the digram_table_compiler.

If the user supplies his own versions of random_unit and random_vowel, these subroutines will have to supply legal units that are recognized by the random_word_ subroutine. The include file "digram_structure.incl.pl1" can be used to reference the digram table to determine which units are available. If included in the source program, appropriate references to the following variables of interest in "digrams" will be generated:

```
dcl n_units fixed bin defined digrams$n_units;
```

```
dcl letters(0:n_units) char(2) aligned
     based(addr(digrams$letters));
dcl 1 rules(n_units) aligned based(addr(digrams$rules)),
                2 vowel bit(1),
                2 alternate_vowel bit(1),
                .....
```

where:

n_units — is the number of different units.

letters(i) — contains 1 or 2 characters (left justified) for the i'th unit.

rules.vowel(i), rules.alternate_vowel(i) — One of these two bits are set for the units that may be returned by a call to random_vowel.

When random_unit is called, a number from 1 to n_units must be returned. When random_vowel is called, a number from 1 to n_units, where one of the two bits in rules(i) is marked, must be returned.

Entry: random_word_$debug_on

This entry sets a switch in random_word_ that causes printing (on user_output) of partial words that could not be completed. This entry is of interest during debugging of random_word_ or for checking the consistency of the digram table prepared by the user.

Usage

```
dcl random_word_$debug_on entry;

call random_word_$debug_on;
```

Entry: random_word_$debug_off

This entry resets the switch set by debug_on.

Additional notes

The random_word_ subroutine can be used for certain special applications (such as the application used by hyphenate_), and there are certain features that help support some of these applications. The features described below are of little interest to most users.

The first feature allows the caller-supplied random_unit (and random_vowel) subroutine to find out whether random_word_ "accepted" or "rejected" the previous unit supplied by random_unit. Each time random_unit is invoked by random_word_, the value of the argument passed is the index of the previous unit that random_unit_ returned (or zero on the first call to random_unit in a given invocation of random_word_). The sign of the argument will be positive if this last unit was accepted. "Accepted" means that the last unit was inserted into the random word and the word index maintained by random_word_ was incremented. Once a unit is accepted, it is never removed. Thus a positive value of the unit index passed to random_unit means that a unit for the next position of the word is requested.

If the unit index passed to random_unit has a negative sign, the last unit was rejected according to the rules used by random_word_ and information supplied in the digram table. If the unit is rejected, random_word_ does not advance its word index and calls random_unit again for another unit for that same word position. With this information random_unit can keep track of the "progress" of the word being generated.

The feature described above is used by the special random_unit routine provided by hyphenate_. Since the random_unit routine for hyphenate_ is not really supplying random units (but is supplying units of the word to be hyphenated), it must know whether any particular unit is rejected by random_word_. Rejection then implies that the word is illegal according to random_word_ rules.

The second feature allows random_unit to "try" a certain unit without committing that unit to actually be used in the random word. The sign of each unit supplied to random_word_ by random_unit is checked. If the sign of the word is positive, random_word_ will accept or reject the unit according to its rules, and will indicate this on the subsequent call to random_unit.

If the sign of the unit passed to random_word_ is negative, random_word_ will merely indicate (on the subsequent call to random_unit) whether that unit would have been accepted, but it never actually updates the word index. In other words, random_word_ always rejects the unit, but lets random_unit know whether the unit was acceptable.

This latter feature is used by hyphenate_$probability in order to determine which of all possible units are acceptable in a given position of the word. The random_unit routine used by hyphenate_$probability tries all possible units in each word position, and only allows random_word_ to accept the unit that actually appears in that position.

Name: read_table_

This subroutine is the compiler for the digram table for random_word_. It is called by digram_table_compiler.

Usage

```
declare read_table_ entry (ptr, fixed bin(24), returns (bit(1));

flag = read_table_ (source_ptr, bitcount);
```

1) source_ptr is a pointer to the source segment to be compiled. (Input)

2) bitcount is the bit count of the source segment. (Input)

3) flag is "0"b if compilation was successful. It is "1"b if an error was encountered.

