DESIGN OF AN AUTOMATED HYPHENATION PROCEDURE

by

Steven Scott Ranker

B.S., KANSAS STATE UNIVERSITY, 1980

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

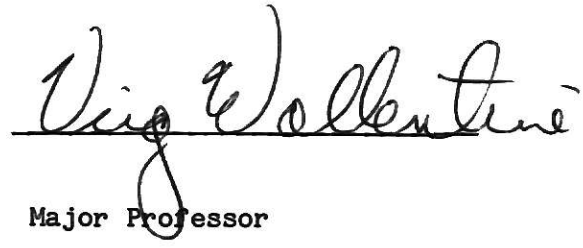
Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1982

Approved by



Major Professor

TABLE OF CONTENTS

TABLE OF CONTENTS (cont.)

1.0 INTRODUCTION

Hyphenation is defined as dividing or connecting (syllables or word elements) with a hyphen. The ability to do hyphenation adds overall readability and professionalism to a word processor. Hyphenation in word processing systems is usually performed on what is called the overset word. The overset word is the last word on the current output line, and is the word that crosses over the right margin. If the overset word can be hyphenated, the word processor places the hyphenated part on the current output line and then adjusts the interword spacing of the line to yield the correct right margin. The remaining part of the word is placed at the beginning of the next output line. If the overset word cannot be hyphenated, the word processor places the whole word on the next output line and the next overset word is then located.

1.1 METHODS OF HYPHENATION

There are basically two types of hyphenation methods: algorithmic and dictionary driven. The dictionary driven method uses a rather large dictionary of words and their hyphenable points. When an overset word is located, the dictionary is searched for that word. These dictionaries usually contain anywhere from 10,000 to 50,000 words. Procedures to handle negative dictionary searches vary from method to method. Some interact with the user, which requires a processing halt

until a user response is forthcoming, while others reject the words and continue processing. For microcomputers, these large dictionaries take up large amounts of disk space, which is scarce and time consuming to access. This project uses an algorithmic method which searches words for some well-defined substrings which are almost always hyphenated. This method also uses dictionary files, but these files should be much smaller than the dictionary in the previously stated method.

There are many types of algorithmic methods. Some rely on a statistical probability table of hyphenable points between each pair of letters. Others rely on so-called "special letter" pairs, which signal a nonbreakable point between pairs of letters. Some check for the position of the vowels in the word. If this and other tests fail, they break arbitrarily at the right margin. Moitra, Mudur, and Narwekar[2] expanded a method by W. H. Ocker[1], which uses an exception dictionary prefix and suffix string files, and a statistical "break value table" to determine hyphenable points. Moitra, Mudur, and Narwekar's[2] method was tested at 92.8% efficiency; that is, if their algorithm is tested on 1000 words, it will process and correctly hyphenate 928 words. This means that only 72 words will need to be stored and checked for in an exception dictionary instead of the 1000 words in the dictionary driven method. They tested their hyphenator on 2700 common technical words to achieve their results. By doing so they skewed the results slightly. I have found that most of the exception words are common non-technical

words of four to six characters in length. However, once sufficient dictionary building has taken place to eliminate this type of word, I believe the efficiency will be similar to the authors results.

1.2 PROJECT OBJECTIVES

In this project, I have implemented and expanded the Moitra, Mudur, and Narwekar[2] method.

The objectives of my project are:

OBJECTIVE 1 :

Design a usable, automated, hyphenation procedure (automated, in that no interactive user action is necessary unless errors are found), to hyphenate overset English language words using as a main outline and reference:

> ABHA MOITRA, S. P. MUDUR and A. W. NARWEKAR,
> Design and Analysis of a Hyphenation Procedure,
> Software - Practice and Experience, Vol. 9,
> 325 - 337 (1979)

OBJECTIVE 2 :

Write this procedure in PASCAL, looking forward to an eventual implementation in SCRIPT.

OBJECTIVE 3 :

Write this procedure, using as small a memory space as possible since any future microcomputer implementation will need to fit in a machine with a very limited memory size. If necessary, this objective can be met by making speed concessions, but not to the point of causing the project to be unusably slow.

OBJECTIVE 4 :

Create and manage an exception dictionary, in order to test the efficiency and usability of the procedure.

OBJECTIVE 5 :

Write and compile documentation, including a users guide and implementation assistance.

2.0 PROJECT DESIGN

The basic strategy of this algorithmic hyphenator is to construct a word sieve. Each sieve layer has holes in it that lets some words pass through. As English language words pass through the hyphenation procedure, each layer of the sieve does its best to catch and hyphenate as

many words as possible and pass or incorrectly hyphenate as few words as possible. The bad words (exceptions) are placed in a dictionary and provide the patches necessary to give as total a closure over the set of English language words as possible. The sieve is constructed of conditions, files of substrings, and a statistical probabilities break value table.

2.1 STRING FILES

The main part of the word sieve is the file of strings. These strings, when found in a word, represent a location in that word in which a hyphen can almost always be placed correctly. The strings are made up of commonly used prefixes, suffixes, and root words. The root words are added to the files in which they most logically fit. An example of a root word is the word "thing". It can be compounded into words like "anything" and "something", and its presence indicates a hyphenable point. Not many words begin with the prefix "thing", so it should be added to the suffix string file. From this point on, the combined suffix-root and prefix-root files will be referred to as the suffix string file and the prefix string file. The combined root word and suffix string file needs to be sorted and stored in reverse order. This is necessary to keep from having to skip around the file while searching. It allows the searching to be sequential in nature once the first string

is found. An example of this is the suffix "tion" - stored "noit". This means that the target word, (the word to be hyphenated), must be searched for the suffix strings from the last character backwards.

2.2 SPECIAL CONDITIONS

Another part of the word sieve is the set of test conditions. This is a group of conditions that check for a variety of things, such as embedded hyphens, placement and number of vowels, double consonants, plural words ending in "s", and words ending in "ed" or "es". These conditions are also used to do hyphenable range reduction and to do word rejection (to increase the speed of the procedure).

2.3 STATISTICAL TABLE

The statistical break table is the least accurate part of the word sieve. Therefore it is used and searched last. It is a boolean table which reflects the probability of a break point between every pair of letters. It was initially taken from Gimple[3] by Moitra, Mudur, and Narwekar[2] and was considerably modified by them. Every pair of letters within the range of available length is checked to see if the probability is high enough for a possible break point.

2.4 EXCEPTION DICTIONARY

The exception dictionary is used to shore up the holes in this strategy. But in order to determine what words are exceptions, the hyphenation procedure itself is first used with an empty exception dictionary. You have to let a few words through the sieve to see where the holes in the sieve are located. In other words, you must see which words, that you use daily in your vocabulary, are "exceptions" to the rules, strings, and the probabilities you have used in your sieve.

3.0 PROJECT IMPLEMENTATION

The implementation of this strategy was done in the following order.

Reject as many unhyphenable words from the start as possible, as processing of these words is a waste of time. This is done by checking the length of the target word for a minimum length and for some special case lengths. The number of vowels the word contains is also checked because words with only one vowel, like "cat", cannot be hyphenated because there are not enough vowels to have one on each side of the hyphen.

Next in the implementation, you reduce the range of all words that survive the rejection cycle, in order to speed up overall processing time. To reduce the hyphenable range of the target word the first vowel

and the last vowel are located. No hyphen can be inserted before the first or after the last vowel since a syllable must contain at least one vowel. This reduced range is then intersected with the range of lengths the hyphenated part of the word must lie within, in order for it to fit on the current output line. This yields the lowest possible hyphenable range. The word "problem" will show how this works. The first vowel is in position 3. Therefore the first possible break point is in position 4 ("LOW_RANGE_INDEX"). The last vowel is in position 6 and therefore the last possible break point is in position 6 ("HIGH_RANGE_INDEX"). This gives us a hyphenable range of (4,6). If the word needs to fit into the five remaining spaces in the output line, the available length is equal to 5. This range is (1,5). The intersection is then made between the two ranges. This intersection yields the final reduced range (4,5), which means that the hyphenator must break the word "problem" with the hyphen occupying either position 4 or position 5. The two possibilities are "pro-" or "prob-". The only correct possibility is "prob-" and the hyphenator returns this to the word processor.

Then the check must be made for the first set of special conditions, such as embedded hyphens or words having too few vowels, to be sure the target word is not a special case. If a word survives this far, the exception dictionary is checked to see if this word has been found to be previously processed in error.

Before an exception dictionary can be implemented, one must be

constructed. The exception dictionary can be built one of two ways. It can be user created or it can be built through the use of a testing program. Both involve physically looking up words in a dictionary.

The testing program approach requires a large number of input words on which hyphenation is attempted. It produces the hyphenated word through all possible ranges of available length. The user then looks up each suspect word and compiles the dictionary.

The user approach lets the user supply the input, and gives him the responsibility for checking each hyphenated word for correctness. If and when an error is found, that word is added to the exception dictionary and the file is rerun through the word processor again.

The user-created approach has a few advantages over the testing approach. Both approaches will achieve the same result, an exception dictionary. However, the dictionary constructed by the user will yield a larger dictionary with a wider range of words, since the input to the hyphenation procedure will be from a larger number of sources. It will take much longer to build, but the quality of the dictionary should be worth the time. Some of the major problems with the testing method of dictionary building are that the input files are hard to come by, often are wordy, repetitious, and generally deal with only one subject and therefore yield fewer potential dictionary words for the amount of effort expended.

The best approach is to use the supplied testing program on as many

input files as one can find, until the number of new dictionary words found is not worth the time it takes to find them. Then the user-created approach can be used to increase the dictionary's size and quality. With this type of algorithm, every time a wrongly hyphenated word is found and entered into the dictionary, the procedure "learns" from its mistake and never makes the same mistake again. Looking down the road, a time will come when the number of words this algorithm correctly processes, (the reciprocal of the efficiency of the procedure, multiplied by the number of entries in the dictionary), will surpass the vocabularies of the users, thereby giving the illusion of closure over the set of English language words. If no match is found in the dictionary, one of two things is true. Either the procedure may have or may yet possess the ability to correctly process the word, or the procedure has not yet processed this word and will not be able to do so correctly. If the latter is true, an addition needs to be made to the exception dictionary.

After the dictionary search, more special condition checks are made. The check for double consonants and the check for plural words ending is "s" must be made after the dictionary search because these conditions are not totally accurate. There are "exceptions" to these rules. Therefore the exception dictionary must contain these "exceptions". A note about these post-dictionary search conditions: Moitra, Mudur, and Narwekar[2] have found that almost all double conso-

nants signify correct break points. The pairs "ss" and "ll" are the most frequent exceptions and are therefore excluded from the set of breakable double consonants. The check for plural "s" simply removes the "s" at the end of the word and recursively calls the hyphenation procedure on this "new" word.

The search for suffix strings is the next step in the hyphenation procedure. Either string search (prefix or suffix) could come next, but suffixes occur more frequently and thus the speed of the procedure can be increased by searching for suffixes first. The suffix string file (stored in reverse order) is matched against the target word from the last letter backwards, leaving at least one vowel in the last syllable and having a length not exceeding the longest string in the file.

The prefix string search follows a negative suffix search. The prefix strings are matched against the target word from the first letter forward to the longest string in the prefix string file, provided at least one vowel is present in these prefixes.

Last in the implemented procedure, is the break value table search and the search for "ed" and "es". If a word makes it this far through the procedure, chances are very good that the procedure will be in error for this particular word. The break value table is searched for every pair of letters within the reduced range, until a break point is found, if one exists. Only when all other tests and string checks have failed is the target word searched for and broken at the string "ed" and "es"

when those strings occupy the last two positions of that word. This is a highly inaccurate test and will probably generate the highest number of errors, (besides exception words), and is only included to catch those few words that it can. At this point in the procedure, to omit this would only generate a different type of error: where a word with a hyphenable point present is found by the procedure to have none.

4.0 PROJECT PROCEDURE EXPLANATIONS

As with any PASCAL procedure, the caller must do certain things to enable efficient and error free integration. This procedure is no exception. The calling program must define the hyphenation procedure as "FORWARD" to enable recursion. It also needs to contain a standard "SVC_BLOCK" record and must define "SVC1" as an external procedure to enable block reading capabilities. The hyphenation procedure will reject all four letter words; therefore the calling program should only try to hyphenate words greater than four in length to speed things up. The arrays that are passed in and out of the procedure must be type defined by the caller to achieve type matching. The procedure uses these array definitions a number of different places and therefore they must be present in the calling program. Since this hyphenator is a procedure and will be called every time a word needs to be hyphenated, certain things that need to be done only once can be done more easily

and efficiently by the calling program. The array "BREAK_VALUE_TABLE", a 26 by 26 boolean array, needs to be included as a calling program variable because the break value table never changes once initialized. This initializtion can be done through the use of a procedure executed by the calling program. Appendix A contains this procedure.

What follows is a detailed explanation of each procedure in the hyphentor, where it is called from, what procedures it calls, if any, and the input or output it does, if any.

4.1 PROCEDURE HYPH

The hyphenation procedure's name is "HYPH". It must first define its procedures and then its execution cycle (the begin - end section), calls them in the order they are to be performed. It calls the procedures "INIT" and "REJECT" everytime a word needs to be hyphenated. It calls "CK_HYPHEN", "SPECIAL_REJECT", "CK_DICTIONARY", "RECURSIVE_S_CK", "CK_SUFFIX", "CK_PREFIX", "CK_BREAK_VALUE_TABLE" and executes the "ED" - "ES" string check only if previous processing allows the procedure to continue. Processing continues only while the "CONTINUE_FLAG" is true. The reason for this is simple; any one of these procedures can process a particular word correctly. Once this has been completed, further processing is useless and wasteful. Therefore once the word is hyphenated or rejected, the "CONTINUE_FLAG" is set to false and processing ceases.

The input to the procedure is either passed in through the use of parameters, or is read in through the use of "SVC1" page reads. The inputs to the procedure from disk are the exception dictionary and the two string files. They are read 18 lines at a time and are stored in arrays for subsequent processing. The input parameters are variable parameters in that they are changed and passed back to the calling program. The calling program passes in the overset word in a 25 position packed character array called "WORD_TABLE", and the integer length to which this word must be hyphenated in the parameter "AVAIL_LENG". The target word must be less than or equal in length to 25 and must end with a new line, carriage return, end-of-medium, or space character. The word passed to the procedure may end with any type of punctuation mark as long as the punctuation mark is followed somewhere by a space. This is necessary because the procedure scans the word until any punctuation mark, space, new line, carriage return, or end-of-medium character is found. The string, up to this delimiter, is considered the overset word. All post-word characters from the first delimiter to the first space are transferred to the return parameter "WORD_TABLE_REMAINING". The requirement that every word end with a space, new line, carriage return, or an end-of-medium character, delimits these post-word characters. To recognize these post-word characters a scan of the input parameter "WORD_TABLE" is made. If any of the four ending characters is found, its position will mark the total length

of the array. The "TOTAL_LENG" variable is used in every procedure where the overset word is broken. It tells when to stop transferring characters to "WORD_TABLE_REMAINING". In other words, the "TOTAL_LENG" of the array is the position where the overset word plus any and all post-word characters ends, while the "LENG" is the length of the overset word in alphabetic characters. If the calling program uses means other than space delimiters to locate overset words, a few changes to the procedure will need to be made. These are:

1. Delete the variable "TOTAL_LENG" from line 339
2. Change the variable "TOTAL_LENG" on line 381 to "LENG + 1"
3. Delete lines 583 - 593
4. Change the variable "TOTAL_LENG" on line 635 and 641 to "LENG + 1"
5. Delete "STORE_TOTAL_LENG from 708
6. Delete line 716
7. Delete lines 723 - 733

The use of "LENG + 1" will return the word plus the delimiter. If the word is all that is required to be passed back, (no delimiters), use "LENG". This implementation uses the total length scan due to the fact SCRIPT us spaces to denote the ending of words.

The hyphenation procedure uses variable parameters to do output. It passes these parameters back to the caller: The two input parameters described above, plus two boolean flag variables "HYPEN_FLAG" and "CONTINUE_FLAG", and an array and integer length field for the part of the word remaining after the hyphen, if one is inserted. The "HYPHEN_FLAG" is the key output parameter. If this flag is true, it

means that the procedure has found what it believes to be a correct hyphenable point in the word contained in "WORD_TABLE", and that this break point is within or equal to the remaining length available on the current output line. When the "HYPHEN_FLAG" is true, the hyphenated part of the word is output in "WORD_TABLE", and the length of that part, up to and including the hyphen, is contained in the output parameter "AVAIL_LENG". The remaining part of the word and its length are returned in the output parameters "WORD_TABLE_REMAINING", and "REMAINING_LENG". If the "HYPHEN_FLAG" is returned false, the "WORD_TABLE" parameter and the "AVAIL_LENG" parameter are returned untouched. The last output parameter, the boolean flag variable "CONTINUE_FLAG", is used to give the calling program an idea of the reason, if any, why the procedure did not hyphenate the word. When the "HYPHEN_FLAG" is returned false and the target word has hyphenable points, the word sieve does not correctly process the word and does not even deflect the word by hyphenating it incorrectly. These words are anomalies, because it takes the human cognitive process, and sometimes a dictionary, to determine whether the procedure's failure to hyphenate a word was in error. If the procedure could determine this, it would not make the error in the first place. The calling program can overcome this anomaly. The anomaly is a word that has a hyphenable point that is found by the procedure to have none, after all the conditions, all the strings, and all the exceptions have been checked. These anomalies

would never be hyphenated, either correctly or incorrectly. Therefore they would be transparent to the user. The calling program can make these visible by checking the "CONTINUE_FLAG" and "HYPHEN_FLAG" upon return from the procedure. The "HYPHEN_FLAG" tells whether or not hyphenation was performed by the procedure. On these anomalies, that information is not enough, since you need to know why it was not performed. Through the use of the "CONTINUE_FLAG", which tells if the processing in the procedure was cut short by the procedure itself, you can detect these anomalies. Their presence can be determined when processing was never stopped, (i.e. CONTINUE_FLAG = TRUE), and the word was not hyphenated, (i.e. HYPHEN_FLAG = FALSE). When these two conditions are present the word should be displayed to a special file of anomalies. This can be done easily by the calling program. This file, called "DIS.TXT" for DISPLAY FILE.TEXT should be checked occasionally and the words that are in it, if any, should be looked up to confirm the correct break point and then added to the exception dictionary.

4.2 PROCEDURE INIT

The procedure "INIT" is called by HYPH's execution cycle. It calls no other procedures and does no input or output. It does the initialization of global variables and "SVC" blocks for the whole procedure. The set of delimiters is also loaded from this procedure so any change of delimiters should be included here.

4.3 PROCEDURE REJECT

The "REJECT" procedure is called by HYPH's execution cycle. It calls no other procedures and does no input or output. It checks to make sure there are no capitals or special characters in the target word. The only special character it accepts is a hyphen. If there is one present, "REJECT" records its location for later processing. "REJECT" also does word range reduction on the target word by locating the first and last vowel (with 'y' included in the vowel set), since no break is possible after the last vowel or before the first. "REJECT" is also responsible for determining the true length of the word and the total length of the array. The "TOTAL_LENG" is the length of the string in "WORD_TABLE" up to and including the first space. The "LENG" of the word is the word is the number of characters up to any delimiter in the set "END_CHAR".

4.4 PROCEDURE CK_HYPHEN

The procedure "CK_HYPHEN" is called by the main procedure's execution cycle, calls no other procedures, and does no input or output. It is called only if the previous two procedures have let it, (i.e. "CONTINUE_FLAG" = TRUE) and executes only if the "REJECT" procedure locates a hyphen in the word. It then breaks the word at the correct location and stops further execution.

4.5 PROCEDURE SPECIAL_REJECT

The "SPECIAL_REJECT" procedure is called by the main procedure's execution cycle if no proceding procedure has stopped processing, calls no other procedures and does no input or output. It does further range reduction by intersecting the already reduced range (in "REJECT"), and the length available on the current output line. It checks and rejects: words of length 4 and under, words with only one vowel, words that are 5 characters long and end with "s" or "d", and words where the now reduced range is empty, (the highest possible break point is equal to or less than the lowest possible break point). These rules can cause exceptions, but they save more than enough processing time and errors to justify them.

4.6 PROCEDURE CK_DICTIONARY

The procedure "CK_DICTIONARY" is called by the main procedure's execution cycle as long as the "CONTINUE_FLAG" has not been set to false. It calls and tells "MATCHSTRING", though the use of its parameters, to execute all three of its procedures, which search the exception dictionary for a match to the string in "WORD_TABLE" up to the character in the that array's "LENG" position. By matching only as far as the length, it does root word matching on words that are not changed by suffixes. An example of this is the word "search". The prefix string

procedure finds the prefix "sea", which in this case is an error, so we enter it in the exception dictionary. But what of the past tense and plural derivatives of this word ("searched" and "searches")? By doing the matching up to the target word's length, 6, we will find positive matches on the word "search" from either "searched" or "searches". Unfortunately, this only works correctly on words that are unchanged by adding suffixes. A word that drops an "e" when adding "ing", for example, will require multiple entries. In simpler terms, the root can sometimes be matched from the derivative but not vice versa. If the target word is found to be an exception word, "CK_DICTIONARY" first computes the break point of the matched target word from the dictionary's 26th and 27th positions, and then calls the procedure "BREAKWORD". It passes "BREAKWORD" that break point. The only output "CK_DICTIONARY" does is to rewind the exception dictionary after the searching is completed.

4.7 PROCEDURE CK_DOUBLE_CONSONANT

The procedure "CK_DOUBLE_CONSONANT" is called by HYPH's execution cycle if processing has not been halted. If a double consonant exists in the target word, it calls the procedure "BREAKWORD" with the location of the second of a double consonant pair (other than "ss" and "ll"). "CK_DOUBLE_CONSONANT" does no input or output.

4.8 PROCEDURE RECURSIVE_S_CK

The procedure "RECURSIVE_S_CK" is called by HYPH's execution cycle as long as no previous procedure has stopped processing. It does no input or output. It checks for an "s" in the last position of the target word. If the word ends in "s" and not "ss" the procedure replaces the "s" with a space and recursively calls HYPH. It replaces the "s" upon return, whether the word has been hyphenated or not.

4.9 PROCEDURE CK_SUFFIX

The procedure "CK_SUFFIX" is called by HYPH as long as the "CONTINUE_FLAG" is true. It checks for all possible suffixes from the end of the target word to the "LOW_RANGE_INDEX" or for a length of "SUFFIX_LONGEST", which ever is smaller. It checks these strings in reverse order. It calls the procedure "MATCHSTRING" and passes parameters which tell "MATCHSTRING" to not only search the suffix file for the target strings, but to read in those strings only once if certain conditions are present. "MATCHSTRING" need read in an array of strings more than once only if it finds that all the potential matching strings are not present in the current block of the array. If a match to the string is found, "CK_SUFFIX" calls the procedure "BREAKWORD" with the position at which the reversed string ends. "CK_SUFFIX" rewinds its own file at the end of string processing or if "MATCHSTRING" tells it to.

