



(11) Publication number: **0 481 569 A2**

(12) **EUROPEAN PATENT APPLICATION**

(21) Application number: **91202690.3**

(51) Int. Cl.⁵: **B07C 3/00**

(22) Date of filing: **16.10.91**

(30) Priority: **16.10.90 US 598189**

(43) Date of publication of application:
22.04.92 Bulletin 92/17

(84) Designated Contracting States:
CH DE FR GB IT LI

(71) Applicant: **BELL & HOWELL PHILLIPSBURG COMPANY**
5215 Old Orchard Road
Skokie, Illinois 60077(US)

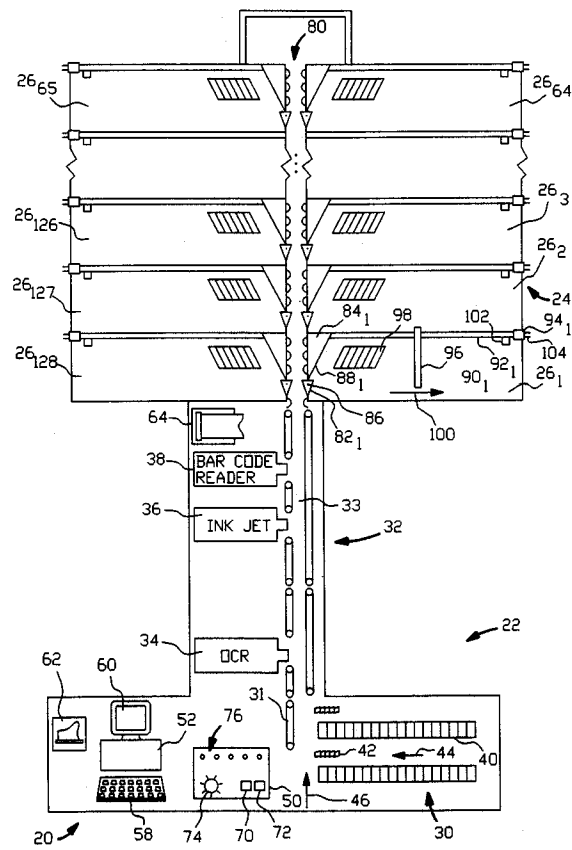
(72) Inventor: **Kostyniuk, Paul F.**
539 Park Avenue
Wilmette, Illinois 60091(US)

(74) Representative: **Mittler, Enrico et al**
c/o Marchi & Mittler s.r.l. Viale Lombardia, 20
I-20131 Milano(IT)

(54) **Mail sorting apparatus and method.**

(57) A mail sorting machine (20) includes an input hopper (30); a mailpiece reading and processing section (22); and, a sorting bin section (24) comprising a plurality of bins (26₁ -26₁₂₈). The reading and processing section (22) includes a CPU (54) which executes a program ANALYZE_MAIL for sorting third class mailpieces. The program ANALYZE_MAIL sorts the mailpieces included in a batch into packages, and then associates the packages into sacks or bags. The program ANALYZE_MAIL constructs the packages and sacks to obtain maximum postage discounts. Upon an initial pass of all mailpieces of a batch through the sorting machine (20), the program ANALYZE_MAIL generates output (TABLE 1) advising how the bins (26) are to be grouped for subsequent passes. The program ANALYZE_MAIL also generates output (TABLES 2A - 2E) advising, for each group, which bins (26) are to have their packages associated together for insertion into the same bag or sack. Advantageously, the mailpieces are sorted so that the bins (26) to be associated together are physically adjacent one another in the sorting machine (20). Bag tags are generated to tell an operator which bins are to be collected together to form a sack or bag, as well as the sack number and group number. The program ANALYZE_MAIL also includes an accounting capability for billing postage to a possible plurality of clients having mailstreams included in the batch, and for allocating postage costs in accordance with whether the client's mailpieces qualify for postage discounts.

FIG. 1



BACKGROUND OF THE INVENTION

I. FIELD OF THE INVENTION

5 This invention pertains to apparatus and method for sorting postal envelopes prior to mailing, and particularly to such apparatus and method for presorting envelopes in order to obtain postage discounts offered by the United States Postal Service.

II. PRIOR ART AND OTHER CONSIDERATIONS

10

The United States Postal Service is handling an ever increasing volume of domestic mail. A large component of the domestic mail volume is attributed to postal patrons who introduce large or bulk shipments of mail into the Postal Service. Examples of such postal patrons include financial institutions (such as banks and credit card companies that mail out periodic statements to their customers); utilities (which mail out monthly or quarterly bills); charitable and non-profit institutions; and, advertising agencies.

15

The United States Postal Service affords more favorable postage rates for postal patrons who cooperate with the United States Postal Service by preparing bulk shipments of mail in a manner more easily handled by the United States Postal Service. The United States Postal Service defines its postage class structure, and the requirements for obtaining the more favorable postage rates for bulk mail patrons, in a publication called the Domestic Mail Manual (also known as the "DMM").

20

By way of example for the foregoing, the DMM sets forth a schedule of rates and fees for third class mail, with the postage rate for third class mail depending upon a presort level of the mail. In this respect, for third class mail the DMM prescribes the following presort levels for bulk rate mail: basic; basic ZIP + 4; 5 digit; 5 digit ZIP + 4; ZIP + 4 barcoded; and, carrier route. Of these presort levels, the basic level is the most expensive, the basic ZIP + 4 the second most expensive, and so on with the carrier route presort level being the least expensive. Indeed, a patron preparing a bulk shipment of mail can achieve a considerable postage savings depending upon the extent to which the mailpieces included in the shipment qualify for the less expensive presort levels.

25

Qualifying for a particular presort level involves more than the degree (five or nine digit) and manner (barcoded or not) by which ZIP code information is provided on the mailpieces. For a mailpiece to qualify for most of the presort levels, the DMM further requires that the mailpiece be included as a part of a package (a specified number, such as 10 or more) of mailpieces packaged (i.e., associated by a rubberband) in accordance with specified criteria (such as the same carrier route, same 5 digit ZIP code destination, same 3 digit ZIP code prefix destination, for examples). In addition, to qualify for most of the presort levels, in accordance with specified criteria the postal patron must place the packages in a sack along with other mailpieces, and the sack must contain at least a specified minimum number of pieces (or have at least a specified minimum weight). Examples of such specified criteria for inclusion of mailpieces in the same sack are that the mailpieces either be destined to the same carrier route, the same 5 digit ZIP code destination, or the same 3 digit ZIP code prefix destination.

35

In addition to complying with the foregoing package and sack requirements, the postal patron must apply a label or tag, having a prescribed format, to each sack. As required by the DMM, the sack tag or label must include select information regarding the contents of the sack.

40

Thus, in order to obtain the maximum possible postage savings for each mailpiece, a postal patron must presort the mailpieces in accordance with ZIP code; must attempt to associate mailpieces in accordance with DMM specifications into packages; must attempt to associate packages in accordance with DMM specifications into sacks; and, must generate a label for each sack in accordance with the format prescribed by the DMM. It should become apparent that factors such as insufficient quantity and thin geographical distribution may disqualify many of the mailpieces included in a bulk shipment from receiving the most favored presort level. A greater postage rate associated with a less favored presort level must be paid for a mailpieces disqualified from the most favored presort level.

45

50

Large bulk shipments of mail can be presorted in ZIP code groupings using automated sorting machines, such as those provided by the Bell & Howell Phillipsburg Company. Examples of such automated sorting machines include the Bell & Howell Phillipsburg Company model 1000, 800, and 600 Mail Processing Systems. These automated sorting machines read optical characters and/or bar code and sort mail into bins.

55

While the prior art automated sorting machines cited above perform admirably for their initially intended purposes, the machines still required much human thought in the preparation of packages and associating of packages into sacks for obtaining the more favorable presort levels. In this respect, a human operator

must mentally determine how to associate into packages and sacks mailpieces from numerous and often non-adjacent bins of the sorting machine. Such tedious determinations are subject to human error. Erroneous packaging and sacking of mailpieces causes considerable consternation with the United States Postal Service, and may jeopardize or render suspect the entire bulk mail shipment.

5 In order to qualify as many mailpieces as possible for the most favored presort levels, many companies and organizations send their bulk shipments to a third party company such as a presort agency for combining with the bulk shipments of other companies and organizations. By combining the bulk mail shipments of several postal patrons, and by presorting the combined mail for the several patrons on the automated sorting machines described above, the presort agencies are often able to leverage the quantity
10 and geographical distribution factors in order to qualify the maximum number of mailpieces for the most favored presort levels. Unfortunately, since some mailpieces do not achieve the most favored presort levels, it is very difficult for the presort agencies to allocate the postage costs (e.g., qualified discount vs. non-qualified postage rate) incurred among the contributing patrons.

Moreover, some postal patrons meter the mailpieces included in a bulk shipment with postage prior to
15 conducting their own in-house sorting or prior to sending the shipment to a presort agency. In such cases, it may turn out that a mailpiece pre-metered at a rate for a favored presort level may not qualify for that presort level, with the result that additional postage must be applied to that mailpiece. When the mailpieces of more than one patron are combined or commingled, as at a presort agency, it is very difficult from an accounting standpoint to allocate the resultant postage increase triggered by the non-qualifying pre-metered
20 mailpiece to the postal patron from whom the mailpiece came.

As mentioned above, some postal patrons meter the mailpieces included in a bulk shipment prior to the sorting operation (either in-house or at a presort agency). Other postal patrons use the "permit" mail provisions of the United States Postal Service. Traditionally the United States Postal Service has refused to accept bulk shipments that include both pre-metered and permit mail, in view inter alia of the difficulty in
25 verifying the accuracy of the computed postage amounts.

In view of the foregoing, it is an object of the present invention to provide a sorting method and apparatus for associating mailpieces in a manner conducive for collection and associating into packages and sacks for obtaining desired postage presort levels.

An advantage of the present invention is the provision of method and apparatus for sorting mailpieces
30 and for providing reports indicative of the postage presort levels into which mailpieces are classified.

Another advantage of the present invention is the provision of method and apparatus for sorting mailpieces wherein sack labels are automatically generated for sacks of mail.

SUMMARY

35 A mail sorting machine includes an input hopper; a mailpiece reading and processing section; and, a sorting bin section comprising a plurality of bins. The reading and processing section includes a CPU which executes a program ANALYZE__MAIL for sorting third class mailpieces. The program ANALYZE__MAIL sorts the mailpieces included in a batch into packages, and then associates the packages into sacks or
40 bags. The program ANALYZE__MAIL constructs the packages and sacks to obtain maximum postage discounts.

Upon an initial physical pass of all mailpieces of a batch through the sorting machine, the program ANALYZE__MAIL generates output advising how the bins are to be associated for subsequent passes. As a result of the first physical pass, the program ANALYZE__MAIL classifies the mailpieces of the batch into a
45 plurality of "groups". A "group" is a set of mailpieces which is to be separately sorted, independently from the remainder of the batch, during one or more "passes" of the sorting machine. For example, after the first or initial physical pass, for a first subsequent pass only a first group of mailpieces is loaded into the input hopper; for a second subsequent pass only a second group of mailpieces is loaded into the input hopper; and so forth. In general, during the first physical pass, mailpieces belonging to a first group are assigned to
50 a first set of bins; mailpieces belonging to a second group are assigned to a second set of the bins; and so forth.

In addition to assigning mailpieces to specified groups (i.e., bins) during the first physical pass, the program ANALYZE__MAIL also generates a report which informs the operator from which bins to collect each group. Advantageously, each group is collected from physically adjacent bins. Moreover, the program
55 ANALYZE__MAIL also generates a report for each group, which report indicates from which bins mailpieces are to be collected into sacks. Advantageously, the sack is composed of mailpieces from physically adjacent bins. Further, the program ANALYZE__MAIL generates "bag tags" (also known as "sack tags") for each sack, with the bag tag bearing information to apprise the operator from which bins to gather the

contents of the sack, as well as the sack and group number.

The program ANALYZE_MAIL also includes an accounting capability for billing postage to a possible plurality of clients having mailstreams included in the batch, and for allocating postage costs in accordance with whether the client's mailpieces qualify for postage discounts.

5

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects, features, and advantages of the invention will be apparent from the following more particular description of preferred embodiments as illustrated in the accompanying drawings in which reference characters refer to the same parts throughout the various views. The drawings are not necessarily to scale, emphasis instead being placed upon illustrating the principles of the invention.

10

Fig. 1 is a top schematic view of a sorter apparatus according to an embodiment of the invention.

Fig. 2 is a schematic view of electronic circuitry included in the sorter apparatus of the embodiment of Fig. 1.

15

Fig. 3 is an isometric view of an edge post and hook assembly provided thereon according to the embodiment of Fig. 1.

Fig. 4 is a schematic view showing the interrelationships between Figs. 4A - 4N.

Figs. 4A - 4N are schematic views showing a series of functions and their constituent steps executed in accordance with a program ANALYZE_MAIL by the sorter apparatus of the embodiment of Fig. 1.

20

DETAILED DESCRIPTION OF THE DRAWINGS

Fig. 1 shows a sorter apparatus 20 according to an embodiment of the invention. The sorter 20 includes a reading and processing section 22 and a sorting bin section 24. The sorting bin section 24 includes a plurality of bins 26 into which mailpieces are ultimately sorted. In one embodiment, 128 such bins 26, numbered as bins 26₁-26₁₂₈, are provided (although not all of the 128 such bins 26 are illustrated in Fig. 1). It should be understood that in other embodiments a different number of bins are provided.

25

The processing section 22 includes an input hopper 30; a feeder 31; and a mailpiece transport assembly 32 which directs mailpieces along a processing path 33. Along the processing path 33 are various processing stations also included in the processing section 22, including an optical character recognition (OCR) station 34; an ink jet printer station 36; a bar code reader station 38.

30

As shown in Fig. 1, a plurality of hopper floor belts 40 and hopper augers 42 transport incoming mailpieces on edge in the direction of arrow 44 toward the feeder 31. The feeder 31, being of a rotating belt variety, feeds the leading mailpiece in the hopper 31 in the direction of the processing path 33 (i.e., in the direction of arrow 46). The mailpieces travel on edge down the processing path 33 along the OCR station 34, the ink jet printer station 36, the bar code reader section 38, and into the sorting bin section 24.

35

The processing section 22 also includes an operator console 50 and a data processing system 52. The data processing system 52 includes a central processing unit (CPU) 54 and an I/O interface 56 (see Fig. 2). The CPU 54 communicates through the I/O interface 56 to various peripheral devices, including a keyboard 58; a monitor 60; a report printer 62; a bag tag or sack tag printer 64; and, a disk drive 66 (see Fig. 2). In addition, the CPU 54 communicates through the I/O interface 56 to electronics for the aforementioned OCR station 34; ink jet printer station 36; bar code reader station 38; and, operator console 50.

40

The operator console 50 includes a start switch 70 and stop switch 72, as well a feed select switch 74 and a series of status indicator lights 76. The feed select switch 74 is used to control the rate at which the feeder 31 feeds mailpieces from the input hopper 30 toward the mail processing path 33.

45

The sorting bin section 24 includes a central transport assembly 80 which directs mailpieces along a sorting path that is collinear with the processing path 33 of the processing section 22. That is, the central transport assembly 80 of the sorting bin section 24 continues to transport a mailpiece in the direction of arrow 46 through the sorting bin section 24 until the mailpiece is deflected into an appropriate one of the bins 26. The central transport assembly 80 includes a plurality of unillustrated transport belts.

50

As shown in Fig. 1, bins are provided on both sides of the sorting path in paired relationship. That is, at the same distance from the processing section 22, bin 26₁ is paired with bin 26₁₂₈; a little further downstream bin 26₂ is paired with bin 26₁₂₇; and so forth until the downstream-most pair of bins 26₆₄ and 26₆₅.

55

The sorting path is defined by a plurality of diverter gates 82 and sorting path walls 84. Each bin 26 has a diverter gate 82 and a sorting path wall 82 associated therewith. When activated, a diverter gate 84 pivots about a pivot point (such as pivot point 86 shown with respect to diverter gate 82₁ of bin 26₁) for diverting a mailpiece from the sorting path into its respective bin. Each sorting path wall 84 has a rear ramp surface 88

inclined with respect to the sorting path. The rear ramp surface 88 and the diverter gate 82 form a planar surface inclined with respect to the sorting path when the diverter gate 82 is pivoted to deflect mailpieces out of the sorting path and into the bin 26. The angle of inclination of this planar surface including the rear ramp surface 88 facilitates direction of the on edge mailpiece into the queue of mailpieces developing in the bin.

In addition to the rear ramp surface 88, a bin 26 is defined by a horizontal bin floor 90 and a leading edge abutment rail 92. The leading edge abutment rail is suspended above the horizontal bin floor 90 between the sorting path wall 84 and a vertical edge post 94. As shown in Figs. 1 and 3, the leading edge abutment rail extends through an aperture provided in a travelling vertical plate 96. The bin floor 90 has an auger 98 provide therein which transports mailpieces diverted into the bin 26 in a direction perpendicular and away from the sorting path. For example, with reference to bin 26₁, the auger 98₁ directs deflected mailpieces away from the sorting path in the direction of arrow 100. The leading or first such mailpiece deflected into a bin 26 contacts the travelling plate 96. As successive mailpieces are diverted into a bin 26 and interposed between the ramp surface 88 and the previous mailpiece, the travelling plate 96 is slidably pushed along the abutment rail 92 away from the sorting path toward the edge post 94. Near the edge post 94 a pressure sensor switch 102 is provided to detect when a bin 26 is becoming full. In this respect, when a sufficient number of mailpieces are diverted into a single bin 26 such that the travelling plate 96 closes the pressure sensor switch 102 for that bin, the sorting operation is temporarily halted and a diagnostic message is provided to the operator so that mailpieces diverted to the bin can be manually removed for accommodating additional mailpieces in that bin.

As is shown in Fig. 3, each edge post 94 has hook assembly 104 provided thereon for engaging mail sacks, for example. The hook assembly 104 comprises two perpendicular bracket members 106a and 106b. The hook assembly 104 is mounted on the edge post 94 by fasteners 108 which extend through the bracket member 106a. The bracket member 106b has two U-shaped hooks 110 provided thereon. The hooks 110 engage one of the metal rings 112 provided around the mouth of a mail sack 114.