Notes

If compilation was successful, the compiled table will be placed in the working directory with the name "digrams". If unsuccessful, the digrams segment may or may not have been created, and may be left in an inconsistent state (i.e., unusable by random_word_). Error messages are printed out on user_output as the errors are encountered, except that file system errors are printed on error_output.

This subroutine uses the ALM assembler for part of its work. As a result, the letters "ALM" will be printed on user_output sometime during the compilation.

APPENDIX VII

MODIFIED SOFTWARE FOR UNIFORM DISTRIBUTION

The following pages contain documentation and source listings for the two modules that have been altered to produce uniformly distributed random words as discussed near the end of Section IV. The two modules are generate_word_ and generate_words. In addition, a listing of the random number generator encipher_ is included for those interested in the algorithm used to obtain random numbers.

Except for generate_words and generate_word_, all other modules are unchanged from those shown in Appendix III and Appendix VI. These two modules have been modified in an upward compatible manner. Thus, when called as described in Appendix VI, they will perform exactly as described. In order to get uniformly distributed words, an additional entry point in generate_word_ and an additional control argument to the generate_words command have been provided.

Name: generate_word_

This subroutine returns a random pronounceable word as an ASCII character string. It also returns the same word split by hyphens into syllables as an aid to pronunciation.

Usage

```
declare generate_word_ entry (char(*), char(*), fixed bin, fixed
     bin);

call generate_word_ (word, hyphenated_word, min, max);
```

1) word — is the random word, padded on the right with blanks. This string must be long enough to hold the word (at least as long as max). (Output)

2) hyphenated_word — is the same word split into syllables. The length of this string must be greater than max to allow for the hyphens. A length of 3*max/2 + 1 will always be sufficient. (Output)

3) min — is the minimum length of the word to be generated. This value must be greater than 3 and less than 21. (Input)

4) max — is the maximum length of the word to be generated. The actual length of the word will be uniformly random between min and max. The value of max must be greater then or equal to min, and less than 21. (Input)

Note

Each call to generate_word_ should produce a different random word, regardless of when the call is made. However, as with any random generator, there is no guarantee that there will be no duplicates. The probability of duplication is greater with shorter words.

Entry: generate_word_$init_seed

This entry allows the user to specify a starting seed for generating random words. If a seed is specified, the exact same sequence of random words will always be generated on subsequent calls to generate_word_ providing the same values of min and max are specified. If this entry is not called in a process, the value of the clock is used as the initial seed on the first call to generate_word_, thereby "guaranteeing" different sequences of words in different processes.

Usage

```
declare generate_word_$init_seed entry (fixed bin(35));

call generate_word_$init_seed (seed);
```

1) seed — is the initial seed value. If zero, the system clock will be used as the seed. (Input)

Note

If the seed is a small integer, the first few words generated may not be quite as random as one might like, i.e., if 5 is specified for the value of seed the first word generated will be almost the same as when some other small integer is specified.

Entry: generate_word_$uniform

This entry is the same as generate_word_, except that the words produced are uniformly distributed. The probabilities of the words produced by generate_word_ are not all the same. This entry provides words with equal probability. The method used to generate uniformly distributed words results in a speed degradation that is worse with longer words. For eight letter words, factor of at least 10 should be expected. In addition, the set of words that may be produced is not quite as large (although certainly within 90% of the set produced by generate_word_).

Usage

```
declare generate_word_$uniform entry (char(*), char(*), fixed
     bin, fixed bin);
```

```
call generate_word_ (word, hyphenated_word, min, max);
```

Arguments are the same as above.

Name: generate_words, gw

This command will print random pronounceable "words" on the user´s terminal.