"MATCHSTRING" relays its need for a file rewind by sending back a zeroed "AFFIX_NUM" parameter, which means a possible string match lies a block ahead of the one currently being searched.

4.10 PROCEDURE CK_PREFIX

The procedure "CK_PREFIX" is called by HYPH only after all the other procedures have executed with negative results. It checks for all the prefixes in the target word from the beginning of the target word to the "HIGH_RANGE_INDEX" or to "PREFIX_LONGEST", which ever is smaller. It calls "MATCHSTRING" in the same way "CK_SUFFIX" does. It calls the procedure "BREAKWORD" if a match to the string is found and gives it the location of the end of the prefix or root word it has matched. "CK_PREFIX" also rewinds its own file at the end of string processing or if "MATCHSTRING" requests it to, and does no other input or output.

4.11 PROCEDURE CK_BREAK_VALUE_TABLE

The procedure "CK_BREAK_VALUE_TABLE" is called by HYPH if execution makes it this far. It looks up each pair of letters in the word in the array "BREAK_VALUE_TABLE", from the "LOW_RANGE_INDEX" to the "HIGH_RANGE_INDEX". If a break point is found, "BREAKWORD" is called with the correct "BREAK_INDEX" passed in. "CK_BREAK_VALUE_TABLE" uses the array initialized by the word processor and does no input or output.

4.12 PROCEDURE BREAKWORD

The procedure "BREAKWORD" is called anytime another procedure has determined the location of a break point within the target word. Its parameter, "BREAK_INDEX", is the location where the hyphen is to be placed. It breaks the target word and loads all of HYPH's return parameters. It does no input or output.

4.13 PROCEDURE MATCHSTRING

The procedure "MATCHSTRING" is called by "CK_DICTIONARY", "CK_SUFFIX", and "CK_PREFIX". Its parameters are: the "SVC1_BLOCK" used in input, the two arrays containing the strings to be matched, and the number of the procedure requesting service. "MATCHSTRING" is made up of three procedures "LOADER", "FIRST_LETTER_SEARCH", and "STRING_MATCHER". The number of the calling procedure is needed because "LOADER" and "FIRST_LETTER_SEARCH" are only performed once if the caller is "CK_SUFFIX" or "CK_PREFIX" and if the target strings are contained in one block of the file. If a block of potential matching strings overlap the block boundary, the possibility exists that some strings will be on one side and others will be on the other side. Therefore if the strings are not all in one block, "MATCHSTRING" requests the calling procedure to rewind that file. "LOADER" calls a procedure called "SVC_LOAD" and passes it the "SVC1_BLOCK" and the array to load the file into.

"SVC_LOAD" returns the number of rows it has loaded into that array. "LOADER" continually calls "SVCLOAD" until the array's last row contains either the first character in the target string or a character that is greater that the first character in the target string. When "LOADER" is finished, the array contains the block that the target string is in or should be in. "LOADER" then calls "FIRST_LETTER_SEARCH", which binarily searches the loaded array for the first letter in the target string. If this search yields a match, this procedure finds all other first letter matches by sequentially searching up and down from the first match. If a match is found to exist in row one of the array, the previous block may contain a string that will be searched for at a later time and therefore a rewind is signaled for. If at least one match is found, "FIRST_LETTER_SEARCH" calls "STRING_MATCHER". "STRING_MATCHER" sequentially checks all strings in the array from the upper limit to the lower limit (found by "FIRST_LETTER_SEARCH"), for a match to the target string. If a match to the string is found, the global variable "MATCH_FOUND" is set to true and "CONTINUE_FLAG" is set to false.

4.14 PROCEDURE SVC_LOAD

The procedure "SVC_LOAD" was originally designed to read disk files into packed character arrays. It was meant to be able to read in block

sizes that could be varied and thereby tuned to yield the greatest efficiency. Perkin - Elmer's 8/32 has a problem with this in that one can only block disk records 512 without having difficulty transferring them correctly to the array. This is the main reason for the file format being 18 records to the 512 block. Should future operating systems remedy this, one needs to change these few lines: the arrays "EXCEPT" and "AFFIX" need to have their sizes increased (type declaration); the constant "18" (in "SVC_LOAD") needs to reflect that increase; and the file formatter/changer (Appendix H) may need to be be restructured.

5.0 INPUT FILES

There are three files needed by the procedure: the exception dictionary, the reversed suffix string file, and the prefix string file. The decision to page these files in and out of memory was made for the following reasons. The exception dictionary must be read in because it will be ever expanding in size and no array can be expanded indefinitely. The two string files need to remain expandable, but the reason to page these files in was due to the memory space required. These files would take up an additional 7 K of space and many microcomputers cannot spare that much space.

The exception dictionary is made up of exception words of length 25. The 25 character word is followed by a two character number which

is the position in the word that the hyphen should be placed. For example, the word "problem" would be broken correctly with the hyphen occupying position 5 in the hyphenated string. Therefore the entry in position 26 and 27 of the dictionary would be 05. These entries must be formatted to fit 512 byte records due to a Perkin-Elmer disk blocking restriction. I have supplied this formatter with the procedure under the file name "FORMAT.PAS". The steps to update the dictionary and string files are as follows:

1. Use an editor to add, delete, or change the desired record, remembering that these files are in sorted order and all insertions must be made in the correct alphabetic location.

   NOTE The record must be 27 characters to the new line character, and the editor must not change the record length of the file. I suggest the use of PEDIT.

2. When editing is completed, load the formatter assigning logical units 1 and 2 to the file to be formatted and start.

When the formatter has completed its job, the 27 character records are grouped 18 in number along with 8 spaces to round off the length to 512.

The reversed suffix and prefix strings are stored in the same format as the exception dictionary. This is done to standardize the input to the procedure and eliminate memory used in the duplication of code for separate input and formatting. The string, in both files occupies a 25 character length field, followed by a two character numeric field. The numeric field is not used by the procedure during string

processing, but contains a non-space entry, "0", to keep the editor from eliminating trailing blanks and thus keeping the format the correct length. The suffix string file is stored in reverse order so updates must also be made in reversed order. The string files can be added to, but this should not be as necessary as additions to the exception dictionary. Three simple rules can be followed to determine when a string or root word should be added.

Rule 1: the word or string starts or ends with a vowel.

Rule 2: the word or string has a letter combination that does not follow the break value table.

Rule 3: the word or string is used, or occurs often enough that an entry in the exception dictionary would not suffice. Rule 3 can be explained easily by example. During dictionary building tests, the word "analyze" was found to be in error. It can be made past tense by adding "ed" to make "analyzed". The suffix in this word is "lyzed" and could make processing correct if this suffix were added to the reversed suffix string file. Looking ahead however, this suffix is quite rare and does not occur on a large number of words. Therefore, an exception entry would serve the user better. The suffix and prefix file entries vary in length from two to five characters as they are now. But as the dictionary and string files are expanded, this may change. It is unwise to search for strings that are six characters long and longer when there are none present in the files. Hence two constants, "suffix_longest"

and "prefix_longest" are both needed in this procedure. They are set equal to five. However this creates another update problem. If a new root word or string is six characters or longer, these constants must be changed and the procedure recompiled. For this amount of effort, the string must occur quite frequently, as all subsequent string searches will look for six character strings and there will only be one six character string in the file. Again the balance between the string file entry and the exception dictionary must be weighed by the user. I suggest entering the word in the exception dictionary and the correct string file without changing the two constants. Further processing errors of words with the same string present could justify resetting the constants and deleting the exception dictionary entries with these strings.

6.0 UTILIZATION TOOLS

Some of the program tools that are supplied with this project are located in Appendixes F through G. They include: a program called "LINEBUILDER", that shows how to implement the hyphenator, a program called "DICTIONARYBUILDER", that hyphenates every word in any text file, to be used in the construction of the exception dictionary, and the file formatter/changer, which formats the three files used by the hyphenator, into 512 blocks.

6.1 LINEBUILDER PROGRAM

The "LINEBUILDER" program supplies all the required types and definitions needed by the hyphenator. It begins by executing the "BREAK_VALUE_TABLE" loading procedure. (Appendix A). It then begins building a line that will be "LINE_LIMIT" long (line 149). It scans the input file loading each word into a temporary buffer. Each time it adds a character it increases a counter. ("LINE_COUNT"). Everytime a word is ended with a space, new line, carriage return or end of medium character, the location and number of that charcter is recorded. (SPACE_TABLE [SPACE_COUNT]). If a word is ended with a new line or a carriage return, these are replaced by a space. (lines 890 - 894). Everytime the end of a word is found, the "LINE_COUNT" is checked to see if it is greater than, less than, or equal to the "LINE_LIMIT" + 1. The plus one is the ending space, which will be placed in the right margin. If the "LINE_COUNT" is less than the limit, the word in the temporary buffer is placed in the "LINE_BUFFER" (line 896 - 901), and the next word is read into the temporary buffer. If the "LINE_COUNT" is equal to the limit plus one, the temporary buffer is emptied into the "LINE_BUFFER" and the "LINE_BUFFER" is output and reset (line 901 - 915). If the "LINE_COUNT" is greater than the limit plus one the word in the temporary buffer is overset. HYPH is called, passing in the overset word in the temporary buffer, and the "AVAIL_LENG", which is found by subtracting the position of the space following the overset word, from the "LINE_LIMIT". Upon

return from the hyphenator, the anomaly check is performed (line 927 - 932). The "HYPHEN_FLAG" is then checked to see if the hyphenator hyphenated the overset word or not. If it did, the interword spacing is calculated with the hyphenated part of the overset word included. If the "HYPHEN_FLAG" is false, the interword spacing is calculated without the overset word. (line 933 - 959). The "LINE_BUFFER" is then output (line 960 - 985). Once this is done, the remaining part of the word or the whole overset word, depending on "HYPHEN_FLAG", is placed on the now reset "LINE_BUFFER" (line 986 - 1005). This process is repeated until end-of-medium is reached. When the end-of-medium is reached, the remaining fragment of "LINE_BUFFER" is flushed to output.

6.2 DICTIONARYBUILDER PROGRAM

The dictionary builder program scans text files for words, hyphenating each word in the file. Each word is hyphenated using its length for the "AVAIL_LENG". Its length is then decreased to four and the word is hyphenated again and again with "AVAIL_LENG" increasing by one until "AVAIL_LENG" equals the word length. The "AVAIL_LENG" is output along with the section code of the section of the hyphenator that processed the word. The section codes are:

"D" for CK_DICTIONARY
"H" for CK_HYPHEN
"C" for CK_DOUBLE_CONSONANT
"S" for CK_SUFFIX

"P" for CK_PREFIX
"B" for CK_BREAK_VALUE_TABLE
or ED - ES CHECK

6.3 FORMATTING PROGRAM

The formatter program simply blocks file records 512 characters long. Since each row in the array is 28 characters long, (25 character word, two character break index, and a new line), the formatter reads in 28 characters and checks the 28th to see if it is a new line character. If it is a new line character, the line is output and count is kept. If the 28th character is not a new line the buffer is flushed. When the count reaches 18, (18 x 28 = 504), the eight pad spaces are added. Processing like this continues until end-of-medium is reached.

7.0 OBSERVATIONS AND CONCLUSIONS

Overall, the project worked very well. Including the initializing, the program uses just over 8 K of memory space. The speed of the program is difficult to measure since the number of overset words in a file vary. The dictionary builder takes quite a while to get through a large file, but remember each word is hyphenated the number of times that it is long. Utilization of the procedure in my line building program appears to be very useable in terms of speed. The errors generated by the

hyphenator fall into one of these six categories:

1. The rejection of four and some five letter words rejected hyphenalble words.

2. The "ed" - "es" check found a hyphenable point when none existed.

3. The incorrect words should have been included in the exception dictionary.

4. A prefix of suffix string generated an incorrect break point.

5. The word was a double consonant rule exception.

6. The word passed all tests, conditions, and string match tests (the anomaly word).

There are two major obstacles to the successful usage of this procedure: getting the procedure into the word processor and getting the users to make additions to the exception dictionary. Setting aside some time to integrate the procedure is the major factor in the first obstacle. The user's attitude and willingness to take part in the dictionary building process is the major factor is the second. The improved readability and professionalism gained through the use of this procedure may not be enough to get the user to put out the effort of contributing.

REFERENCES

1. W. A. Ocker, 'A Program to hyphenate English words', IEEE Trans Prof. Comm. PC-18, 54-63 (1975)

2. Abha Moitra, S. P. Mudur and A. W. Narwekar, 'Design and Analysis of a Hyphenation Procedure', SOFTWARE - PRACTICE and EXPERIENCE, VOL. 9, 325 - 337 (1979)

3. J. F. Gimpel, Algorithms in SNOBOL4 John Wiley, New York, 1976

APPENDIX A - PROCEDURE INIT_BREAKVT

This is the VAR declaration and

procedure to initialize the BREAK_VALUE_TABLE

```
; BREAK_VALUE_TABLE : ARRAY [ 1 .. 26 , 1 .. 26 ] OF BOOLEAN

; PROCEDURE INIT_BREAKVT

; BEGIN
    FOR I := 1 TO 26
    DO FOR J := 1 TO 26 DO BREAK_VALUE_TABLE [ I , J ] := TRUE
  ; FOR I := 1 TO 14
    DO BEGIN
      BREAK_VALUE_TABLE [ 1 , I ] := FALSE
    ; BREAK_VALUE_TABLE [ I , 1 ] := FALSE
    ; BREAK_VALUE_TABLE [ 5 , I ] := FALSE
    ; BREAK_VALUE_TABLE [ I , 5 ] := FALSE
    ; BREAK_VALUE_TABLE [ 9 , I ] := FALSE
    ; BREAK_VALUE_TABLE [ I , 9 ] := FALSE
    ; BREAK_VALUE_TABLE [ 15 , I ] := FALSE
    ; BREAK_VALUE_TABLE [ I , 15 ] := FALSE
    ; BREAK_VALUE_TABLE [ 21 , I ] := FALSE
    ; BREAK_VALUE_TABLE [ I , 21 ] := FALSE
    END
  ; FOR I := 15 TO 23
    DO BEGIN
      BREAK_VALUE_TABLE [ I , 1 ] := FALSE
    ; BREAK_VALUE_TABLE [ I , 5 ] := FALSE
    ; BREAK_VALUE_TABLE [ I , 9 ] := FALSE
    ; BREAK_VALUE_TABLE [ I , 15 ] := FALSE
    ; BREAK_VALUE_TABLE [ I , 21 ] := FALSE
    END
  ; FOR I := 18 TO 24
    DO BEGIN
      BREAK_VALUE_TABLE [ 1 , I ] := FALSE
    ; BREAK_VALUE_TABLE [ 5 , I ] := FALSE
    ; BREAK_VALUE_TABLE [ 9 , I ] := FALSE
    ; BREAK_VALUE_TABLE [ 15 , I ] := FALSE
    ; BREAK_VALUE_TABLE [ 21 , I ] := FALSE
    END
  ; BREAK_VALUE_TABLE [ 12 , 4 ] := FALSE
  ; BREAK_VALUE_TABLE [ 14 , 3 ] := FALSE
```

```
; BREAK_VALUE_TABLE [ 14 , 4 ] := FALSE
; BREAK_VALUE_TABLE [ 18 , 3 ] := FALSE
; BREAK_VALUE_TABLE [ 18 , 4 ] := FALSE
; BREAK_VALUE_TABLE [ 19 , 3 ] := FALSE
; BREAK_VALUE_TABLE [ 20 , 3 ] := FALSE
; BREAK_VALUE_TABLE [ 25 , 1 ] := FALSE
; BREAK_VALUE_TABLE [ 26 , 1 ] := FALSE
; BREAK_VALUE_TABLE [ 3 , 8 ] := FALSE
; BREAK_VALUE_TABLE [ 7 , 8 ] := FALSE
; BREAK_VALUE_TABLE [ 14 , 7 ] := FALSE
; BREAK_VALUE_TABLE [ 16 , 8 ] := FALSE
; BREAK_VALUE_TABLE [ 18 , 7 ] := FALSE
; BREAK_VALUE_TABLE [ 19 , 8 ] := FALSE
; BREAK_VALUE_TABLE [ 20 , 8 ] := FALSE
; BREAK_VALUE_TABLE [ 2 , 18 ] := FALSE
; BREAK_VALUE_TABLE [ 3 , 18 ] := FALSE
; BREAK_VALUE_TABLE [ 4 , 18 ] := FALSE
; BREAK_VALUE_TABLE [ 6 , 18 ] := FALSE
; BREAK_VALUE_TABLE [ 7 , 18 ] := FALSE
; BREAK_VALUE_TABLE [ 8 , 18 ] := FALSE
; BREAK_VALUE_TABLE [ 16 , 18 ] := FALSE
; BREAK_VALUE_TABLE [ 17 , 18 ] := FALSE
; BREAK_VALUE_TABLE [ 23 , 8 ] := FALSE
; BREAK_VALUE_TABLE [ 26 , 5 ] := FALSE
; BREAK_VALUE_TABLE [ 25 , 9 ] := FALSE
; BREAK_VALUE_TABLE [ 26 , 9 ] := FALSE
; BREAK_VALUE_TABLE [ 2 , 12 ] := FALSE
; BREAK_VALUE_TABLE [ 3 , 11 ] := FALSE
; BREAK_VALUE_TABLE [ 3 , 12 ] := FALSE
; BREAK_VALUE_TABLE [ 4 , 12 ] := FALSE
; BREAK_VALUE_TABLE [ 6 , 12 ] := FALSE
; BREAK_VALUE_TABLE [ 7 , 12 ] := FALSE
; BREAK_VALUE_TABLE [ 7 , 14 ] := FALSE
; BREAK_VALUE_TABLE [ 11 , 14 ] := FALSE
; BREAK_VALUE_TABLE [ 12 , 12 ] := FALSE
; BREAK_VALUE_TABLE [ 12 , 11 ] := FALSE
; BREAK_VALUE_TABLE [ 14 , 11 ] := FALSE
; BREAK_VALUE_TABLE [ 16 , 12 ] := FALSE
; BREAK_VALUE_TABLE [ 18 , 13 ] := FALSE
; BREAK_VALUE_TABLE [ 18 , 14 ] := FALSE
; BREAK_VALUE_TABLE [ 20 , 18 ] := FALSE
; BREAK_VALUE_TABLE [ 23 , 18 ] := FALSE
; BREAK_VALUE_TABLE [ 25 , 18 ] := FALSE
; BREAK_VALUE_TABLE [ 11 , 19 ] := FALSE
; BREAK_VALUE_TABLE [ 12 , 19 ] := FALSE
; BREAK_VALUE_TABLE [ 14 , 19 ] := FALSE
```

```
; BREAK_VALUE_TABLE [ 18 , 19 ] := FALSE
; BREAK_VALUE_TABLE [ 19 , 19 ] := FALSE
; BREAK_VALUE_TABLE [ 23 , 19 ] := FALSE
; BREAK_VALUE_TABLE [ 25 , 19 ] := FALSE
; BREAK_VALUE_TABLE [ 19 , 11 ] := FALSE
; BREAK_VALUE_TABLE [ 19 , 14 ] := FALSE
; BREAK_VALUE_TABLE [ 20 , 12 ] := FALSE
; BREAK_VALUE_TABLE [ 25 , 12 ] := FALSE
; BREAK_VALUE_TABLE [ 25 , 13 ] := FALSE
; BREAK_VALUE_TABLE [ 25 , 14 ] := FALSE
; BREAK_VALUE_TABLE [ 25 , 15 ] := FALSE
; BREAK_VALUE_TABLE [ 1 , 16 ] := FALSE
; BREAK_VALUE_TABLE [ 26 , 15 ] := FALSE
; BREAK_VALUE_TABLE [ 5 , 16 ] := FALSE
; BREAK_VALUE_TABLE [ 9 , 16 ] := FALSE
; BREAK_VALUE_TABLE [ 13 , 16 ] := FALSE
; BREAK_VALUE_TABLE [ 15 , 16 ] := FALSE
; BREAK_VALUE_TABLE [ 19 , 16 ] := FALSE
; BREAK_VALUE_TABLE [ 19 , 17 ] := FALSE
; BREAK_VALUE_TABLE [ 21 , 16 ] := FALSE
; BREAK_VALUE_TABLE [ 24 , 16 ] := FALSE
; BREAK_VALUE_TABLE [ 20 , 23 ] := FALSE
; BREAK_VALUE_TABLE [ 21 , 26 ] := FALSE
; BREAK_VALUE_TABLE [ 23 , 25 ] := FALSE
; BREAK_VALUE_TABLE [ 24 , 25 ] := FALSE
; BREAK_VALUE_TABLE [ 25 , 25 ] := FALSE
; BREAK_VALUE_TABLE [ 3 , 20 ] := FALSE
; BREAK_VALUE_TABLE [ 6 , 20 ] := FALSE
; BREAK_VALUE_TABLE [ 8 , 20 ] := FALSE
; BREAK_VALUE_TABLE [ 14 , 20 ] := FALSE
; BREAK_VALUE_TABLE [ 16 , 20 ] := FALSE
; BREAK_VALUE_TABLE [ 18 , 20 ] := FALSE
; BREAK_VALUE_TABLE [ 19 , 20 ] := FALSE
; BREAK_VALUE_TABLE [ 25 , 20 ] := FALSE
; BREAK_VALUE_TABLE [ 25 , 21 ] := FALSE
; BREAK_VALUE_TABLE [ 26 , 21 ] := FALSE
; BREAK_VALUE_TABLE [ 1 , 25 ] := FALSE
; BREAK_VALUE_TABLE [ 1 , 26 ] := FALSE
; BREAK_VALUE_TABLE [ 3 , 26 ] := FALSE
; BREAK_VALUE_TABLE [ 5 , 26 ] := FALSE
; BREAK_VALUE_TABLE [ 6 , 25 ] := FALSE
; BREAK_VALUE_TABLE [ 9 , 26 ] := FALSE
; BREAK_VALUE_TABLE [ 12 , 22 ] := FALSE
; BREAK_VALUE_TABLE [ 12 , 25 ] := FALSE
; BREAK_VALUE_TABLE [ 15 , 25 ] := FALSE
; BREAK_VALUE_TABLE [ 15 , 26 ] := FALSE
```

```
; BREAK_VALUE_TABLE [ 17 , 25 ] := FALSE
; BREAK_VALUE_TABLE [ 18 , 22 ] := FALSE
; BREAK_VALUE_TABLE [ 18 , 25 ] := FALSE
; BREAK_VALUE_TABLE [ 18 , 26 ] := FALSE
; BREAK_VALUE_TABLE [ 19 , 23 ] := FALSE
; BREAK_VALUE_TABLE [ 1 , 15 ] := TRUE
END
```

APPENDIX B - SUFFIX-ROOT STRING FILE

This is the formatted suffix-root word file.