Referring again to the diverter gates 82 provided along the sorting path, each diverter gate 82 is activated by a solenoid 120. The solenoids 120 of each pair of bins are controlled by bin pair controller 122, there being 64 such bin pair controllers 122 shown in the embodiment of Fig. 1. Bin pair controller 122₁ controls the solenoids 120₁ and 120₁₂₈ for bins 26₁ and 26₁₂₈, respectively; bin pair controller 122₂ controls the solenoids 120₂ and 120₁₂₇ for bins 26₂ and 26₁₂₇, respectively; and so forth.

The bin pair controllers 122 are connected in series to the I/O interface 56 of the data processing system 52 in shift register fashion along line 124. The signal carried on line 124 is a digital signal indicative of the bin number to which a mailpiece should be directed in accordance with the sorting operation. The signal from the I/O interface 56 is first applied to the bin pair controller 122₁ as a mailpiece approaches diverter gates 82₁ and 82₁₂₈. If the signal indicates that the mailpiece is destined for either bin 26₁ or bin 26₁₂₈, the bin pair controller, upon evaluating the signal, causes activation of the appropriate solenoid 120. If the signal indicates that the mailpiece is destined for another downstream bin 26, the signal for that mailpiece is shifted downstream to the bin pair controller 122₂ as the mailpiece approaches diverter gates 82₂ and 82₁₂₇ associated with bins 26₂ and 26₁₂₇. The bin pair controller 122₂ then either activates an appropriate solenoid 120 or shifts the signal yet further downstream along with the travelling mailpiece.

The data processing system 52, and particularly the CPU 54 executes a set of instructions that control the operation of the sorter 20. That set of instructions is collectively referred to as program ANALYZE_MAIL. The program ANALYZE_MAIL consists of numerous subsets of instructions coded in the "C" programming language, which subsets are referred to herein as "functions". Execution of the program ANALYZE_MAIL and its constituent functions causes the sorting machine 20 to operate in the manner described below. In connection with the ensuing description of the operation of the sorting machine 20, it should be understood that the word "bundle" is often used interchangeably with "package", and that the word "bag" is often used interchangeably with "sack".

OPERATION

In the operation of the sorting machine 20 of Fig. 1, a batch of third class mail is placed in the input hopper 30. The batch may comprise a plurality of mailstreams from a plurality of patrons. In the example discussed hereinafter particularly with reference to the TABLES, the batch includes mailstreams from an insurance company patron, a utility company patron, and a publishing company patron. The insurance company actually contributes three separate mailstreams to the batch, in particular an automotive insurance mailstream, a life insurance mailstream, and a health insurance mailstream.

After the mailstreams are all loaded into the input hopper 30, and when the CPU 54 is running the

program ANALYZE__MAIL and the start switch 70 is turned on, the operator activates the feed select switch 74. Activation of the feed select switch 74 initiates a first physical pass of the mailpieces through the sorter 20, which includes transport of the mailpieces through the input hopper 30 in the direction of arrow 44; through the processing section 32 (in the direction of arrow 46); and, into the sorting bin section 24. As
 5 each mailpiece is transported through the processing section 22, the OCR 34 reads the character address; the ink jet printer 36 prints a barcode corresponding to the character-read ZIP code; and, the bar code reader 38 verifies the printed bar code.

As discussed hereinbefore, the program ANALYZE__MAIL executed by sorting machine 20 collectively sorts the entire batch, comprising the mailstreams of all the patrons, in order to achieve the optimum third
 10 class postage discounts in accordance with the DMM. In order to do so, the program ANALYZE__MAIL associates the mailpieces of the batch into "packages" (also known as "bundles"), and the packages are associated into sacks or bags.

In the above regard, the program ANALYZE__MAIL creates four different package types, notably: five digit (or "5") packages; three digit ("3") packages; state ("S") packages; and, mixed state (or "M")
 15 packages. Generally, packages must consist of ten (10) or more mailpieces satisfying the same package ZIP code rule. For example, all the mailpieces in a FIVE__DIGIT package must be destined for the same five digit zip code. All the mailpieces included in a THREE__DIGIT package must have the same three initial three ZIP code digits (e.g., 22151, 22153, 22155, 22165). All the mailpieces included in a single STATE package must have initial ZIP code digits which destine the mailpieces to the same state (e.g., to Illinois).

As is more fully explained by the DMM, to qualify for certain third class postage discounts, the packages must in turn be placed into sacks consisting of a minimum number of mailpieces (or a minimum weight). Typically the minimum number of mailpieces per sack is 125, or alternatively the minimum weight is 15 pounds. Accordingly, the program ANALYZE__MAIL creates four different sack types: FIVE__DIGIT
 20 sacks; THREE__DIGIT sacks; STATE sacks; and MIXED__STATE sacks.

As a result of the first physical pass, the program ANALYZE__MAIL classifies the mailpieces of the batch into a plurality of "groups". A "group" is a set of mailpieces which is to be separately sorted, independently from the remainder of the batch, during one or more "passes" of the sorting machine 20. For example, after the first or initial physical pass, for a first subsequent pass only a first group of mailpieces is loaded into the input hopper 30; for a second subsequent pass only a second group of mailpieces is loaded
 30 into the input hopper 30; and so forth. In general, during the first physical pass, mailpieces belonging to a first group are assigned to a first set of bins 26; mailpieces belonging to a second group are assigned to a second set of the bins 26; and so forth.

In addition to assigning mailpieces to specified groups (i.e., bins) during the first physical pass, the program ANALYZE__MAIL also generates a report in the form of TABLE 1 which informs the operator from
 35 which bins 26 to collect each group. Advantageously, each group is collected from physically adjacent bins 26. Moreover, the program ANALYZE__MAIL also generates a report in the form of TABLE 2A for each group, which report indicates from which bins mailpieces are to be collected into sacks. Advantageously, the sack is composed of mailpieces from physically adjacent bins 26. Further, the program ANALYZE__MAIL generates "bag tags" (also known as "sack tags") for each sack, with the bag tag bearing
 40 information to apprise the operator from which bins 26 to gather the contents of the sack, as well as the sack and group number. A format for a plurality of bag tags is illustrated in TABLE 3.

In addition, during the first physical pass, the program ANALYZE__MAIL generates a number of accounting reports as exemplified by TABLES 4, 5, 6, and 6A discussed infra.

45 **Function FIRST__SORT__PASS**

As a result of the execution of a function FIRST__SORT__PASS, three files are created during the first physical pass, including file COUNT.DAT, file AGGR.DAT, and CLIENT1.DAT. The creation of these three files is indicated by steps 200, 202, and 204 in Fig. 4A.

50 The size of the file COUNT.DAT (i.e., the number of records in the file) depends on the number of unique zip codes and mailstreams encountered during the first physical pass.

The format of each record in file COUNT.DAT is as follows:

			<u>byte offset</u>
#			
	Zip Code	- 4 bytes (long integer)	0
5	Stream Index	- 1 byte (hex/binary value)	4
	Client Index	- 1 byte (hex/binary value)	5
	Bin	- 1 byte (hex/binary value)	6
	5 Digit OCR/BCR count	- 2 bytes (unsigned integer)	7
	Zip + 4 OCR count	- 2 bytes (unsigned integer)	9
10	Zip + 4 Barcoded count	- 2 bytes (unsigned integer)	11

After the first physical pass, the records in file COUNT.DAT are sorted in ascending order. In this respect, a primary sortation is done by ZIP code. For ZIP codes repeated due to their usage in different client/mailstreams, a secondary sortation is performed by first sorting the client index number, followed by the stream index number.

The following is an example of how multiple records in the file COUNT.DAT are stored (note that the binary values are converted to ascii for display purposes):

ZIP CODE	stream	client	bin	5 Digit	ZIP + 4	ZIP + 4 Barcoded
203460000	2	1	3	12	3	7
203500000	1	1	3	1	0	3
203500000	3	1	3	0	0	1
203500000	2	3	3	1	0	0
203500000	0	7	3	0	2	0
302530000	2	1	4	0	0	5
406770000	1	1	5	0	1	3

File AGGR.DAT is a binary file of fixed length. Each entry is four bytes long representing a long integer. The first 256 entries is an array of bin counts where each individual bin count is indexed by bin. The 257th entry represents the total number of mailpieces fed, the 258th entry represents the total number of mailpieces read, and the last entry represents the total 5 Digit ZIP count (OCR and Barcoded mailpieces). The following summarizes the file format for file AGGR.DAT:

counts by bin - 256 entries, each entry 4 bytes
total fed - 1 entry 4 bytes long
total read - 1 entry 4 bytes long
total 5 Digit count - 1 entry 4 bytes long

The file CLIENT1.DAT is a binary file containing ten bytes per record. Each record contains the following data and is presented in order:

			<u>byte offset #</u>
45	Stream Index	- 1 byte (hex/binary value)	0
	Client Index	- 1 byte (hex/binary value)	1
	Total Fed Count	- 4 bytes (long integer)	2
	Total Reject Count	- 4 bytes (long integer)	6

File CLIENT1.DAT contains a variable number of records, depending on the number of unique clients and mailstreams used. The records in this file are sorted in ascending order. A sortation is performed by first sorting the client index number followed by the stream index number.

The following discussion describes other steps executed by the CPU 54 in connection with a plurality of functions included in the program ANALYZE__MAIL.

Function ASSIGN__PACKAGES

The function ASSIGN_PACKAGES (see Fig. 4A) takes records from file COUNT.DAT and assigns package types to each zip code. The package types are reflected by the numbers "5" and "3" and the letters "S" and "M". A package type of "5" refers to a five digit package; a package type of "3" refers to a three digit package; a package type of "M" refers to mixed states package; a package type of "S" refers to a state package.

During the first pass through the records in the file COUNT.DAT, the function ASSIGN_PACKAGES initially examines each record in the file COUNT.DAT in the record order noted above in order to determine potential three digit packages (i.e., packages wherein all the mailpieces have identical first three ZIP code digits, but which do not qualify as five digit packages) [reflected by step 250] and potential state packages [reflected by step 252]. In this regard, the function ASSIGN_PACKAGES examines the field ZIP CODE for the current record to determine whether the record has the same ZIP code as did the next previous record. (Consecutive records might have the same ZIP codes when the same ZIP codes are present in different mailstreams). If the ZIP code for the currently examined record is the same as the previous record, the function ASSIGN_PACKAGES determines the total number of mailpieces represented by the record by adding the values in the "5 Digit", "ZIP + 4", and "ZIP + 4 Barcoded" fields of the record and adds that sum to a running total mailpiece counter (in location CNT) for this ZIP code.

If the ZIP code for the currently examined record differs from the previous record, the function ASSIGN_PACKAGES momentarily lays aside the current record to process the mailpiece count for the previous ZIP code (i.e., the ZIP code for the previous record). In this respect, the function ASSIGN_PACKAGES determines whether the number of mailpieces for the previous ZIP code was less than the predetermined minimum bundle size. If it was, the function ASSIGN_PACKAGES realizes that the previous ZIP code might qualify for a three digit package. To do this, the function ASSIGN_PACKAGES determines to what 3 Digit ZIP code the previous ZIP code belongs (i.e., determines the first three digits of the previous ZIP code).

The function ASSIGN_PACKAGES then assigns the total count of the number of mailpieces for the previous ZIP code (stored at location CNT) to a three digit package counter for the 3 Digit ZIP code to which the previous ZIP code belongs (i.e., to counter THREE_DIGIT_PACK_CNT[3_DIGIT_ZIP]).

After processing the mailpiece count for a previous ZIP code in the manner described above, the function ASSIGN_PACKAGES returns to processing the ZIP code for the current record. At this juncture, the function ASSIGN_PACKAGES assigns to the mailpiece counter CNT the sum of the values in the "5 Digit", "ZIP + 4", and "ZIP + 4 Barcoded" fields of the current record. The function ASSIGN_PACKAGES then examines the next record in the file COUNT.DAT, and continues the process described above for each such record until an end of file is encountered.

After determining the potential three digit packages at step 250, the function ASSIGN_PACKAGES attempts to locate potential state packages (step 252). In this regard, the function ASSIGN_PACKAGES compares the number of mailpieces assigned to each 3 Digit ZIP code with the predetermined minimum number of mailpieces necessary to make up a package (e.g., 10). If the actual number of counted mailpieces for an 3 Digit ZIP code is less than the predetermined minimum, the function ASSIGN_PACKAGES determines with which state the 3 Digit ZIP code is associated, and increments a state package counter for that state (i.e., STATE_PACK_CNT(i); where i = a number corresponding to the associated state). Also, if the number of counted mailpieces for an 3 Digit ZIP code is less than the predetermined minimum, the corresponding three digit package counter (THREE_DIGIT_PACK_CNT) is set to zero.

Having determined potential three digit packages and potential state packages in the manner described above (steps 250 and 252), the function ASSIGN_PACKAGES then conducts a second pass through the file COUNT.DAT in order to create a temporary file SACK1.TMP (step 254). As will be seen later, the file SACK1.TMP is used by function ASSIGN_SACKS to generate a file SACK2.TMP. The file SACK1.TMP contains multiple records of the following structure:

```

50 struct PACKAGE
    {
        unsigned long ZIP_ID;           /* zip identifier */
        unsigned long CNT;              /* count */
        char PTYPE;                    /* package type */
55 unsigned char BIN;                 /* bin assignment */
    }

```

Thus, the file SACK1.TMP is much reduced in size from the file COUNT.DAT since the multiple zip_id entries due to client/mailstream selections are combined. Also, the three different count categories ("ZIP + 4"; "5 Digit"; and "ZIP + 4 Barcoded") are combined into one count value (CNT) since the sortation of mail only depends on the combined count. The file SACK1.TMP is created in ZIP code order.

As mentioned above, the function ASSIGN__PACKAGES creates the file SACK1.TMP during the second pass through the file COUNT.DAT (step 254). During the second pass through file COUNT.DAT the function ASSIGN__PACKAGES examines each record of file COUNT.DAT in a manner similar to the first pass. The function ASSIGN__PACKAGES determines whether the ZIP code for each currently examined record in file COUNT.DAT is the same as for the previous record. If the ZIP code is the same, the function ASSIGN__PACKAGES determines the total number of mailpieces represented by the record and adds that number to a cumulative counter (CNT) for the total number of mailpieces for that ZIP code.

When, during the second pass through file COUNT.DAT, the function ASSIGN__PACKAGES encounters a record having a ZIP code which differs from the previous ZIP code (i.e., the ZIP code of the previous record), the function ASSIGN__PACKAGES prepares a record for file SACK1.TMP for the previous ZIP code. In this respect, in creating the record, the previous ZIP code is stored in the ZIP_ID field of the structure PACKAGE and the current value of the counter CNT is stored in the CNT field of the structure PACKAGE. A value for the bin assignment (BIN) for the mailpieces for this package is obtained from the corresponding "bin" field from the record in the COUNT.DAT file. The value for the package type (PTYPE) is determined as follows.

In determining the package type (PTYPE) for a package of mailpieces, the function ASSIGN__PACKAGES determines whether the total number of mailpieces for the ZIP code (stored in location CNT) exceeds the predetermined minimum package size (e.g., 10). If the predetermined minimum is equalled or exceeded, the function ASSIGN__PACKAGES assigns a "5" value to the PTYPE field for the record in the structure PACKAGE associated with this ZIP code. The "5" value in the PTYPE field is indicative of the fact that the package is a "5 Digit Package", meaning that all the mailpieces in this package have the identical first five digit ZIP codes.

If the total number of mailpieces for the ZIP code is less than the predetermined minimum package size, the function ASSIGN__PACKAGES determines to what 3 Digit ZIP code this ZIP code belongs. Then the function ASSIGN__PACKAGES determines whether the three digit package counter (THREE_DIGIT_PACK_CNT[3_DIGIT_ZIP]) for the 3 Digit ZIP code equals or exceeds the predetermined minimum package size. If (THREE_DIGIT_PACK_CNT[3_DIGIT_ZIP]) for the 3 Digit ZIP code equals or exceeds the predetermined minimum package size, then the function ASSIGN__PACKAGES assigns a "3" value to the PTYPE field for the record in the structure PACKAGE.

If the total number of mailpieces for the 3 Digit ZIP code is less than the predetermined minimum package size, the function ASSIGN__PACKAGES determines to what state this 3 Digit ZIP code belongs. Then the function ASSIGN__PACKAGES determines whether the state package counter (STATE_PACK_CNT) for this state equals or exceeds the predetermined minimum package size. If the state package counter for this state equals or exceeds the predetermined minimum package size, then the function ASSIGN__PACKAGES assigns a "S" value to the PTYPE field for the record in the structure PACKAGE.

If the function ASSIGN__PACKAGES cannot assign a "5", "3", or "S" value to the PTYPE field for this record in the SACK1.TMP file, a "M" value (indicative of "Mixed State Packages") is assigned to the PTYPE field.

Function ASSIGN__SACKS

Function ASSIGN__SACKS (see Fig. 4A) uses the file SACK1.TMP created by function ASSIGN__PACKAGES to generate another file (file SACK2.TMP). In so doing, function ASSIGN__SACKS makes tentative sack assignments.

The file SACK2.TMP is a temporary work file for making sack assignments to ZIP codes. Once created by function ASSIGN__SACKS, the file SACK2.TMP contains multiple records of the following structure:

```

    struct SACK
    {
        struct PACKAGE package;
        int NO;
        char STYPE
    }

```

5

10 where the structure PACKAGE is as formerly defined.

There are five possible sack assignments: FIVE__DIGIT; MIXED__FIVE; THREE__DIGIT; STATE; and MIXED__STATES.

The function ASSIGN__SACKS reads through the file SACK1.TMP in two passes. During the first pass, the function ASSIGN__SACKS obtains certain sack counts. During the second pass, the function

15

ASSIGN__SACKS generates the file SACK2.TMP.