Usage

generate_words -control_args-

1) control_args may be selected from the following:

nwords	is the number of words to print. If not specified, one word is printed.
-min n	specifies the minimum length, in characters, of the words to be generated.
-max n	specifies the maximum length of the words to be generated.
-length n, -ln n	specifies the length of the words to be generated. If this argument is specified, all words will be this length, and -min or -max may not be specified.
-hyphenate, -hph	causes the hyphenated form (divided into syllables) of each word to be printed alongside the original word.
-seed SEED	On the first call to generate_words in a process, the system clock is used to obtain a starting "seed" for generating random words. This seed is updated for every word generated, and subsequent values of the seed depend on previous values (in a rather complex way). If the -seed argument is specified, SEED must be a positive decimal integer between 1 and 9999. For a given value of SEED, the sequence of random words will always be the same providing the same length values are specified. When no -seed argument is specified, the last value of the updated seed from the previous call to generate_words will be used. To revert back to

using the system clock as the seed, specify a zero value for SEED, i.e., -seed 0.

-uniform, -uf
: The probability of the words produced by this command is not the same for all words, i.e., some words are more probable than others. This option changes the way in which the words are generated so that all words are equally probable. The number of different words that can result when this option is specified is a little smaller than the number of words that may be produced without this option. (One result of using this option is that the letter "q" will never appear.) The use of this argument results in a speed degradation by a factor of about 10 for eight letter words, and greater for longer words. This argument is useful when it is desirable to generate words whose probabilities are equal.

Notes

If neither -min, -max, nor -length are specified, the defaults are -min 6 and -max 8. In all other cases, the defaults are -min 4 and -max 20.