It is stored on USR6 as SUF.TXT/38

(The blank lines are pad characters, 1 every 18 entries)

alp	0
ca	0
ci	0
cib	0
cif	0
cil	0
cim	0
cir	0
cis	0
cit	0
daol	0
deif	0
dloh	0
dnal	0
dnats	0
dnec	0
dnop	0
dra	0
draw	0
dray	0
ecal	0
ecalp	0
ecif	0
ecit	0
ecna	0
ecnac	0
ecnan	0
ecnat	0
ecne	0
ecnec	0
ecned	0
eda	0
edac	0
edam	0
edut	0

ega	0
egal	0
egats	0
egit	0
ekil	0
elb	0
elba	0
elbi	0
elbis	0
elc	0
eld	0
elg	0
eloh	0
elp	0
elt	0
emit	0
enihs	0
enil	0
enit	0
enot	0
epahs	0
epip	0
epol	0
erehp	0
erud	0
erus	0
erut	0
esac	0
esim	0
esin	0
esit	0
esuoh	0
esyl	0
eta	0
etac	0
etad	0
etag	0
etal	0
etam	0
etan	0
etap	0
etar	0
etart	0

etas	0
etat	0
etav	0
etis	0
etuc	0
eug	0
euq	0
evic	0
evig	0
evis	0
evit	0
ezic	0
ezid	0
ezig	0
ezil	0
ezim	0
ezin	0
ezir	0
ezit	0
ezyl	0
flah	0
gni	0
gniht	0
gnild	0
gnilp	0
hsi	0
hsil	0
hsin	0
hsiug	0
kcalb	0
koob	0
krow	0
la	0
lab	0
lac	0
lacit	0
lag	0
laic	0
laid	0
laim	0
lain	0
lair	0
lait	0
lam	0

lar	0
lart	0
lat	0
le	0
leb	0
led	0
let	0
luf	0
met	0
mod	0
mrow	0
msi	0
msic	0
msin	0
muic	0
muin	0
muir	0
muis	0
mum	0
murt	0
nac	0
nai	0
naic	0
naid	0
naig	0
nais	0
nait	0
nam	0
ne	0
neet	0
nek	0
nem	0
nes	0
nez	0
niam	0
niat	0
noeg	0
noic	0
noig	0
noir	0
nois	0
noit	0
nort	0

nos	0
nroc	0
nrow	0
nwod	0
pihs	0
pohs	0
rad	0
ral	0
re	0
reb	0
red	0
ref	0
reg	0
reht	0
rehto	0
rei	0
reif	0
rel	0
rep	0
ret	0
retfa	0
reve	0
revo	0
roir	0
ros	0
rot	0
sei	0
seic	0
seif	0
seil	0
seilp	0
sein	0
seir	0
seirt	0
seit	0
sis	0
ssel	0
ssen	0
suo	0
suog	0
suoic	0
suoig	0
suois	0
suoit	0

suol	0
suom	0
suon	0
suot	0
sut	0
tarc	0
tcel	0
tcep	0
tcet	0
tek	0
tel	0
thgil	0
tic	0
tid	0
tiuc	0
tluc	0
tna	0
tnac	0
tnad	0
tnalp	0
tnan	0
tne	0
tnec	0
tned	0
tneg	0
tneic	0
tnein	0
tnel	0
tnem	0
tnen	0
tnes	0
tnet	0
tneuq	0
tnev	0
tse	0
tsin	0
tuo	0
yad	0
yaw	0
yb	0
yc	0
ycne	0
yf	0

yg	0
yhp	0
yl	0
ylb	0
ylba	0
ylf	0
yllac	0
ylp	0
ym	0
yn	0
yob	0
yr	0
yran	0
yrat	0
yre	0
yrot	0
yrt	0
yt	0
ytir	0
ytis	0
ytiv	0
ylba	0
ylf	0
yllac	0
ylp	0
ym	0
yn	0
yob	0
yr	0
yran	0
yrat	0
yre	0
yrot	0
yrt	0
yt	0
ytir	0
ytis	0
ytiv	0

The following is the reversed suffix entries in a readable format.

able
ably
ac
ade
after
age
al
ance
ant
ard
ate
bal
bel
ber
bic
black
ble
bly
book
boy
by
cade
cal
cally
can
cance
cant
case
cate
cence
cend
cent
cial
cian
cient
cies
cion
cious
cism
cit
cium
cive

cize
cle
corn
crat
cuit
cult
cute
cy
dant
dar
date
day
del
dence
dent
der
dial
dian
dit
dize
dle
dling
dom
down
dure
el
en
ence
ency
ent
er
ery
est
ever
fer
fic
fice
fied
fier
fies
fly
ful
fy
gal
gate
gent

geon
ger
gian
gion
gious
give
gize
gle
gous
gue
guish
gy
half
hold
hole
house
ian
ible
ic
ier
ies
ing
ish
ism
ken
ket
lace
lage
land
lar
late
lect
lent
ler
less
let
lic
lies
light
like
line
lish
lize
load
lope
lous

ly
lyse
lyze
made
main
mal
man
mate
men
ment
mial
mic
mise
mize
mous
mum
my
nance
nant
nary
nate
nent
ness
nial
nient
nies
nise
nish
nism
nist
nium
nize
nous
ny
other
ous
out
over
pate
pect
per
phere
phy
pipe
pla
place

plant
ple
plies
pling
ply
pond
que
quent
ral
rate
rial
ric
ries
rion
rior
rity
rium
rize
ry
sate
sen
sent
shape
shine
ship
shop
sian
sible
sic
sion
sious
sis
site
sity
sium
sive
son
sor
stage
stand
sure
tain
tal
tance
tary
tate

tect
teen
tel
tem
tent
ter
ther
thing
tial
tian
tic
tical
tice
ties
tige
time
tine
tion
tious
tise
tive
tize
tle
tone
tor
tory
tous
tral
trate
tries
tron
trum
try
tude
ture
tus
ty
vate
vent
vity
ward
way
work
worm
worn
yard
zen

APPENDIX C - PREFIX-ROOT STRING FILE

This is the formatted prefix-root word file.

It is stored on USR6 as PRE.TXT/38

(blank lines are pad charecter lines, 1 every 18 entries)

ab	0
abs	0
actu	0
ad	0
aero	0
ag	0
ah	0
an	0
ante	0
anti	0
any	0
ap	0
arbi	0
au	0
auto	0
bare	0
be	0
bio	0
blow	0
blue	0
book	0
bour	0
boy	0
by	0
cap	0
cen	0
cis	0
civ	0
co	0
col	0
com	0
con	0
corn	0
count	0
cov	0

cover	0
cul	0
cus	0
cut	0
cym	0
cyn	0
dark	0
day	0
de	0
dead	0
dean	0
dear	0
death	0
desp	0
dest	0
di	0
dic	0
dif	0
din	0
dis	0
doc	0
earth	0
econ	0
elec	0
emb	0
en	0
epi	0
es	0
every	0
ex	0
extra	0
eye	0
fail	0
fig	0
fire	0
fly	0
for	0
force	0
fore	0
forth	0
free	0
gen	0
geo	0

guar	0
guide	0
half	0
heart	0
hemi	0
here	0
home	0
hon	0
horse	0
house	0
hy	0
hyper	0
hypo	0
im	0
in	0
inch	0
inter	0
intro	0
iron	0
iso	0
key	0
land	0
lead	0
left	0
liber	0
life	0
line	0
long	0
mach	0
mag	0
mal	0
man	0
mat	0
match	0
mer	0
merge	0
meta	0
micro	0
min	0
mind	0
mis	0
mod	0
mono	0
multi	0

non	0
old	0
one	0
op	0
open	0
ortho	0
out	0
over	0
pan	0
panto	0
par	0
pea	0
per	0
peri	0
pheno	0
photo	0
phy	0
pipe	0
poly	0
poss	0
post	0
pre	0
pri	0
prin	0
pro	0
prob	0
prom	0
prop	0
pros	0
proto	0
pub	0
race	0
radi	0
re	0
reach	0
read	0
ream	0
reap	0
rear	0
retro	0
rose	0
safe	0
sand	0

sci	0
sea	0
sec	0
self	0
semi	0
sen	0
sep	0
ser	0
sharp	0
ship	0
shoe	0
shop	0
short	0
show	0
side	0
sig	0
sim	0
sin	0
sky	0
snake	0
snow	0
some	0
state	0
step	0
sub	0
sul	0
sun	0
super	0
sur	0
sus	0
sym	0
syn	0
sys	0
tele	0
ter	0
theo	0
there	0
time	0
trade	0
trans	0
trav	0
tri	0
turn	0
two	0

type	0
ultra	0
un	0
under	0
uni	0
up	0
util	0
ver	0
vice	0
vol	0
wave	0
where	0
white	0
wide	0
work	0

APPENDIX D - EXCEPTION DICTIONARY

This is the formatted exception dictionary.

It is stored on USR6 as EXCEPT.TXT

(The blanks are pad charecters, 1 every 18 entries)

about	02
above	02
absolute	03
acquire	03
against	02
allow	03
amount	02
analysis	02
analyze	03
analyzed	03
analyzing	03
asset	03
assume	03
auxiliary	04
benefit	04
between	03
called	00
capable	03
capabilities	05
changed	00
changes	00
changing	00
computer	04
course	00
crosses	00
declining	03
definition	04
deleted	03
deleting	03
delimiter	03
designation	04
determinable	03
dictionary	04
disadvantage	04
dividing	03

embodied	03
event	00
every	04
familiar	03
fashion	05
follow	04
force	00
foreign	04
greater	06
hierarchy	03
hyphenate	05
ideal	02
including	03
independent	03
inflationary	03
initiating	03
language	04
later	00
latest	00
longer	00
longest	00
looked	00
looking	00
machine	03
margin	04
matched	00
matching	00
metal	04
method	05
necessary	04
needed	00
negotiable	03
offset	04
often	03
operating	03
original	02
other	04
paragraph	05
parse	00
phrases	00
placed	00
places	00
possess	04

preferred	04
priced	00
prices	00
priority	04
processing	05
profit	05
profitability	05
purchase	04
reached	00
reaching	00
reader	05
reading	00
reconcile	04
reconciling	04
referring	03
regard	03
request	03
revenues	04
rewritting	03
searched	00
searches	00
sealed	05
secure	03
securities	03
security	03
select	03
separate	04
several	04
shares	00
simplest	04
software	05
spacing	00
specific	04
stated	00
states	00
target	04
technique	05
temporary	04
terminal	04
there	00
transferred	06
using	00
window	04

APPENDIX E - THE HYPHENATION PROCEDURE

```
; PROCEDURE HYPH
    ( VAR WORD_TABLE : SCANTABLE
    ; VAR AVAIL_LENG : INTEGER
    ; VAR HYPHEN_FLAG : BOOLEAN
    ; VAR WORD_TABLE_REMAINING : SCANTABLE
    ; VAR REMAINING_LENG : INTEGER
    ; VAR CONTINUE_FLAG : BOOLEAN
    )
  ; FORWARD

; PROCEDURE HYPH

  ; CONST SUFFIX_LONGEST = 5
    ; PREFIX_LONGEST = 5

  ; TYPE ARRAYCHAR = PACKED ARRAY [ 1 .. 18 , 1 .. 28 ] OF CHAR

  ; VAR I , J , K , L , M , HYPHEN , LENG , HIGH_RANGE_INDEX
       , LOW_RANGE_INDEX , VOWEL_COUNT , DOUBLE_CONSONANT
       , LENG_STRING , TABLE_FILLED_TO , TOTAL_LENG , AFFIX_NUM
       : INTEGER
    ; FIRSTFLAG , SEARCH_FLAG , MATCH_FOUND : BOOLEAN
    ; SVC1_IN , SVC1_SUFF , SVC1_PRE : SVC1_BLOCK
    ; EXCEPT , AFFIX : ARRAYCHAR
    ; FIRSTARRAY : ARRAY [ 1 .. 2 ] OF BOOLEAN
    ; VOWEL_SET , ENDCHAR : SET OF CHAR
    ; SHORT_TABLE : PACKED ARRAY [ 1 .. 25 ] OF CHAR

  ; PROCEDURE BREAKWORD ( BREAK_INDEX : INTEGER )

    ; VAR M : INTEGER

    ; BEGIN
        IF ( ( BREAK_INDEX <> 0 ) AND ( BREAK_INDEX <= AVAIL_LENG )
           )
        THEN BEGIN
          WORD_TABLE_REMAINING [ 1 ] := WORD_TABLE [ BREAK_INDEX ]
        ; WORD_TABLE [ BREAK_INDEX ] := '-'
        ; AVAIL_LENG := BREAK_INDEX
        ; HYPHEN_FLAG := TRUE
        ; CONTINUE_FLAG := FALSE
        ; J := 2
```

```
      ; FOR M := ( BREAK_INDEX + 1 ) TO TOTAL_LENG
        DO BEGIN
          WORD_TABLE_REMAINING [ J ] := WORD_TABLE [ M ]
        ; WORD_TABLE [ M ] := ' '
        ; J := J + 1
        END
      ; REMAINING_LENG := J - 1
      END
      ELSE CONTINUE_FLAG := FALSE
    END

; PROCEDURE SVC_LOAD
    ( BLOCK : SVC1_BLOCK
    ; ARRAY1 : ARRAYCHAR
    ; VAR TABLE_FILLED_TO : INTEGER
    )

  ; VAR I : INTEGER

  ; BEGIN
      I := 1
    ; IF ( BLOCK . SVC1_STAT <> 136 )
      THEN BEGIN
        SVC1 ( BLOCK )
      ; IF ( BLOCK . SVC1_STAT = 136 )
        THEN SEARCH_FLAG := FALSE
      ; I := 2
      ; REPEAT I := I + 1
        UNTIL ( ( ARRAY1 [ I - 1 , 1 ] > ARRAY1 [ I , 1 ] )
                OR ( I = 18 )
                OR ( ARRAY1 [ I , 1 ] < 'a' )
                OR ( ARRAY1 [ I , 1 ] > 'z' )
              )
      END
    ; TABLE_FILLED_TO := I
    END

; PROCEDURE MATCHSTRING
    ( BLOCK : SVC1_BLOCK
    ; ARRAY1 : ARRAYCHAR
    ; ARRAY2 : SCANTABLE
    ; LENG_STRING : INTEGER
    ; VAR AFFIX_NUM : INTEGER
    )

  ; VAR I , J , K , L : INTEGER
```

```
  ; NOT_LAST_ENTRY , IN_TABLE_FLAG : BOOLEAN

; PROCEDURE STRING_MATCHER

  ; BEGIN
      I := 1
    ; L := K
    ; WHILE ( ( IN_TABLE_FLAG ) AND ( L <= J ) )
      DO BEGIN
      ; WHILE ( ( IN_TABLE_FLAG )
                AND ( I <= LENG_STRING )
                AND ( ARRAY1 [ L , I ] = ARRAY2 [ I ] )
              )
        DO I := I + 1
      ; IF I = LENG_STRING + 1
        THEN BEGIN
          MATCH_FOUND := TRUE
        ; CONTINUE_FLAG := FALSE
        ; IN_TABLE_FLAG := FALSE
        ; TABLE_FILLED_TO := L
        END
        ELSE BEGIN I := 1 ; L := L + 1 END
      ; IF CONTINUE_FLAG
        THEN SEARCH_FLAG := FALSE
      END
    END

; PROCEDURE FIRST_LETTER_SEARCH

  ; BEGIN
    ; L := 1
    ; IN_TABLE_FLAG := FALSE
    ; I := 1
    ; J := TABLE_FILLED_TO
    ; REPEAT K := ( L + J ) DIV 2
      ; IF ARRAY1 [ K , I ] = ARRAY2 [ I ]
        THEN IN_TABLE_FLAG := TRUE
        ELSE IF ARRAY1 [ K , I ] < ARRAY2 [ I ]
          THEN L := K + 1
          ELSE J := K - 1
      UNTIL IN_TABLE_FLAG OR ( L > J )
    ; IF NOT IN_TABLE_FLAG
      THEN SEARCH_FLAG := FALSE
      ELSE BEGIN
        J := K
      ; WHILE ( ( ARRAY1 [ K , I ] = ARRAY2 [ I ] )
```

```
              AND ( K > 1 )
            )
      DO K := K - 1
    ; IF K = 1
      THEN AFFIX_NUM := 0
    ; WHILE ( ( ARRAY1 [ J , I ] = ARRAY2 [ I ] )
              AND ( J <= 17 )
            )
      DO J := J + 1
    ; J := J - 1
    ; STRING_MATCHER
    END
  END

; PROCEDURE LOADER

  ; BEGIN
    ; NOT_LAST_ENTRY := FALSE
    ; SEARCH_FLAG := TRUE
    ; SVC_LOAD ( BLOCK , ARRAY1 , TABLE_FILLED_TO )
    ; WHILE SEARCH_FLAG
      DO BEGIN
        IF ARRAY1 [ TABLE_FILLED_TO , 1 ] < ARRAY2 [ 1 ]
        THEN SVC_LOAD ( BLOCK , ARRAY1 , TABLE_FILLED_TO )
      ; IF ARRAY1 [ TABLE_FILLED_TO , 1 ] = ARRAY2 [ 1 ]
        THEN BEGIN
          I := 2
        ; WHILE ( ( I <= LENG_STRING )
                  AND ( ARRAY1 [ TABLE_FILLED_TO , I ]
                        = ARRAY2 [ I ]
                      )
                )
          DO I := I + 1
        ; IF I = LENG_STRING + 1
          THEN BEGIN
            CONTINUE_FLAG := FALSE
          ; SEARCH_FLAG := FALSE
          ; MATCH_FOUND := TRUE
          END
          ELSE IF ARRAY1 [ TABLE_FILLED_TO , I ] < ARRAY2 [ I ]
            THEN SVC_LOAD ( BLOCK , ARRAY1 , TABLE_FILLED_TO )
            ELSE NOT_LAST_ENTRY := TRUE
        END
      ; IF ( ( ( ARRAY1 [ TABLE_FILLED_TO , 1 ] > ARRAY2 [ 1 ] )
               OR ( NOT_LAST_ENTRY )
             )
```

```
            AND CONTINUE_FLAG
          )
        THEN FIRST_LETTER_SEARCH
      END
    END

; BEGIN
    MATCH_FOUND := FALSE
  ; IF AFFIX_NUM = 0
    THEN LOADER
    ELSE IF NOT FIRSTARRAY [ AFFIX_NUM ]
      THEN BEGIN FIRSTARRAY [ AFFIX_NUM ] := TRUE ; LOADER END
      ELSE STRING_MATCHER
  END

; PROCEDURE INITIALIZE

  ; BEGIN
      VOWEL_SET := [ 'a' , 'e' , 'i' , 'o' , 'u' , 'y' ]
    ; ENDCHAR := [ ',' , '.' , ':' , ';' , ' ' , '!' , '?' , NL ]
    ; WITH SVC1_IN
      DO BEGIN
        SVC1_FUNC := 73
      ; SVC1_LU := 3
      ; SVC1_STAT := 0
      ; SVC1_DEV_STAT := 0
      ; SVC1_BUFSTART := ADDRESS ( EXCEPT [ 1 , 1 ] )
      ; SVC1_BUFEND := ADDRESS ( EXCEPT [ 1 , 1 ] ) + 503
      END
    ; WITH SVC1_SUFF
      DO BEGIN
        SVC1_FUNC := 73
      ; SVC1_LU := 4
      ; SVC1_STAT := 0
      ; SVC1_DEV_STAT := 0
      ; SVC1_BUFSTART := ADDRESS ( AFFIX [ 1 , 1 ] )
      ; SVC1_BUFEND := ADDRESS ( AFFIX [ 1 , 1 ] ) + 503
      END
    ; WITH SVC1_PRE
      DO BEGIN
        SVC1_FUNC := 73
      ; SVC1_LU := 5
      ; SVC1_STAT := 0
      ; SVC1_DEV_STAT := 0
      ; SVC1_BUFSTART := ADDRESS ( AFFIX [ 1 , 1 ] )
      ; SVC1_BUFEND := ADDRESS ( AFFIX [ 1 , 1 ] ) + 503
```

```
      END
    ; I := 1
    ; FIRSTFLAG := FALSE
    ; FIRSTARRAY [ 1 ] := FALSE
    ; FIRSTARRAY [ 2 ] := FALSE
    ; HYPHEN_FLAG := FALSE
    ; CONTINUE_FLAG := TRUE
    ; DOUBLE_CONSONANT := 0
    ; HYPHEN := 0
    ; LENG := 0
    ; HIGH_RANGE_INDEX := 0
    ; LOW_RANGE_INDEX := 0
    ; VOWEL_COUNT := 0
    END

; PROCEDURE REJECT

  ; BEGIN
      K := 1
    ; WHILE ( ( WORD_TABLE [ K ] <> ' ' )
               AND ( WORD_TABLE [ K ] <> NL )
               AND ( WORD_TABLE [ K ] <> CR )
               AND ( WORD_TABLE [ K ] <> EM )
               AND ( K <> TABLE_LENGTH )
             )
      DO K := K + 1
    ; TOTAL_LENG := K
    ; WORD_TABLE [ K ] := ' '
    ; WHILE ( NOT ( WORD_TABLE [ I ] IN ENDCHAR )
               AND ( I <= TABLE_LENGTH )
             )
      DO BEGIN
        IF ( ( ( WORD_TABLE [ I ] < 'a' )
               OR ( WORD_TABLE [ I ] > 'z' )
             )
             AND ( WORD_TABLE [ I ] <> '-' )
           )
        THEN BEGIN
          HYPHEN_FLAG := FALSE
        ; CONTINUE_FLAG := FALSE
        END
      ; IF WORD_TABLE [ I ] = '-'
        THEN HYPHEN := I
      ; IF WORD_TABLE [ I ] IN VOWEL_SET
        THEN BEGIN
          VOWEL_COUNT := VOWEL_COUNT + 1
```

```
        ; IF NOT FIRSTFLAG
          THEN BEGIN
            LOW_RANGE_INDEX := I + 1
          ; FIRSTFLAG := TRUE
          END
          ELSE HIGH_RANGE_INDEX := I
        END
      ; LENG := LENG + 1
      ; I := I + 1
      END
    END

; PROCEDURE CK_HYPHEN

  ; BEGIN
      IF HYPHEN <> 0
      THEN BEGIN
        CONTINUE_FLAG := FALSE
      ; IF HYPHEN <= AVAIL_LENG
        THEN BEGIN
        ; AVAIL_LENG := HYPHEN
        ; J := 1
        ; FOR I := ( HYPHEN + 1 ) TO TOTAL_LENG
          DO BEGIN
            WORD_TABLE_REMAINING [ J ] := WORD_TABLE [ I ]
          ; J := J + 1
          END
        ; HYPHEN_FLAG := TRUE
        ; REMAINING_LENG := TOTAL_LENG - HYPHEN
        END
      END
    END

; PROCEDURE SPECIAL_REJECT

  ; BEGIN
      IF AVAIL_LENG < HIGH_RANGE_INDEX
      THEN HIGH_RANGE_INDEX := AVAIL_LENG
    ; IF ( ( VOWEL_COUNT <= 1 )
           OR ( HIGH_RANGE_INDEX <= LOW_RANGE_INDEX )
           OR ( LENG <= 3 )
         )
      THEN CONTINUE_FLAG := FALSE
    ; IF ( ( LENG <= 5 ) AND CONTINUE_FLAG )
      THEN BEGIN
        IF ( ( LENG = 5 )
```

```
            AND ( ( WORD_TABLE [ 5 ] = 's' )
                  OR ( WORD_TABLE [ 5 ] = 'd' )
                )
          )
        THEN CONTINUE_FLAG := FALSE
      ; IF ( ( LENG = 4 ) AND ( WORD_TABLE [ 4 ] <> 'y' ) )
        THEN CONTINUE_FLAG := FALSE
      END
    END

; PROCEDURE CK_DICTIONARY

  ; VAR K : INTEGER

  ; BEGIN
      AFFIX_NUM := 0
    ; MATCHSTRING
        ( SVC1_IN , EXCEPT , WORD_TABLE , LENG , AFFIX_NUM )
    ; IF MATCH_FOUND
      THEN BEGIN
        K := 0
      ; CASE EXCEPT [ TABLE_FILLED_TO , 26 ]
        OF
          '1' : K := K + 10
        ; '2' : K := K + 20
        ; ELSE : K := 0
        END
      ; CASE EXCEPT [ TABLE_FILLED_TO , 27 ]
        OF
          '0' , '1' , '2' , '3' , '4' , '5' , '6' , '7' , '8'
          , '9'
          : K
            := K
               + ( ORD ( EXCEPT [ TABLE_FILLED_TO , 27 ] ) - 48 )
        ; ELSE : K := 0
        END
      ; BREAKWORD ( K )
      END
    ; SVC1_IN . SVC1_FUNC := 192
    ; SVC1 ( SVC1_IN )
    END

; PROCEDURE CK_DOUBLE_CONSONANT

  ; BEGIN
      FOR I := 2 TO LENG
```

```
      DO BEGIN
        IF ( ( NOT ( WORD_TABLE [ I ] IN VOWEL_SET )
               AND ( ( WORD_TABLE [ I - 1 ] = WORD_TABLE [ I ] )
                     AND ( WORD_TABLE [ I ] <> 'l' )
                     AND ( WORD_TABLE [ I ] <> 's' )
                   )
             )
           )
        THEN DOUBLE_CONSONANT := I
        END
      ; IF ( ( DOUBLE_CONSONANT <> 0 )
             AND ( DOUBLE_CONSONANT <= HIGH_RANGE_INDEX )
             AND ( DOUBLE_CONSONANT >= LOW_RANGE_INDEX )
           )
        THEN BREAKWORD ( DOUBLE_CONSONANT )
      END

  ; PROCEDURE RECURSIVE_S_CK

    ; VAR STORE_WORD_TABLE : SCANTABLE
      ; STORE_LENG , STORE_TOTAL_LENG : INTEGER

    ; BEGIN
        IF ( ( WORD_TABLE [ LENG ] = 's' )
             AND ( WORD_TABLE [ LENG - 1 ] <> 's' )
           )
        THEN BEGIN
        ; STORE_LENG := LENG + 1
        ; STORE_TOTAL_LENG := TOTAL_LENG
        ; FOR I := 1 TO STORE_TOTAL_LENG
          DO STORE_WORD_TABLE [ I ] := WORD_TABLE [ I ]
        ; WORD_TABLE [ LENG ] := ' '
        ; HYPH
            ( WORD_TABLE , AVAIL_LENG , HYPHEN_FLAG
            , WORD_TABLE_REMAINING , REMAINING_LENG , CONTINUE_FLAG
            )
        ; IF HYPHEN_FLAG
          THEN BEGIN
            WORD_TABLE_REMAINING [ REMAINING_LENG ] := 's'
          ; REMAINING_LENG := REMAINING_LENG + 1
          ; FOR I := STORE_LENG TO STORE_TOTAL_LENG
            DO BEGIN
              WORD_TABLE_REMAINING [ REMAINING_LENG ]
              := STORE_WORD_TABLE [ I ]
            ; REMAINING_LENG := REMAINING_LENG + 1
            END
```

```
        ; REMAINING_LENG := REMAINING_LENG - 1
        ; CONTINUE_FLAG := FALSE
        END
        ELSE WORD_TABLE [ LENG ] := 's'
      END
    END

; PROCEDURE CK_SUFFIX

  ; BEGIN
      IF ( LENG - ( SUFFIX_LONGEST - 1 ) ) < LOW_RANGE_INDEX
      THEN K := LOW_RANGE_INDEX
      ELSE K := LENG - ( SUFFIX_LONGEST - 1 )
    ; AFFIX_NUM := 1
    ; WHILE ( ( K <= HIGH_RANGE_INDEX ) AND CONTINUE_FLAG )
      DO BEGIN
        J := 1
      ; M := LENG
      ; FOR I := K TO LENG
        DO BEGIN
          SHORT_TABLE [ J ] := WORD_TABLE [ M ]
        ; J := J + 1
        ; M := M - 1
        END
      ; LENG_STRING := J
      ; SHORT_TABLE [ J ] := ' '
      ; MATCHSTRING
          ( SVC1_SUFF , AFFIX , SHORT_TABLE , LENG_STRING
          , AFFIX_NUM )
      ; IF MATCH_FOUND
        THEN BREAKWORD ( K )
      ; K := K + 1
      ; IF AFFIX_NUM <> 1
        THEN BEGIN
        ; SVC1_SUFF . SVC1_FUNC := 192
        ; SVC1 ( SVC1_SUFF )
        ; SVC1_SUFF . SVC1_FUNC := 72
        END
      END
    ; SVC1_SUFF . SVC1_FUNC := 192
    ; SVC1 ( SVC1_SUFF )
    END

; PROCEDURE CK_PREFIX

  ; BEGIN
```

```
      IF ( HIGH_RANGE_INDEX - 1 ) < PREFIX_LONGEST
      THEN I := ( HIGH_RANGE_INDEX - 1 )
      ELSE I := PREFIX_LONGEST
    ; AFFIX_NUM := 2
    ; WHILE ( ( I >= ( LOW_RANGE_INDEX - 1 ) ) AND CONTINUE_FLAG )
      DO BEGIN
        J := 1
      ; FOR K := 1 TO I
        DO BEGIN
          SHORT_TABLE [ J ] := WORD_TABLE [ K ]
        ; J := J + 1
        END
      ; LENG_STRING := J
      ; SHORT_TABLE [ J ] := ' '
      ; MATCHSTRING
          ( SVC1_PRE , AFFIX , SHORT_TABLE , LENG_STRING
          , AFFIX_NUM )
      ; IF MATCH_FOUND
        THEN BREAKWORD ( LENG_STRING )
      ; I := I - 1
      ; IF AFFIX_NUM <> 2
        THEN BEGIN
        ; SVC1_PRE . SVC1_FUNC := 192
        ; SVC1 ( SVC1_PRE )
        ; SVC1_PRE . SVC1_FUNC := 72
        END
      END
    ; SVC1_PRE . SVC1_FUNC := 192
    ; SVC1 ( SVC1_PRE )
    END

; PROCEDURE CK_BREAK_VALUE_TABLE

  ; BEGIN
      I := LOW_RANGE_INDEX - 1
    ; J := LOW_RANGE_INDEX
    ; WHILE ( ( J <= HIGH_RANGE_INDEX ) AND CONTINUE_FLAG )
      DO BEGIN
        L := ORD ( WORD_TABLE [ I ] ) - 96
      ; K := ORD ( WORD_TABLE [ J ] ) - 96
      ; IF BREAK_VALUE_TABLE [ L , K ]
        THEN BREAKWORD ( J )
      ; I := I + 1
      ; J := I + 1
      END
    END
```

```
    ; BEGIN
        INITIALIZE
      ; REJECT
      ; CK_HYPHEN
      ; IF CONTINUE_FLAG
        THEN SPECIAL_REJECT
      ; IF CONTINUE_FLAG
        THEN CK_DICTIONARY
      ; IF CONTINUE_FLAG
        THEN CK_DOUBLE_CONSONANT
      ; IF CONTINUE_FLAG
        THEN RECURSIVE_S_CK
      ; IF CONTINUE_FLAG
        THEN CK_SUFFIX
      ; IF CONTINUE_FLAG
        THEN CK_PREFIX
      ; IF CONTINUE_FLAG
        THEN CK_BREAK_VALUE_TABLE
(*  ED  AND  ES CHECK  *)
      ; IF ( ( CONTINUE_FLAG )
             AND ( ( WORD_TABLE [ LENG - 1 ] = 'e' )
                   AND ( ( WORD_TABLE [ LENG ] = 'd' )
                         OR ( WORD_TABLE [ LENG ] = 's' )
                       )
                 )
           )
        THEN BREAKWORD ( LENG - 1 )
      END
```