During the first pass through the file SACK1.TMP, function ASSIGN__SACKS determines the 3 Digit ZIP code and the package type (PTYPE) assigned to each record in the file SACK1.TMP. If the package type (PTYPE) of a record is "5", the package count (CNT) for that package is checked to determine whether the package contains enough mailpieces to be its own sack (step 260). This is done by comparing the package

20

count (CNT) value of the package to a predetermined minimum mailpiece number necessary to make up a sack ("MIN__MAIL"). If the package count (CNT) does not qualify the package to be its own sack, then the function ASSIGN__SACKS realizes that this "5" type package may be part of a MIXED__FIVE; THREE__DIGIT; STATE; or MIXED__STATES sack. Accordingly, for the 3 Digit ZIP code corresponding to the ZIP code for the current package, the function ASSIGN__SACKS adds the package count (CNT) to a

25

"mixed five sack counter" (MIXED__FIVE__SACK__CNT[3__DIGIT__ZIP]) [step 262].

For packages belonging to the other types (PTYPE = 3, S, or M), the function ASSIGN__SACKS adds the package count (CNT) to the appropriate counter (step 262). For example, for a record in SACK1.TMP having a PTYPE = 3, the function ASSIGN__SACKS adds the package count (CNT) of that record to a

30

"three digit sack counter" (THREE__DIGIT__SACK__CNT[3__DIGIT__ZIP]) for the 3 Digit ZIP code for the ZIP code of the record. Similarly, a unique counter exists for each state (STATE__SACK__CNT) and each mixed state (MS__SACK__CNT).

The function ASSIGN__SACKS then checks the potential mixed five, three digit, and state sack counts (step 264). First, for each 3 Digit ZIP code, the function ASSIGN__SACKS checks the value of the counter MIXED__FIVE__SACK__CNT to determine whether each potential mixed five sack has the predetermined

35

minimum number of mailpieces to be a sack. If the counter MIXED__FIVE__SACK__CNT falls short of the predetermined minimum number, then it is assumed for the moment that these mailpieces, previously thought to comprise a mixed five sack, should now comprise a three digit sack. To this end, the value of the counter MIXED__FIVE__SACK__CNT[3__DIGIT__ZIP] for this 3 Digit ZIP code is added to the counter THREE__DIGIT__SACK__CNT[3__DIGIT__ZIP] for the 3 Digit ZIP code. The value of the counter

40

MIXED__FIVE__SACK__CNT for this 3 Digit ZIP code is then re-initialized at zero.

In the same manner the function ASSIGN__SACKS checks the number of mailpieces in each potential three digit sack to insure that the potential sack has the predetermined minimum number of mailpieces to qualify as a sack. If the value of THREE__DIGIT__SACK__CNT[3__DIGIT__ZIP] does not have the predetermined minimum number of mailpieces, it is assumed that these mailpieces should now be part of a state

45

sack. Accordingly, a state sack counter (STATE__SACK__CNT) for the state having the concerned 3 Digit ZIP code is incremented by the value of the THREE__DIGIT__SACK__CNT[3__DIGIT__ZIP], and the value of the counter THREE__DIGIT__SACK__CNT[3__DIGIT__ZIP] is reinitialized at zero.

In the same manner the function ASSIGN__SACKS checks the number of mailpieces in each potential state sack to insure that the potential state sack has the predetermined minimum number of mailpieces to

50

qualify as a sack. If the value of STATE__SACK__CNT for the state does not have the predetermined minimum number of mailpieces, it is assumed that these mailpieces should now be part of a mixed state sack. Accordingly, a mixed state sack counter (MS__SACK__CNT) is incremented by the value of the STATE__SACK__CNT for the affected state, and the value of the counter STATE__SACK__CNT for the affected state is reinitialized at zero.

55

During the second pass through the file SACK1.TMP the function ASSIGN__SACKS uses the data in the file SACK1.TMP and the various sack counters described above (MIXED__FIVE__SACK__CNT; THREE__DIGIT__SACK__CNT; STATE__SACK__CNT; and, MS__SACK__CNT) to create the new file SACK2.TMP (step 266). As the function ASSIGN__SACKS reads through each record in file SACK1.TMP,

the function ASSIGN__SACKS again examines the package type (PTYPE) and obtains the 3 Digit ZIP Code corresponding to the ZIP code stored in the record. For records in SACK1.TMP having a record type "5" (i.e., PTYPE = 5) and a package count (CNT) exceeding the predetermined minimum number required to form sack, a record is created in file SACK2.TMP having a five digit sack type value (SType = FIVE__DIGIT).

If a record has a record type "5" but does not represent the predetermined minimum number of mailpieces for a FIVE__DIGIT sack, the associated count is compared to see if it exceeds the minimum sack count for the next lowest priority sack type (i.e., sack type MIXED__FIVE) to which it belongs. If it does, a record is created in file SACK2.TMP having a mixed five sack type value (SType = MIXED__FIVE). If the record does not represent the minimum sack count, the same type process is repeated for the remaining lower priority sacks in the order THREE__DIGIT, followed by STATE. If the record does not meet the minimum number for any of the above sack types, it is assigned a MIXED__STATES sack type.

For records in SACK1.TMP having a package record type of "3", a similar count comparison is performed as above. However, the check begins with the THREE__DIGIT sack count level so only THREE__DIGIT, STATE, and MIXED__STATES assignments can be made.

For records in SACK1.TMP having a package record type of "S" a similar count comparison is performed as above. However, the check begins with the STATE sack count level so that only STATE and MIXED__STATES assignments can be made.

Finally, all records having package type "M" are assigned a MIXED__STATES sack type.

Function FIRST__PASS__PACKS

The function FIRST__PASS__PACKS (see Fig. 4B) determines which bins, as a result of the first sort pass, consist of completely sorted packages. In this regard, function FIRST__PASS__PACKS prepares an array FULLSORT__BIN having elements corresponding to each of the 128 bins included in the sorter. As a result of the execution of function FIRST__PASS__PACKS, bins which do not contain a fully sorted package as a result of the first sort pass have a zero value stored in their corresponding element in array FULLSORT__BIN. For example, if bin 26₃ does not include a fully sorted package, a "zero" value is stored in FULLSORT__BIN[3]. For any bin consisting of fully sorted packages, a unique non-zero number is stored in the element in array FULLSORT__BIN corresponding to that bin number.

In the simple case (reflected by step 270), the determination of function FIRST__PASS__PACKS is made by comparing the number of mailpieces in a bin with the sum of the number of mailpieces included in all the packages in the bin. If the number of mailpieces in a bin equals the sum of the number of mailpieces included in all the packages in the bin, then the bin consists of fully sorted packages. For any bin consisting of fully sorted packages, a unique non-zero number is stored in the element in array FULLSORT__BIN corresponding to that bin number.

For three digit and state packages the comparison is complicated by the fact that completely sorted package contents could end up in several first pass bins. Therefore, for three digit and state packages, the partial package counts from several bins making up one complete package are compared against the same several bins total piece count.

In order to process the three digit and state packages, the function FIRST__PASS__PACKS reads through every record in the file SACK2.TMP (which was created by function ASSIGN__SACKS) and creates two additional work files, i.e., STATE__PACK__FILE and THREE__PACK__FILE (step 272). For every record encountered in file SACK2.TMP that concerns a state package (PTYPE = S), the function FIRST__PASS__PACKS duplicates that record in the file STATE__PACK__FILE. Likewise, for every record encountered in file SACK2.TMP that concerns a three digit package (PTYPE = 3), the function FIRST__PASS__PACKS duplicates that record in the file THREE__PACK__FILE.

To determine whether a potentially completely sorted state package is spread through a plurality of bins (step 274), for each state the function FIRST__PASS__PACKS determines what records in the file STATE__PACKS__FILE have ZIP codes belonging to that state. When a record in file STATE__PACKS__FILE pertains to the state, the function FIRST__PASS__PACKS sets a flag in an element of an array BIN__USAGE corresponding to the bin number indicated in the record. For example, if bins 26₇, 26₅₅, and 26₉₂ all have mailpieces belonging to a completely sorted state package, flags are set at BIN__USAGE[7], BIN__USAGE[55], and BIN__USAGE[92].

Having noted the bins in which completely sorted state packages may reside, the function FIRST__PASS__PACKS then compares the total number of mailpieces in the state package with the total number of mailpieces in all the bins for which a flag was set in array BIN__USAGE for the state of interest.

Recall that the number of mailpieces for each bin 26 is obtainable from the file AGGR.DAT which was created during the first sort pass (i.e., at step 202).

If the total number of mailpieces in the state package is equal to the total number of mailpieces in all the bins for which a flag was set in array BIN_USAGE, then those bins are known to include mailpieces for the fully sorted state package. For the bins including mailpieces for the fully sorted state package, the same non-zero number is placed in the elements of array FULLSORT_BIN corresponding to those bins. The non-zero number placed in each of the elements of array FULLSORT_BIN for the state is unique number which does not appear in array FULLSORT_BIN for any other state or any other purpose.

It should be understood that the foregoing processing related to state packages is conducted separately for each state. This requires that certain parameters, including the array BIN_USAGE, be reinitialized for each state. Likewise, whatever number entered into one or more elements of the array FULLSORT_BIN for a particular state will be a number unique to that state.

The function FIRST_PASS_PACKS also determines whether fully sorted three digit packages are spread through more than one bin (step 276). This determination is made in a similar manner as was the determination for state packages. That is, for each 3 Digit ZIP Code value the function FIRST_PASS_PACKS reads through the file THREE_PACK_FILE which it created, and determines whether mailpieces belonging to that 3 Digit ZIP Code are in a plurality of bins. If so, the function FIRST_PASS_PACKS sets flags in array BIN_USAGE in the same manner as with the state packages. Then, in like manner as with the state packages, the function FIRST_PASS_PACKS determines whether the total mailpiece count of the particular three digit package equals the sum of the bin counts for each of the bins in which the three digit package is spread. If an equality is determined, then function FIRST_PASS_PACKS realizes that the bins for which flags were set in array BIN_USAGE contain the completely sorted three digit package. As with the completely sorted state packages, a unique number associate with this three digit package is assigned to each element in array FULLSORT_BIN corresponding to the bins wherein mailpieces belonging to this completely sorted three digit package reside.

Thus, upon completion of the execution of function FIRST_PASS_PACKS, an example of the contents of a portion of array FULLSORT_BIN might be as follows:

```

30          FULLSORT_BIN[001] = 0
          FULLSORT_BIN[002] = 1
          FULLSORT_BIN[003] = 0
          FULLSORT_BIN[004] = 2
          FULLSORT_BIN[005] = 2
35          FULLSORT_BIN[006] = 3
          FULLSORT_BIN[007] = 3
              .
              .
              .
40          FULLSORT_BIN[127] = 0
          FULLSORT_BIN[128] = 0

```

Where a FULLSORT_BIN value of "0" indicates that the bin does not contain a completely sorted package; a value of "1" indicates that a first fully sorted package is contained in bin 26₁; a value of "2" indicates that a second fully sorted package (perhaps a state package) is contained in bins 26₄ and 26₅; and, a value of "3" indicates that a third fully sorted package (perhaps a three digit package) is contained in bins 26₆ and 26₇.

50 **Function MAKE_SORT_RECORDS**

The function MAKE_SORT_RECORDS (see Fig. 4B) makes a file (file SORTREC1.TMP) which lists packages for second pass sorting and makes another file (file FULLSORT1.TMP) which lists packages that will not be sorted in a subsequent pass. The function MAKE_SORT_RECORDS creates the files SORTREC1.TMP and FULLSORT1.TMP after reading through the file SACK2.TMP (which was created by function ASSIGN_SACKS).

In reading each record from the file SACK2.TMP, the function MAKE_SORT_RECORDS determines to which type of package the record relates. In particular, the function MAKE_SORT_RECORDS checks to

see if the PTYPE for the record is either "5", "3", or "S".

For a record in file SACK2.TMP having a PTYPE of "5", corresponding to a five digit package, the function MAKE_SORT_RECORDS duplicates the record either in file SORTREC1.TMP or file FULLSORT1.TMP (step 280). To determine to which file to duplicate the record, the function
 5 MAKE_SORT_RECORDS further checks to determine whether this five digit package is in a fully sorted bin. This further check is conducted by noting the bin number included in the record, and then indexing into the array FULLSORT_BIN for that bin number. If the array FULLSORT_BIN contains a non-zero value for that bin, the routine ASSIGN_PACKAGES concludes that the record is fully sorted and duplicates the record from SACK2.TMP in the file FULLSORT1.TMP. Otherwise the function MAKE_SORT_RECORDS
 10 duplicates the record from SACK2.TMP in the file SORTREC1.TMP.

If the function MAKE_SORT_RECORDS determines that the PTYPE for a record in file SACK2.TMP is "3", corresponding to a three digit package, the function MAKE_SORT_RECORDS determines the 3 Digit ZIP Code to which the record pertains. Then the function MAKE_SORT_RECORDS notes the bin number stored in the record, and stores that bin number in an element of an array THREE_DIG_BIN corresponding to the pertinent 3 Digit ZIP Code (step 282). Likewise, the function MAKE_SORT_RECORDS notes
 15 from the record the sack type assignment (from STYPE), and stores that sack type in an element of an array THREE_DIGIT_SACK_TYPE corresponding to the pertinent 3 Digit ZIP Code (step 284).

If the function MAKE_SORT_RECORDS determines that the PTYPE for a record in file SACK2.TMP is "S", corresponding to a state package, the function MAKE_SORT_RECORDS determines the state to which the record pertains. Then the function MAKE_SORT_RECORDS notes the bin number stored in the record, and stores that bin number in an element of an array STATE_BIN corresponding to the pertinent state (step 286). Likewise, the function MAKE_SORT_RECORDS notes from the record the sack type assignment (from STYPE), and stores that sack type in an element of an array STATE_SACK_TYPE corresponding to the pertinent state (288).
 20

After reading all the records in file SACK2.TMP, after storing information in the arrays THREE_DIG_BIN and THREE_DIGIT_SACK_TYPE for three digit package records, and after storing information in arrays STATE_BIN and STATE_SACK_TYPE for state package records, the function MAKE_SORT_RECORDS is prepared to complete the writing of the two output files SORTREC1.TMP and FULLSORT1.TMP. The function MAKE_SORT_RECORDS first writes three digit packages to the appropriate one of the two output files (step 290), and then the state packages to the appropriate one of the two output files (292), with the result that the two output files are sorted first by package type, and then within each package type by package ZIP.
 25

In writing the three digit packages to the appropriate file (file FULLSORT1.TMP or file SORTREC1.TMP) at step 290, the routine MAKE_SORT_RECORDS checks to determine which elements of array
 35 THREE_DIGIT_PACK_CNT (generated by the function ASSIGN_PACKAGES), i.e. which 3 Digit ZIP Codes, have non-zero values, and write a record to the appropriate file only for those 3 Digit ZIP Codes. Similarly, the routine MAKE_SORT_RECORDS writes the state packages to the appropriate file (file FULLSORT1.TMP or file SORTREC1.TMP) at step 292 only for those states having a number of mailpieces exceeding the predetermined minimum bundles size.

The formats for file FULLSORT1.TMP and SORTREC1.TMP are identical. In particular, the formats are both files are prescribed by the structure SACK, which was defined above in connection with the discussion of function ASSIGN_SACKS as including the structure PACKAGE (which, in turn, was defined above in connection with the discussion of the function ASSIGN_PACKAGES). But in creating these two files, the routine ASSIGN_PACKAGES must store the proper information in the structure PACKAGE portion of the structure SACK, particularly the fields for ZIP_ID, CNT and BIN. It should be noted that only one record is written to a 3 Digit ZIP code, e.g., 60202 is written to ZIP_ID 602. Furthermore, only one record per state is written to FULLSOR1.TMP and SORTREC1.TMP.
 40

For each three digit package, the function MAKE_SORT_RECORDS obtains the information for fields CNT and BIN from the corresponding elements in the respective arrays THREE_DIGIT_PACK_CNT and
 50 THREE_DIGIT_BIN. Recall that array THREE_DIGIT_BIN was generated by the function MAKE_SORT_RECORDS and that the array THREE_DIGIT_PACK_CNT was generated by the function ASSIGN_PACKAGES. To determine whether a record should be written to file FULLSORT1.TMP or to file SORTREC1.TMP, the function MAKE_SORT_RECORDS checks the status of array FULLSORT_BIN for the bin whose number is stored in the array THREE_DIGIT_BIN for the pertinent 3 Digit ZIP Code. If the value stored in array FULLSORT_BIN is non-zero, then a record is written to file FULLSORT1.TMP.
 55 Otherwise a record is written to file SORTREC1.TMP.

For each state package, the function MAKE_SORT_RECORDS obtains the information for fields CNT and BIN from the corresponding elements in the respective arrays STATE_PACK_CNT and STATE_BIN.

Recall that array STATE__BIN was generated by the function MAKE__SORT__RECORDS and that the array STATE__PACK__CNT was generated by the function ASSIGN__PACKAGES. To determine whether a record should be written to file FULLSORT1.TMP or to file SORTREC1.TMP, the function MAKE__SORT__RECORDS checks the status of array FULLSORT__BIN for the state. If the value stored in array FULLSORT__BIN for the pertinent state is non-zero, then a record is written to file FULLSORT1.TMP. Otherwise a record is written to file SORTREC1.TMP.

Function SACK__SORT

The function SACK__SORT (see Fig. 4C) creates a file SORTREC2.TMP using file SORTREC1.TMP (step 300). Each record in file SORTREC2.TMP has the format of the structure SACK described above.

The function SACK__SORT sorts the records in file SORTREC1.TMP by sack type (step 302), then within sack type by package type (step 304), and within package type by package ZIP id (step 306). The resultant sort creates the file SORTREC2.TMP. At the end of execution of the function SACK__SORT, mail is not yet in the final "sack and bag" order, since further sorting is required by major tree as described below.

Function SACK__SORT begins by scanning the file SORTREC1.TMP and writing all FIVE__DIBIT sack package entries to the output file SORTREC2.TMP. While scanning the input file, the count for each package type entry is recorded. Since the file SORTREC1.TMP has been sorted by package type, the package type counts are used to determine the starting position of each different package type's entries.

Next, the file SORTREC1.TMP is scanned from the beginning and all MIXED__FIVE SACK package entries are appended to the output file.