If -length is not specified, the lengths of the random words will be uniformly distributed between min and max. Words generated are printed one per line, with the hyphenated forms, if specified, lined up in a column alongside the original words.

```
ASSEMBLY LISTING OF SEGMENT >user_dir_dir>AOruid>Gasser>p>encipher_.alm
ASSEMBLED ON:        08/19/75  1038.4 edt Tue
OPTIONS USED:        ls  symbols  new_call  new_object
ASSEMBLED BY:        ALM Version 4.5, September 1974
ASSEMBLER CREATED:   04/29/75  1343.9 edt Tue

                                  """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
                                  " This procedure enciphers an arrya of double words, i.e., fixed bin(71),
                                  " using the key that is provided.  It has entries to both encipher and decipher:
                                  "
                                  "          call encipher_(key,input_array,output_array,array_length)
                                  "
                                  "          call decipher_(key,input_array,output_array,array_length)
                                  "
                                  " where:   key                   is fixed bin(71) key for coding
                                  "          input_array(array_length) is fixed bin(71) array
                                  "          output_array(array_length) is fixed bin(71) array
                                  "          array_length          is fixed bin(17) length (double words) of array
                                  "
                                  "          Coded 1 April 1973 by Roger R. Schell, Major, USAF
                                  """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

    000000                                   followon
    000000              000000               entry      encipher_
    000000              000004               entry      decipher_

    000000              000002               equ        key,2
    000000              000004               equ        input_array,4
    000000              000006               equ        output_array,6
    000000              000010               equ        array_length,8

                                  "
                                  "          Entry to encipher
                                  "

    000000                        encipher_:
    000000  aa   000100 6270 00              push
    000001  aa  7 00040 2721 20
    000002  aa  0 00006 3701 20              epplp      ap|output_array,*    "LP -> cipher text
    000003  0a   000007 7100 00              tra        setup_keys

                                  "
                                  "          Entry to decipher
                                  "

    000004                        decipher_:
    000004  aa   000100 6270 00              push
    000005  aa  7 00040 2721 20
    000006  aa  0 00004 3701 20              epplp      ap|input_array,*     "set LP -> cipher text

    000007                        setup_keys:
                                                                    "First create internal keying variables
                                  "Use Tausworth pseudo-random number generator on key

    000007              000013               equ        shift,11             "Shift for generator
    000007              000044               equ        size,36              "Word size used for generator

                                             tempd      variables(12)        "Internal keying variables

    000007  aa   000000 6260 00              eax6       0                    "loop index in x6
```

```
000010  aa  0 00002 2371 20           ldaq     ap|key,*           "Start with input key

000011                          mask_loop:
                                                                  "Create masks
000011  aa  6 00050 7571 16           staq     variables,6        "save copy of generator seed
000012  aa    000013 7720 00          qrl      shift              "Now generate pseudo-random number
000013  aa    000013 7710 00          arl      shift
000014  aa  6 00050 6771 16           eraq     variables,6
000015  aa  6 00050 7571 16           staq     variables,6
000016  aa    000031 7360 00          qls      size-shift
000017  aa    000031 7350 00          als      size-shift
000020  aa  6 00050 6771 16           eraq     variables,6
000021  aa  6 00050 7571 16           staq     variables,6        "Save result

000022  aa    000002 6260 16          eax6     2,6
000023  aa    000022 1060 03          cmpx6    18,du              "Generate 9 double words
000024  0a    000011 6010 00          tnz      mask_loop

                                "
                                "Next create 7-bit shift variables

000025  aa    000000 6260 00          eax6     0
000026  aa    000013 7730 00          lrl      11                 "First 7 bits to upper A-reg
000027  aa    000000 6200 00          eax0     0                  "Zero for clearing half word
000030                          shift_loop:
000030  aa  6 00070 7551 16           sta      variables+A1,6     "Upper A-reg is shift variable
000031  aa  6 00070 4401 16           sxl0     variables+A1,6     "Zero lower half word
000032  aa    000007 7370 00          lls      7
000033  0a    000116 3750 00          ana      =o000177777777     "Save 7 bits in upper A-reg
000034  aa    000001 6260 16          eax6     1,6
000035  aa    000007 1060 03          cmpx6    7,du               "Generate 7 shift variables
000036  0a    000030 6010 00          tnz      shift_loop

                                "
                                "     Now that we have needed variables, aply the cipher
                                "

                                "Declaration of offsets of keying variables
000037            000000              equ      C0,0               "Initial cipher text from key
000037            000002              equ      M1,2               "Mask variables
000037            000004              equ      M2,4
000037            000006              equ      M3,6
000037            000010              equ      M4,8
000037            000012              equ      M5,10
000037            000014              equ      M6,12
000037            000016              equ      M7,14
000037            000020              equ      A1,16              "Amount of shift -- as address
000037            000021              equ      A2,17
000037            000022              equ      A3,18
000037            000023              equ      A4,19
000037            000024              equ      A5,20
000037            000025              equ      A6,21
000037            000026              equ      A7,22

000037  aa  0 00010 7251 20           lxl5     ap|array_length,*  "Get length (double words)
000040  aa    777777 6250 15          eax5     -1,5               "Check for zero or negative
000041  0a    000103 6040 00          tmi      return
000042  aa    000000 6260 00          eax6     0                  "X6 is index into arrays
```

```
000043  aa  6 00050 3521 00              eppbp     variables+C0          "Initial cipher text from key
000044                          cipher_loop:
000044  aa  2 00000 2371 00              ldaq      bp|0
                                "First compute select function
000045  aa  6 00075 7771 20              llr       variables+A6,*
000046  aa  6 00064 0371 00              adlaq     variables+M6
000047  aa  6 00076 7771 20              llr       variables+A7,*
000050  aa  6 00066 6771 00              eraq      variables+M7
000051  aa    000000 6210 06             eax1      0,ql                  "Save select function
                                "
                                "Compute value
                                "
000052  aa  2 00000 2371 00              ldaq      bp|0