APPENDIX F - LINEBUILDER PROGRAM

This program uses the hyphenator procedure to build printable text.

It shows how to integrate the hyphenator into a word processor.

```
      1
      2 "ROBERT YOUNG"
      3 "KANSAS STATE UNIVERSITY"
      4   CONST COPYRIGHT = 'COPYRIGHT ROBERT YOUNG 1978'
      5
      6 "###########
C     7 #  PREFIX  #
C     8 ###########"
      9
     10 ; CONST NL = '(:10:)'
     11   ; FF = '(:12:)'
     12   ; CR = '(:13:)'
     13   ; EM = '(:25:)'
     14
     15 ; CONST PAGELENGTH = 512
     16
     17 ; TYPE PAGE = ARRAY (. 1 .. PAGELENGTH .) OF CHAR
     18
     19 ; CONST LINELENGTH = 132
     20
     21 ; TYPE LINE = ARRAY (. 1 .. LINELENGTH .) OF CHAR
     22
     23 ; CONST IDLENGTH = 12
     24
     25 ; TYPE IDENTIFIER = ARRAY (. 1 .. IDLENGTH .) OF CHAR
     26
     27 ; TYPE FILE = 1 .. 2
     28
     29 ; TYPE FILEKIND = ( EMPTY , SCRATCH , ASCII , SEQCODE , CONCODE )
     30
     31 ; TYPE FILEATTR
     32     = RECORD
     33         KIND : FILEKIND
     34       ; ADDR : INTEGER
     35       ; PROTECTED : BOOLEAN
     36       ; NOTUSED : ARRAY (. 1 .. 5 .) OF INTEGER
     37       END
```

```
38
39 ; TYPE IODEVICE
40     = ( TYPEDEVICE , DISKDEVICE , TAPEDEVICE , PRINTDEVICE
41       , CARDDEVICE )
42
43 ; TYPE IOOPERATION = ( INPUT , OUTPUT , MOVE , CONTROL )
44
45 ; TYPE IOARG = ( WRITEEOF , REWIND , UPSPACE , BACKSPACE )
46
47 ; TYPE IORESULT
48     = ( COMPLETE , INTERVENTION , TRANSMISSION , FAILURE , ENDFILE
49       , ENDMEDIUM , STARTMEDIUM )
50
51 ; TYPE IOPARAM
52     = RECORD
53         OPERATION : IOOPERATION
54       ; STATUS : IORESULT
55       ; ARG : IOARG
56       END
57
58 ; TYPE TASKKIND = ( INPUTTASK , JOBTASK , OUTPUTTASK )
59
60 ; TYPE ARGTAG = ( NILTYPE , BOOLTYPE , INTTYPE , IDTYPE , PTRTYPE )
61
62 ; TYPE POINTER = @ BOOLEAN
63
64 ; TYPE PASSPTR = @ PASSLINK
65
66 ; TYPE PASSLINK
67     = RECORD
68         OPTIONS : SET OF CHAR
69       ; FILLER1 : ARRAY (. 1 .. 7 .) OF INTEGER
70       ; FILLER2 : BOOLEAN
71       ; RESET_POINT : INTEGER
72       ; FILLER3 : ARRAY (. 1 .. 11 .) OF POINTER
73       END
74
75 ; TYPE ARGTYPE
76     = RECORD
77         CASE TAG : ARGTAG
78         OF NILTYPE , BOOLTYPE : ( BOOL : BOOLEAN )
79         ; INTTYPE : ( INT : INTEGER )
80         ; IDTYPE : ( ID : IDENTIFIER )
81         ; PTRTYPE : ( PTR : PASSPTR )
82       END
83
```

```
 84 ; CONST MAXARG = 10
 85
 86 ; TYPE ARGLIST = ARRAY (. 1 .. MAXARG .) OF ARGTYPE
 87
 88 ; TYPE ARGSEQ = ( INP , OUT )
 89
 90 ; TYPE PROGRESULT
 91     = ( TERMINATED , OVERFLOW , POINTERERROR , RANGEERROR
 92       , VARIANTERROR , HEAPLIMIT , STACKLIMIT , CODELIMIT , TIMELIMIT
 93       , CALLERROR )
 94
 95 ; PROCEDURE READ ( VAR C : CHAR )
 96
 97 ; PROCEDURE WRITE ( C : CHAR )
 98
 99 ; PROCEDURE OPEN ( F : FILE ; ID : IDENTIFIER ; VAR FOUND : BOOLEAN )
100
101 ; PROCEDURE CLOSE ( F : FILE )
102
103 ; PROCEDURE GET ( F : FILE ; P : INTEGER ; VAR BLOCK : UNIV PAGE )
104
105 ; PROCEDURE PUT ( F : FILE ; P : INTEGER ; VAR BLOCK : UNIV PAGE )
106
107 ; FUNCTION LENGTH ( F : FILE ) : INTEGER
108
109 ; PROCEDURE MARK ( VAR TOP : INTEGER )
110
111 ; PROCEDURE RELEASE ( TOP : INTEGER )
112
113 ; PROCEDURE IDENTIFY ( HEADER : LINE )
114
115 ; PROCEDURE ACCEPT ( VAR C : CHAR )
116
117 ; PROCEDURE DISPLAY ( C : CHAR )
118
119 ; PROCEDURE NOTUSED
120
121 ; PROCEDURE NOTUSED2
122
123 ; PROCEDURE NOTUSED3
124
125 ; PROCEDURE NOTUSED4
126
127 ; PROCEDURE NOTUSED5
128
129 ; PROCEDURE NOTUSED6
```

```
130
131 ; PROCEDURE NOTUSED7
132
133 ; PROCEDURE NOTUSED8
134
135 ; PROCEDURE NOTUSED9
136
137 ; PROCEDURE NOTUSED10
138
139 ; PROCEDURE RUN
140     ( ID : IDENTIFIER
141     ; VAR PARAM : ARGLIST
142     ; VAR LINE : INTEGER
143     ; VAR RESULT : PROGRESULT
144     )
145
146 ; PROGRAM HYPHNO1 ( PARAM : LINE )
147
148   ; CONST TABLE_LENGTH = 25
149     ; LINE_LIMIT = 72
150
151   ; TYPE SCANTABLE = PACKED ARRAY [ 1 .. TABLE_LENGTH ] OF CHAR
152     ; SVC1_BLOCK
153       = RECORD
154           SVC1_FUNC : BYTE
155         ; SVC1_LU : BYTE
156         ; SVC1_STAT : BYTE
157         ; SVC1_DEV_STAT : BYTE
158         ; SVC1_BUFSTART : INTEGER
159         ; SVC1_BUFEND : INTEGER
160         ; SVC1_RANDOM_ADDR : INTEGER
161         ; SVC1_XFER_LEN : INTEGER
162         ; SVC1_RESERVED : INTEGER
163         ;
164         END
165
166   ; VAR I , J , K : INTEGER
167     ; HYPHEN_FLAG , STOPPED_FLAG : BOOLEAN
168     ; WORD_COUNT : INTEGER
169     ; WORD_COUNT_REMAINING : INTEGER
170     ; Z : CHAR
171     ; CHAR_TABLE , CHAR_TABLE_REMAINING : SCANTABLE
172     ; STORE_COUNT : INTEGER
173     ; BREAK_VALUE_TABLE : ARRAY [ 1 .. 26 , 1 .. 26 ] OF BOOLEAN
174     ; RIGHT , LINE_START : BOOLEAN
175     ; LINE_COUNT , SPACE_COUNT , BUFFER_COUNT , AMT_EACH_SPACE
```

```
176       , LEFT_OVER_SPACES , WAIT , NEXTSPACE
177       : INTEGER
178     ; SPACE_TABLE : ARRAY [ 1 .. 30 ] OF INTEGER
179     ; LINE_BUFFER : PACKED ARRAY [ 1 .. 200 ] OF CHAR
180
181   ; PROCEDURE INIT_BREAKVT
182
183     ; BEGIN
184         FOR I := 1 TO 26
185         DO FOR J := 1 TO 26 DO BREAK_VALUE_TABLE [ I , J ] := TRUE
186       ; FOR I := 1 TO 14
187         DO BEGIN
188           BREAK_VALUE_TABLE [ 1 , I ] := FALSE
189         ; BREAK_VALUE_TABLE [ I , 1 ] := FALSE
190         ; BREAK_VALUE_TABLE [ 5 , I ] := FALSE
191         ; BREAK_VALUE_TABLE [ I , 5 ] := FALSE
192         ; BREAK_VALUE_TABLE [ 9 , I ] := FALSE
193         ; BREAK_VALUE_TABLE [ I , 9 ] := FALSE
194         ; BREAK_VALUE_TABLE [ 15 , I ] := FALSE
195         ; BREAK_VALUE_TABLE [ I , 15 ] := FALSE
196         ; BREAK_VALUE_TABLE [ 21 , I ] := FALSE
197         ; BREAK_VALUE_TABLE [ I , 21 ] := FALSE
198         END
199       ; FOR I := 15 TO 23
200         DO BEGIN
201           BREAK_VALUE_TABLE [ I , 1 ] := FALSE
202         ; BREAK_VALUE_TABLE [ I , 5 ] := FALSE
203         ; BREAK_VALUE_TABLE [ I , 9 ] := FALSE
204         ; BREAK_VALUE_TABLE [ I , 15 ] := FALSE
205         ; BREAK_VALUE_TABLE [ I , 21 ] := FALSE
206         END
207       ; FOR I := 18 TO 24
208         DO BEGIN
209           BREAK_VALUE_TABLE [ 1 , I ] := FALSE
210         ; BREAK_VALUE_TABLE [ 5 , I ] := FALSE
211         ; BREAK_VALUE_TABLE [ 9 , I ] := FALSE
212         ; BREAK_VALUE_TABLE [ 15 , I ] := FALSE
213         ; BREAK_VALUE_TABLE [ 21 , I ] := FALSE
214         END
215       ; BREAK_VALUE_TABLE [ 12 , 4 ] := FALSE
216       ; BREAK_VALUE_TABLE [ 14 , 3 ] := FALSE
217       ; BREAK_VALUE_TABLE [ 14 , 4 ] := FALSE
218       ; BREAK_VALUE_TABLE [ 18 , 3 ] := FALSE
219       ; BREAK_VALUE_TABLE [ 18 , 4 ] := FALSE
220       ; BREAK_VALUE_TABLE [ 19 , 3 ] := FALSE
221       ; BREAK_VALUE_TABLE [ 20 , 3 ] := FALSE
```

```
222       ; BREAK_VALUE_TABLE [ 25 , 1 ] := FALSE
223       ; BREAK_VALUE_TABLE [ 26 , 1 ] := FALSE
224       ; BREAK_VALUE_TABLE [ 3 , 8 ] := FALSE
225       ; BREAK_VALUE_TABLE [ 7 , 8 ] := FALSE
226       ; BREAK_VALUE_TABLE [ 14 , 7 ] := FALSE
227       ; BREAK_VALUE_TABLE [ 16 , 8 ] := FALSE
228       ; BREAK_VALUE_TABLE [ 18 , 7 ] := FALSE
229       ; BREAK_VALUE_TABLE [ 19 , 8 ] := FALSE
230       ; BREAK_VALUE_TABLE [ 20 , 8 ] := FALSE
231       ; BREAK_VALUE_TABLE [ 2 , 18 ] := FALSE
232       ; BREAK_VALUE_TABLE [ 3 , 18 ] := FALSE
233       ; BREAK_VALUE_TABLE [ 4 , 18 ] := FALSE
234       ; BREAK_VALUE_TABLE [ 6 , 18 ] := FALSE
235       ; BREAK_VALUE_TABLE [ 7 , 18 ] := FALSE
236       ; BREAK_VALUE_TABLE [ 8 , 18 ] := FALSE
237       ; BREAK_VALUE_TABLE [ 16 , 18 ] := FALSE
238       ; BREAK_VALUE_TABLE [ 17 , 18 ] := FALSE
239       ; BREAK_VALUE_TABLE [ 23 , 8 ] := FALSE
240       ; BREAK_VALUE_TABLE [ 26 , 5 ] := FALSE
241       ; BREAK_VALUE_TABLE [ 25 , 9 ] := FALSE
242       ; BREAK_VALUE_TABLE [ 26 , 9 ] := FALSE
243       ; BREAK_VALUE_TABLE [ 2 , 12 ] := FALSE
244       ; BREAK_VALUE_TABLE [ 3 , 11 ] := FALSE
245       ; BREAK_VALUE_TABLE [ 3 , 12 ] := FALSE
246       ; BREAK_VALUE_TABLE [ 4 , 12 ] := FALSE
247       ; BREAK_VALUE_TABLE [ 6 , 12 ] := FALSE
248       ; BREAK_VALUE_TABLE [ 7 , 12 ] := FALSE
249       ; BREAK_VALUE_TABLE [ 7 , 14 ] := FALSE
250       ; BREAK_VALUE_TABLE [ 11 , 14 ] := FALSE
251       ; BREAK_VALUE_TABLE [ 12 , 12 ] := FALSE
252       ; BREAK_VALUE_TABLE [ 12 , 11 ] := FALSE
253       ; BREAK_VALUE_TABLE [ 14 , 11 ] := FALSE
254       ; BREAK_VALUE_TABLE [ 16 , 12 ] := FALSE
255       ; BREAK_VALUE_TABLE [ 18 , 13 ] := FALSE
256       ; BREAK_VALUE_TABLE [ 18 , 14 ] := FALSE
257       ; BREAK_VALUE_TABLE [ 20 , 18 ] := FALSE
258       ; BREAK_VALUE_TABLE [ 23 , 18 ] := FALSE
259       ; BREAK_VALUE_TABLE [ 25 , 18 ] := FALSE
260       ; BREAK_VALUE_TABLE [ 11 , 19 ] := FALSE
261       ; BREAK_VALUE_TABLE [ 12 , 19 ] := FALSE
262       ; BREAK_VALUE_TABLE [ 14 , 19 ] := FALSE
263       ; BREAK_VALUE_TABLE [ 18 , 19 ] := FALSE
264       ; BREAK_VALUE_TABLE [ 19 , 19 ] := FALSE
265       ; BREAK_VALUE_TABLE [ 23 , 19 ] := FALSE
266       ; BREAK_VALUE_TABLE [ 25 , 19 ] := FALSE
267       ; BREAK_VALUE_TABLE [ 19 , 11 ] := FALSE
```

```
268       ; BREAK_VALUE_TABLE [ 19 , 14 ] := FALSE
269       ; BREAK_VALUE_TABLE [ 20 , 12 ] := FALSE
270       ; BREAK_VALUE_TABLE [ 25 , 12 ] := FALSE
271       ; BREAK_VALUE_TABLE [ 25 , 13 ] := FALSE
272       ; BREAK_VALUE_TABLE [ 25 , 14 ] := FALSE
273       ; BREAK_VALUE_TABLE [ 25 , 15 ] := FALSE
274       ; BREAK_VALUE_TABLE [ 1 , 16 ] := FALSE
275       ; BREAK_VALUE_TABLE [ 26 , 15 ] := FALSE
276       ; BREAK_VALUE_TABLE [ 5 , 16 ] := FALSE
277       ; BREAK_VALUE_TABLE [ 9 , 16 ] := FALSE
278       ; BREAK_VALUE_TABLE [ 13 , 16 ] := FALSE
279       ; BREAK_VALUE_TABLE [ 15 , 16 ] := FALSE
280       ; BREAK_VALUE_TABLE [ 19 , 16 ] := FALSE
281       ; BREAK_VALUE_TABLE [ 19 , 17 ] := FALSE
282       ; BREAK_VALUE_TABLE [ 21 , 16 ] := FALSE
283       ; BREAK_VALUE_TABLE [ 24 , 16 ] := FALSE
284       ; BREAK_VALUE_TABLE [ 20 , 23 ] := FALSE
285       ; BREAK_VALUE_TABLE [ 21 , 26 ] := FALSE
286       ; BREAK_VALUE_TABLE [ 23 , 25 ] := FALSE
287       ; BREAK_VALUE_TABLE [ 24 , 25 ] := FALSE
288       ; BREAK_VALUE_TABLE [ 25 , 25 ] := FALSE
289       ; BREAK_VALUE_TABLE [ 3 , 20 ] := FALSE
290       ; BREAK_VALUE_TABLE [ 6 , 20 ] := FALSE
291       ; BREAK_VALUE_TABLE [ 8 , 20 ] := FALSE
292       ; BREAK_VALUE_TABLE [ 14 , 20 ] := FALSE
293       ; BREAK_VALUE_TABLE [ 16 , 20 ] := FALSE
294       ; BREAK_VALUE_TABLE [ 18 , 20 ] := FALSE
295       ; BREAK_VALUE_TABLE [ 19 , 20 ] := FALSE
296       ; BREAK_VALUE_TABLE [ 25 , 20 ] := FALSE
297       ; BREAK_VALUE_TABLE [ 25 , 21 ] := FALSE
298       ; BREAK_VALUE_TABLE [ 26 , 21 ] := FALSE
299       ; BREAK_VALUE_TABLE [ 1 , 25 ] := FALSE
300       ; BREAK_VALUE_TABLE [ 1 , 26 ] := FALSE
301       ; BREAK_VALUE_TABLE [ 3 , 26 ] := FALSE
302       ; BREAK_VALUE_TABLE [ 5 , 26 ] := FALSE
303       ; BREAK_VALUE_TABLE [ 6 , 25 ] := FALSE
304       ; BREAK_VALUE_TABLE [ 9 , 26 ] := FALSE
305       ; BREAK_VALUE_TABLE [ 12 , 22 ] := FALSE
306       ; BREAK_VALUE_TABLE [ 12 , 25 ] := FALSE
307       ; BREAK_VALUE_TABLE [ 15 , 25 ] := FALSE
308       ; BREAK_VALUE_TABLE [ 15 , 26 ] := FALSE
309       ; BREAK_VALUE_TABLE [ 17 , 25 ] := FALSE
310       ; BREAK_VALUE_TABLE [ 18 , 22 ] := FALSE
311       ; BREAK_VALUE_TABLE [ 18 , 25 ] := FALSE
312       ; BREAK_VALUE_TABLE [ 18 , 26 ] := FALSE
313       ; BREAK_VALUE_TABLE [ 19 , 23 ] := FALSE
```

```
314       ; BREAK_VALUE_TABLE [ 1 , 15 ] := TRUE
315       END
316
317   ; PROCEDURE SVC1 ( SVC1_IN : SVC1_BLOCK )
318     ; EXTERN
319
320   ; PROCEDURE HYPH
321       ( VAR WORD_TABLE : SCANTABLE
322       ; VAR AVAIL_LENG : INTEGER
323       ; VAR HYPHEN_FLAG : BOOLEAN
324       ; VAR WORD_TABLE_REMAINING : SCANTABLE
325       ; VAR REMAINING_LENG : INTEGER
326       ; VAR CONTINUE_FLAG : BOOLEAN
327       )
328     ; FORWARD
329
330   ; PROCEDURE HYPH
331
332     ; CONST SUFFIX_LONGEST = 5
333       ; PREFIX_LONGEST = 5
334
335     ; TYPE ARRAYCHAR = PACKED ARRAY [ 1 .. 18 , 1 .. 28 ] OF CHAR
336
337     ; VAR I , J , K , L , M , HYPHEN , LENG , HIGH_RANGE_INDEX
338           , LOW_RANGE_INDEX , VOWEL_COUNT , DOUBLE_CONSONANT
339           , LENG_STRING , TABLE_FILLED_TO , TOTAL_LENG , AFFIX_NUM
340           : INTEGER
341       ; FIRSTFLAG , SEARCH_FLAG , MATCH_FOUND : BOOLEAN
342       ; SVC1_IN , SVC1_SUFF , SVC1_PRE : SVC1_BLOCK
343       ; EXCEPT , AFFIX : ARRAYCHAR
344       ; FIRSTARRAY : ARRAY [ 1 .. 2 ] OF BOOLEAN
345       ; VOWEL_SET , ENDCHAR : SET OF CHAR
346       ; SHORT_TABLE : PACKED ARRAY [ 1 .. 25 ] OF CHAR
347
348     ; PROCEDURE BREAKWORD ( BREAK_INDEX : INTEGER )
349
350       ; VAR M : INTEGER
351
352       ; BEGIN
353           IF ( ( BREAK_INDEX <> 0 ) AND ( BREAK_INDEX <= AVAIL_LENG )
354              )
355           THEN BEGIN
356             WORD_TABLE_REMAINING [ 1 ] := WORD_TABLE [ BREAK_INDEX ]
357           ; WORD_TABLE [ BREAK_INDEX ] := '-'
358           ; AVAIL_LENG := BREAK_INDEX
359           ; HYPHEN_FLAG := TRUE
```

```
360           ; CONTINUE_FLAG := FALSE
361           ; J := 2
362           ; FOR M := ( BREAK_INDEX + 1 ) TO TOTAL_LENG
363             DO BEGIN
364               WORD_TABLE_REMAINING [ J ] := WORD_TABLE [ M ]
365             ; WORD_TABLE [ M ] := ' '
366             ; J := J + 1
367             END
368           ; REMAINING_LENG := J - 1
369           END
370           ELSE CONTINUE_FLAG := FALSE
371         END
372
373       ; PROCEDURE SVC_LOAD
374           ( BLOCK : SVC1_BLOCK
375           ; ARRAY1 : ARRAYCHAR
376           ; VAR TABLE_FILLED_TO : INTEGER
377           )
378
379         ; VAR I : INTEGER
380
381         ; BEGIN
382             I := 1
383           ; IF ( BLOCK . SVC1_STAT <> 136 )
384             THEN BEGIN
385               SVC1 ( BLOCK )
386             ; IF ( BLOCK . SVC1_STAT = 136 )
387               THEN SEARCH_FLAG := FALSE
388             ; I := 2
389             ; REPEAT I := I + 1
390               UNTIL ( ( ARRAY1 [ I - 1 , 1 ] > ARRAY1 [ I , 1 ] )
391                       OR ( I = 18 )
392                       OR ( ARRAY1 [ I , 1 ] < 'a' )
393                       OR ( ARRAY1 [ I , 1 ] > 'z' )
394                     )
395             END
396           ; TABLE_FILLED_TO := I
397           END
398
399       ; PROCEDURE MATCHSTRING
400           ( BLOCK : SVC1_BLOCK
401           ; ARRAY1 : ARRAYCHAR
402           ; ARRAY2 : SCANTABLE
403           ; LENG_STRING : INTEGER
404           ; VAR AFFIX_NUM : INTEGER
405           )
```

```
406
407       ; VAR I , J , K , L : INTEGER
408         ; NOT_LAST_ENTRY , IN_TABLE_FLAG : BOOLEAN
409
410       ; PROCEDURE STRING_MATCHER
411
412         ; BEGIN
413             I := 1
414           ; L := K
415           ; WHILE ( ( IN_TABLE_FLAG ) AND ( L <= J ) )
416             DO BEGIN
417             ; WHILE ( ( IN_TABLE_FLAG )
418                       AND ( I <= LENG_STRING )
419                       AND ( ARRAY1 [ L , I ] = ARRAY2 [ I ] )
420                     )
421               DO I := I + 1
422             ; IF I = LENG_STRING + 1
423               THEN BEGIN
424                 MATCH_FOUND := TRUE
425               ; CONTINUE_FLAG := FALSE
426               ; IN_TABLE_FLAG := FALSE
427               ; TABLE_FILLED_TO := L
428               END
429               ELSE BEGIN I := 1 ; L := L + 1 END
430             ; IF CONTINUE_FLAG
431               THEN SEARCH_FLAG := FALSE
432             END
433           END
434
435       ; PROCEDURE FIRST_LETTER_SEARCH
436
437         ; BEGIN
438           ; L := 1
439           ; IN_TABLE_FLAG := FALSE
440           ; I := 1
441           ; J := TABLE_FILLED_TO
442           ; REPEAT K := ( L + J ) DIV 2
443             ; IF ARRAY1 [ K , I ] = ARRAY2 [ I ]
444               THEN IN_TABLE_FLAG := TRUE
445               ELSE IF ARRAY1 [ K , I ] < ARRAY2 [ I ]
446                 THEN L := K + 1
447                 ELSE J := K - 1
448             UNTIL IN_TABLE_FLAG OR ( L > J )
449           ; IF NOT IN_TABLE_FLAG
450             THEN SEARCH_FLAG := FALSE
451             ELSE BEGIN
```

```
452             J := K
453           ; WHILE ( ( ARRAY1 [ K , I ] = ARRAY2 [ I ] )
454                     AND ( K > 1 )
455                   )
456             DO K := K - 1
457           ; IF K = 1
458             THEN AFFIX_NUM := 0
459           ; WHILE ( ( ARRAY1 [ J , I ] = ARRAY2 [ I ] )
460                     AND ( J <= 17 )
461                   )
462             DO J := J + 1
463           ; J := J - 1
464           ; STRING_MATCHER
465           END
466         END
467
468       ; PROCEDURE LOADER
469
470         ; BEGIN
471           ; NOT_LAST_ENTRY := FALSE
472           ; SEARCH_FLAG := TRUE
473           ; SVC_LOAD ( BLOCK , ARRAY1 , TABLE_FILLED_TO )
474           ; WHILE SEARCH_FLAG
475             DO BEGIN
476               IF ARRAY1 [ TABLE_FILLED_TO , 1 ] < ARRAY2 [ 1 ]
477               THEN SVC_LOAD ( BLOCK , ARRAY1 , TABLE_FILLED_TO )
478             ; IF ARRAY1 [ TABLE_FILLED_TO , 1 ] = ARRAY2 [ 1 ]
479               THEN BEGIN
480                 I := 2
481               ; WHILE ( ( I <= LENG_STRING )
482                         AND ( ARRAY1 [ TABLE_FILLED_TO , I ]
483                               = ARRAY2 [ I ]
484                             )
485                       )
486                 DO I := I + 1
487               ; IF I = LENG_STRING + 1
488                 THEN BEGIN
489                   CONTINUE_FLAG := FALSE
490                 ; SEARCH_FLAG := FALSE
491                 ; MATCH_FOUND := TRUE
492                 END
493                 ELSE IF ARRAY1 [ TABLE_FILLED_TO , I ] < ARRAY2 [ I ]
494                   THEN SVC_LOAD ( BLOCK , ARRAY1 , TABLE_FILLED_TO )
495                   ELSE NOT_LAST_ENTRY := TRUE
496               END
497             ; IF ( ( ( ARRAY1 [ TABLE_FILLED_TO , 1 ] > ARRAY2 [ 1 ] )
```

```
498                     OR ( NOT_LAST_ENTRY )
499                   )
500                   AND CONTINUE_FLAG
501                 )
502               THEN FIRST_LETTER_SEARCH
503             END
504           END
505
506       ; BEGIN
507           MATCH_FOUND := FALSE
508         ; IF AFFIX_NUM = 0
509           THEN LOADER
510           ELSE IF NOT FIRSTARRAY [ AFFIX_NUM ]
511             THEN BEGIN FIRSTARRAY [ AFFIX_NUM ] := TRUE ; LOADER END
512             ELSE STRING_MATCHER
513         END
514
515     ; PROCEDURE INITIALIZE
516
517         ; BEGIN
518             VOWEL_SET := [ 'a' , 'e' , 'i' , 'o' , 'u' , 'y' ]