Since only one package type could go into FIVE__DIGIT and MIXED__FIVE sacks, it was a simple matter of copying sequential records marked with the appropriate sack type to the output file. For the remaining sack types (THREE__DIGIT, STATE, and MIXED__STATES) multiple package types are allowed. Therefore, much searching is required to find the appropriate next entry for the output file.

The THREE__DIGIT sack can be composed of both FIVE__DIGIT and THREE__DIGIT packages. The first step is to determine if any entries exist for all possible package types. Then the function SACK__SORT repeatedly determines which of the existing package entries of sack type THREE__DIGIT (which are located in the input file by the starting position previously saved and count values of records already processed) corresponds to the first THREE__DIGIT sack entry and the appropriate record is appended to the output file with the ZIP__ID corresponding to the three digit ZIP of the sack. In addition, the processed record count for the selected package type is incremented. This is done until all package entries have been processed.

The MIXED__STATES sack can be composed of both FIVE__DIGIT, THREE__DIGIT, STATE, and MIXED__STATE packages. The first step is to determine if any entries exist for all possible package types. Since there is only one MIXED__STATES sack, packages are written in package priority order until all entries are exhausted.

Upon completion of the execution of function SACK__SORT, the ZIP__ID field in a record in file SORTREC2.TMP is the package ZIP id, which is no longer necessarily the long integer value of the full 5 digit zip code with four zeros trailing in the "+ 4" position. The ZIP__ID for non-5 digit package mail may be the 3 digit zip (range 0 - 999) for 3 digit packages and a state index number (range 0 - 99) for state packages. The BAG field of the records in file SORTREC2.TMP are undefined at this point, since further processing is necessary to determine the appropriate values for this field.

Function MAKE__COMBOS

The function MAKE__COMBOS (Fig. 4D) determines mandatory combinations of first pass bins for making up state packages and three digit packages. Function MAKE__COMBOS sets up the required data in an array BIN__COMBOS to force mailpieces scattered across numerous bins by ineffective first pass sort schemes back into a single second pass group. Bins fully sorted on the first sort pass are pulled out since they are not part of second pass records.

The function MAKE__COMBOS first handles state packages. For each state at step 310 the function MAKE__COMBOS reads each record in the STATE__PACK__FILE (which was created by function FIRST__PASS__PACKS) and determines (by reference to array FULLSORT__BIN) whether the particular bin number contained in the record is a fully sorted bin (step 312). If the bin number for that record is fully sorted and the record belongs to the current state, the function MAKE__COMBOS goes on to read the next record in the array STATE__PACK__FILE (i.e., to step 310). If the bin is not fully sorted, the function MAKE__COMBOS sets a flag in an element of array BIN__USAGE corresponding to that bin, thereby

indicating that the bin has mailpieces for the current state (step 314).

After the entire file STATE_PACK_FILE has been read for a particular state (determined at step 316), at step 318 the function MAKE_COMBOS calls another function, function CHECK_COMBOS, which actually sets up the data in array BIN_COMBOS giving consideration to possible conflicting bin assignments. A detailed discussion of the function CHECK_COMBOS is provided below.

After the function CHECK_COMBOS has been called for a particular state, as indicated by the affirmative result at step 320 the function MAKE_COMBOS moves on the next state and repeats the afore-described state package handling for that next state, including a call to function CHECK_COMBOS after reading through the entire STATE_PACK_FILE for that next state. The function MAKE_COMBOS conducts the afore-described state package handling procedure for each state.

The function MAKE_COMBOS executes much the same procedure for the three digit packages as it did for the state packages. In handling the three digit packages, at step 330 the function MAKE_COMBOS reads the first record in the file THREE_DIGIT_PACK_FILE (which was created by function FIRST_PASS_PACKS). As reflected by step 332 for example, in handling the state packages, the function MAKE_COMBOS ignores any records in the file THREE_DIGIT_PACK_FILE which pertain to fully sorted bins (determined by reference to array FULLSORT_BINS).

At step 334 the function MAKE_COMBOS obtains the first three digit ZIP code from the first record in the file THREE_DIGIT_PACK_FILE and stores that first ZIP code in a location THREE_DIG_ZIP. At step 336 the function MAKE_COMBOS determines the bin number contained in that record; and, sets a flag in the element of array BIN_USAGE corresponding to that bin number.

Having processed the first record in file THREE_DIGIT_PACK_FILE, the function MAKE_COMBOS then reads through further records in the file THREE_DIGIT_PACK_FILE (as reflected by step 340), noting the ZIP code stored in the record and storing the first three digits of the ZIP code in a location SCF (step 342). At step 344 the function MAKE_COMBOS checks to determine if the value in location SCF (the ZIP code for the most recently read record) is the same as the value in THREE_DIGIT_ZIP (see the preceding paragraph). If so, the function MAKE_COMBOS (1) at step 346 sets a flag in an element of the array BIN_USAGE corresponding to the bin number included in the most recently read record from file THREE_DIGIT_PACK_FILE (thereby indicating that the three digit ZIP code has mailpieces in that bin as well), and (2) goes on to read the next record in file THREE_DIGIT_PACK_FILE (i.e., returns to step 340). If the next record in file THREE_DIGIT_PACK_FILE is read at this point, the function MAKE_COMBOS repeats the steps described in this paragraph with respect to that next record.

If the function MAKE_COMBOS determines at step 344 that the value in location SCF (the ZIP code for the most recently read record) is not the same as the value in THREE_DIGIT_ZIP, the function MAKE_COMBOS concludes its processing of the ZIP code whose value is stored in location THREE_DIGIT_ZIP by: (1) calling function CHECK_COMBOS (described below) at step 348 to set appropriate values in the array BIN_COMBO; (2) setting the value in location THREE_DIGIT_ZIP equal to the value in location SCF (at step 350); (3) at step 352 setting a flag in an element of the array BIN_USAGE corresponding to the bin number included in the most recently read record from file THREE_DIGIT_PACK_FILE; and, (4) repeating the steps of the preceding paragraph for the next record in file THREE_DIGIT_PACK_FILE (e.g., by returning to step 340). The processing of three digit packages continues in this manner until all three digit packages have been processed (step 354), after which the function MAKE_SACK_COMBOS is called (step 356).

Function CHECK_COMBOS

The function CHECK_COMBOS is called by the function MAKE_COMBOS (described above) in order to resolve any conflicting bin assignments and to store data in the array BIN_COMBOS. The routine MAKE_COMBOS calls the function CHECK_COMBOS as the routine MAKE_COMBOS finishes with each state, and as the routine MAKE_COMBOS finishes with each three digit ZIP value.

As indicated above, the function CHECK_COMBOS stores values in the array BIN_COMBO to indicate which first pass bins are to be mandatorily combined together. As an example of how the array BIN_COMBOS might appear upon completion of the execution of function CHECK_COMBOS, consider the following:


```

BIN_COMBO[001] = 0
BIN_COMBO[002] = 0
BIN_COMBO[003] = 1
BIN_COMBO[004] = 0
BIN_COMBO[005] = 2
BIN_COMBO[006] = 1
BIN_COMBO[007] = 2
BIN_COMBO[008] = 0
.
.
BIN_COMBO[127] = 0
BIN_COMBO[128] = 0

```

15 where a zero ("0") assignment indicates that there is no forced combination for a bin; where bins assigned a "1" are to be forced into combination (i.e., bins 3 and 6); and, where bins assigned a "2" are to be forced into combination (i.e., bins 5 and 7). Thus, each forced combination has a unique combination number associated therewith, and the bins forced into combination together have the same combination number assigned to their corresponding elements in the array BIN_COMBOS.

20 When function CHECK_COMBOS is called, function CHECK_COMBOS initially executes two steps: (1) initializes a counter COLLISION_CNT; and, (2) counts the number of bins for which a flag has been set in array BIN_USAGE for the current package. If the number of flags set is only one, then function CHECK_COMBOS knows that no combination of bins is required and returns control to the calling function MAKE_COMBOS.

25 Assuming that the function CHECK_COMBOS does not immediately return control to the calling function MAKE_COMBOS, the function CHECK_COMBOS checks to determine whether any of the bins for which a flag was set in array BIN_USAGE has already been forced into a combination. This check is implemented by checking the element in array BIN_COMBO corresponding to that bin to determine if a non-zero combination number has already been assigned to the bin. If a non-zero combination number has already been assigned, the function CHECK_COMBOS notes a "collision".

30 The function CHECK_COMBOS counts the number of collisions detected using a counter COLLISION_CNT. In addition, the function CHECK_COMBOS stores the numbers of the bins subject to collision in an array COLLISION_COMBO. For example, if a first collision occurred in bin 67 and a second collision occurred in bin 48, COLLISION_COMBO[1] = 34 and COLLISION_COMBO[2] = 48.

35 Having searched for collisions, the function CHECK_COMBOS executes different steps in determining the combination number to be stored in BIN_COMBOS, depending upon whether no collisions, one collision, or multiple collisions were detected.

40 If no collisions were encountered during the execution of function CHECK_COMBOS, the function CHECK_COMBOS would then select the next available number for use as a combination number. For example, in the example, if the highest number thus far stored in the array BIN_COMBOS was "2", the function CHECK_COMBOS would then store a "3" in every element in array BIN_COMBOS for which a flag was set in a corresponding element of array BIN_USAGE. The function CHECK_COMBOS would then return control to the calling function MAKE_COMBOS.

45 If only one collision were encountered during the execution of function CHECK_COMBOS, the function CHECK_COMBOS would assign the combination number stored in the first element of COLLISION_COMBO to the elements in array BIN_COMBOS corresponding to the elements in array BIN_USAGE having flags set. For example, if bins 67 and 102 were utilized for the current package as indicated by flags being set in the 67th and 102nd elements of array BIN_USAGE, if the only collision detected by function CHECK_COMBOS for the current package occurred with respect to bin 67, and if the combination number assigned to bin 67 were "5", then the function CHECK_COMBOS would assign the combination number "5" to bins 67 and 102 (i.e., to BIN_COMBO[67] and BIN_COMBO[102]). The function CHECK_COMBOS would then return control to the calling function MAKE_COMBOS.

55 In the more complex case where multiple collisions are encountered, the function CHECK_COMBOS replaces all the elements of array BIN_COMBO affected by the collision with the combination number of the first detected collision, i.e., all collision bins are assigned the value COLLISION_COMBO[0] and are properly placed in the BIN_COMBO array. Then the combination number in COLLISION_COMBO[0] is assigned to any new bins that might not have collided. This is done by setting the array elements in BIN_COMBO to COLLISION_COMBO[0] using the corresponding elements marked in array BIN_USAGE.

Since the value in COLLISION_COMBO[0] may not have been the lowest forced combination indicator out of multiple such indicators, a numbering gap could possibly occur.

A numbering gap, if it occurs, is fixed by placing all non-zero elements from the BIN_COMBO array into another array UNIQUE_COLLISIONS along with counting the number of elements in array
 5 UNIQUE_COLLISIONS. All elements in the array UNIQUE_COLLISIONS are sorted in numerical order, then each element repeating the value of the previous element is removed from the array, thereby shrinking the array in size. The array UNIQUE_COLLISIONS makes it possible to easily reassign forced bin combination numbers without gaps to the BIN_COMBO array. This is accomplished by searching both the
 10 BIN_COMBO array and the UNIQUE_COLLISIONS array for matching contents. When the contents match, a new sequential number is assigned into yet another array COMBO_TRANSLATION corresponding to the position of the UNIQUE_COLLISIONS array. Finally, the BIN_COMBO array is updated by assigning the value in the COMBO_TRANSLATION array which is indexed by the contents of the BIN_COMBO array.

15 **Function MAKE_SACK_COMBOS**

The function MAKE_SACK_COMBOS (see Fig. 4E) determines mandatory combinations of first pass bins for forming sacks. That is, the function MAKE_SACK_COMBOS sets up the required data to force a sack's packages scattered across bins by the first pass sort scheme back into a single second pass group.

20 The function MAKE_SACK_COMBOS (which forces a sack's packages together) is somewhat simpler than the function MAKE_COMBOS (which determined combinations of first pass bins for making up state packages and three digit packages). The simplicity results from two factors. First, the function MAKE_COMBOS has already forced zip codes scattered across bins to form packages. Second, an input file (i.e., SORTREC2.TMP) of subsequent pass packages sorted by sack zip ID already exists. The file
 25 SORTREC2.TMP was created by the function SACK_SORT. The file SORTREC2.TMP is sorted by sack type, within sack type by package type, and within package type by package zip. Therefore, the file SORTREC2.TMP is effectively sorted by sack zip ID.

At step 370 the function MAKE_SACK_COMBOS reads the initial record in the file SORTREC2.TMP. At step 372 the sack ID value obtained from the first record is stored in location LAST_SACK_ZIP_ID. At
 30 step 374 the sack type value obtained from the first record is stored in location LAST_SACK_TYPE. At step 376 a flag is set in the element of array BIN_USAGE corresponding to the bin value obtained from the first record of file SORTREC2.TMP.

At step 378 the function MAKE_SACK_COMBOS begins a loop of reading and processing records in the file SORTREC2.TMP. The next record is read at step 378. If the record was not the last record (as
 35 determined at step 380), at step 382 the function MAKE_SACK_COMBOS checks whether the sack type value of the record indicates a "MIXED STATES" sack. There is no forced combination with respect to any MIXED STATES sacks, so that an affirmative result at step 382 results in the continuation of the loop with the reading of the next record at step 378. If the record does not indicate a MIXED STATES sack, the ZIP ID value from the record is stored at location SACK_ZIP_ID (step 384).

40 At step 386 the function MAKE_SACK_COMBOS determines whether the next record indicates a change of sack types from the previous record. A change of sack types occurs when the current records has two critical parameters that differ from the previous records. That is, when SACK_ZIP_ID is not equal to LAST_SACK_ZIP_ID, and when the sack type read from the current record is not equal to LAST_SACK_TYPE, a change of sack type has occurred.

45 When, at step 386, a change in sack type is determined not to have occurred, the function MAKE_SACK_COMBOS sets a flag in an element of the array BIN_USAGE corresponding to the bin value obtained from the current record in the SORTREC2.TMP file (step 388). The value in location SACK_ZIP_ID is then stored in location LAST_SACK_ZIP_ID (step 390), and a value indicative of the sack type obtained from the current record is stored in the location LAST_SACK_TYPE (step 392). The
 50 function MAKE_SACK_COMBOS then loops back to step 378 for the reading of another record from file SORTREC2.TMP.

When, at step 386, a change in sack type is determined to have occurred, at step 394 the function MAKE_SACK_COMBOS calls the function CHECK_COMBOS (described above) to update the array
 55 BIN_COMBOS. The function CHECK_COMBOS resolves conflicting bin assignments for sacks in essentially the same manner as described above in connection with the resolution of conflicting bin assignments for packages. After the call to function CHECK_COMBOS, at step 396 the function MAKE_SACK_COMBOS clears the array BIN_USAGE in anticipating of processing the next sack type. The function MAKE_SACK_COMBOS then sets a flag in an element of the array BIN_USAGE cor-

responding to the bin value obtained from the current record in the SORTREC2.TMP file (step 388). The value in location SACK_ZIP_ID is then stored in location LAST_SACK_ZIP_ID (step 390), and a value indicative of the sack type obtained from the current record is stored in the location LAST_SACK_TYPE (step 392). The function MAKE_SACK_COMBOS then loops back to step 378 for the reading of another

record from file SORTREC2.TMP.
After the last record is read from file SORTREC2.TMP (as determined at step 380), the function MAKE_SACK_COMBOS calls the function CHECK_COMBOS at step to update the array BIN_COMBOS with respect to the last record. Thereafter, as indicated by step 398, processing continues with the function BUILD_TREES, which is described immediately below.

Function BUILD_TREES

The function BUILD_TREES (see Figs. 4F and 4G) builds a major tree array, i.e. array MAJ_TREE, which associates particular bins with a sort tree. In addition, the function BUILD_TREES develops pointer information in a file SRTREE.DAT. The file SRTREE.DAT contains key pointer and offset information used to process individual subsequent pass groups within a properly ordered input file containing package information for multiple groups.

As an example of how the major trees built by function BUILD_TREES appear, after the execution of function BUILD_TREES, the array MAJ_TREE might have values such as the following:

```
MAJ_TREE[000] = 0 (bin 0 not used)
MAJ_TREE[001] = 0 (no major tree assignment)
MAJ_TREE[002] = 1 (first sort tree)
MAJ_TREE[003] = 2 (second sort tree)
MAJ_TREE[004] = 2 (second sort tree)
MAJ_TREE[005] = 0 (no major tree assignment)
MAJ_TREE[006] = 3 (third sort tree)
      .
      .
      .
MAJ_TREE[128] = 0 (no major tree assignment)
```

The function BUILD_TREES takes the number of different packages coming from a first pass bin or a required combination of bins and forms information for "groups" (including "groups" requiring even more passes). This is done by comparing the package count with the number of available bins. Several first pass bins may be combined together if the total package count is less than the number of bins available.

At the beginning of the execution of function BUILD_TREES, as reflected by step 400, the function BUILD_TREES determines the number of packages in each state/3-digit forced bin combination and stores that value in an array QUAL_PER_COMBO. This is done by determining which bins have the same value stored in the array BIN_COMBO (e.g., which bins are forced into combination), and summing the number of packages for all bins combined together. The number of packages in each bin is available from the array PACKS_PER_BIN, which was developed in the previous function MAKE_SORT_RECORDS.

At steps 402 - 408 the function BUILD_TREES initializes various parameters. At step 402 the counter CUTOFF_CNT is initialized at a value equaling the number of available bins. This initialization value may not be 128, since some bins may be designated as reject bins, or may not be used, or may be used for other purposes. At step 404 the counter TREE_CNT, which counts the number of sort trees, is initialized at 1. At steps 406 and 408, respectively, the counters QUAL_CNT and TEMP_CNT are initialized at zero.

Commencing at step 410 the function BUILD_TREES attempts to find the first bin belonging to a tree. In so doing, the function BUILD_TREES considers only bins having packages stored therein which are not fully sorted during the first pass, or bins which have been forced into combination (e.g., bins having a non-zero value in their corresponding element in array BIN_COMBO).