000053  aa  6 00070 7771 20              llr       variables+A1,*
000054  aa  6 00052 0371 00              adlaq     variables+M1
000055  aa    000010 3010 03             canx1     =o10,du
000056  aa    000002 6010 04             tnz       2,ic
000057  aa  6 00071 7771 20              llr       variables+A2,*
000060  aa  6 00054 6771 00              eraq      variables+M2
000061  aa    000004 3010 03             canx1     =o4,du
000062  aa    000002 6010 04             tnz       2,ic
000063  aa  6 00072 7771 20              llr       variables+A3,*
000064  aa  6 00056 0371 00              adlaq     variables+M3
000065  aa    000002 3010 03             canx1     =o2,du
000066  aa    000002 6010 04             tnz       2,ic
000067  aa  6 00073 7771 20              llr       variables+A4,*
000070  aa  6 00060 6771 00              eraq      variables+M4
000071  aa    000001 3010 03             canx1     =o1,du
000072  aa    000002 6010 04             tnz       2,ic
000073  aa  6 00074 7771 20              llr       variables+A5,*
000074  aa  6 00062 0371 00              adlaq     variables+M5          "AQ contains computed key
000075  aa  4 00000 3521 16              eppbp     lp|0,6                "set BP -> next cipher text autokey
000076  aa  0 00004 6771 76              eraq      ap|input_array,*6
000077  aa  0 00006 7571 76              staq      ap|output_array,*6    "return ciphered value
000100  aa    000002 6260 16             eax6      2,6                   "Increment array offset
000101  aa    777777 6250 15             eax5      -1,5                  "Check for end of array
000102  0a    000044 6050 00             tpl       cipher_loop
000103                          return:
                                "
                                "Clean up the 'dirty blackboard' before returning
000103             005202                bool      rpt,5202              "RPT instruction
000103  0a    000103 2370 00             ldaq      *                     " Load AQ with garbage
000104  aa    000000 6260 00             eax6      0
000105  aa    026200 520202              vfd       8/11,2/0,1/1,7/0,12/rpt,6/2 "RPT instruction
000106  aa  6 00050 7571 16              staq      variables,6           "Overwrite keying variables
000107  aa  7 00042 7101 20              return
                                         end
```

```
COMPILATION LISTING OF SEGMENT generate_word_
Compiled by: Multics PL/I Compiler of July 2, 1975.
Compiled on: 06/19/75  1016.9 edt Tue
    Options: check source

/* This procedure is a modification of the standard subroutine interface to generate random words.
   A change has been made to add the entry point generate_word_$uniform.
   Except for that, the original functioning of generate_word_ is the same.
*/
generate_word_: procedure (word, hyphenated_word, min, max);
 dcl word char(*);
 dcl hyphenated_word char(*);
 dcl min fixed bin;
 dcl max fixed bin;
 dcl (random_unit_, random_unit_$random_vowel) entry (fixed bin);
 dcl convert_word_ entry ((0:*) fixed bin, (0:*) bit(1) aligned,
     fixed bin, char(*), char(*));
 dcl random_word_ entry ((0:*) fixed bin, (0:*) bit(1) aligned,
     fixed bin, fixed bin, entry, entry);
 dcl hyphens (0:20) bit(1) aligned;
 dcl random_word (0:20) fixed bin;
 dcl length_in_units fixed bin;
 dcl random_length fixed bin;
 dcl unique_bits_ entry returns (bit(70));
 dcl encipher_ entry (fixed bin(71), (*) fixed bin (71), (*) fixed bin(71), fixed bin);
 dcl random_unit_stat_$seed fixed bin(71) external;
 dcl saved_seed(1) fixed bin(71) static;
 dcl first_call bit(1) static aligned init("1"b);

 /* On the very first call to this procedure in a process (if the
    init_seed entry was not called), use unique_bits to get a
    random number to initialize the random seed. */

 if first_call then do;
     saved_seed(1), random_unit_stat_$seed = fixed (unique_bits_ ());
     first_call = "0"b;
     end;

 /* encipher the seed to get a random number and the next value of the seed */

 call encipher_ (saved_seed(1), saved_seed, saved_seed, 1);

 /* Get the length of the word desired.
    We calculate this to be a uniformly distributed random number between
    min and max. */

 random_length = mod (abs (fixed (saved_seed(1), 17)), (max - min + 1)) + min;

 /* Get the random word and convert it to characters */

 call random_word_ (random_word, hyphens, random_length, length_in_units, random_unit_, random_unit_$random_vowel);
convert:
 call convert_word_ (random_word, hyphens, length_in_units, word, hyphenated_word);
 return;

/* This entry allows the user to set the seed.  If the seed argument is zero, we
   go back to using the clock value.
*/
generate_word_$init_seed: entry (seed);
 dcl seed fixed bin(35);
```

```
dcl whole_seed fixed bin(71);
dcl 1 half_seeds based (addr (whole_seed)),
      2 (first, second) fixed bin(35);

if seed = 0 then first_call = "1"b;
else do;
  half_seeds = seed;
 random_bits, saved_seed(1), random_unit_stat_$seed = whole_seed;
 first_call = "0"b;
 index = array_size;
 end;
return;
/*
```

```
*/
/* This entry point generates uniformly distributed words.
   It does this by "giving" random_word_ uniformly distributed random
   words and keeps trying until an acceptable word turns up.
   For long words, this may take some time.  If any letter of a
   word is rejected by random_word_, the word is abandoned and another word is tried.

   The random numbers are obtained from an array of 72-bit numbers generated by
   encipher_.  This array is overlayed by 5-bit numbers to give random
   numbers in the range -16 to +15.  The random_unit and random_vowel routines
   below that supply the next random unit number only supply the numbers
   of single-letter units.  This is because it's difficult to obtain
   a uniform distribution of words with double letter units.
   Note that only unit numbers 32 or less are returned.  Thus, if there is
   a single letter unit with a number greater than 32, it will never
   appear.
*/

generate_word_$uniform: entry (word, hyphenated_word, min, max);
 dcl bits_size fixed bin static init (10);
 dcl array_size fixed bin static init (144);
 dcl index fixed bin static init(9999);
 dcl array (array_size) based (addr(random_bits)) fixed bin(4) unaligned;
 dcl random_bits (10) fixed bin (71) static;
 dcl number fixed bin;
 dcl max_number fixed bin;

/* ******** include file digram_structure.incl.pl1 ******* */

dcl  digrams$digrams external;
dcl  digrams$n_units fixed bin external;
dcl  digrams$letters external;
dcl  digrams$rules external;

/* This array contains information about all possible pairs of units */

dcl 1 digrams (n_units, n_units) based (addr (digrams$digrams)),
      2 begin bit (1),                          /* on if this pair must begin syllable */
      2 not_begin bit (1),                      /* on if this pair must not begin */
      2 end bit (1),                            /* on if this pair must end syllable */
      2 not_end bit (1),                        /* on if this pair must not end */
      2 break bit (1),                          /* on if this pair is a break pair */
      2 prefix bit (1),                         /* on if vowel must precede this pair in same syllable */
      2 suffix bit (1),                         /* on if vowel must follow this pair in same syllable */
      2 illegal_pair bit (1),                   /* on if this pair may not appear */
      2 pad bit (1);                            /* this makes 9 bits/entry */

/* This array contains left justified 1 or 2-letter pairs representing each unit */

dcl  letters (0:n_units) char (2) aligned based (addr (digrams$letters));

/* This is the same as letters, but allows reference to individual characters */

dcl 1 letters_split (0:n_units) based (addr (digrams$letters)),
      2 first char (1),
      2 second char (1),
      2 pad char (2);