519           ; ENDCHAR := [ ',' , '.' , ':' , ';' , ' ' , '!' , '?' , NL ]
520           ; WITH SVC1_IN
521             DO BEGIN
522               SVC1_FUNC := 73
523             ; SVC1_LU := 3
524             ; SVC1_STAT := 0
525             ; SVC1_DEV_STAT := 0
526             ; SVC1_BUFSTART := ADDRESS ( EXCEPT [ 1 , 1 ] )
527             ; SVC1_BUFEND := ADDRESS ( EXCEPT [ 1 , 1 ] ) + 503
528             END
529           ; WITH SVC1_SUFF
530             DO BEGIN
531               SVC1_FUNC := 73
532             ; SVC1_LU := 4
533             ; SVC1_STAT := 0
534             ; SVC1_DEV_STAT := 0
535             ; SVC1_BUFSTART := ADDRESS ( AFFIX [ 1 , 1 ] )
536             ; SVC1_BUFEND := ADDRESS ( AFFIX [ 1 , 1 ] ) + 503
537             END
538           ; WITH SVC1_PRE
539             DO BEGIN
540               SVC1_FUNC := 73
541             ; SVC1_LU := 5
542             ; SVC1_STAT := 0
543             ; SVC1_DEV_STAT := 0
```

```
544           ; SVC1_BUFSTART := ADDRESS ( AFFIX [ 1 , 1 ] )
545           ; SVC1_BUFEND := ADDRESS ( AFFIX [ 1 , 1 ] ) + 503
546           END
547         ; I := 1
548         ; FIRSTFLAG := FALSE
549         ; FIRSTARRAY [ 1 ] := FALSE
550         ; FIRSTARRAY [ 2 ] := FALSE
551         ; HYPHEN_FLAG := FALSE
552         ; CONTINUE_FLAG := TRUE
553         ; DOUBLE_CONSONANT := 0
554         ; HYPHEN := 0
555         ; LENG := 0
556         ; HIGH_RANGE_INDEX := 0
557         ; LOW_RANGE_INDEX := 0
558         ; VOWEL_COUNT := 0
559         END
560
561     ; PROCEDURE REJECT
562
563       ; BEGIN
564           K := 1
565         ; WHILE ( ( WORD_TABLE [ K ] <> ' ' )
566                   AND ( WORD_TABLE [ K ] <> NL )
567                   AND ( WORD_TABLE [ K ] <> CR )
568                   AND ( WORD_TABLE [ K ] <> EM )
569                   AND ( K <> TABLE_LENGTH )
570                 )
571           DO K := K + 1
572         ; TOTAL_LENG := K
573         ; WORD_TABLE [ K ] := ' '
574         ; WHILE ( NOT ( WORD_TABLE [ I ] IN ENDCHAR )
575                   AND ( I <= TABLE_LENGTH )
576                 )
577           DO BEGIN
578             IF ( ( ( WORD_TABLE [ I ] < 'a' )
579                    OR ( WORD_TABLE [ I ] > 'z' )
580                  )
581                  AND ( WORD_TABLE [ I ] <> '-' )
582                )
583             THEN BEGIN
584               HYPHEN_FLAG := FALSE
585             ; CONTINUE_FLAG := FALSE
586             END
587           ; IF WORD_TABLE [ I ] = '-'
588             THEN HYPHEN := I
589           ; IF WORD_TABLE [ I ] IN VOWEL_SET
```

```
590             THEN BEGIN
591               VOWEL_COUNT := VOWEL_COUNT + 1
592             ; IF NOT FIRSTFLAG
593               THEN BEGIN
594                 LOW_RANGE_INDEX := I + 1
595               ; FIRSTFLAG := TRUE
596               END
597               ELSE HIGH_RANGE_INDEX := I
598             END
599           ; LENG := LENG + 1
600           ; I := I + 1
601           END
602         END
603
604     ; PROCEDURE CK_HYPHEN
605
606       ; BEGIN
607           IF HYPHEN <> 0
608           THEN BEGIN
609             CONTINUE_FLAG := FALSE
610           ; IF HYPHEN <= AVAIL_LENG
611             THEN BEGIN
612             ; AVAIL_LENG := HYPHEN
613             ; J := 1
614             ; FOR I := ( HYPHEN + 1 ) TO TOTAL_LENG
615               DO BEGIN
616                 WORD_TABLE_REMAINING [ J ] := WORD_TABLE [ I ]
617               ; J := J + 1
618               END
619             ; HYPHEN_FLAG := TRUE
620             ; REMAINING_LENG := TOTAL_LENG - HYPHEN
621             END
622           END
623         END
624
625     ; PROCEDURE SPECIAL_REJECT
626
627       ; BEGIN
628           IF AVAIL_LENG < HIGH_RANGE_INDEX
629           THEN HIGH_RANGE_INDEX := AVAIL_LENG
630         ; IF ( ( VOWEL_COUNT <= 1 )
631                OR ( HIGH_RANGE_INDEX <= LOW_RANGE_INDEX )
632                OR ( LENG <= 3 )
633              )
634           THEN CONTINUE_FLAG := FALSE
635         ; IF ( ( LENG <= 5 ) AND CONTINUE_FLAG )
```

```
636           THEN BEGIN
637             IF ( ( LENG = 5 )
638                  AND ( ( WORD_TABLE [ 5 ] = 's' )
639                        OR ( WORD_TABLE [ 5 ] = 'd' )
640                      )
641                )
642             THEN CONTINUE_FLAG := FALSE
643           ; IF ( ( LENG = 4 ) AND ( WORD_TABLE [ 4 ] <> 'y' ) )
644             THEN CONTINUE_FLAG := FALSE
645           END
646         END
647
648       ; PROCEDURE CK_DICTIONARY
649
650         ; VAR K : INTEGER
651
652         ; BEGIN
653             AFFIX_NUM := 0
654           ; MATCHSTRING
655               ( SVC1_IN , EXCEPT , WORD_TABLE , LENG , AFFIX_NUM )
656           ; IF MATCH_FOUND
657             THEN BEGIN
658               K := 0
659             ; CASE EXCEPT [ TABLE_FILLED_TO , 26 ]
660               OF
661                 '1' : K := K + 10
662               ; '2' : K := K + 20
663               ; ELSE : K := 0
664               END
665             ; CASE EXCEPT [ TABLE_FILLED_TO , 27 ]
666               OF
667                 '0' , '1' , '2' , '3' , '4' , '5' , '6' , '7' , '8'
668                 , '9'
669                 : K
670                   := K
671                      + ( ORD ( EXCEPT [ TABLE_FILLED_TO , 27 ] ) - 48 )
672               ; ELSE : K := 0
673               END
674             ; BREAKWORD ( K )
675             END
676           ; SVC1_IN . SVC1_FUNC := 192
677           ; SVC1 ( SVC1_IN )
678           END
679
680       ; PROCEDURE CK_DOUBLE_CONSONANT
681
```

```
682       ; BEGIN
683           FOR I := 2 TO LENG
684           DO BEGIN
685             IF ( ( NOT ( WORD_TABLE [ I ] IN VOWEL_SET )
686                    AND ( ( WORD_TABLE [ I - 1 ] = WORD_TABLE [ I ] )
687                          AND ( WORD_TABLE [ I ] <> 'l' )
688                          AND ( WORD_TABLE [ I ] <> 's' )
689                        )
690                  )
691                )
692             THEN DOUBLE_CONSONANT := I
693           END
694         ; IF ( ( DOUBLE_CONSONANT <> 0 )
695                AND ( DOUBLE_CONSONANT <= HIGH_RANGE_INDEX )
696                AND ( DOUBLE_CONSONANT >= LOW_RANGE_INDEX )
697              )
698           THEN BREAKWORD ( DOUBLE_CONSONANT )
699         END
700
701     ; PROCEDURE RECURSIVE_S_CK
702
703       ; VAR STORE_WORD_TABLE : SCANTABLE
704         ; STORE_LENG , STORE_TOTAL_LENG : INTEGER
705
706       ; BEGIN
707           IF ( ( WORD_TABLE [ LENG ] = 's' )
708                AND ( WORD_TABLE [ LENG - 1 ] <> 's' )
709              )
710           THEN BEGIN
711           ; STORE_LENG := LENG + 1
712           ; STORE_TOTAL_LENG := TOTAL_LENG
713           ; FOR I := 1 TO STORE_TOTAL_LENG
714             DO STORE_WORD_TABLE [ I ] := WORD_TABLE [ I ]
715           ; WORD_TABLE [ LENG ] := ' '
716           ; HYPH
717               ( WORD_TABLE , AVAIL_LENG , HYPHEN_FLAG
718               , WORD_TABLE_REMAINING , REMAINING_LENG , CONTINUE_FLAG
719               )
720           ; IF HYPHEN_FLAG
721             THEN BEGIN
722               WORD_TABLE_REMAINING [ REMAINING_LENG ] := 's'
723             ; REMAINING_LENG := REMAINING_LENG + 1
724             ; FOR I := STORE_LENG TO STORE_TOTAL_LENG
725               DO BEGIN
726                 WORD_TABLE_REMAINING [ REMAINING_LENG ]
727                 := STORE_WORD_TABLE [ I ]
```

```
728               ; REMAINING_LENG := REMAINING_LENG + 1
729               END
730             ; REMAINING_LENG := REMAINING_LENG - 1
731             ; CONTINUE_FLAG := FALSE
732             END
733             ELSE WORD_TABLE [ LENG ] := 's'
734           END
735         END
736
737     ; PROCEDURE CK_SUFFIX
738
739       ; BEGIN
740           IF ( LENG - ( SUFFIX_LONGEST - 1 ) ) < LOW_RANGE_INDEX
741           THEN K := LOW_RANGE_INDEX
742           ELSE K := LENG - ( SUFFIX_LONGEST - 1 )
743         ; AFFIX_NUM := 1
744         ; WHILE ( ( K <= HIGH_RANGE_INDEX ) AND CONTINUE_FLAG )
745           DO BEGIN
746             J := 1
747           ; M := LENG
748           ; FOR I := K TO LENG
749             DO BEGIN
750               SHORT_TABLE [ J ] := WORD_TABLE [ M ]
751             ; J := J + 1
752             ; M := M - 1
753             END
754           ; LENG_STRING := J
755           ; SHORT_TABLE [ J ] := ' '
756           ; MATCHSTRING
757               ( SVC1_SUFF , AFFIX , SHORT_TABLE , LENG_STRING
758               , AFFIX_NUM )
759           ; IF MATCH_FOUND
760             THEN BREAKWORD ( K )
761           ; K := K + 1
762           ; IF AFFIX_NUM <> 1
763             THEN BEGIN
764             ; SVC1_SUFF . SVC1_FUNC := 192
765             ; SVC1 ( SVC1_SUFF )
766             ; SVC1_SUFF . SVC1_FUNC := 72
767             END
768           END
769         ; SVC1_SUFF . SVC1_FUNC := 192
770         ; SVC1 ( SVC1_SUFF )
771         END
772
773     ; PROCEDURE CK_PREFIX
```

```
774
775       ; BEGIN
776           IF ( HIGH_RANGE_INDEX - 1 ) < PREFIX_LONGEST
777           THEN I := ( HIGH_RANGE_INDEX - 1 )
778           ELSE I := PREFIX_LONGEST
779         ; AFFIX_NUM := 2
780         ; WHILE ( ( I >= ( LOW_RANGE_INDEX - 1 ) ) AND CONTINUE_FLAG )
781           DO BEGIN
782             J := 1
783           ; FOR K := 1 TO I
784             DO BEGIN
785               SHORT_TABLE [ J ] := WORD_TABLE [ K ]
786             ; J := J + 1
787             END
788           ; LENG_STRING := J
789           ; SHORT_TABLE [ J ] := ' '
790           ; MATCHSTRING
791               ( SVC1_PRE , AFFIX , SHORT_TABLE , LENG_STRING
792               , AFFIX_NUM )
793           ; IF MATCH_FOUND
794             THEN BREAKWORD ( LENG_STRING )
795           ; I := I - 1
796           ; IF AFFIX_NUM <> 2
797             THEN BEGIN
798             ; SVC1_PRE . SVC1_FUNC := 192
799             ; SVC1 ( SVC1_PRE )
800             ; SVC1_PRE . SVC1_FUNC := 72
801             END
802           END
803         ; SVC1_PRE . SVC1_FUNC := 192
804         ; SVC1 ( SVC1_PRE )
805         END
806
807     ; PROCEDURE CK_BREAK_VALUE_TABLE
808
809       ; BEGIN
810           I := LOW_RANGE_INDEX - 1
811         ; J := LOW_RANGE_INDEX
812         ; WHILE ( ( J <= HIGH_RANGE_INDEX ) AND CONTINUE_FLAG )
813           DO BEGIN
814             L := ORD ( WORD_TABLE [ I ] ) - 96
815           ; K := ORD ( WORD_TABLE [ J ] ) - 96
816           ; IF BREAK_VALUE_TABLE [ L , K ]
817             THEN BREAKWORD ( J )
818           ; I := I + 1
819           ; J := I + 1
```

```
820           END
821         END
822
823     ; BEGIN
824         INITIALIZE
825       ; REJECT
826       ; CK_HYPHEN
827       ; IF CONTINUE_FLAG
828         THEN SPECIAL_REJECT
829       ; IF CONTINUE_FLAG
830         THEN CK_DICTIONARY
831       ; IF CONTINUE_FLAG
832         THEN CK_DOUBLE_CONSONANT
833       ; IF CONTINUE_FLAG
834         THEN RECURSIVE_S_CK
835       ; IF CONTINUE_FLAG
836         THEN CK_SUFFIX
837       ; IF CONTINUE_FLAG
838         THEN CK_PREFIX
839       ; IF CONTINUE_FLAG
840         THEN CK_BREAK_VALUE_TABLE
841 (*  ED  AND  ES CHECK  *)
842       ; IF ( ( CONTINUE_FLAG )
843              AND ( ( WORD_TABLE [ LENG - 1 ] = 'e' )
844                    AND ( ( WORD_TABLE [ LENG ] = 'd' )
845                          OR ( WORD_TABLE [ LENG ] = 's' )
846                        )
847                  )
848            )
849         THEN BREAKWORD ( LENG - 1 )
850       END
851
852   ; BEGIN
853       INIT_BREAKVT
854     ; RIGHT := FALSE
855     ; LINE_START := TRUE
856     ; LINE_COUNT := 0
857     ; SPACE_COUNT := 1
858     ; BUFFER_COUNT := 1
859     ; WHILE Z <> EM
860       DO BEGIN
861         READ ( Z )
862       ; LINE_COUNT := LINE_COUNT + 1
863       ; IF Z <> EM
864         THEN BEGIN
865           IF ( Z = ' ' )
```

```
866           THEN BEGIN
867             IF LINE_START
868             THEN BEGIN
869               FOR I := 1 TO BUFFER_COUNT - 1
870               DO WRITE ( LINE_BUFFER [ I ] )
871             ; WRITE ( NL )
872             ; WRITE ( NL )
873             ; BUFFER_COUNT := 1
874             ; LINE_COUNT := 1
875             ; SPACE_COUNT := 1
876             ; LINE_START := FALSE
877             END
878             ELSE LINE_START := FALSE
879           END
880         ; IF ( ( Z = ' ' ) OR ( Z = CR ) OR ( Z = NL ) )
881           THEN BEGIN
882             LINE_BUFFER [ BUFFER_COUNT ] := Z
883           ; BUFFER_COUNT := BUFFER_COUNT + 1
884           END
885           ELSE BEGIN
886             WORD_COUNT := 1
887           ; CHAR_TABLE [ 1 ] := Z
888           ; REPEAT READ ( Z )
889             ; LINE_COUNT := LINE_COUNT + 1
890             ; IF Z <> EM
891               THEN BEGIN
892                 WORD_COUNT := WORD_COUNT + 1
893               ; CHAR_TABLE [ WORD_COUNT ] := Z
894               END
895             UNTIL ( ( Z = EM )
896                     OR ( Z = ' ' )
897                     OR ( Z = CR )
898                     OR ( Z = NL )
899                   )
900           ; SPACE_TABLE [ SPACE_COUNT ] := LINE_COUNT
901           ; SPACE_COUNT := SPACE_COUNT + 1
902           ; IF ( ( Z <> ' ' ) AND ( Z <> EM ) )
903             THEN BEGIN
904               LINE_START := TRUE
905             ; CHAR_TABLE [ WORD_COUNT ] := ' '
906             END
907             ELSE LINE_START := FALSE
908           ; IF ( LINE_COUNT <= LINE_LIMIT )
909             THEN FOR I := 1 TO WORD_COUNT
910               DO BEGIN
911                 LINE_BUFFER [ BUFFER_COUNT ] := CHAR_TABLE [ I ]
```

```
912               ; BUFFER_COUNT := BUFFER_COUNT + 1
913               END
914             ; IF ( LINE_COUNT = LINE_LIMIT + 1 )
915               THEN BEGIN
916               ; FOR I := 1 TO WORD_COUNT
917                 DO BEGIN
918                   LINE_BUFFER [ BUFFER_COUNT ] := CHAR_TABLE [ I ]
919                 ; BUFFER_COUNT := BUFFER_COUNT + 1
920                 END
921               ; FOR I := 1 TO LINE_LIMIT DO WRITE ( LINE_BUFFER [ I ] )
922               ; WRITE ( NL )
923               ; WRITE ( NL )
924               ; LINE_COUNT := 0
925               ; SPACE_COUNT := 1
926               ; BUFFER_COUNT := 1
927               END
928             ; IF ( LINE_COUNT > LINE_LIMIT + 1 )
929               THEN BEGIN
930               ; HYPHEN_FLAG := FALSE
931               ; STORE_COUNT := WORD_COUNT
932               ; WORD_COUNT
933                 := LINE_LIMIT - SPACE_TABLE [ SPACE_COUNT - 2 ]
934               ; IF WORD_COUNT > 4
935                 THEN HYPH
936                     ( CHAR_TABLE , WORD_COUNT , HYPHEN_FLAG
937                     , CHAR_TABLE_REMAINING , WORD_COUNT_REMAINING
938                     , STOPPED_FLAG )
939               ; IF STOPPED_FLAG AND NOT HYPHEN_FLAG
940                 THEN BEGIN
941                   FOR K := 1 TO TABLE_LENGTH
942                   DO DISPLAY ( CHAR_TABLE [ K ] )
943                 ; DISPLAY ( CR )
944                 END
945               ; IF HYPHEN_FLAG
946                 THEN BEGIN
947                   AMT_EACH_SPACE
948                   := ( LINE_LIMIT
949                        - SPACE_TABLE [ SPACE_COUNT - 2 ]
950                        - WORD_COUNT
951                      )
952                      DIV ( SPACE_COUNT - 2 )
953                 ; LEFT_OVER_SPACES
954                   := ( LINE_LIMIT
955                        - SPACE_TABLE [ SPACE_COUNT - 2 ]
956                        - WORD_COUNT
957                      )
```

```
958                 MOD ( SPACE_COUNT - 2 )
959               END
960               ELSE BEGIN
961                 AMT_EACH_SPACE
962                 := ( ( LINE_LIMIT + 1 )
963                      - SPACE_TABLE [ SPACE_COUNT - 2 ]
964                    )
965                    DIV ( SPACE_COUNT - 2 )
966               ; LEFT_OVER_SPACES
967                 := ( ( LINE_LIMIT + 1 )
968                      - SPACE_TABLE [ SPACE_COUNT - 2 ]
969                    )
970                    MOD ( SPACE_COUNT - 2 )
971               END
972             ; NEXTSPACE := 1
973             ; J := 1
974             ; IF RIGHT
975               THEN WAIT
976                 := ( ( ( SPACE_COUNT - 2 ) - LEFT_OVER_SPACES ) - 1 )
977             ; WHILE ( NEXTSPACE <= ( SPACE_COUNT - 2 ) )
978               DO BEGIN
979                 FOR I := J TO SPACE_TABLE [ NEXTSPACE ]
980                 DO WRITE ( LINE_BUFFER [ I ] )
981               ; IF AMT_EACH_SPACE <> 0
982                 THEN FOR I := 1 TO AMT_EACH_SPACE DO WRITE ( ' ' )
983               ; IF ( ( LEFT_OVER_SPACES <> 0 ) AND ( NOT RIGHT ) )
984                 THEN BEGIN
985                   WRITE ( ' ' )
986                 ; LEFT_OVER_SPACES := LEFT_OVER_SPACES - 1
987                 END
988               ; IF ( ( LEFT_OVER_SPACES <> 0 ) AND RIGHT )
989                 THEN IF ( WAIT <> 0 )
990                   THEN WAIT := WAIT - 1
991                   ELSE BEGIN
992                     WRITE ( ' ' )
993                   ; LEFT_OVER_SPACES := LEFT_OVER_SPACES - 1
994                   END
995               ; J := SPACE_TABLE [ NEXTSPACE ] + 1
996               ; NEXTSPACE := NEXTSPACE + 1
997               END
998             ; IF HYPHEN_FLAG
999               THEN BEGIN
1000                FOR I := 1 TO WORD_COUNT_REMAINING
1001                DO LINE_BUFFER [ I ] := CHAR_TABLE_REMAINING [ I ]
1002              ; BUFFER_COUNT := I + 1
1003              ; LINE_COUNT := WORD_COUNT_REMAINING
```

```
1004              ; SPACE_COUNT := 1
1005              ; SPACE_TABLE [ SPACE_COUNT ] := LINE_COUNT
1006              ; SPACE_COUNT := SPACE_COUNT + 1
1007              ; FOR I := 1 TO WORD_COUNT DO WRITE ( CHAR_TABLE [ I ] )
1008              END
1009              ELSE BEGIN
1010                FOR I := 1 TO STORE_COUNT
1011                DO LINE_BUFFER [ I ] := CHAR_TABLE [ I ]
1012              ; BUFFER_COUNT := I + 1
1013              ; LINE_COUNT := STORE_COUNT
1014              ; SPACE_COUNT := 1
1015              ; SPACE_TABLE [ SPACE_COUNT ] := LINE_COUNT
1016              ; SPACE_COUNT := SPACE_COUNT + 1
1017              END
1018            ; IF RIGHT
1019              THEN RIGHT := FALSE
1020              ELSE RIGHT := TRUE
1021            ; WRITE ( NL )
1022            ; WRITE ( NL )
1023            END
1024          END
1025        END
1026      END
1027    ; FOR I := 1 TO BUFFER_COUNT - 1 DO WRITE ( LINE_BUFFER [ I ] )
1028    ; WRITE ( NL )
1029    END
1030 .
1031
```