At step 412 the function BUILD_TREES determines whether the considered bin was forced into combination by checking for a non-zero value at the corresponding element in array BIN_COMBO. If the considered bin was involved in a forced combination, at step 414 the qualifying package counter QUAL_CNT has the value of the number of packages included in the combined bins added thereto. Where "i" stands for the bin under consideration, the number of packages included in the combined bins is

obtained from array QUAL_PER_COMBO[BIN_COMBO[i]]. Values were stored in array QUAL_PER_COMBO at step 400 of function BUILD_TREES. Also for the considered bin involved in the forced combination, for each bin included in the forced combination, at step 416 the function BUILD_TREES sets elements corresponding to each such bin in array MAJ_TREE equal to the value in counter TREE_CNT, which is presently "1". The function BUILD_TREES then sets BIN_COMBO_USAGE[BIN_COMBO[i]] = 1 at step 417, indicating that all bins in this combination have been accounted for. Processing then continues at step 420 as indicated by path 422.

The function BUILD_TREES locates a bin not involved in a combination (e.g., a bin for which the corresponding element in array BIN_COMBO is zero) at step 412. Upon locating a non-combined bin, the function BUILD_TREES stores the current value of counter TREE_CNT (i.e., "1"), in the element of array MAJ_TREE corresponding to the located non-combined bin (step 418). From thence processing continues with step 420 as indicated by path 422.

At step 420 the function BUILD_TREES computes the number of the next-highest numbered bin by incrementing the number of the bin located at step 412. Then, using the incremented bin number (symbolically expressed by "i"), at step 422 the function BUILD_TREES checks to see if the bin was involved in a forced combination by checking the value of BIN_COMBO[i]. If the bin was involved in a forced combination, the number of packages involved in the combination (i.e., QUAL_PER_COMBO[BIN_COMBO]) is stored at the temporary counter TEMP_CNT (step 424). If the bin was not involved in a forced combination, the number of packages in the bin (i.e., PACKS_PER_BIN[i]) is stored at the temporary counter TEMP_CNT (step 424). After execution of step 424 or step 426, the function BUILD_TREES adds the value of TEMP_CNT to the counter QUAL_CNT (step 428).

At step 430 the function BUILD_TREES determines whether the value in counter QUAL_CNT (updated at step 428) exceeds the number of available bins (i.e., exceeds the value of the counter CUTOFF_CNT initialized at step 402). If an excess is determined at step 430, at steps 432 and 434, respectively, the function BUILD_TREES increments the value in counter TREE_CNT and stores the value in TEMP_CNT in counter QUAL_CNT before proceeding to step 436. In so doing, the function BUILD_TREES begins another sort tree beginning with the current bin and initializes the value in counter QUAL_CNT for the new tree on the basis of the count determined at the appropriate one of steps 424 or 426.

At step 436 the function BUILD_TREES again discerns whether the current bin was involved in a forced combination. If the current bin was involved in a forced combination, at step 438 the function BUILD_TREES sets the corresponding elements in array MAJ_TREE equal to the current value of counter TREE_CNT for each bin included in forced combination with the current bin. Then, at step 440, a flag is set in array BIN_COMBO_USAGE at element BIN_COMBO[i] thereof.

If at step 436 the function BUILD_TREES discerns that the current bin was not involved in a forced combination, at step 442 the function BUILD_TREES assigns the value of the counter TREE_CNT to the element i in array MAJ_TREE for this bin (i.e., MAJ_TREE[i] = TREE_CNT).

After processing either step 442 or step 440, the function BUILD_TREES checks at step 444 if all bins have been processed. If not, execution loops back to step 420 for processing the next bin.

After all bins have been processed by function BUILD_TREES as determined at step 444, at step 446 the function BUILD_TREES prepares the file SRTREE.DAT (described below). The file SRTREE.DAT contains key pointer and offset information eventually used to process individual "groups" within a properly ordered input file, containing package information for multiple groups.

The file SRTREE.DAT contains records of the following structure:

```
typedef struct
{
  unsigned int QUAL_PTR; /* number of package data entries
prior to this major tree */
  unsigned int QUAL_TRE: /* number packages (bins) required
per major tree */
  unsigned int GRPS_TRE; /* groups per major tree */
  unsigned int GROF_TRE: /* number of groups prior to this
major tree
} TREE_DATA
```

In preparing the file SRTREE.DAT, the function BUILD_TREES finds the bins belonging to each sort tree. Then, for each sort tree, the number of qualifying packages for all the bins included in the tree is summed to obtain a total package count for the tree (stored in location QUAL_CNT [which is re-initialized at zero before checking each tree]). This value is stored in field QUAL_TRE for the appropriate record in file SRTREE.DAT for the tree of interest.

To determine the value for field GRPS_TRE for a record in file SRTREE.DAT, the function BUILD_TREES evaluates the expression:

$$1 + (\text{QUAL_CNT} - 2) / (\text{TOT_BINS} - 1)$$

assuming QUAL_CNT is greater than zero, and wherein TOT_BINS is as described with respect to step 402 supra. If the value of counter QUAL_CNT is zero, the value for field GRPS_TRE becomes "1".

The values of fields QUAL_PTR and GROF_TRE, respectively, for each record in file SRTREE.DAT, are obtained by maintaining running summations of the values QUAL_TRE and GRPS_TRE for previous trees.

Function TREESORT

The function TREESORT (see Fig. 4H) takes the input file SORTREC2.TMP and produces an output file SORTREC3.TMP. Whereas the records in input file SORTREC2.TMP are sorted by sack type, then within sack type by package type, and within package type by ZIP ID, the records in output file SORTREC3.TMP are sorted by major tree, then within major tree by sack type, then within sack type by package type, then within package type by ZIP ID.

At step 450, the function TREESORT determines the number of entries (i.e., the number of records) belonging to each tree. This is done by making a first pass through the input file SORTREC2.TMP. As each record in file SORTREC2.TMP is read, the bin number for that record is obtained from the record. Using the bin number extracted from the record as an index, the function TREESORT determines the major tree to which the record belongs by checking the array MAJ_TREE. The entries for each tree are counted as the input file SORTREC2.TMP is read.

All the records from input file SORTREC2.TMP for as many trees as possible are stored in dynamic memory by the function TREESORT. All the records for all the trees may not fit into dynamic memory simultaneously, so for each execution of a loop (consisting of steps 452, 454, 456, and 458), the records for as many trees as possible are stored in dynamic memory. The lowest numbered trees are handled during the first execution of the loop, with successive loop executions involving progressively higher numbered trees.

Before conducting another pass of the records included in the input file SORTREC2.TMP, at step 452 the function TREESORT sets a memory pointer for each tree which will fit into dynamic memory for the current execution of the loop. The memory pointer is easily determined since the number of entries for each tree is known from step 450, and the size of each record is standardized in accordance with the format discussed supra.

At step 454 each record in the input file SORTREC2.TMP is again read. As a record is read, it is copied into dynamic memory at the location specified by the memory pointer for the tree to which the record belongs. After each record is written to dynamic memory, the memory pointer for its tree is advanced to the next record location for that tree in dynamic memory.

When all the trees being handled by this execution of the loop have been written into dynamic memory, at step 456 the contents of the dynamic memory is written to the output file SORTREC3.TMP. Thus, the output file SORTREC3.TMP is sorted first by tree, then by sack, then by package, and then by ZIP ID.

At step 458 the function TREESORT checks to determine if all trees have been processed. If further trees remain, the function TREESORT goes back to the beginning of the loop (i.e., back to step 452) to handle further trees and to continue writing to the output file SORTREC3.TMP in the manner just described. If all trees have been written, processing continues with function FIRST_PASS_SACKS (as indicated by step 460).

Function FIRST_PASS_SACKS

Function FIRST_PASS_SACKS (see Fig. 4I) determines first pass bins containing completed sorted sacks. In this regard, the function FIRST_PASS_SACKS uses the file FULLSOR2.TMP to prepare an array FULL_SACK_BIN, which is an array of completely sorted first pass sacks indexed by first pass bin. An example of the appearance of a portion of array FULL_SACK_BIN upon completion of execution of function FIRST_PASS_SACKS is as follows::

```

FULL_SACK_BIN[000] = 0 - bin 0 not used
FULL_SACK_BIN[001] = 0
FULL_SACK_BIN[002] = 0
5  FULL_SACK_BIN[003] = 1 - bins 3 and 5 form one sack
FULL_SACK_BIN[004] = 0
FULL_SACK_BIN[005] = 1 - bins 3 and 5 form one sack
FULL_SACK_BIN[006] = 2 - 2nd sack from only one bin
FULL_SACK_BIN[007] = 0
10 FULL_SACK_BIN[008] = 0
.
.
FULL_SACK_BIN[127] = 0
FULL_SACK_BIN[128] = 0
15

```

In the simple case, function FIRST_PASS_SACKS determines completely sorted sacks by comparing the sack count with the bin count of the bin where the sack contents was assigned. If the two counts are equal, the bin is completely sorted.

20 For Mixed five digit, three digit, and state sacks the comparison is complicated because the sack contents could end up in several first pass bins. Therefore the partial sack counts from several bins making up one complete sack are compared against the same several bins total piece count. In addition, only the file containing the first pass package records is scanned, so records that end up in the subsequent pass package record file are not included. To account for this, the sack count arrays containing both first pass
25 and subsequent pass data are compared against the first pass sack count.

Describing now in detail the steps executed by function FIRST_PASS_SACKS, various arrays and parameters are initialized at step 470. For example, the output array FULL_SACK_BIN has all its elements set equal to zero, and the array BIN_USAGE is set to a logical FALSE value at step 470. In addition, the counters SACK_BIN_CNT and FIRST_PASS_SACKS, and the location SACK_CNT are initialized at
30 zero.

At step 472 the function FIRST_PASS_SACKS reads the first record from file FULLSOR2.TMP. The file FULLSOR2.TMP was created by the function SACK_SORT, and is a file containing one record per completely sorted first pass package. In reading the first record from file FULLSOR2.TMP at step 472, the function FIRST_PASS_SACKS obtains the ZIP ID and the sack type from the initial record, and stores
35 those values at the respective locations LAST_SACK_ZIP_ID and LAST_SACK_TYPE.

At step 474 the function FIRST_PASS_SACKS begins a loop of reading and processing further records in the file FULLSOR2.TMP. In so doing, the function FIRST_PASS_SACKS obtains the ZIP ID and the sack type from the new record, and stores those values at the respective locations SACK_ZIP_ID and SACK_TYPE.

40 At step 476 the function FIRST_PASS_SACKS determines whether the most-recently read record is in the same sack as the previous record. This is affirmatively determined when SACK_ZIP_ID = LAST_SACK_ZIP_ID and SACK_ZIP_ID = LAST_SACK_ZIP_ID. If it is determined at step 476 that the most-recently read record is not in the same sack as the Previous record, processing continues at step 478. Otherwise processing branches to step 480.

45 At step 480 the function FIRST_PASS_SACKS determines whether the most recently read record involves a new bin. A new bin is involved if the element corresponding to the new bin in array BIN_USAGE is still FALSE. If a new bin is not involved, processing continues with step 482. Otherwise, processing branches to step 484.

When a new bin is involved, at step 484 the function FIRST_PASS_SACKS sets the element in array
50 BIN_USAGE corresponding to the new bin to a TRUE value. Then, at step 486, the function FIRST_PASS_SACKS adds the count of the number of mailpieces in that new bin to the counter SACK_BIN_CNT. It will be remembered that the count of the number of mailpieces in the new bin is obtained from the file AGGR.DAT, which was created at step 202.

At step 482, reached either from step 480 or step 486, the function FIRST_PASS_SACKS sums the
55 partial package counts by adding the adding the package count from the most recent record to the counter SACK_CNT. At step 478, reached either from step 476 or step 482, the function FIRST_PASS_SACKS initializes the logical flag ALL_FROM_PASS1 to a logical zero.

At step 488 the function FIRST_PASS_SACKS examines the value of the location

LAST_SACK_TYPE to determine what type of sack is being processed, with a view to determining whether the flag ALL_FROM_PASS1 should be changed from FALSE to TRUE for this sack, thereby indicating that the sack is not split into first and second pass records. FIVE_DIGIT sacks cannot be split into first and second pass records, so if the value of LAST_SACK_TYPE is FIVE_DIGIT, the flag ALL-
 5 FROM_PASS1 is set TRUE.

For the other sack types, function FIRST_PASS_SACKS at step 488 determines whether the value of an appropriate counter equals the current value of the counter SACK_CNT (calculated at step 482). In this respect, the equality determination at step 488 is made with respect to the appropriate one of the counters MIXED_FIVE_SACK_CNT, THREE_DIGIT_SACK_CNT, or STATE_SACK_CNT for the last sack
 10 ZIP_ID.

At step 490 the function FIRST_PASS_SACKS determines whether the flag ALL_FROM_PASS1 is TRUE and whether the value in counter SACK_CNT equals the value in the counter SACK_BIN_CNT (see step 486). When both determinations are affirmative, the function FIRST_PASS_SACKS realizes that it has encountered a completely sorted first pass sack. If either determination is negative, the function
 15 FIRST_PASS_SACKS continues processing at step 492; otherwise the function FIRST_PASS_SACKS branches to steps 493 followed by step 494.

Both determinations at step 490 being affirmative reflect the location of a completely sorted sack, and cause a branch in processing to step 493. At step 493 the counter FIRST_PASS_SACKS is incremented to a value which will be used as a unique identifying value for the just-located completely sorted sack. At
 20 step 494 the value of FIRST_PASS_SACKS is stored in every element of array FULL_SACK_BIN which corresponds to a bin which has mailpieces included in the completely sorted sack. The bins which have mailpieces included in this most-recently located completely sorted sack are reflected by the elements in array BIN_USAGE which have been set to a logical TRUE value.

At step 492, reached either from step 490 or step 494, the function FIRST_PASS_SACKS reinitializes
 25 the counters SACK_CNT and SACK_BIN_CNT at zero; sets every element in array BIN_USAGE to a FALSE value; and, stores the value from location SACK_ZIP_ID in location LAST_SACK_ZIP_ID and the value from location SACK_TYPE in the location LAST_SACK_TYPE.

At step 495 the function FIRST_PASS_SACKS determines whether any more records remain for reading in file FULLSOR2.TMP. If records remain, processing loops back to step 474, at which the repetition
 30 of the above-described steps occurs for the next record. If no further records remain in file FULLSOR2.TMP, the function FIRST_PASS_SACKS processes the last-read record at step 496. In this regard, the processing of step 496 is essentially the same as steps 488 through 494 inclusive, except there is no step corresponding to reinitialization step 492. At step 497 processing is transferred to function MAKE_BAGS.

Function MAKE_BAGS

Function MAKE-BAGS (see Figs. 4J and 4K) assigns unique identification numbers to each bag destination and determines the number of pieces assigned to a destination. In addition, the bag identifica-
 40 tion numbers get corresponding assignments to subsequent pass groups and bins.

At step 500 the function MAKE_BAGS initializes the value of location LAST_BAG_ASSIGNED at zero. At step 502 the function MAKE_BAGS handles mixed states bags, assigning all mixed states records to the bag number "one". As part of step 502, the function MAKE_BAGS increments the value at location
 45 LAST_BAG_ASSIGNED (so that the value is "1"), and then sets BAG_NO = 1. Further, at step 502 the function MAKE_BAGS creates a record in a file BAGTAG_HANDLE, with the record having the following format and values:

```

50      BAG_DATA.ZIP_ID      = 0
      BAG_DATA.BAG_ID_NO   = BAG.NO
      BAG_DATA.BEG_GROUP   = 0
      BAG_DATA.BEG_BIN     = 0
      BAG_DATA.END_GROUP   = 0
      BAG_DATA.END_BIN     = 0
55      BAG_DATA.S_TYPE     = MIXED STATES
      BAG_DATA.CNT         = MS_SACK_CNT
  
```

Having handled the mixed states bags at step 502, the function MAKE__BAGS determines the number of first pass bags containing either some or all completely sorted first pass packages. These sacks must have their bag ID assignment before subsequent pass groups are handled. The completely sorted first pass bags are processed by steps 504 through 550 of the function MAKE__BAGS. Thereafter, at steps 554 through 572, the subsequent bags are processed.

In the above regard, at step 504 the function MAKE__BAGS initializes various values for handling the first pass bags. The function MAKE__BAGS sets LAST__ID = -1; LAST__TYPE = -1; and, FRST__PASS__BAGS.CNT = 0.

At step 506 the function MAKE__BAGS reads, as a first step in a loop, a record from file FULLSOR2.TMP. The file FULLSOR2.TMP was created by the function SACK__SORT. Assuming at step 508 that the record read is not from the same sack as the last record in the file FULLSOR2.TMP, at step 510 the function MAKE__BAGS checks whether the record just read from the file FULLSOR2.TMP pertained to a mixed states sack. If so, the function MAKE__BAGS realizes that it has already handled (at step 502) the mixed states bag, and at step 512 sets SACK__NO = 1. Also, at step 514, the function MAKE__BAGS writes a corresponding record (i.e., the record read from file FULLSOR2.TMP with the bag number updated) to file FULLSORT. After execution of step 514, processing loops back to step 506 for the reading of another record from file FULLSOR2.TMP.

Assuming that the function MAKE__BAGS determined at step 510 that the record just-read pertained to a sack type other than a mixed states sack, at step 516 the function MAKE__BAGS obtains the ZIP ID from the record and stores that ZIP ID in location SACK__ZIP__ID. At step 518 the function MAKE__BAGS checks whether the record just-read from file FULLSOR2.TMP signals a change of sack. A change of sack is signalled when SACK__ZIP__ID does not equal the value stored in location LAST__ID.

If a change of sack is not encountered at step 518, the function MAKE__BAGS adds the value in the field SACK.PACKAGE.CNT from the record just-read to the counter FRST__PASS__BAG.CNT (step 520). Location SACK.NO is then set to the value at location BAG.NO (step 522). At step 524, a corresponding record is written to file FULLSORT. After the corresponding record is written at step 524, execution loops back to step 506 for the reading of yet another record from the file FULLSOR2.TMP.