```

```
/* This array has rules for each unit */

dcl 1 rules (n_units) aligned based (addr (digrams$rules)),
     2 no_final_split bit (1),                     /* can't be the only vowel in last syllable */
     2 not_begin_syllable bit (1),                 /* can't begin a syllable */
     2 vowel bit (1),                              /* this is a vowel */
     2 alternate_vowel bit (1);                    /* this is an alternate vowel, (i.e., "y") */

dcl  n_units defined digrams$n_units fixed bin;

/* ******** end include file digram_structure.incl.pl1 *********** */


 if first_call then do;
   saved_seed(1) = fixed (unique_bits_());
   first_call = "0"b;
  end;

 /* get length of word, if a range was specified */

 if max ^= min then do;
   random_length = max - min + 1;
   max_number = divide (32, random_length, 17, 0) * random_length;
   number = 33;
   do while (number > max_number);
     number = random_number();
   end;
   random_length = number - random_length * divide (number, random_length, 17, 0) + min;
   end;
 else random_length = max;

/* now get the random word */

try_again: call random_word_ (random_word, hyphens, random_length, length_in_units, random_unit,random_vowel);
          goto convert;

/* specialized random_unit and random_vowel routines */

/* These routines return a random unit number.  If the previous
   unit was not accepted, they do a nonlocal goto to
   try the word all over. */

random_unit: proc (n);
 dcl n fixed bin;
 if n < 0 then goto try_again;
loop:
 n = 999;
 do while (n > n_units);
          n = random_number();
 end;
 if substr (letters(n), 2, 1) ^= " " then goto loop; /* keep trying until a onme-letter unit is found */
end;

random_vowel: proc (n);
 dcl n fixed bin;
 if n < 0 then goto try_again;
loop:
 n = 999;
```

```
 do while (n > n_units);
    n = random_number();
 end;
 if substr (letters(n), 2, 1) ^= " " then goto loop; /* keep trying until a one-letter unit is found */
 if ^rules.vowel(n) then goto try_again;
end;

/* routine to generate a random number 1 to 32 */

random_number: proc returns (fixed bin(5));
 if index >= array_size then do; /* no more numbers left in array */
   call encipher_ (saved_seed(1), random_bits, random_bits, bits_size); /* get an array of numbers */
   saved_seed(1) = random_bits (1);
   index = 16;
   end;
 else index = index + 1;
 return (array (index) + 17); /* return the next random number from array */
end;

end;
```