The commands used to execute this program are:

```
LOAD USR6:PROJECT.TSK

ASSIGN 0, DIS.TXT        "ANOMALY FILE"

ASSIGN 1, (YOUR INPUT FILE)

ASSIGN 2, (YOUR OUTPUT FILE)

ASSIGN 3, EXCEPT.TXT     "EXCEPTION DICTIONARY"

ASSIGN 4, SUF.TXT        "REVERSED SUFFIX FILE"

ASSIGN 5, PRE.TXT        "PREFIX FILE"

START
```

APPENDIX G - DICTIONARYBUILDER PROGRAM

This program uses the hyphenator to help a person build the exception dictionary by showing how the hyphenator processes each word through all possible lengths. The section codes are located on page 30. They tell which section hyphenated the word.

```
    1
    2 "ROBERT YOUNG"
    3 "KANSAS STATE UNIVERSITY"
    4   CONST COPYRIGHT = 'COPYRIGHT ROBERT YOUNG 1978'
    5
    6 "############
C   7 #  PREFIX  #
C   8 ############"
    9
   10 ; CONST NL = '(:10:)'
   11   ; FF = '(:12:)'
   12   ; CR = '(:13:)'
   13   ; EM = '(:25:)'
   14
   15 ; CONST PAGELENGTH = 512
   16
   17 ; TYPE PAGE = ARRAY (. 1 .. PAGELENGTH .) OF CHAR
   18
   19 ; CONST LINELENGTH = 132
   20
   21 ; TYPE LINE = ARRAY (. 1 .. LINELENGTH .) OF CHAR
   22
   23 ; CONST IDLENGTH = 12
   24
   25 ; TYPE IDENTIFIER = ARRAY (. 1 .. IDLENGTH .) OF CHAR
   26
   27 ; TYPE FILE = 1 .. 2
   28
   29 ; TYPE FILEKIND = ( EMPTY , SCRATCH , ASCII , SEQCODE , CONCODE )
   30
   31 ; TYPE FILEATTR
   32     = RECORD
   33         KIND : FILEKIND
   34       ; ADDR : INTEGER
   35       ; PROTECTED : BOOLEAN
```

```
36        ; NOTUSED : ARRAY (. 1 .. 5 .) OF INTEGER
37        END
38
39  ; TYPE IODEVICE
40      = ( TYPEDEVICE , DISKDEVICE , TAPEDEVICE , PRINTDEVICE
41        , CARDDEVICE )
42
43  ; TYPE IOOPERATION = ( INPUT , OUTPUT , MOVE , CONTROL )
44
45  ; TYPE IOARG = ( WRITEEOF , REWIND , UPSPACE , BACKSPACE )
46
47  ; TYPE IORESULT
48      = ( COMPLETE , INTERVENTION , TRANSMISSION , FAILURE , ENDFILE
49        , ENDMEDIUM , STARTMEDIUM )
50
51  ; TYPE IOPARAM
52      = RECORD
53          OPERATION : IOOPERATION
54        ; STATUS : IORESULT
55        ; ARG : IOARG
56        END
57
58  ; TYPE TASKKIND = ( INPUTTASK , JOBTASK , OUTPUTTASK )
59
60  ; TYPE ARGTAG = ( NILTYPE , BOOLTYPE , INTTYPE , IDTYPE , PTRTYPE )
61
62  ; TYPE POINTER = @ BOOLEAN
63
64  ; TYPE PASSPTR = @ PASSLINK
65
66  ; TYPE PASSLINK
67      = RECORD
68          OPTIONS : SET OF CHAR
69        ; FILLER1 : ARRAY (. 1 .. 7 .) OF INTEGER
70        ; FILLER2 : BOOLEAN
71        ; RESET_POINT : INTEGER
72        ; FILLER3 : ARRAY (. 1 .. 11 .) OF POINTER
73        END
74
75  ; TYPE ARGTYPE
76      = RECORD
77          CASE TAG : ARGTAG
78          OF NILTYPE , BOOLTYPE : ( BOOL : BOOLEAN )
79          ; INTTYPE : ( INT : INTEGER )
80          ; IDTYPE : ( ID : IDENTIFIER )
81          ; PTRTYPE : ( PTR : PASSPTR )
```

```
 82       END
 83
 84 ; CONST MAXARG = 10
 85
 86 ; TYPE ARGLIST = ARRAY (. 1 .. MAXARG .) OF ARGTYPE
 87
 88 ; TYPE ARGSEQ = ( INP , OUT )
 89
 90 ; TYPE PROGRESULT
 91     = ( TERMINATED , OVERFLOW , POINTERERROR , RANGEERROR
 92       , VARIANTERROR , HEAPLIMIT , STACKLIMIT , CODELIMIT , TIMELIMIT
 93       , CALLERROR )
 94
 95 ; PROCEDURE READ ( VAR C : CHAR )
 96
 97 ; PROCEDURE WRITE ( C : CHAR )
 98
 99 ; PROCEDURE OPEN ( F : FILE ; ID : IDENTIFIER ; VAR FOUND : BOOLEAN )
100
101 ; PROCEDURE CLOSE ( F : FILE )
102
103 ; PROCEDURE GET ( F : FILE ; P : INTEGER ; VAR BLOCK : UNIV PAGE )
104
105 ; PROCEDURE PUT ( F : FILE ; P : INTEGER ; VAR BLOCK : UNIV PAGE )
106
107 ; FUNCTION LENGTH ( F : FILE ) : INTEGER
108
109 ; PROCEDURE MARK ( VAR TOP : INTEGER )
110
111 ; PROCEDURE RELEASE ( TOP : INTEGER )
112
113 ; PROCEDURE IDENTIFY ( HEADER : LINE )
114
115 ; PROCEDURE ACCEPT ( VAR C : CHAR )
116
117 ; PROCEDURE DISPLAY ( C : CHAR )
118
119 ; PROCEDURE NOTUSED
120
121 ; PROCEDURE NOTUSED2
122
123 ; PROCEDURE NOTUSED3
124
125 ; PROCEDURE NOTUSED4
126
127 ; PROCEDURE NOTUSED5
```

```
128
129 ; PROCEDURE NOTUSED6
130
131 ; PROCEDURE NOTUSED7
132
133 ; PROCEDURE NOTUSED8
134
135 ; PROCEDURE NOTUUED9
136
137 ; PROCEDURE NOTUSED10
138
139 ; PROCEDURE RUN
140     ( ID : IDENTIFIER
141     ; VAR PARAM : ARGLIST
142     ; VAR LINE : INTEGER
143     ; VAR RESULT : PROGRESULT
144     )
145
146 ; PROGRAM HYPHNO1 ( PARAM : LINE )
147
148   ; CONST TABLE_LENGTH = 30
149
150   ; TYPE SCANTABLE = ARRAY [ 1 .. TABLE_LENGTH ] OF CHAR
151     ; SVC1_BLOCK
152       = RECORD
153           SVC1_FUNC : BYTE
154         ; SVC1_LU : BYTE
155         ; SVC1_STAT : BYTE
156         ; SVC1_DEV_STAT : BYTE
157         ; SVC1_BUFSTART : INTEGER
158         ; SVC1_BUFEND : INTEGER
159         ; SVC1_RANDOM_ADDR : INTEGER
160         ; SVC1_XFER_LEN : INTEGER
161         ; SVC1_RESERVED : INTEGER
162         ;
163         END
164
165   ; VAR I , J , K : INTEGER
166     ; HYPHFLAG , STOPPED_FLAG : BOOLEAN
167     ; WORDCOUNT : INTEGER
168     ; WORDCOUNTR : INTEGER
169     ; X : CHAR
170     ; U : CHAR
171     ; CHARTABLE , STORETABLE : SCANTABLE
172     ; CHARTABLER : SCANTABLE
173     ; STORECOUNT : INTEGER
```

```
174     ; BREAK_VALUE_TABLE : ARRAY [ 1 .. 26 , 1 .. 26 ] OF BOOLEAN
175
176   ; PROCEDURE SVC1 ( SVC1_IN : SVC1_BLOCK )
177     ; EXTERN
178
179   ; PROCEDURE WRITETABLE
180
181     ; BEGIN
182         IF HYPHFLAG
183         THEN BEGIN
184           FOR K := 1 TO WORDCOUNT DO WRITE ( CHARTABLE [ K ] )
185         ; WRITE ( ' ' )
186         ; FOR K := 1 TO WORDCOUNTR - 1 DO WRITE ( CHARTABLER [ K ] )
187         ; WRITE ( ' ' )
188         ; WRITE ( ' ' )
189         ; WRITE ( U )
190         END
191         ELSE BEGIN
192           FOR K := 1 TO WORDCOUNT + 1 DO WRITE ( CHARTABLE [ K ] )
193         ; WRITE ( U )
194         END
195       END
196
197   ; PROCEDURE WRITE_INT ( I : INTEGER )
198
199     ; VAR DIGIT , J , INT : INTEGER
200       ; INT_STRING : ARRAY [ 1 .. 11 ] OF CHAR
201
202     ; BEGIN
203         INT := ABS ( I )
204       ; FOR J := 1 TO 11 DO INT_STRING [ J ] := ' '
205       ; J := 11
206       ; REPEAT DIGIT := INT MOD 10
207         ; INT_STRING [ J ] := CHR ( DIGIT + 48 )
208         ; INT := INT DIV 10
209         ; J := J - 1
210         UNTIL INT = 0
211       ; IF I < 0
212         THEN INT_STRING [ J ] := '-'
213       ; FOR J := 1 TO 11 DO WRITE ( INT_STRING [ J ] )
214       END
215
216   ; PROCEDURE INIT_BREAKVT
217
218     ; BEGIN
219         FOR I := 1 TO 26
```

```
220         DO FOR J := 1 TO 26 DO BREAK_VALUE_TABLE [ I , J ] := TRUE
221       ; FOR I := 1 TO 14
222         DO BEGIN
223           BREAK_VALUE_TABLE [ 1 , I ] := FALSE
224         ; BREAK_VALUE_TABLE [ I , 1 ] := FALSE
225         ; BREAK_VALUE_TABLE [ 5 , I ] := FALSE
226         ; BREAK_VALUE_TABLE [ I , 5 ] := FALSE
227         ; BREAK_VALUE_TABLE [ 9 , I ] := FALSE
228         ; BREAK_VALUE_TABLE [ I , 9 ] := FALSE
229         ; BREAK_VALUE_TABLE [ 15 , I ] := FALSE
230         ; BREAK_VALUE_TABLE [ I , 15 ] := FALSE
231         ; BREAK_VALUE_TABLE [ 21 , I ] := FALSE
232         ; BREAK_VALUE_TABLE [ I , 21 ] := FALSE
233         END
234       ; FOR I := 15 TO 23
235         DO BEGIN
236           BREAK_VALUE_TABLE [ I , 1 ] := FALSE
237         ; BREAK_VALUE_TABLE [ I , 5 ] := FALSE
238         ; BREAK_VALUE_TABLE [ I , 9 ] := FALSE
239         ; BREAK_VALUE_TABLE [ I , 15 ] := FALSE
240         ; BREAK_VALUE_TABLE [ I , 21 ] := FALSE
241         END
242       ; FOR I := 18 TO 24
243         DO BEGIN
244           BREAK_VALUE_TABLE [ 1 , I ] := FALSE
245         ; BREAK_VALUE_TABLE [ 5 , I ] := FALSE
246         ; BREAK_VALUE_TABLE [ 9 , I ] := FALSE
247         ; BREAK_VALUE_TABLE [ 15 , I ] := FALSE
248         ; BREAK_VALUE_TABLE [ 21 , I ] := FALSE
249         END
250       ; BREAK_VALUE_TABLE [ 12 , 4 ] := FALSE
251       ; BREAK_VALUE_TABLE [ 14 , 3 ] := FALSE
252       ; BREAK_VALUE_TABLE [ 14 , 4 ] := FALSE
253       ; BREAK_VALUE_TABLE [ 18 , 3 ] := FALSE
254       ; BREAK_VALUE_TABLE [ 18 , 4 ] := FALSE
255       ; BREAK_VALUE_TABLE [ 19 , 3 ] := FALSE
256       ; BREAK_VALUE_TABLE [ 20 , 3 ] := FALSE
257       ; BREAK_VALUE_TABLE [ 25 , 1 ] := FALSE
258       ; BREAK_VALUE_TABLE [ 26 , 1 ] := FALSE
259       ; BREAK_VALUE_TABLE [ 3 , 8 ] := FALSE
260       ; BREAK_VALUE_TABLE [ 7 , 8 ] := FALSE
261       ; BREAK_VALUE_TABLE [ 14 , 7 ] := FALSE
262       ; BREAK_VALUE_TABLE [ 16 , 8 ] := FALSE
263       ; BREAK_VALUE_TABLE [ 18 , 7 ] := FALSE
264       ; BREAK_VALUE_TABLE [ 19 , 8 ] := FALSE
265       ; BREAK_VALUE_TABLE [ 20 , 8 ] := FALSE
```

```
266       ; BREAK_VALUE_TABLE [ 2 , 18 ] := FALSE
267       ; BREAK_VALUE_TABLE [ 3 , 18 ] := FALSE
268       ; BREAK_VALUE_TABLE [ 4 , 18 ] := FALSE
269       ; BREAK_VALUE_TABLE [ 6 , 18 ] := FALSE
270       ; BREAK_VALUE_TABLE [ 7 , 18 ] := FALSE
271       ; BREAK_VALUE_TABLE [ 8 , 18 ] := FALSE
272       ; BREAK_VALUE_TABLE [ 16 , 18 ] := FALSE
273       ; BREAK_VALUE_TABLE [ 17 , 18 ] := FALSE
274       ; BREAK_VALUE_TABLE [ 23 , 8 ] := FALSE
275       ; BREAK_VALUE_TABLE [ 26 , 5 ] := FALSE
276       ; BREAK_VALUE_TABLE [ 25 , 9 ] := FALSE
277       ; BREAK_VALUE_TABLE [ 26 , 9 ] := FALSE
278       ; BREAK_VALUE_TABLE [ 2 , 12 ] := FALSE
279       ; BREAK_VALUE_TABLE [ 3 , 11 ] := FALSE
280       ; BREAK_VALUE_TABLE [ 3 , 12 ] := FALSE
281       ; BREAK_VALUE_TABLE [ 4 , 12 ] := FALSE
282       ; BREAK_VALUE_TABLE [ 6 , 12 ] := FALSE
283       ; BREAK_VALUE_TABLE [ 7 , 12 ] := FALSE
284       ; BREAK_VALUE_TABLE [ 7 , 14 ] := FALSE
285       ; BREAK_VALUE_TABLE [ 11 , 14 ] := FALSE
286       ; BREAK_VALUE_TABLE [ 12 , 12 ] := FALSE
287       ; BREAK_VALUE_TABLE [ 12 , 11 ] := FALSE
288       ; BREAK_VALUE_TABLE [ 14 , 11 ] := FALSE
289       ; BREAK_VALUE_TABLE [ 16 , 12 ] := FALSE
290       ; BREAK_VALUE_TABLE [ 18 , 13 ] := FALSE
291       ; BREAK_VALUE_TABLE [ 18 , 14 ] := FALSE
292       ; BREAK_VALUE_TABLE [ 20 , 18 ] := FALSE
293       ; BREAK_VALUE_TABLE [ 23 , 18 ] := FALSE
294       ; BREAK_VALUE_TABLE [ 25 , 18 ] := FALSE
295       ; BREAK_VALUE_TABLE [ 11 , 19 ] := FALSE
296       ; BREAK_VALUE_TABLE [ 12 , 19 ] := FALSE
297       ; BREAK_VALUE_TABLE [ 14 , 19 ] := FALSE
298       ; BREAK_VALUE_TABLE [ 18 , 19 ] := FALSE
299       ; BREAK_VALUE_TABLE [ 19 , 19 ] := FALSE
300       ; BREAK_VALUE_TABLE [ 23 , 19 ] := FALSE
301       ; BREAK_VALUE_TABLE [ 25 , 19 ] := FALSE
302       ; BREAK_VALUE_TABLE [ 19 , 11 ] := FALSE
303       ; BREAK_VALUE_TABLE [ 19 , 14 ] := FALSE
304       ; BREAK_VALUE_TABLE [ 20 , 12 ] := FALSE
305       ; BREAK_VALUE_TABLE [ 25 , 12 ] := FALSE
306       ; BREAK_VALUE_TABLE [ 25 , 13 ] := FALSE
307       ; BREAK_VALUE_TABLE [ 25 , 14 ] := FALSE
308       ; BREAK_VALUE_TABLE [ 25 , 15 ] := FALSE
309       ; BREAK_VALUE_TABLE [ 1 , 16 ] := FALSE
310       ; BREAK_VALUE_TABLE [ 26 , 15 ] := FALSE
311       ; BREAK_VALUE_TABLE [ 5 , 16 ] := FALSE
```

```
312       ; BREAK_VALUE_TABLE [ 9 , 16 ] := FALSE
313       ; BREAK_VALUE_TABLE [ 13 , 16 ] := FALSE
314       ; BREAK_VALUE_TABLE [ 15 , 16 ] := FALSE
315       ; BREAK_VALUE_TABLE [ 19 , 16 ] := FALSE
316       ; BREAK_VALUE_TABLE [ 19 , 17 ] := FALSE
317       ; BREAK_VALUE_TABLE [ 21 , 16 ] := FALSE
318       ; BREAK_VALUE_TABLE [ 24 , 16 ] := FALSE
319       ; BREAK_VALUE_TABLE [ 20 , 23 ] := FALSE
320       ; BREAK_VALUE_TABLE [ 21 , 26 ] := FALSE
321       ; BREAK_VALUE_TABLE [ 23 , 25 ] := FALSE
322       ; BREAK_VALUE_TABLE [ 24 , 25 ] := FALSE
323       ; BREAK_VALUE_TABLE [ 25 , 25 ] := FALSE
324       ; BREAK_VALUE_TABLE [ 3 , 20 ] := FALSE
325       ; BREAK_VALUE_TABLE [ 6 , 20 ] := FALSE
326       ; BREAK_VALUE_TABLE [ 8 , 20 ] := FALSE
327       ; BREAK_VALUE_TABLE [ 14 , 20 ] := FALSE
328       ; BREAK_VALUE_TABLE [ 16 , 20 ] := FALSE
329       ; BREAK_VALUE_TABLE [ 18 , 20 ] := FALSE
330       ; BREAK_VALUE_TABLE [ 19 , 20 ] := FALSE
331       ; BREAK_VALUE_TABLE [ 25 , 20 ] := FALSE
332       ; BREAK_VALUE_TABLE [ 25 , 21 ] := FALSE
333       ; BREAK_VALUE_TABLE [ 26 , 21 ] := FALSE
334       ; BREAK_VALUE_TABLE [ 1 , 25 ] := FALSE
335       ; BREAK_VALUE_TABLE [ 1 , 26 ] := FALSE
336       ; BREAK_VALUE_TABLE [ 3 , 26 ] := FALSE
337       ; BREAK_VALUE_TABLE [ 5 , 26 ] := FALSE
338       ; BREAK_VALUE_TABLE [ 6 , 25 ] := FALSE
339       ; BREAK_VALUE_TABLE [ 9 , 26 ] := FALSE
340       ; BREAK_VALUE_TABLE [ 12 , 22 ] := FALSE
341       ; BREAK_VALUE_TABLE [ 12 , 25 ] := FALSE
342       ; BREAK_VALUE_TABLE [ 15 , 25 ] := FALSE
343       ; BREAK_VALUE_TABLE [ 15 , 26 ] := FALSE
344       ; BREAK_VALUE_TABLE [ 17 , 25 ] := FALSE
345       ; BREAK_VALUE_TABLE [ 18 , 22 ] := FALSE
346       ; BREAK_VALUE_TABLE [ 18 , 25 ] := FALSE
347       ; BREAK_VALUE_TABLE [ 18 , 26 ] := FALSE
348       ; BREAK_VALUE_TABLE [ 19 , 23 ] := FALSE
349       ; BREAK_VALUE_TABLE [ 1 , 15 ] := TRUE
350       END
351
352   ; PROCEDURE HYPH
353       ( VAR WORD_TABLE : SCANTABLE
354       ; VAR AVAIL_LENG : INTEGER
355       ; VAR HYPHEN_FLAG : BOOLEAN
356       ; VAR WORD_TABLE_REMAINING : SCANTABLE
357       ; VAR REMAINING_LENG : INTEGER
```

```
358       ; VAR CONTINUE_FLAG : BOOLEAN
359       )
360     ; FORWARD
361
362   ; PROCEDURE HYPH
363
364     ; CONST SUFFIX_LONGEST = 5
365       ; PREFIX_LONGEST = 5
366
367     ; TYPE ARRAYCHAR = PACKED ARRAY [ 1 .. 18 , 1 .. 28 ] OF CHAR
368
369     ; VAR I , J , K , L , M , HYPHEN , LENG , HIGH_RANGE_INDEX
370           , LOW_RANGE_INDEX , VOWEL_COUNT , DOUBLE_CONSONANT
371           , LENG_STRING , TABLE_FILLED_TO , TOTAL_LENG , AFFIX_NUM
372           : INTEGER
373       ; FIRSTFLAG , SEARCH_FLAG , MATCH_FOUND : BOOLEAN
374       ; SVC1_IN , SVC1_SUFF , SVC1_PRE : SVC1_BLOCK
375       ; EXCEPT , AFFIX : ARRAYCHAR
376       ; FIRSTARRAY : ARRAY [ 1 .. 2 ] OF BOOLEAN
377       ; VOWEL_SET , ENDCHAR : SET OF CHAR
378       ; SHORT_TABLE : PACKED ARRAY [ 1 .. 25 ] OF CHAR
379
380     ; PROCEDURE BREAKWORD ( BREAK_INDEX : INTEGER )
381
382       ; VAR M : INTEGER
383
384       ; BEGIN
385           IF ( ( BREAK_INDEX <> 0 ) AND ( BREAK_INDEX <= AVAIL_LENG )
386              )
387           THEN BEGIN
388             WORD_TABLE_REMAINING [ 1 ] := WORD_TABLE [ BREAK_INDEX ]
389           ; WORD_TABLE [ BREAK_INDEX ] := '-'
390           ; AVAIL_LENG := BREAK_INDEX
391           ; HYPHEN_FLAG := TRUE
392           ; CONTINUE_FLAG := FALSE
393           ; J := 2
394           ; FOR M := ( BREAK_INDEX + 1 ) TO TOTAL_LENG
395             DO BEGIN
396               WORD_TABLE_REMAINING [ J ] := WORD_TABLE [ M ]
397             ; WORD_TABLE [ M ] := ' '
398             ; J := J + 1
399             END
400           ; REMAINING_LENG := J - 1
401           END
402           ELSE CONTINUE_FLAG := FALSE
403         END
```

```
404
405     ; PROCEDURE SVC_LOAD
406         ( BLOCK : SVC1_BLOCK
407         ; ARRAY1 : ARRAYCHAR
408         ; VAR TABLE_FILLED_TO : INTEGER
409         )
410
411       ; VAR I : INTEGER
412
413       ; BEGIN
414           I := 1
415         ; IF ( BLOCK . SVC1_STAT <> 136 )
416           THEN BEGIN
417             SVC1 ( BLOCK )
418           ; IF ( BLOCK . SVC1_STAT = 136 )
419             THEN SEARCH_FLAG := FALSE
420           ; I := 2
421           ; REPEAT I := I + 1
422             UNTIL ( ( ARRAY1 [ I - 1 , 1 ] > ARRAY1 [ I , 1 ] )
423                     OR ( I = 18 )
424                     OR ( ARRAY1 [ I , 1 ] < 'a' )
425                     OR ( ARRAY1 [ I , 1 ] > 'z' )
426                   )
427           END
428         ; TABLE_FILLED_TO := I
429         END
430
431     ; PROCEDURE MATCHSTRING
432         ( BLOCK : SVC1_BLOCK
433         ; ARRAY1 : ARRAYCHAR
434         ; ARRAY2 : SCANTABLE
435         ; LENG_STRING : INTEGER
436         ; VAR AFFIX_NUM : INTEGER
437         )
438
439       ; VAR I , J , K , L : INTEGER
440         ; NOT_LAST_ENTRY , IN_TABLE_FLAG : BOOLEAN
441
442       ; PROCEDURE STRING_MATCHER
443
444         ; BEGIN
445             I := 1
446           ; L := K
447           ; WHILE ( ( IN_TABLE_FLAG ) AND ( L <= J ) )
448             DO BEGIN
449             ; WHILE ( ( IN_TABLE_FLAG )
```

```
450                     AND ( I <= LENG_STRING )
451                     AND ( ARRAY1 [ L , I ] = ARRAY2 [ I ] )
452                   )
453             DO I := I + 1
454           ; IF I = LENG_STRING + 1
455             THEN BEGIN
456               MATCH_FOUND := TRUE
457             ; CONTINUE_FLAG := FALSE
458             ; IN_TABLE_FLAG := FALSE
459             ; TABLE_FILLED_TO := L
460             END
461             ELSE BEGIN I := 1 ; L := L + 1 END
462           ; IF CONTINUE_FLAG
463             THEN SEARCH_FLAG := FALSE
464           END
465         END
466
467       ; PROCEDURE FIRST_LETTER_SEARCH