If a change of sack is encountered at step 518, the function MAKE__BAGS determines whether the record just-read from file FULLSOR2.TMP was the very first record in file FULLSOR2.TMP (step 526). For all but the very first record in file FULLSOR2.TMP, the function MAKE__BAGS writes a corresponding record in the file BAGTAG__HANDLE (step 528).

At step 530 the function MAKE__BAGS computes a value for pointer PREV__ASSIGN__PTR. In this regard, at step 530 the function MAKE__BAGS uses the value of SACK__ZIP__ID obtained from the record just-read as an index for an appropriate one of arrays MIXED__FIVE__BAG__NO; STATE__BAG__NO; THREE__DIGIT__BAG__NO; depending on the value of SACK.STYPE obtained for the record just-read. (These arrays were also initialized at step 500). The value obtained by indexing into the appropriate array is stored in the pointer PREV__ASSIGN__PTR. If the value of SACK.STYPE obtained from the record just-read is other than MIXED__FIVE; THREE__DIGIT; or STATE; the pointer PREV__ASSIGN__PTR is assigned the value NULL at step 530.

At step 532 the function MAKE__BAGS checks to determine if PREV__ASSIGN__PTR is NULL or the contents thereof is zero. If either value is stored in pointer PREV__ASSIGN__PTR, at step 534 LAST__BAG__ASSIGNED is incremented and at step 536 that incremented value is stored in location BAG__NO. Otherwise, at step 538, BAG__NO has the value from *PREV__ASSIGN__PTR stored therein.

Step 540 is reached either from step 536 or step 538. At step 540, the function MAKE__BAGS checks whether the pointer PREV__ASSIGN__PTR has the value NULL stored therein. If so, at step 542 the value of BAG__NO is stored in location *PREV__ASSIGN__PTR.

At step 544, which follows either step 540 or step 542, the function MAKE__BAGS creates a record for the array FRST__PASS__BAGS. The record created at step 544 has the following format and values:


```

FRST_PASS_BAGS.ZIP_ID      = SACK_ZIP_ID
FRST_PASS_BAGS.BAD_ID_NO   = BAG_NO
FRST_PASS_BAGS.BEG_GROUP   = 0
5  FRST_PASS_BAGS.BEG_BIN    = 0
FRST_PASS_BAGS.END_GROUP   = 0
FRST_PASS_BAGS.END_BIN     = 0
FRST_PASS_BAGS.S_TYPE      = SACK.STYPE
10 FRST_PASS_BAGS.CNT        = SACK.PACKAGE.CNT

```

After creating a record for file FRST_PASS_BAGS at step 544, the function MAKE_BAGS stores the value in location SACK_ZIP_ID in location LAST_ID (step 546) and stores the value in location SACK.STYPE in location LAST_TYPE (step 548). Then function MAKE_BAGS jumps back to execute steps 522 and 524 before reading another record from file FULLSOR2.TMP at step 506. In this respect, as explained before, at step 522 the location SACK.NO is then set to the value at location BAG.NO and, at step 524, a corresponding record is written to file FULLSORT.

After all records from file FULLSOR2.TMP have been read as determined by step 508, the final FIRST_PASS_BAGS record is written to file BAGTAG_HANDLE at step 549. Then, as indicated by symbol 550 processing continues at step 554 (see Fig. 4K) for handling package records for subsequent pass sorting (i.e., package records from file SORTREC3.TMP).

At step 554, which is somewhat akin to step 504, various parameters are initialized, i.e., CUS_GROUP = 0; LAST_ID = -1; LAST_TYPE = -1; and, BAG_DATA.CNT = 0. The file SRTREE.DAT is read to determine if any more tree structures remain. In the case of remaining tree structures, CUS_GROUP is incremented by one (step 560). Then a check at step 562 is made to determine if there are any more groups defined by the present tree record. If there are not more groups, the function continues at step 556 where the file SRTREE.DAT is read for another tree structure. In the case where a group remains in the present tree (determined at step 562), at step 564 an array BIN_CNT is set declaring the number of packages to be placed in the present group's bins. In addition, at step 565 a file seek position is set in file SORTREC3.TMP, for the first record in the present group being processed, by using tree record information and the present group within the tree. At step 567 the array BIN_CNT contents are checked for a value of one. This indicates that no more groups are required to sort the next record found in SORTREC3.TMP. For all bins such that BIN_CNT[i] = 1 a record is read from the file SORTREC3.TMP (step 568). In the case where no more bins remain, processing proceeds to step 560 where a new group is used. Where a record is read from SORTREC3.TMP the record is processed at step 570 in the same way as for a first pass sort record. Step 570 is the same as steps 506 through 549 except in step 506 where a record is read from file FULLSOR2.TMP it is now read from file SORTREC3.TMP. After processing the package record from file SORTREC3.TMP the process continues by going to step 566 where a check is made to determine if any more bins have been assigned to the present group.

In the case where no more trees are left at step 558, the program is ready to go to the function MAKE_CLIENT_COUNTS (as indicated by step 572).

Function MAKE_CLIENT_COUNTS

The function MAKE_CLIENT_COUNTS (see Fig. 4L) sets up several count categories for each client. These counts are subsequently used in postage reporting and client billing.

The function MAKE_CLIENT_COUNTS uses the input file COUNT.DAT created by function FIRST_SORT_PASS and the input file SORTREC2.TMP created by the function SACK_SORT. It will be recalled that the format of file COUNT.DAT is as follows:

		<u>byte</u> <u>offset #</u>
	Zip Code - 4 bytes (long integer)	0
5	Stream Index - 1 byte (hex/binary value)	4
	Client Index - 1 byte (hex/binary value)	5
	Bin - 1 byte (hex/binary value)	6
	5 Digit OCR/BCR count - 2 bytes (unsigned integer)	7
	Zip + 4 OCR count - 2 bytes (unsigned integer)	9
10	Zip + 4 Barcoded count - 2 bytes (unsigned integer)	11

The records in file COUNT.DAT are sorted in ascending order. In this respect, a primary sortation is done by ZIP code. For ZIP codes repeated due to their usage in different client/mailstreams, a secondary
 15 sortation is performed by first sorting the client index number, followed by the stream index number.

At step 600 the function MAKE_CLIENT_COUNTS initializes several counter arrays. In this respect, at step 600 the following are initialized at zero:

```

20      CLIENT[i].QUAL_TOT.COUNT5
      CLIENT[i].QUAL_TOT.ZIP4
      CLIENT[i].QUAL_TOT.BARCDE
      CLIENT[i].NQUAL_TOT.COUNT5
      CLIENT[i].NQUAL_TOT.ZIP4
25      CLIENT[i].NQUAL_TOT.BARCDE

```

where i represents the client index (the number associated with a particular client). CLIENT[i].QUAL_TOT.COUNT5 will ultimately contain the total number of mailpieces from client "i" which qualify for
 30 the 5 Digit OCR/barcode postage discount; CLIENT[i].QUAL_TOT.ZIP4 will ultimately contain the total number of mailpieces from client "i" which qualify for the ZIP+4 OCR postage discount; CLIENT[i].QUAL_TOT.BARCDE will ultimately contain the total number of mailpieces from client "i" which qualify for the ZIP+4 barcoded postage discount; CLIENT[i].NQUAL_TOT.COUNT5 will ultimately contain the total number of 5 Digit OCR/barcoded mailpieces from client "i" which do not qualify for the 5 Digit
 35 OCR/barcoded postage discount; CLIENT[i].NQUAL_TOT.ZIP4 will ultimately contain the total number of ZIP+4 OCR mailpieces from client "i" which do not qualify for the ZIP+4 OCR postage discount; and, CLIENT[i].NQUAL_TOT.BARCDE will ultimately contain the total number of ZIP+4 Barcoded mailpieces from client "i" which do not qualify for the ZIP+4 Barcoded postage discount

At step 600 the following are also initialized at zero:

```

40      TOTALS[0].COUNT5
      TOTALS[0].COUNT9
      TOTALS[0].BAR_CNT

45      TOTALS[1].COUNT5
      TOTALS[1].COUNT9
      TOTALS[1].BAR_CNT

50      TOTALS[2].COUNT5
      TOTALS[2].COUNT9
      TOTALS[2].BAR_CNT

```

At step 602 the function MAKE_CLIENT_COUNTS reads a record in the file COUNT.DAT. As noted
 55 above, each record in file COUNT.DAT has a ZIP ID field. At step 604 the function MAKE_CLIENT_COUNTS searches the file SACK2.TMP to find the record in file SACK2.TMP having the same ZIP ID as the current record in file COUNT.DAT. At step 606 the function MAKE_CLIENT_COUNTS consults the record found in file SACK2.TMP to determine the sack type (STYPE) assigned to the sack

containing mailpieces for the current ZIP ID.

At step 608 a determination is made whether the record in file SACK2.TMP for the current ZIP ID indicates that mailpieces having the current ZIP ID are contained in FIVE__DIGIT or MIXED__FIVE sacks (thereby qualifying for the applicable postage discounts). If the determination at step 608 is affirmative, the function MAKE__CLIENT__COUNTS adds values from the appropriate fields of the current COUNT.DAT record to "qualifying" counters for the client having the client index borne by the current COUNT.DAT record. In this respect, for client "i" at step 610 the counter CLIENT[i].QUAL__TOT.COUNT5 is incremented by the value contained at byte offset 7 in the COUNT.DAT record; the counter CLIENT[i].QUAL__TOT.ZIP4 is incremented by the value contained at byte offset 9 in the COUNT.DAT record; and, the counter CLIENT-
[i].QUAL__TOT.BARCODE is incremented by the value contained at byte offset 11 in the COUNT.DAT record.

On the otherhand, if the determination at step 608 is negative, at step 612 other "non-qualifying" counters for client "i" are incremented by the values contained at byte offsets 7, 9, and 11, namely counters CLIENT[i].NQUAL__TOT.COUNT5, CLIENT[i].NQUAL__TOT.ZIP4, and CLIENT[i].NQUAL__TOT.BARCODE, respectively.

If other records remain in file COUNT.DAT (as determined at step 614), the function MAKE__CLIENT__COUNTS loops back to step 602 to obtain the next record and to execute the steps of Fig. 4L for that next record. After all records in file COUNT.DAT have been processed by function MAKE__CLIENT__COUNTS, several "totals" are computed at step 616.

At step 616 the function MAKE__CLIENT__COUNTS determines the following totals:

20	TOTALS[0].COUNT5	(The number of mailpieces for all clients qualifying for the 5 Digit OCR/barcode postage discount)
25	TOTALS[0].COUNT4	(The number of mailpieces for all clients qualifying for the ZIP+4 OCR postage discount)
30	TOTALS[0].BAR_CNT	(The number of mailpieces for all clients qualifying for the ZIP+4 barcoded postage discount)
35	TOTALS[1].COUNT5	(The number of non-qualifying 5 Digit OCR/barcode mailpieces for all clients)
40	TOTALS[1].COUNT4	(The number of non-qualifying ZIP+4 OCR mailpieces for all clients)
45	TOTALS[1].BAR_CNT	(The number of non-qualifying ZIP+4 barcoded mailpieces for all clients)
	TOTALS[2].COUNT5 =	TOTALS[0].COUNT5 + TOTALS[1].COUNT5
	TOTALS[2].COUNT4 =	TOTALS[0].COUNT4 + TOTALS[1].COUNT4
	TOTALS[2].BAR_CNT=	TOTALS[0].BAR_CNT + TOTALS[1].BAR_CNT

Then, at step 616, for $j = 0, 1,$ and 2 , the function MAKE__CLIENT__COUNTS determines TOTALS[j].TOTAL, which is evaluated for each j by the expression $TOTALS[j].TOTAL = TOTALS[j].COUNT5 + TOTALS[j].COUNT9 + TOTALS[j].BAR_CNT$.

Function CORRELATE__BAGS

The function CORRELATE__BAGS (see Fig. 4M) determines the bag number (i.e., the sack number) for each Zip Code and creates a file SACK3.TMP. The file SACK3.TMP is similar to the file SACK2.TMP which is used to create file SACK3.TMP, but unlike file SACK2.TMP the file SACK3.TMP has a bag number assigned to the "no" field in each record.

The input files utilized by function CORRELATE__BAGS are file SACK2.TMP, file SORTREC.DAT, and file FULLSORT.DAT. These input files are created by the functions ASSIGN__SACKS, MAKE__BAGS, and MAKE__BAGS, respectively.

Function CORRELATE__BAGS reads successive records from the file SACK2.TMP and attempts to first match the current record with a record from the file SORTREC.DAT. If a match is found, the function

CORRELATE__BAGS can assign a bag number for the Zip Code for the current record from the file SACK2.TMP, and writes a record including that bag number to the new file SACK3.TMP. If a match is not found in the file SORTREC.DAT, the function CORRELATE__BAGS then attempts to match the current record from the file SACK2.TMP with a record from the file FULLSORT.DAT. If a match is found, the function CORRELATE__BAGS assigns a bag number for the Zip Code for the current record from the file SACK2.TMP, and writes a record including that bag number to the new file SACK3.TMP.

At step 640 the function CORRELATE__BAGS reads a record from the file SACK2.TMP. At step 642 the function CORRELATE__BAGS obtains the value in the PTYPE field for the record just read from the file SACK2.TMP. At step 644 a check is made to determine if the PTYPE value is "M", indicating a mixed states package. If the PTYPE is "M", the function CORRELATE__BAGS knows that all MIXED STATES packages are to go into the first bin, and accordingly at step 646 assigns SACK.NO the value "1". Then, at step 648, a record is written to the new file SACK3.TMP, with the "no" field of the record having stored therein the value of SACK.NO (i.e., "1").

If the PTYPE value for the current record from file SACK2.TMP is not an "M", at step 650 the function CORRELATE__BAGS obtains the PACKAGE__ZIP_ID value from the zip identifier field of the struct PACKAGE included in the struct SACK comprising the record for the file SACK2.TMP. Then, preparatory to a loop of reading records from file SORTREC.DAT, at step 652 the function CORRELATE__BAGS initializes the flag MATCH__FLAG to have a TRUE value and the index CURRENT__INDEX to have the value "0".

As indicated above, the function CORRELATE__BAGS first attempts to match the current record in the file SACK2.TMP with a record in file SORTREC.DAT. In this regard, at step 654 the function CORRELATE__BAGS requires the reading of a record from the file SORTREC.DAT. Then, at step 656, a value for pointer CURRENT__PTR is determined, which value reflects the physical position of the current record in the file SORTREC.DAT relative to the beginning of the file SORTREC.DAT. As will be seen below, the value of pointer CURRENT__PTR is ultimately used to determine the bag number for the package referred to by the current record in file SACK2.TMP.

At step 658 the function CORRELATE__BAGS determines whether information from the current record in file SACK2.TMP matches the corresponding information for the current record in file SORTREC.DAT. Specifically, the PTYPE and zip identifier fields for the two current records are compared. In this regard, the zip identifier information for the current record in file SACK2.TMP is stored in the location PACKAGE__ZIP_ID previously determined at step 650.

If a "match" is located at step 658, the function CORRELATE__BAGS performs three operations depicted by steps 660, 662, and 664. At step 660 the value of CURRENT__PTR is used to find the bag number and set the SACK.NO. At step 662 a record is written to the new file SACK3.TMP, with the value of SACK.NO as determined at step 660 being stored in the "no" field of the record. At step 664, the flag MATCH__FLAG is set to a TRUE value.

At step 666, reached either from step 664 after a "match" or from step 658 when a match is not found, the index CURRENT__INDEX is incremented. As explained above, the value of CURRENT__INDEX is used at step 656 to determine the value of CURRENT__PTR, which in turn is used at step 660 to determine the value of SACK.NO.

At step 668 the function CORRELATE__BAGS checks to see if the flag MATCH__FLAG has a TRUE value, indicating that a match has just been found. If so, the function CORRELATE__BAGS knows that it is finished with the current record in file SACK2.TMP, and can go on to process the next record in file SACK2.TMP, with the hope of finding a match for that next record as well. In this regard, an affirmative determination at step 668 results in a branching back to step 640 for reading the next record in file SACK2.TMP.

If a match were not found comparing the current record in file SACK2.TMP with the current record in file SORTREC.DAT, at step 670 the function CORRELATE__BAGS checks to determine whether there are yet further records in the file SORTREC.DAT for which a comparison for prospective match can be made. If additional records remain in file SORTREC.DAT, the function CORRELATE__BAGS branches back to step 654 for reading the next record in file SORTREC.DAT. For that next record, the steps 658 through 668 of Fig. 4M are executed, with that next record from file SORTREC.DAT becoming the "current" record from file SORTREC.DAT.

If, at step 670, it is determined that the file SORTREC.DAT has been exhausted with no match for the current record in file SACK2.TMP, as indicated above the function CORRELATE BAGS goes on to check if a match for the current record in file SACK2.TMP can be found with a record in the file FULLSORT.DAT. Before reading a record from the file FULLSORT.DAT, however, at step 672 a loop parameter "i" is initialized at "0". As seen hereinafter, this loop parameter "i" plays a role in determining the SACK.NO should a match occur.

At step 674 the function CORRELATE__BAGS requires the reading of a record from the file FULL-SORT.DAT. At step 678 the function CORRELATE__BAGS determines whether information from the current record in file SACK2.TMP matches the corresponding information for the current record in file FULL-SORT.DAT. Specifically, the PTYPE and zip identifier fields for the two current records are compared. In this regard, the zip identifier information for the current record in file SACK2.TMP is stored in the location PACKAGE__ZIP__ID previously determined at step 650.

If a "match" is located at step 678, the function CORRELATE__BAGS performs three operations depicted by steps 680, 682, and 684. At step 680 the value of the loop parameter "i" (which points to the first pass record with a match) is used to find the assigned bag number and to set SACK.NO. At step 682 a record is written to the new file SACK3.TMP, with the value of SACK.NO as determined at step 680 being stored in the "no" field of the record. At step 684, the flag MATCH__FLAG is set to a TRUE value.

At step 686, reached either from step 684 after a "match" or from step 678 when a match is not found, the loop parameter "i" is incremented. As explained above, the value of the loop parameter "i" is used to determine the value of SACK.NO.