COMPILATION LISTING OF SEGMENT generate_words
Compiled by: Multics PL/I Compiler of July 2, 1975.
Compiled on: 08/19/75 1017.1 edt Tue
    Options: check source

```
/* This procedure is a modified version of the standard generate_words command.
   The change is to accept the -uniform control argument and to call generate_word_$uniform
   when that argument is specified, instead of generate_word_.  Otherwise, it is identical
   to the standard version,
*/
generate_words: gw: procedure;
dcl cu_$arg_ptr entry (fixed,ptr,fixed,fixed bin(35));
dcl cu_$arg_ptr_rel entry (fixed bin, ptr, fixed bin, fixed bin(35), ptr);
dcl cu_$arg_list_ptr entry (ptr);
dcl argno fixed;
dcl new_line char(1) init("
");
dcl error_table_$badopt external fixed bin(35);
dcl arglen fixed bin;
dcl (generate_word_, generate_word_$uniform) entry (char(*), char(*), fixed bin, fixed bin);
dcl generate_word entry (char(*), char(*), fixed bin, fixed bin) variable init (generate_word_);
dcl generate_word_$init_seed entry (fixed bin(35));
dcl ios_$write_ptr entry (ptr, fixed bin, fixed bin);
dcl argptr ptr;
dcl hyphenate bit(1) init("0"b);
dcl cv_dec_check_ entry (char(*), fixed bin) returns (fixed bin(35));
dcl maximum_length fixed bin init(-1); /* set to maximum length of words */
dcl minimum_length fixed bin init(-1); /* minimum length of words */
dcl seed_value fixed bin(35) init(-1);     /* value of seed typed by user */
dcl com_err_ entry options(variable);
dcl i fixed, code fixed bin(35) init(0);
dcl unique_bits_ entry returns (fixed bin(70));
dcl result fixed bin;
dcl nwords fixed init(0);
dcl max_words fixed init(0);
dcl arg char(arglen) based (argptr) unaligned;
dcl maximum_hyphenated fixed bin;
dcl area char(58); /* where output line goes */
dcl output_line_length fixed bin; /* length of the output line in area */
dcl unhyphenated_word char (maximum_length) based (addr(area));
dcl hyphenated_word char (maximum_hyphenated) based (hph_ptr);
dcl hph_ptr ptr; /* pointer to position in area where hyphenated word goes */

dcl arglistptr ptr;

call cu_$arg_list_ptr (arglistptr);
do argno = 1 by 1 while(code = 0);
 call cu_$arg_ptr (argno,argptr,arglen,code);
 if code = 0
 then
  if arg = "-hph" | arg = "-hyphenate"
  then hyphenate = "1"b;
  else
   if arg = "-max"
   then maximum_length = value("maximum");
   else
     if arg = "-min"
     then minimum_length = value("minimum");
     else
       if arg = "-uniform" | arg = "-uf"
```

```
      then generate_word = generate_word_$uniform;
      else
       if arg = "-length" | arg = "-ln"
       then do;
         maximum_length = value("length");
         minimum_length = maximum_length;
         end;
       else
         if arg = "-seed" then do;
          seed_value = value("seed");
          if seed_value ^= 0 /* if seed not zero, use its characters instead of value */
                             /* encipher_ generates non-random numbers if there are a lot of leading
                               zeros in the key. */
          then seed_value = fixed (unspec (char(arg,4))); /* take up to first 4 characters */
          call generate_word_$init_seed (seed_value);
          end;
         else do;
           nwords = cv_dec_check_ (arg, result); /* look for number of words */
           if result = 0 & nwords > 0
           then max_words = nwords;
           else call ugly (error_table_$badopt, arg);
         end;
end;

/* Below we decide whether minimum, maximum, both, or none have been specified,
   and set their default values accordingly. */

if nwords = 0 then max_words = 1;
if minimum_length = -1
then if maximum_length = -1
     then do;
         minimum_length = 6;
         maximum_length = 8;
         end;
     else minimum_length = 4;
else if maximum_length = -1
     then maximum_length = 20;
if minimum_length < 4 | minimum_length > maximum_length |
  maximum_length > 20 then
      call ugly (0, "Bad value of lengths: 3<min<max<21 required.");

maximum_hyphenated = maximum_length + 2*maximum_length/3; /* maximum length of hyphenated word */

hph_ptr = addr (substr (area, maximum_length + 2)); /* where hyphenated word is put */
                                        /* even if we're not printing it out, it needs a place to go */
if hyphenate /* for efficiency, put newline character in expected place in output string */
then do;
   substr (unhyphenated_word, maximum_length + 1, 1) = " ";
   substr (hyphenated_word, maximum_hyphenated + 1, 1) = new_line;
   output_line_length = maximum_length + maximum_hyphenated + 2;
   end;
else do;
   substr (unhyphenated_word, maximum_length + 1, 1) = new_line;
   output_line_length = maximum_length + 1;
   end;

/* generate max_words and write them all out */