468
469         ; BEGIN
470           ; L := 1
471           ; IN_TABLE_FLAG := FALSE
472           ; I := 1
473           ; J := TABLE_FILLED_TO
474           ; REPEAT K := ( L + J ) DIV 2
475             ; IF ARRAY1 [ K , I ] = ARRAY2 [ I ]
476               THEN IN_TABLE_FLAG := TRUE
477               ELSE IF ARRAY1 [ K , I ] < ARRAY2 [ I ]
478                 THEN L := K + 1
479                 ELSE J := K - 1
480             UNTIL IN_TABLE_FLAG OR ( L > J )
481           ; IF NOT IN_TABLE_FLAG
482             THEN SEARCH_FLAG := FALSE
483             ELSE BEGIN
484               J := K
485             ; WHILE ( ( ARRAY1 [ K , I ] = ARRAY2 [ I ] )
486                       AND ( K > 1 )
487                     )
488               DO K := K - 1
489             ; IF K = 1
490               THEN AFFIX_NUM := 0
491             ; WHILE ( ( ARRAY1 [ J , I ] = ARRAY2 [ I ] )
492                       AND ( J <= 17 )
493                     )
494               DO J := J + 1
495             ; J := J - 1
```

```
496             ; STRING_MATCHER
497             END
498           END
499
500       ; PROCEDURE LOADER
501
502         ; BEGIN
503           ; NOT_LAST_ENTRY := FALSE
504           ; SEARCH_FLAG := TRUE
505           ; SVC_LOAD ( BLOCK , ARRAY1 , TABLE_FILLED_TO )
506           ; WHILE SEARCH_FLAG
507             DO BEGIN
508               IF ARRAY1 [ TABLE_FILLED_TO , 1 ] < ARRAY2 [ 1 ]
509               THEN SVC_LOAD ( BLOCK , ARRAY1 , TABLE_FILLED_TO )
510             ; IF ARRAY1 [ TABLE_FILLED_TO , 1 ] = ARRAY2 [ 1 ]
511               THEN BEGIN
512                 I := 2
513               ; WHILE ( ( I <= LENG_STRING )
514                         AND ( ARRAY1 [ TABLE_FILLED_TO , I ]
515                               = ARRAY2 [ I ]
516                             )
517                       )
518               DO I := I + 1
519             ; IF I = LENG_STRING + 1
520               THEN BEGIN
521                 CONTINUE_FLAG := FALSE
522               ; SEARCH_FLAG := FALSE
523               ; MATCH_FOUND := TRUE
524               END
525               ELSE IF ARRAY1 [ TABLE_FILLED_TO , I ] < ARRAY2 [ I ]
526                 THEN SVC_LOAD ( BLOCK , ARRAY1 , TABLE_FILLED_TO )
527                 ELSE NOT_LAST_ENTRY := TRUE
528               END
529             ; IF ( ( ( ARRAY1 [ TABLE_FILLED_TO , 1 ] > ARRAY2 [ 1 ] )
530                      OR ( NOT_LAST_ENTRY )
531                    )
532                    AND CONTINUE_FLAG
533                  )
534               THEN FIRST_LETTER_SEARCH
535             END
536           END
537
538       ; BEGIN
539           MATCH_FOUND := FALSE
540         ; IF AFFIX_NUM = 0
541           THEN LOADER
```

```
542           ELSE IF NOT FIRSTARRAY [ AFFIX_NUM ]
543             THEN BEGIN FIRSTARRAY [ AFFIX_NUM ] := TRUE ; LOADER END
544             ELSE STRING_MATCHER
545         END
546
547     ; PROCEDURE INITIALIZE
548
549       ; BEGIN
550           VOWEL_SET := [ 'a' , 'e' , 'i' , 'o' , 'u' , 'y' ]
551         ; ENDCHAR := [ ',' , '.' , ':' , ';' , ' ' , '!' , '?' , NL ]
552         ; WITH SVC1_IN
553           DO BEGIN
554             SVC1_FUNC := 73
555           ; SVC1_LU := 3
556           ; SVC1_STAT := 0
557           ; SVC1_DEV_STAT := 0
558           ; SVC1_BUFSTART := ADDRESS ( EXCEPT [ 1 , 1 ] )
559           ; SVC1_BUFEND := ADDRESS ( EXCEPT [ 1 , 1 ] ) + 503
560           END
561         ; WITH SVC1_SUFF
562           DO BEGIN
563             SVC1_FUNC := 73
564           ; SVC1_LU := 4
565           ; SVC1_STAT := 0
566           ; SVC1_DEV_STAT := 0
567           ; SVC1_BUFSTART := ADDRESS ( AFFIX [ 1 , 1 ] )
568           ; SVC1_BUFEND := ADDRESS ( AFFIX [ 1 , 1 ] ) + 503
569           END
570         ; WITH SVC1_PRE
571           DO BEGIN
572             SVC1_FUNC := 73
573           ; SVC1_LU := 5
574           ; SVC1_STAT := 0
575           ; SVC1_DEV_STAT := 0
576           ; SVC1_BUFSTART := ADDRESS ( AFFIX [ 1 , 1 ] )
577           ; SVC1_BUFEND := ADDRESS ( AFFIX [ 1 , 1 ] ) + 503
578           END
579         ; I := 1
580         ; FIRSTFLAG := FALSE
581         ; FIRSTARRAY [ 1 ] := FALSE
582         ; FIRSTARRAY [ 2 ] := FALSE
583         ; HYPHEN_FLAG := FALSE
584         ; CONTINUE_FLAG := TRUE
585         ; DOUBLE_CONSONANT := 0
586         ; HYPHEN := 0
587         ; LENG := 0
```

```
588         ; HIGH_RANGE_INDEX := 0
589         ; LOW_RANGE_INDEX := 0
590         ; VOWEL_COUNT := 0
591         END
592
593       ; PROCEDURE REJECT
594
595         ; BEGIN
596             K := 1
597           ; WRITE_INT ( AVAIL_LENG )
598           ; WHILE ( ( WORD_TABLE [ K ] <> ' ' )
599                     AND ( WORD_TABLE [ K ] <> NL )
600                     AND ( WORD_TABLE [ K ] <> CR )
601                     AND ( WORD_TABLE [ K ] <> EM )
602                     AND ( K <> TABLE_LENGTH )
603                   )
604             DO K := K + 1
605           ; TOTAL_LENG := K
606           ; WORD_TABLE [ K ] := ' '
607           ; WHILE ( NOT ( WORD_TABLE [ I ] IN ENDCHAR )
608                     AND ( I <= TABLE_LENGTH )
609                   )
610             DO BEGIN
611               IF ( ( ( WORD_TABLE [ I ] < 'a' )
612                       OR ( WORD_TABLE [ I ] > 'z' )
613                    )
614                    AND ( WORD_TABLE [ I ] <> '-' )
615                  )
616               THEN BEGIN
617                 HYPHEN_FLAG := FALSE
618               ; CONTINUE_FLAG := FALSE
619               END
620             ; IF WORD_TABLE [ I ] = '-'
621               THEN HYPHEN := I
622             ; IF WORD_TABLE [ I ] IN VOWEL_SET
623               THEN BEGIN
624                 VOWEL_COUNT := VOWEL_COUNT + 1
625               ; IF NOT FIRSTFLAG
626                 THEN BEGIN
627                   LOW_RANGE_INDEX := I + 1
628                 ; FIRSTFLAG := TRUE
629                 END
630                 ELSE HIGH_RANGE_INDEX := I
631               END
632             ; LENG := LENG + 1
633             ; I := I + 1
```

```
634           END
635         END
636
637     ; PROCEDURE CK_HYPHEN
638
639       ; BEGIN
640           IF HYPHEN <> 0
641           THEN BEGIN
642             U := 'H'
643           ; CONTINUE_FLAG := FALSE
644           ; IF HYPHEN <= AVAIL_LENG
645             THEN BEGIN
646             ; AVAIL_LENG := HYPHEN
647             ; J := 1
648             ; FOR I := ( HYPHEN + 1 ) TO TOTAL_LENG
649               DO BEGIN
650                 WORD_TABLE_REMAINING [ J ] := WORD_TABLE [ I ]
651               ; J := J + 1
652               END
653             ; HYPHEN_FLAG := TRUE
654             ; REMAINING_LENG := TOTAL_LENG - HYPHEN
655             END
656           END
657         END
658
659     ; PROCEDURE SPECIAL_REJECT
660
661       ; BEGIN
662           IF AVAIL_LENG < HIGH_RANGE_INDEX
663           THEN HIGH_RANGE_INDEX := AVAIL_LENG
664         ; IF ( ( VOWEL_COUNT <= 1 )
665                OR ( HIGH_RANGE_INDEX <= LOW_RANGE_INDEX )
666                OR ( LENG <= 3 )
667              )
668           THEN CONTINUE_FLAG := FALSE
669         ; IF ( ( LENG <= 5 ) AND CONTINUE_FLAG )
670           THEN BEGIN
671             IF ( ( LENG = 5 )
672                  AND ( ( WORD_TABLE [ 5 ] = 's' )
673                        OR ( WORD_TABLE [ 5 ] = 'd' )
674                      )
675                )
676             THEN CONTINUE_FLAG := FALSE
677           ; IF ( ( LENG = 4 ) AND ( WORD_TABLE [ 4 ] <> 'y' ) )
678             THEN CONTINUE_FLAG := FALSE
679           END
```

```
680         END
681
682     ; PROCEDURE CK_DICTIONARY
683
684       ; VAR K : INTEGER
685
686       ; BEGIN
687           AFFIX_NUM := 0
688         ; U := 'D'
689         ; MATCHSTRING
690             ( SVC1_IN , EXCEPT , WORD_TABLE , LENG , AFFIX_NUM )
691         ; IF MATCH_FOUND
692           THEN BEGIN
693             K := 0
694           ; CASE EXCEPT [ TABLE_FILLED_TO , 26 ]
695             OF
696               '1' : K := K + 10
697             ; '2' : K := K + 20
698             ; ELSE : K := 0
699             END
700           ; CASE EXCEPT [ TABLE_FILLED_TO , 27 ]
701             OF
702               '0' , '1' , '2' , '3' , '4' , '5' , '6' , '7' , '8'
703               , '9'
704               : K
705                 := K
706                    + ( ORD ( EXCEPT [ TABLE_FILLED_TO , 27 ] ) - 48 )
707             ; ELSE : K := 0
708             END
709           ; BREAKWORD ( K )
710           END
711         ; SVC1_IN . SVC1_FUNC := 192
712         ; SVC1 ( SVC1_IN )
713         END
714
715     ; PROCEDURE CK_DOUBLE_CONSONANT
716
717       ; BEGIN
718           FOR I := 2 TO LENG
719           DO BEGIN
720             U := 'C'
721           ; IF ( ( NOT ( WORD_TABLE [ I ] IN VOWEL_SET )
722                    AND ( ( WORD_TABLE [ I - 1 ] = WORD_TABLE [ I ] )
723                          AND ( WORD_TABLE [ I ] <> 'l' )
724                          AND ( WORD_TABLE [ I ] <> 's' )
725                        )
```

```
726                 )
727               )
728           THEN DOUBLE_CONSONANT := I
729         END
730       ; IF ( ( DOUBLE_CONSONANT <> 0 )
731              AND ( DOUBLE_CONSONANT <= HIGH_RANGE_INDEX )
732              AND ( DOUBLE_CONSONANT >= LOW_RANGE_INDEX )
733            )
734         THEN BREAKWORD ( DOUBLE_CONSONANT )
735       END
736
737   ; PROCEDURE RECURSIVE_S_CK
738
739     ; VAR STORE_WORD_TABLE : SCANTABLE
740       ; STORE_LENG , STORE_TOTAL_LENG : INTEGER
741
742     ; BEGIN
743         IF ( ( WORD_TABLE [ LENG ] = 's' )
744              AND ( WORD_TABLE [ LENG - 1 ] <> 's' )
745            )
746         THEN BEGIN
747         ; STORE_LENG := LENG + 1
748         ; STORE_TOTAL_LENG := TOTAL_LENG
749         ; FOR I := 1 TO STORE_TOTAL_LENG
750           DO STORE_WORD_TABLE [ I ] := WORD_TABLE [ I ]
751         ; WORD_TABLE [ LENG ] := ' '
752         ; HYPH
753             ( WORD_TABLE , AVAIL_LENG , HYPHEN_FLAG
754             , WORD_TABLE_REMAINING , REMAINING_LENG , CONTINUE_FLAG
755             )
756         ; IF HYPHEN_FLAG
757           THEN BEGIN
758             WORD_TABLE_REMAINING [ REMAINING_LENG ] := 's'
759           ; REMAINING_LENG := REMAINING_LENG + 1
760           ; FOR I := STORE_LENG TO STORE_TOTAL_LENG
761             DO BEGIN
762               WORD_TABLE_REMAINING [ REMAINING_LENG ]
763               := STORE_WORD_TABLE [ I ]
764             ; REMAINING_LENG := REMAINING_LENG + 1
765             END
766           ; REMAINING_LENG := REMAINING_LENG - 1
767           ; CONTINUE_FLAG := FALSE
768           END
769           ELSE WORD_TABLE [ LENG ] := 's'
770         END
771       END
```

```
772
773     ; PROCEDURE CK_SUFFIX
774
775       ; BEGIN
776           IF ( LENG - ( SUFFIX_LONGEST - 1 ) ) < LOW_RANGE_INDEX
777           THEN K := LOW_RANGE_INDEX
778           ELSE K := LENG - ( SUFFIX_LONGEST - 1 )
779         ; U := 'S'
780         ; AFFIX_NUM := 1
781         ; WHILE ( ( K <= HIGH_RANGE_INDEX ) AND CONTINUE_FLAG )
782           DO BEGIN
783             J := 1
784           ; M := LENG
785           ; FOR I := K TO LENG
786             DO BEGIN
787               SHORT_TABLE [ J ] := WORD_TABLE [ M ]
788             ; J := J + 1
789             ; M := M - 1
790             END
791           ; LENG_STRING := J
792           ; SHORT_TABLE [ J ] := ' '
793           ; MATCHSTRING
794               ( SVC1_SUFF , AFFIX , SHORT_TABLE , LENG_STRING
795               , AFFIX_NUM )
796           ; IF MATCH_FOUND
797             THEN BREAKWORD ( K )
798           ; K := K + 1
799           ; IF AFFIX_NUM <> 1
800             THEN BEGIN
801             ; SVC1_SUFF . SVC1_FUNC := 192
802             ; SVC1 ( SVC1_SUFF )
803             ; SVC1_SUFF . SVC1_FUNC := 72
804             END
805           END
806         ; SVC1_SUFF . SVC1_FUNC := 192
807         ; SVC1 ( SVC1_SUFF )
808         END
809
810     ; PROCEDURE CK_PREFIX
811
812       ; BEGIN
813           IF ( HIGH_RANGE_INDEX - 1 ) < PREFIX_LONGEST
814           THEN I := ( HIGH_RANGE_INDEX - 1 )
815           ELSE I := PREFIX_LONGEST
816         ; U := 'P'
817         ; AFFIX_NUM := 2
```

```
818         ; WHILE ( ( I >= ( LOW_RANGE_INDEX - 1 ) ) AND CONTINUE_FLAG )
819           DO BEGIN
820             J := 1
821           ; FOR K := 1 TO I
822             DO BEGIN
823               SHORT_TABLE [ J ] := WORD_TABLE [ K ]
824             ; J := J + 1
825             END
826           ; LENG_STRING := J
827           ; SHORT_TABLE [ J ] := ' '
828           ; MATCHSTRING
829               ( SVC1_PRE , AFFIX , SHORT_TABLE , LENG_STRING
830               , AFFIX_NUM )
831           ; IF MATCH_FOUND
832             THEN BREAKWORD ( LENG_STRING )
833           ; I := I - 1
834           ; IF AFFIX_NUM <> 2
835             THEN BEGIN
836             ; SVC1_PRE . SVC1_FUNC := 192
837             ; SVC1 ( SVC1_PRE )
838             ; SVC1_PRE . SVC1_FUNC := 72
839             END
840           END
841         ; SVC1_PRE . SVC1_FUNC := 192
842         ; SVC1 ( SVC1_PRE )
843         END
844
845       ; PROCEDURE CK_BREAK_VALUE_TABLE
846
847         ; BEGIN
848             I := LOW_RANGE_INDEX - 1
849           ; J := LOW_RANGE_INDEX
850           ; U := 'B'
851           ; WHILE ( ( J <= HIGH_RANGE_INDEX ) AND CONTINUE_FLAG )
852             DO BEGIN
853               L := ORD ( WORD_TABLE [ I ] ) - 96
854             ; K := ORD ( WORD_TABLE [ J ] ) - 96
855             ; IF BREAK_VALUE_TABLE [ L , K ]
856               THEN BREAKWORD ( J )
857             ; I := I + 1
858             ; J := I + 1
859             END
860           END
861
862       ; BEGIN
863           INITIALIZE
```

```
      ; REJECT
      ; CK_HYPHEN
      ; IF CONTINUE_FLAG
        THEN SPECIAL_REJECT
      ; IF CONTINUE_FLAG
        THEN CK_DICTIONARY
      ; IF CONTINUE_FLAG
        THEN CK_DOUBLE_CONSONANT
      ; IF CONTINUE_FLAG
        THEN RECURSIVE_S_CK
      ; IF CONTINUE_FLAG
        THEN CK_SUFFIX
      ; IF CONTINUE_FLAG
        THEN CK_PREFIX
      ; IF CONTINUE_FLAG
        THEN CK_BREAK_VALUE_TABLE
(*  ED  AND  ES CHECK  *)
      ; IF ( ( CONTINUE_FLAG )
             AND ( ( WORD_TABLE [ LENG - 1 ] = 'e' )
                   AND ( ( WORD_TABLE [ LENG ] = 'd' )
                         OR ( WORD_TABLE [ LENG ] = 's' )
                       )
                 )
           )
        THEN BREAKWORD ( LENG - 1 )
      END

  ; BEGIN
      INIT_BREAKVT
    ; WHILE X <> EM
      DO BEGIN
        FOR I := 1 TO 25 DO CHARTABLE [ I ] := ' '
      ; READ ( X )
      ; IF X <> EM
        THEN BEGIN
          IF ( ( X = ' ' ) OR ( X = CR ) OR ( X = NL ) )
          THEN WRITE ( X )
          ELSE BEGIN
            WORDCOUNT := 1
          ; CHARTABLE [ 1 ] := X
          ; REPEAT READ ( X )
            ; IF X <> EM
              THEN BEGIN
                WORDCOUNT := WORDCOUNT + 1
              ; CHARTABLE [ WORDCOUNT ] := X
              ;
```

```
910             END
911           UNTIL ( ( X = EM )
912                   OR ( X = ' ' )
913                   OR ( X = CR )
914                   OR ( X = NL )
915                 )
916         ; FOR I := 1 TO 25 DO STORETABLE [ I ] := CHARTABLE [ I ]
917         ; STORECOUNT := WORDCOUNT
918         ; HYPHFLAG := FALSE
919         ; WORDCOUNTR := WORDCOUNT
920         ; HYPH
921             ( CHARTABLE , WORDCOUNT , HYPHFLAG , CHARTABLER
922             , WORDCOUNTR , STOPPED_FLAG )
923         ; IF STOPPED_FLAG AND NOT HYPHFLAG
924           THEN BEGIN
925             FOR K := 1 TO TABLE_LENGTH
926             DO DISPLAY ( CHARTABLE [ K ] )
927           ; DISPLAY ( CR )
928           END
929         ; IF NOT STOPPED_FLAG AND NOT HYPHFLAG
930           THEN U := ' '
931         ; WRITETABLE
932         ; WRITE ( NL )
933         ; IF HYPHFLAG
934           THEN BEGIN
935             FOR I := 4 TO STORECOUNT
936             DO BEGIN
937             ; IF ( ( U <> 'D' ) AND ( U <> 'H' ) )
938               THEN BEGIN
939                 HYPHFLAG := FALSE
940               ; FOR J := 1 TO 25 DO CHARTABLE [ J ] := ' '
941               ; FOR J := 1 TO 25
942                 DO CHARTABLE [ J ] := STORETABLE [ J ]
943               ; J := I
944               ; HYPH
945                   ( CHARTABLE , I , HYPHFLAG , CHARTABLER
946                   , WORDCOUNTR , STOPPED_FLAG )
947               ; I := J
948               ; WORDCOUNT := I
949               ; WRITETABLE
950               ; WRITE ( NL )
951               END
952             END
953           END
954       END
955     END
```

```
956       END
957     END
958 .
959
```