At step 688 the function CORRELATE__BAGS checks to see if the flag MATCH__FLAG has a TRUE value, indicating that a match has just been found. If so, the function CORRELATE__BAGS knows that it is finished with the current record in file SACK2.TMP, and can go on to process the next record in file SACK2.TMP, with the hope of finding a match for that next record as well. In this regard, an affirmative determination at step 688 results in a branching back to step 640 for reading the next record in file SACK2.TMP.

If a match were not found comparing the current record in file SACK2.TMP with the current record in file SORTREC.DAT, at step 690 the function CORRELATE__BAGS checks to determine whether there are yet further records in the file FULLSORT.DAT for which a comparison for prospective match can be made. If additional records remain in file FULLSORT.DAT, the function CORRELATE__BAGS branches back to step 674 for reading the next record in file FULLSORT.DAT. For that next record, the steps 678 through 688 of Fig. 4M are executed, with that next record from file FULLSORT.DAT becoming the "current" record from file FULLSORT.DAT.

If, at step 690, it is determined that the file FULLSORT.DAT has been exhausted with no match for the current record in file SACK2.TMP, an error message is created at step 692. When, at step 640, it is determined that the file SACK2.TMP has been exhausted, and a match found for each record therein, processing continues with the function SAVE__ANAL__CNT described below.

After the last record is read at step 640, processing continues with the function SAVE__ANAL__CNT.

Function SAVE__ANAL__CNT

The function SAVE__ANAL__CNT (see Fig. 4L) creates a first pass count file ANAL__CNT.DAT which resembles the file COUNT.DAT, except that the file ANAL__CNT.DAT has the parameters package type (PTYPE), bag type (SType), and bag id (SACK.NO) appended to each record.

In the above regard, the function SAVE__ANAL__CNT uses the files COUNT.DAT and SACK3.TMP as input. The file COUNT.DAT was created by the function FIRST__SORT__PASS (see Fig. 4A); the file SACK3.TMP was created by the file CORRELATE__BAGS (see Fig. 4M).

The file ANAL__CNT.DAT has its records sorted by zip code, then within zip code by client, and within client by mailstream. Each record includes zip code counts by 5 Digit, ZIP + 4, and ZIP + 4 Barcoded categories, first pass destination bin, package type (PTYPE), bag type (SType), and bag ID number (SACK.NO).

Multiple records of the following structure are contained in the file ANAL__CNT.DAT:

```

typedef struct {
    CNT_DATA cnt_dat;
    char      PTYPE;      /*package type*/
    char      STYPE;      /*sack type*/
    unsigned int BAG ID;   /*bag ID number*/
} ANAL_CNTS;

```

where

```

typedef struct
{
    long      ZIP ID;      /*zip code*/
    unsigned int STREAM;   /*client-mailstream*/
    unsigned char BIN;     /*bin assignment 1st pass*/
    unsigned int CNT5;     /*5 Digit count*/
    unsigned int ZIP4;     /*ZIP+4 count*/
    unsigned int BARCODE;  /*ZIP+4 Barcoded count*/
} CNT_DATA;

```

Thus, the file ANAL__CNT.DAT is sorted by Zip code, then within Zip code by client, then within client by mailstream.

At step 700, the function SAVE__ANAL__CNT (see Fig. 4N) reads an initial SACK3.TMP record. Then successive COUNT.DAT records are read and a corresponding ANAL__CNT.DAT record is written for every COUNT.DAT record read. Since there is only one record per package in file SACK3.TMP, and since packages may be made up of multiple records, there will be more COUNT.DAT records than SACK3.TMP records. After a COUNT.DAT record is read, at steps 702 and 704 the ZIP code is checked to see if it belongs to the package from the SACK3.TMP record. If the ZIP code belongs to the packages, the package type, bag type, and bag number information from the package is appended to the information in the COUNT.DAT record and written to ANAL__CNT.DAT at step 708. If the ZIP code did not belong to the current SACK3.TMP record, another SACK3.TMP record is read and the new information is written to ANAL__CNT.DAT as in step 708 discussed above. This process repeats until all records in COUNT.DAT have been processed.

Function SET__POST__CNTS

The function SET__POST__CNTS (see Fig. 4N) sets up initial counts files for postage reporting based on the final sorting pass. In this respect, the function SET__POST__CNTS uses input files ANAL__CNT.DAT (generated by the function SAVE__ANAL__CNT) and AGGR.DAT (generated by the function FIRST_SORT_PASS) to create two new files, file PASS1AGGR.DAT and file PASS2AGGR.DAT. The file PASS1AGGR.DAT contains counts for all mailpieces that will not be fed during subsequent pass sorting. The file PASS2AGGR.DAT contains counts for subsequent pass sorting. Both files PASS1AGGR.DAT and PASS2AGGR.DAT include counts for both 5 Digit level rate (qualifying) and Basic level rate (non-qualifying) mailpieces by 5 Digit, ZIP + 4, and ZIP + 4 Barcoded categories, and also include rejects.

The following data structure is employed for both files PASS1AGGR.DAT and PASS2AGGR.DAT:

```

typedef struct
{
    struct
    {
        long COUNT5;
        long COUNT9;
        long BAR_CNT;
    } QUAL, NQUAL;
    long REJECTS;
} POST_SUM_CNT;

```

The function SET__POST__CNTS basically creates the new files PASS1AGGR.DAT and

PASS2AGGR.DAT after reading all the records in the file ANAL_CNT.DAT. At the beginning of a loop commencing with step 730, the function SET_POST_CNTS reads a record from the file ANAL_CNT.DAT. At step 732 the function SET_POST_CNTS determines whether the bin number included in the bin field from the record just read from file ANAL_CNT.DAT is a bin containing fully sorted packages. This is done
 5 by checking whether the element of array FULLSORT corresponding to that bin has a non-zero value. If a zero value exists for the element in array FULLSORT corresponding to that bin, the function SET_POST_CNTS loops back to step 730 for the reading of another record from the file ANAL_CNT.DAT. Otherwise, execution continues with step 734.

At step 734 the function SET_POST_CNTS examines the sack type (STYPE) field of the current
 10 record from the file ANAL_CNT.DAT. If the value of STYPE is FIVE_DIGIT or MIXED_FIVE, the function SET_POST_CNTS knows to go to step 736 to increase certain "qualifying" counters. Otherwise the function SET_POST_CNTS will go to step 738 to increase certain "non-qualifying" counters.

In the above regard, at step 736 the function SET_POST_CNTS increases the following counters by the values stored in corresponding fields in the current record from file ANAL_CNT.DAT: counter
 15 PASS1.QUAL.COUNT5; counter PASS1.QUAL.COUNT9; and, counter PASS1.QUAL.BAR_CNT. Alternatively, at step 738 the function SET_POST_CNTS increases the following counters by the values stored in corresponding fields in the current record from file ANAL_CNT.DAT: counter PASS1.NQUAL.COUNT5; counter PASS1.NQUAL.COUNT9; and, counter PASS1.NQUAL.BAR_CNT.

At step 740 the function SET_POST_CNTS determines whether the current record read from file
 20 ANAL_CNT.DAT was the last record. If not, processing loops back to step 730 for the reading of a new record from file ANAL_CNT.DAT, after which the steps 732 et seq. of function SET_POST_CNTS are repeated, with the next record becoming the "current" record in accordance with the preceding discussion.

Upon the exhaustion of file ANAL_CNT.DAT as determined at step 740, the function SET_POST_CNTS reads the file AGGR.DAT in order to include reject counts (step 742). Then, at step
 25 744, the function SET_POST_CNTS writes the entire file PASS1AGGR.DAT, which has the format described above. Thereafter, at step 746, the function SET_POST_CNTS initializes all count values to zero in the file PASS2AGGR.DAT in preparation for subsequent use.

Function INIT_GROUP_CNTS

Function INIT_GROUP_CNTS produces a file GRPCNTS.DAT that maintains counts, by group
 30 number, of actually fed and rejected mailpieces. The file GRPCNTS.DAT is initialized with all zeros and is intended to be updated during subsequent pass sorting. The file GRPCNTS.DAT is used for second pass sorting display and insures that mailpieces fed in a wrong mode will not allow the reject count to go
 35 negative. Records in the file GRPCNTS.DAT are of the following structure:

```

      struct {
          long FED;
          long REJ;
  40         } GRP_CNT
  
```

Function PRINT_OUT

Function PRINT_OUT serves to print information pertaining to the files created in the manner
 45 described above. In particular, the function PRINT_OUT generates hardcopies of the following reports: Group Listing Report (see TABLES 1, 2A - 2E); Bag Tags Report (see TABLE 3); Job Summary Report (see TABLE 4); Postage Summary Report (see TABLES 5 - 6A); and, Bag Audit Report (see TABLE 7).

TABLE 1 is produced by printing out file ANAL_SUM.DAT; file TOTQUL.DAT; file MAJ_TREE.DAT; file FRST_PAK.DAT; and, file FRST_SAK.DAT. TABLES 1, 2A - 2E show which bins 26 are to be grouped
 50 together for subsequent for subsequent passes through the sorter apparatus 20. For example, bins 26₃ - 26₇ are to be grouped together as Group 1; bin 26₈ forms Group 2; bin 26₉ forms Group 3; bins 26₁₀ - 26₁₁ are to be grouped together as Group 5; and so forth. Some groups are noticeably absent from TABLE 1, such as Group 4, for example. It will be seen below in connection with TABLES 2C and 2D that Group 4 is ultimately generated during a second pass of the Group 3 mailpieces. Likewise, other groups not listed in
 55 TABLE 1 are generated during successive passes (not the first passes) of other groups.

The output of TABLE 1, and of TABLES 2A - 2E explained hereafter, are available upon completion of the program ANALYZE_MAIL after the initial pass of mailpieces through the sorter 20. Using the output of TABLE 1 AND TABLES 2A - 2E, an operator knows how to group together mailpieces for subsequent

passes before those passes are executed. For example, after the initial pass is completed and the program ANALYZE__MAIL has generated TABLE 1 and TABLES 2A - 2E, the operator would manually retrieve the Group 1 mailpieces from bins 26₃ - 26₇ and load those mailpieces into the input hopper 30 of the sorter 20.

TABLES 2A - 2E illustrate the output generated upon the printing of the Group Listing Report, which reflects the contents of the bins 26 after passes of the various groups. Table 2A reflects the contents of the bins 26 after the Group 1 mailpieces (gathered from bins 26₃ - 26₇ after the initial pass). Each bin 26 has a package stored therein, since it is indicated that these bins are fully sorted.

TABLE 2A has five headings: "BIN"; "ZIP"; "P"; "B", and "ID". The "BIN" heading refers to the bins 26 of the sorting machine 20. For example, "bin 3" refers to bin 26₃ according to the nomenclature previously adopted. "ZIP" refers to the Zip Code for the package of mailpieces stored in the associated bin. The heading "P" refers to the type of package (PTYPE) stored in the bin. The heading "B" refers to the type of sack (STYPE) in which the package in the bin is to be inserted. The heading "ID" refers to the bag identification number, or sack number, of the sack which includes the mailpieces of the bin.

For example, from TABLE 2A it is seen that bin 26₃ contains a 5 Digit package for Zip code 02806, which is to be placed in a MIXED__FIVE ("M5") sack bearing sack number ("ID") "2". As TABLE 2A is further read across the page, it is also seen that bin 26₄ contains a 5 Digit package for Zip code 02809, which is to be placed in the same MIXED__FIVE ("M5") sack bearing sack number ("ID") "2". In this regard, if a number is not listed under the heading "ID" for a bin, it is understood that the mailpieces from that bin are to be placed in the same sack with the preceding bin(s). Thus, from TABLE 2A it is apparent that the packages from bins 26₃ - 26₃₄ will all be placed in the same sack (i.e., the sack bearing sack number "2"). Similarly, the packages from bins 26₃₅ - 26₅₁ are to be placed in sack number 3; the three digit packages from bins 26₅₂ - 26₅₄ are to be placed in sack number 4; and so forth. Noticeably, bin 26₁₂₄ houses the MIXED__STATES sack, which bears sack number 1 (see the function MAKE__BAGS, step 502, for an explanation in this regard).

After running Group 1, and loading all the mailpieces from Group 1 into sacks bearing sack numbers 1 - 12 as indicated in TABLE 2A, the machine operator requests that a new sort scheme be loaded into memory with instructions to direct pieces in Group 2 to the proper bins. This is done by referencing file SORTREC.DAT (the creation of which has been described above). The operator also loads the mailpieces of Group 2 (from bin 26₈ from the initial pass) into the input hopper 30 of the sorter 20. TABLE 2B explains how the Group 2 mailpieces will be distributed across the bins 26. The Group 2 mailpieces from bins 26₃ - 26₆ are all to be collected for insertion in a THREE__DIGIT sack which will bear sack number 13; the Group 2 mailpieces from bins 26₇ - 26₁₉ are all to be collected for insertion into a STATE sack which will bear sack number 14.

After collecting the Group 2 mailpieces into sacks 13 and 14 in accordance with TABLE 2B, the operator loads the mailpieces for Group 3 into the input hopper 30 of the sorter 20. The sorter 20 directs the Group 3 mailpieces to the bins 26 in accordance with TABLE 2C. In this regard, TABLE 2C directs how the sacks numbered 15 through 21 inclusive are to be filled (i.e., from which bins packages are gathered for filling the respective sacks). TABLE 2C also indicates that bin 26₁₂₈ is to be further sorted as Group 4. Recall that Group 4 was not listed in TABLE 1, the reason for which is now understood. Group 4 is derived from Group 3, inasmuch as a secondary sorting pass arising from Group 3 necessitated the generation of Group 4.

After collecting the packages from the bins 26 after the running of Group 3 as indicated in TABLE 2C, the operator collects Group 4 from bin 26₁₂₈ and places the Group 4 mailpieces in the input hopper 30 of the sorter 20. TABLE 2D reflects the contents of the bins 26 after the running of the Group 4 mailpieces. From TABLE 2D it is seen that packages from bins 26₃ - 26₁₇ are also to be included in sack number 21 generated during the running of Group 4; that packages from bins 26₁₈ - 26₃₀ are to be collected together for insertion into sack number 22; and so forth through sack number 27.

Subsequent groups are run in accordance with TABLE 1 and in the manner of the foregoing discussion. TABLE 2E reflects the contents of the bins 26 upon the running of the last group, i.e. Group 86. It is thus seen that a total of 722 sacks were filled by the mailpieces run during the illustrative batch.

TABLE 3 shows a partial listing of bag tag data generated by the program ANALYZE__MAIL. The data for generating TABLE 3 is obtained from the file BAGTAG.DAT in conjunction with the table published by the USPS in the DMM. TABLE 3 reflects the contents of bag tags printed for the sacks filled in accordance with the execution of the program ANALYZE__MAIL.

Each bag tag has its first three lines of text generated in accordance with the format prescribed by the Domestic Mail Manual. In addition, a forth line of text tells the operator what group was run, and which bins to collect together for insertion into the bag. For example, the first bag tag generated for Group 1 reads:

PROVIDENCE RI 028
 3C LTRS MXD 5-DG PKG
 EVANSTON IL 602
 1:3 - 1:34 Sack: 2

5

indicating that Group 1 bins 26₃ - 26₃₄ are to be collected together for insertion into a sack bearing sack number ("bag number") "2".

Thus, using the bag tags generated by the sorter 20 as a result of the execution of program
 10 ANALYZE__MAIL, an operator can visibly determine, for each group, which bins 26 are to have their contents loaded into a given sack, as well as the sack number for that sack. Moreover, advantageously the bins having contents for the same sack are consecutively arranged (i.e., arranged in successive physical relationship), so that the operator need not jump around from bin to bin, as by walking around the large sorting machine 20, for example.

15 The Job Summary Report in TABLE 4 is produced using information from the file CLIENT.DAT, which file was produced by the function MAKE__CLIENT__COUNTS. The Job Summary Report demonstrates the accounting capabilities of the program ANALYZE__MAIL. The report is a brief summary of total fed and total reject mailpiece counts maintained by individual mailstreams.

The Postage Summary and Postage Summary by client/mailstream is display in TABLE 5 and TABLES
 20 6 - 6A, respectively. These reports are produced from information obtained from the file CLIENT.DAT. These reports demonstrate the requirements for maintaining detail counts during execution of the program ANALYZE__MAIL.

The Bag Audit Report shown in TABLE 7 demonstrates a unique advantage of the program
 25 ANALYZE__MAIL. This report is generated from information in the file ANAL__CNT.DAT. The program ANALYZE__MAIL organizes data in such a way that counts are made available by package, bag, client, mailstream, and ZIP class categories. This feature is needed to verify the accuracy of the sort process and the accounting.

While the invention has been particularly shown and described with reference to the preferred
 30 embodiments thereof, it will be understood by those skilled in the art that various alterations in form and detail may be made therein without departing from the spirit and scope of the invention. For example, although not specifically mentioned herein, it should be understood that many of the files can be written to random access memory devices, such as a magnetic disk.