```

```
do i = 1 to max_words;
   call generate_word (unhyphenated_word, hyphenated_word, minimum_length, maximum_length);
   call ios_$write_ptr (addr(area), 0, output_line_length);
end;


ugly: procedure (codex, message);
  dcl (code, codex) fixed bin(35);
  dcl message char(*);
  call com_err_ (codex, "generate_words", message);
  goto return;
end;

value: procedure (name) returns (fixed bin(35));
   dcl number fixed bin(35);
   dcl name char (*);
   argno = argno + 1;
   call cu_$arg_ptr_rel (argno, argptr, arglen, code, arglistptr);
   if code ^= 0 then call ugly (code, "Value of " || name);
   number = cv_dec_check_ (arg, result);
   if result ^= 0 | number < 0
   then call ugly (0, "Bad " || name || " value. " || arg);
   return(number);
end;

return:
end;
```

REFERENCES

1. "Easily Guessed Passwords", Memorandum for the Record, ESD/MCIT, October 17, 1974.

2. Design for Multics Security Enhancements, Honeywell Information Systems, Inc., ESD-TR-74-176, Electronic Systems Division/Air Force Systems Command, L.G. Hanscom Field, Bedford, Massachusetts, 1974.

3. Multics Programmers Manual, Honeywell Information Systems, Inc., and Massachusetts Institute of Technology, 1973.