The commands used to execute this program are:

```
LOAD USR6:PRODICT.TSK
ASSIGN 0, DIS.TXT        "ANOMALY FILE"
ASSIGN 1, (YOUR INPUT FILE)
ASSIGN 2, (YOUR OUTPUT FILE)
ASSIGN 3, EXCEPT.TXT     "EXCEPTION DICTIONARY"
ASSIGN 4, SUF.TXT        "REVERSED SUFFIX FILE"
ASSIGN 5, PRE.TXT        "PREFIX FILE"
START
```

The following is a sample of the output this program generates. The first number is the length of the word, and is also the assumed length of the space remaining on the current output line. Words with no hyphen have either been rejected or are anomalies. Words with the hyphen in the wrong place for any length should be loaded into the exception dictionary.

```
    12Hyphenation
3is
8de- fined  P
4de-  fined  P
5de-   fined  P
```

```
 6de-    fined  P
 7de-     fined  P
 8de-      fined  P
 3as
 9di- viding  D
 3or
11con- necting  C
 4con- necting  C
 5con-  necting  C
 6con-   necting  C
 7con-    necting  C
 8con-     necting  C
 9con-      necting  C
10con-       necting  C
11con-        necting  C
11(syllables
 3or
 5word
10elements)
 5with
 2a
 8hyph- en.  D
13Hyphenation,
 3in
 5word
11proc- essing  D
 9          9sys- tems,  S
 4          4sys- tems,  S
 5          5sys-  tems,  S
 6          6sys-   tems,  S
 7          7sys-    tems,  S
 8          8sys-     tems,  S
 9          9sys-      tems,  S
 3is
 8usual- ly  S
 4usualB
 5usuallB
 6usual- ly  S
 7usual-  ly  S
 8usual-   ly  S
10per- formed  P
 4per- formed  P
 5per-  formed  P
 6per-   formed  P
 7per-    formed  P
```

APPENDIX H - FILE FORMATTER

This is used with PEDIT to update and change the hyphenator's files.

This program is stored on USR6 as Format.pas/38 or Format.tsk/38

```
      1
      2 "ROBERT YOUNG"
      3 "KANSAS STATE UNIVERSITY"
      4 "DEPARTMENT OF COMPUTER SCIENCE"
      5   CONST COPYRIGHT = 'COPYRIGHT ROBERT YOUNG 1978'
      6
      7 "############
C     8 #  PREFIX  #
C     9 ############"
     10
     11 ; CONST NL = '(:10:)'
     12   ; FF = '(:12:)'
     13   ; CR = '(:13:)'
     14   ; EM = '(:25:)'
     15
     16 ; CONST PAGELENGTH = 512
     17
     18 ; TYPE PAGE = ARRAY (. 1 .. PAGELENGTH .) OF CHAR
     19
     20 ; CONST LINELENGTH = 132
     21
     22 ; TYPE LINE = ARRAY (. 1 .. LINELENGTH .) OF CHAR
     23
     24 ; CONST IDLENGTH = 12
     25
     26 ; TYPE IDENTIFIER = ARRAY (. 1 .. IDLENGTH .) OF CHAR
     27
     28 ; TYPE FILE = 1 .. 2
     29
     30 ; TYPE FILEKIND = ( EMPTY , SCRATCH , ASCII , SEQCODE , CONCODE )
     31
     32 ; TYPE FILEATTR
     33     = RECORD
     34         KIND : FILEKIND
     35       ; ADDR : INTEGER
     36       ; PROTECTED : BOOLEAN
     37       ; NOTUSED : ARRAY (. 1 .. 5 .) OF INTEGER
     38       END
     39
```

```
40 ; TYPE IODEVICE
41     = ( TYPEDEVICE , DISKDEVICE , TAPEDEVICE , PRINTDEVICE
42       , CARDDEVICE )
43
44 ; TYPE IOOPERATION = ( INPUT , OUTPUT , MOVE , CONTROL )
45
46 ; TYPE IOARG = ( WRITEEOF , REWIND , UPSPACE , BACKSPACE )
47
48 ; TYPE IORESULT
49     = ( COMPLETE , INTERVENTION , TRANSMISSION , FAILURE , ENDFILE
50       , ENDMEDIUM , STARTMEDIUM )
51
52 ; TYPE IOPARAM
53     = RECORD
54         OPERATION : IOOPERATION
55       ; STATUS : IORESULT
56       ; ARG : IOARG
57       END
58
59 ; TYPE TASKKIND = ( INPUTTASK , JOBTASK , OUTPUTTASK )
60
61 ; TYPE ARGTAG = ( NILTYPE , BOOLTYPE , INTTYPE , IDTYPE , PTRTYPE )
62
63 ; TYPE POINTER = @ BOOLEAN
64
65 ; TYPE PASSPTR = @ PASSLINK
66
67 ; TYPE PASSLINK
68     = RECORD
69         OPTIONS : SET OF CHAR
70       ; FILLER1 : ARRAY (. 1 .. 7 .) OF INTEGER
71       ; FILLER2 : BOOLEAN
72       ; RESET_POINT : INTEGER
73       ; FILLER3 : ARRAY (. 1 .. 11 .) OF POINTER
74       END
75
76 ; TYPE ARGTYPE
77     = RECORD
78         CASE TAG : ARGTAG
79         OF NILTYPE , BOOLTYPE : ( BOOL : BOOLEAN )
80         ; INTTYPE : ( INT : INTEGER )
81         ; IDTYPE : ( ID : IDENTIFIER )
82         ; PTRTYPE : ( PTR : PASSPTR )
83       END
84
85 ; CONST MAXARG = 10
```

```
 86
 87 ; TYPE ARGLIST = ARRAY (. 1 .. MAXARG .) OF ARGTYPE
 88
 89 ; TYPE ARGSEQ = ( INP , OUT )
 90
 91 ; TYPE PROGRESULT
 92     = ( TERMINATED , OVERFLOW , POINTERERROR , RANGEERROR
 93       , VARIANTERROR , HEAPLIMIT , STACKLIMIT , CODELIMIT , TIMELIMIT
 94       , CALLERROR )
 95
 96 ; PROCEDURE READ ( VAR C : CHAR )
 97
 98 ; PROCEDURE WRITE ( C : CHAR )
 99
100 ; PROCEDURE OPEN ( F : FILE ; ID : IDENTIFIER ; VAR FOUND : BOOLEAN )
101
102 ; PROCEDURE CLOSE ( F : FILE )
103
104 ; PROCEDURE GET ( F : FILE ; P : INTEGER ; VAR BLOCK : UNIV PAGE )
105
106 ; PROCEDURE PUT ( F : FILE ; P : INTEGER ; VAR BLOCK : UNIV PAGE )
107
108 ; FUNCTION LENGTH ( F : FILE ) : INTEGER
109
110 ; PROCEDURE MARK ( VAR TOP : INTEGER )
111
112 ; PROCEDURE RELEASE ( TOP : INTEGER )
113
114 ; PROCEDURE IDENTIFY ( HEADER : LINE )
115
116 ; PROCEDURE ACCEPT ( VAR C : CHAR )
117
118 ; PROCEDURE DISPLAY ( C : CHAR )
119
120 ; PROCEDURE NOTUSED
121
122 ; PROCEDURE NOTUSED2
123
124 ; PROCEDURE NOTUSED3
125
126 ; PROCEDURE NOTUSED4
127
128 ; PROCEDURE NOTUSED5
129
130 ; PROCEDURE NOTUSED6
131
```

```
136 ; PROCEDURE NOTUUED9
137
138 ; PROCEDURE NOTUSED10
139
140 ; PROCEDURE RUN
141     ( ID : IDENTIFIER
142     ; VAR PARAM : ARGLIST
143     ; VAR LINE : INTEGER
144     ; VAR RESULT : PROGRESULT
145     )
146
147 ; PROGRAM P ( PARAM : LINE )
148
149   ; VAR X , C : CHAR
150     ; I , J , K : INTEGER
151     ; TAB : ARRAY [ 1 .. 512 ] OF CHAR
152
153   ; BEGIN
154       X := ' '
155     ; I := 0
156     ; K := 0
157     ; WHILE X <> EM
158       DO BEGIN
159         REPEAT
160           READ ( X )
161         ; I := I + 1
162         ; IF ( X <> EM )
163           THEN TAB [ I ] := X
164         UNTIL ( X = EM ) OR ( X = NL )
165       ; IF ( ( I = 28 ) AND ( X = NL ) )
166         THEN BEGIN
167           FOR J := 1 TO 27 DO WRITE ( TAB [ J ] )
168         ; WRITE ( NL )
169         ; K := K + 1
170         END
171       ; IF K = 18
172         THEN BEGIN
173           FOR J := 1 TO 7 DO WRITE ( ' ' )
174         ; WRITE ( NL )
175         ; K := 0
176         END
177       ; I := 0
178       END
179     ; WRITE ( EM )
180     END
181 .
```

DESIGN OF AN AUTOMATED HYPHENATION PROCEDURE

by

Steven Scott Ranker

B. S., KANSAS STATE UNIVERSITY, 1980

AN ABSTRACT OF A

MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

Kansas State University

Manhattan, Kansas

1982

ABSTRACT

A procedure to hyphenate English language words has been developed. It uses an algorithmic method rather than a dictionary driven method to accomplish this task. The mainstay of the algorithmic method is a file of prefix and suffix substrings combined with certain compoundable root words, that is used to determine the hyphenable points within a word. A set of test conditions and a statistical probability table are used to supplement the substring file. An exception dictionary has been compiled and is maintained for words that the substrings, conditions, and statistical "break value table" incorrectly hyphenate. A number of automated tools are supplied in addition to the procedure itself. These are used to aid in the implementation by showing how to integrate the hyphenation procedure into a word processor, and by showing how to use the procedure to build the exception dictionary. The procedure uses three files as input to aid it in hyphenating words. These files need to be kept in a specific format for the sake of efficiency. To do the formatting, a program has been supplied that configures the files in the necessary format.