35

40

45

50

55

TABLE I

GROUP INDEX FILE:

FIRST PASS RESULTS

For further sorting...
Mark the mail as follows:

BIN	GROUP #	BIN	GROUP #
3	1	56	46
4	1	57	46
5	1	58	47
6	1	59	47
7	1	60	49
8	2	61	49
9	2	62	49
10	5	63	49
11	5	64	50
12	9	65	51
13	9	66	51
14	9	67	51
15	9	68	55
16	9	69	56
17	11	70	56
18	12	71	57
19	14	72	57
20	15	73	58
21	18	74	58
22	18	75	59
23	18	76	60
24	19	77	60
25	20	78	61
26	20	79	62
27	20	80	63
28	21	81	63
29	21	82	64
30	21	83	64
31	23	84	64
32	23	85	64
33	23	86	65
34	25	87	67
35	26	88	67
36	27	89	67
37	29	90	68
38	31	91	69
39	33	92	71
40	34	93	71
41	34	94	71
42	35	95	72
43	37	96	73
44	38	98	73
45	39	100	73
46	39	102	73
47	40	97	78
48	40	99	79
49	41	101	81
50	41	103	82
51	42	104	84
52	44	105	85
53	44	106	86
54	45	107	86
55	45		

TABLE 2A

GROUP # 1
INPUT FROM: FIRST PASS
BINS: 3, 4, 5, 6, 7
The following bins will be completely sorted:

BIN	ZIP	P	B	ID	BIN	ZIP	P	B	ID	BIN	ZIP	P	B	ID	BIN	ZIP	P	B	ID	BIN	ZIP	P	B	ID
3	02806	M5	M5	2	4	02809	M5	M5		5	02813	M5	M5		6	04046	M5	M5		7	02816	M5	M5	
8	02818	M5	M5		9	02825	M5	M5		10	02828	M5	M5		11	02832	M5	M5		12	02835	M5	M5	
13	02837	M5	M5		14	02840	M5	M5		15	02852	M5	M5		16	02857	M5	M5		17	02860	M5	M5	
18	02861	M5	M5		19	02863	M5	M5		20	02864	M5	M5		21	02865	M5	M5		22	02871	M5	M5	
23	02874	M5	M5		24	02878	M5	M5		25	02879	M5	M5		26	02881	M5	M5		27	02882	M5	M5	
28	02885	M5	M5		29	02886	M5	M5		30	02888	M5	M5		31	02889	M5	M5		32	02891	M5	M5	
33	02893	M5	M5		34	02895	M5	M5		35	02903	M5	M5		36	02904	M5	M5		37	02905	M5	M5	
38	02906	M5	M5		39	02907	M5	M5		40	02908	M5	M5		41	02909	M5	M5		42	02910	M5	M5	
43	02911	M5	M5		44	02912	M5	M5		45	02914	M5	M5		46	02915	M5	M5		47	02916	M5	M5	
48	02917	M5	M5		49	02919	M5	M5		50	02920	M5	M5		51	02921	M5	M5		52	02146	M5	M5	
53	02174	M5	M5		54	021	M5	M5		55	03031	M5	M5		56	03049	M5	M5		57	03051	M5	M5	
58	030	M5	M5		59	038	M5	M5		60	04046	M5	M5		61	040	M5	M5		62	04401	M5	M5	
63	04412	M5	M5		64	04426	M5	M5		65	04468	M5	M5		66	04473	M5	M5		67	044	M5	M5	
68	01085	M5	M5		69	01701	M5	M5		70	01810	M5	M5		71	01890	M5	M5		72	01960	M5	M5	
73	010	M5	M5		74	011	M5	M5		75	012	M5	M5		76	014	M5	M5		77	015	M5	M5	
78	016	M5	M5		79	017	M5	M5		80	018	M5	M5		81	019	M5	M5		82	020	M5	M5	
83	023	M5	M5		84	024	M5	M5		85	025	M5	M5		86	026	M5	M5		87	027	M5	M5	
88	028	M5	M5		89	029	M5	M5		90	03102	M5	M5		91	03103	M5	M5		92	03104	M5	M5	
93	03275	M5	M5		94	03301	M5	M5		95	03743	M5	M5		96	03755	M5	M5		97	03773	M5	M5	
98	03989	M5	M5		99	031	M5	M5		100	032	M5	M5		101	034	M5	M5		102	035	M5	M5	
103	037	M5	M5		104	039	M5	M5		105	04210	M5	M5		106	04240	M5	M5		107	04330	M5	M5	
108	04530	M5	M5		109	04609	M5	M5		110	04736	M5	M5		111	04751	M5	M5		112	04769	M5	M5	
113	04841	M5	M5		114	04901	M5	M5		115	04976	M5	M5		116	041	M5	M5		117	042	M5	M5	
118	043	M5	M5		119	045	M5	M5		120	046	M5	M5		121	047	M5	M5		122	048	M5	M5	
123	049	M5	M5		124	PR	M5	M5																

TABLE 2B

GROUP # 2
FROM: FIRST PASS
INPUT BINS: 8

The following bins will be completely sorted:

BIN	ZIP	P	B	ID	BIN	ZIP	P	B	ID	BIN	ZIP	P	B	ID
3	05701	5	3	13	4	05753	5	3	13	7	05156	5	3	14
8	05201	5	3		9	05301	5	3		12	05819	5	3	
13	050	3	3		14	051	3	3		16	054	3	3	
18	056	3	3		19	058	3	3						

TABLE 2C

GROUP # 3
INPUT FROM: FIRST PASS
BINS: 9

The following bins will be completely sorted:

BIN	ZIP	P	B	ID	BIN	ZIP	P	B	ID	BIN	ZIP	P	B	ID
3	06268	5	5	15	4	06902	5	5	16	7	06010	5	5	19
8	06013	5	M5		9	06016	5	M5		12	06022	5	M5	
13	06026	5	M5		14	06029	5	M5		17	06035	5	M5	
18	06037	5	M5		19	06039	5	M5		22	06050	5	M5	
23	06051	5	M5		24	06052	5	M5		27	06062	5	M5	
28	06066	5	M5		29	06067	5	M5		32	06071	5	M5	
33	06073	5	M5		34	06074	5	M5		37	06081	5	M5	
38	06083	5	M5		39	06084	5	M5		42	06089	5	M5	
43	06092	5	M5		44	06093	5	M5		47	06098	5	M5	
48	06103	5	M5	18	49	06105	5	M5		52	06108	5	M5	
53	06109	5	M5		54	06110	5	M5		57	06114	5	M5	
58	06117	5	M5		59	06118	5	M5		62	06226	5	M5	
63	06231	5	M5		64	06232	5	M5		67	06238	5	M5	
68	06239	5	M5		69	06241	5	M5		72	06250	5	M5	
73	06260	5	M5		74	06269	5	M5	20	77	06280	5	M5	
78	06281	5	M5		79	06320	5	M5		82	06333	5	M5	
83	06334	5	M5		84	06335	5	M5		87	06351	5	M5	
88	06354	5	M5		89	06355	5	M5		92	06360	5	M5	
93	06370	5	M5		94	06371	5	M5		97	06378	5	M5	
98	06379	5	M5		99	06382	5	M5		102	06403	5	M5	
103	06405	5	M5		104	06410	5	M5		107	06416	5	M5	
108	06417	5	M5		109	06418	5	M5		112	06424	5	M5	
113	06426	5	M5		114	06430	5	M5		117	06441	5	M5	
118	06443	5	M5		119	06447	5	M5		122	06457	5	M5	
123	06460	5	M5		124	06468	5	M5		127	06471	5	M5	

For further sorting...
Mark the rail as follows:

BIN 128
GROUP # 4

TABLE 2E														
GROUP # 86														
INPUT FROM: FIRST PASS														
BINS: 106, 107														
The following bins will be completely sorted:														
BIN	ZIP	P	B	ID	BIN	ZIP	P	B	ID	BIN	ZIP	P	B	ID
3	99709	5	MS	710	4	99501	5	MS	711	5	99502	5	MS	
8	99505	5	MS		9	99506	5	MS		10	99507	5	MS	
13	99510	5	MS		14	99511	5	MS		15	99514	5	MS	
18	99517	5	MS		19	99518	5	MS		20	99519	5	MS	
23	99522	5	MS		24	99523	5	MS		25	99524	5	MS	
28	99574	5	MS		29	99576	5	MS		30	99577	5	MS	
33	99615	5	MS		34	99645	5	MS		35	99664	5	MS	
38	99701	5	MS	713	39	99702	5	MS		40	99703	5	MS	
43	99708	5	MS		44	99712	5	MS		45	99723	5	MS	
48	99802	5	MS		49	99803	5	MS		50	99831	5	MS	
53	99804	5	MS	715	54	99806	5	MS		55	99808	5	MS	
58	99872	3	MS		59	99800	3	MS		60	98101	3	MS	
63	99878	3	MS		64	98199	3	MS		65	981	3	MS	
68	99832	3	MS		69	983	3	MS		70	99004	3	MS	
73	99816	3	MS		74	99021	3	MS	718	75	99022	3	MS	
78	99037	3	MS		79	9900	3	MS		80	99111	3	MS	
83	991	3	MS		84	98422	3	MS	721	85	98801	3	MS	
88	99352	3	MS		89	99403	3	MS		90	984	3	MS	
93	986	3	MS		94	989	3	MS		95	992	3	MS	
98	995	3	MS		99	996	3	MS		100	997	3	MS	

EP 0 481 569 A2

TABLE 3

5	***** ** GROUP SEPARATOR ** ** Group 0 ** *****	MXD CHICAGO IL 606 3C LTRS MXD STATES EVANSTON IL 602 0:0 - 0:0 Sack: 1	***** ** GROUP SEPARATOR ** ** Group 1 ** *****	PROVIDENCE RI 028 3C LTRS MXD 5-DG PKG EVANSTON IL 602 1:3 - 1:34 Sack: 2
10	PROVIDENCE RI 028 3C LTRS MXD 5-DG PKG EVANSTON IL 602 1:3 - 1:34 Sack: 2	PROVIDENCE RI 029 3C LTRS MXD 5-DG PKG EVANSTON IL 602 1:35 - 1:51 Sack: 3	PROVIDENCE RI 029 3C LTRS MXD 5-DG PKG EVANSTON IL 602 1:35 - 1:51 Sack: 3	BOSTON MA 021 3C LTRS EVANSTON IL 602 1:52 - 1:54 Sack: 4
15	MANCHESTER NH 030 3C LTRS EVANSTON IL 602 1:55 - 1:58 Sack: 5	PORTSMOUTH NH 038 3C LTRS EVANSTON IL 602 1:59 - 1:59 Sack: 6	PORTLAND ME 040 3C LTRS EVANSTON IL 602 1:60 - 1:61 Sack: 7	BANGOR ME 044 3C LTRS EVANSTON IL 602 1:62 - 1:67 Sack: 8
20	DIS SPRINGFIELD MA 010 3C LTRS MA EVANSTON IL 602 1:68 - 1:87 Sack: 9	DIS SPRINGFIELD MA 010 3C LTRS MA EVANSTON IL 602 1:68 - 1:87 Sack: 9	SCF PROVIDENCE RI 028 3C LTRS RI EVANSTON IL 602 1:88 - 1:89 Sack: 10	DIS MANCHESTER NH 030 3C LTRS NH EVANSTON IL 602 1:90 - 1:104 Sack: 11
25	DIS PORTLAND ME 040 3C LTRS ME EVANSTON IL 602 1:105 - 1:123 Sack: 12	DIS PORTLAND ME 040 3C LTRS ME EVANSTON IL 602 1:105 - 1:123 Sack: 12	***** ** GROUP SEPARATOR ** ** Group 2 ** *****	WHITE RIVER JCT VT 057 3C LTRS EVANSTON IL 602 2:3 - 2:6 Sack: 13
30	DIS WHITE RIVER JCT 050 3C LTRS VT EVANSTON IL 602 2:7 - 2:19 Sack: 14	***** ** GROUP SEPARATOR ** ** Group 3 ** *****	STORRS MANSFIELD CT 06268 3C LTRS EVANSTON IL 602 3:3 - 3:3 Sack: 15	STAMFORD CT 06902 3C LTRS EVANSTON IL 602 3:4 - 3:4 Sack: 16
35	HARTFORD CT 060 3C LTRS MXD 5-DG PKG EVANSTON IL 602 3:5 - 3:47 Sack: 17	HARTFORD CT 060 3C LTRS MXD 5-DG PKG EVANSTON IL 602 3:5 - 3:47 Sack: 17	HARTFORD CT 061 3C LTRS MXD 5-DG PKG EVANSTON IL 602 3:48 - 3:61 Sack: 18	HARTFORD CT 062 3C LTRS MXD 5-DG PKG EVANSTON IL 602 3:62 - 3:78 Sack: 19
40	NEW HAVEN CT 063 3C LTRS MXD 5-DG PKG EVANSTON IL 602 3:79 - 3:100 Sack: 20	NEW HAVEN CT 063 3C LTRS MXD 5-DG PKG EVANSTON IL 602 3:79 - 3:100 Sack: 20	NEW HAVEN CT 064 3C LTRS MXD 5-DG PKG EVANSTON IL 602 3:101 - 4:17 Sack: 21	NEW HAVEN CT 064 3C LTRS MXD 5-DG PKG EVANSTON IL 602 3:101 - 4:17 Sack: 21
45				
50				
55				

TABLE 4

Client/Mailstream Count Summary based on first pass results
 Client (002): Insurance

	FED	REJECTS
(000) Auto	156267	3608
(001) Life	1478	121
(003) Health	12654	114

Client Sub-Total	170399	3843
------------------	--------	------

Client (003): Utility Co.

	FED	REJECTS
(000) Service bills	66385	954

Client Sub-Total	66385	954
------------------	-------	-----

Client (006): Publisher

	FED	REJECTS
(000) Magazine #3	32834	2225

Client Sub-Total	32834	2225
------------------	-------	------

TABLE 5

MASTER POSTAGE SUMMARY METERED AND PERMIT COMBINED
Based upon First Pass Counters

5 DIGIT LEVEL	PIECE COUNT	PER PIECE RATE	COST
ZIP+4 Barcoded	5	0.132	0.660
ZIP+4	0	0.132	0.000
5 DIGIT	179034	0.132	23632.488
5 Digit Total	179039		23633.148

BASIC RATE LEVEL	PIECE COUNT	PER PIECE RATE	COST
ZIP+4 Barcoded	0	0.167	0.000
ZIP+4	0	0.167	0.000
5 DIGIT	83557	0.167	13954.019
Pass 1 Non-Scan	7022	0.167	1172.674
Basic Total	90579		15126.693

5 Digit and Basic	269618		38759.840
-------------------	--------	--	-----------

Qualifying Percentage (Rejects Excluded): 68
 Percentage ZIP+4 Barcoded (Rejects Excluded): 0
 Percentage ZIP+4 (Rejects Excluded): 0

TABLE 6

POSTAGE SUMMARY FOR CLIENT (2): Insurance
Based upon First Pass Counters

PER PIECE METERED RATE by Mailstream

(000)Auto 0.127
(001)Life 0.000
(003)Health 0.000

5

10

15

20

25

30

35

40

45

50

55

5 DIGIT LEVEL	Mailstream	PIECE COUNT	PER PIECE RATE	COST
ZIP+4 Barcoded	(000)Auto	5	0.132	0.660
	(001)Life	0	0.132	0.000
	(003)Health	0	0.132	0.000
	total	5		0.660
ZIP+4	(000)Auto	0	0.132	0.000
	(001)Life	0	0.132	0.000
	(003)Health	0	0.132	0.000
	total	0		0.000
5 DIGIT	(000)Auto	128212	0.132	16923.984
	(001)Life	1074	0.132	141.768
	(003)Health	10988	0.132	1450.416
	total	140274		18516.168
5 Digit Total		140279		18516.828

BASIC RATE LEVEL	Mailstream	PIECE COUNT	PER PIECE RATE	COST
ZIP+4 Barcoded	(000)Auto	0	0.167	0.000
	(001)Life	0	0.167	0.000
	(003)Health	0	0.167	0.000
	total	0		0.000
ZIP+4	(000)Auto	0	0.167	0.000
	(001)Life	0	0.167	0.000
	(003)Health	0	0.167	0.000
	total	0		0.000
5 DIGIT	(000)Auto	24442	0.167	4081.814
	(001)Life	283	0.167	47.261
	(003)Health	1552	0.167	259.184
	total	26277		4388.259
Pass 1 Non-Scan	(000)Auto	3608	0.167	602.536
	(001)Life	121	0.167	20.207
	(003)Health	114	0.167	19.038
	total	3843		641.781

TABLE 6A

5

10

15

20

25

30

35

40

45

50

55

Postage Due (Permit)	14132	1937.874
(Permit Postage Adjusted to exclude rejects)		
Postage Due (Permit Rejects Excluded)	13897	1898.629
Postage Due (Meter)		21608.994
Metered Postage Paid	156267	19845.910
Additional Metered Postage Due		1763.084
(Metered Postage Adjusted to exclude rejects)		
Postage Due (Meter)		21006.458
Metered Postage Paid	152659	19387.694
Additional Metered Postage Due		1618.764
Qualifying Percentage (Rejects Excluded):	84	
Percentage ZIP+4 Barcoded (Rejects Excluded):	@	
Percentage ZIP+4 (Rejects Excluded):	@	

Claims

1. A sorter apparatus comprising:
 - hopper means into which a plurality of mailpieces-to-be-sorted are introduced;
 - reader means for reading postage address destination information provided on said mailpieces-to-be-sorted;
 - a plurality of bins into which said mailpieces are sorted;
 - conveying means for conveying said mailpieces from said hopper means, past said reader means, and toward said plurality of bins;
 - gating means for directing a mailpiece into a selected one of said plurality of bins;
 - control means for controlling said gating means, said control means being connected to said reader means for obtaining therefrom signals indicative of said destination information for said mailpieces, said control means including logic means for determining how said mailpieces should be classified according to predetermined criteria into packages of mailpieces and how said packages of mailpieces

should be classified according to predetermined criteria into sacks, with said control means connected to control the activation of said gating means whereby mailpieces classifiable in a sack but stored by reason of said criteria in a plurality of bins are gated into a plurality of physically adjacent bins without destroying the package classification of said mailpieces.

- 5 **2.** The apparatus of claim 1, further comprising:
 means responsive to said control means for generating labels for attachment to said sacks, said labels bearing information including the bins from which mailpieces should be extracted for inclusion in a sack.
- 10 **3.** The apparatus of claim 2, wherein said labels also bear a sack number.
- 15 **4.** The apparatus of claim 1, wherein upon an initial reading pass of each of said mailpieces-to-be-sorted through the sorter, said control means associates said mailpieces into groups by gating the mailpieces of each group into a set of bins corresponding to their group, said sets of bins having a plurality of bins being comprised of physically adjacent bins of the sorter, and wherein said control means provides an indication of which bins contain each group of mailpieces.
- 20 **5.** The apparatus of claim 4, wherein said control means associates said mailpieces into groups for the purpose of subsequently feeding each group separately into said hopper means for a subsequent pass through said sorter.
- 25 **6.** The apparatus of claim 5, wherein said control means associates said mailpieces into groups and by gating the mailpieces of each group into an associated set of bins, each of said groups being assigned a group number and each bin having a bin number, and whereby, as the group numbers monotonically increase during said assignment, the bin numbers included in the associated set also monotonically increase.
- 30 **7.** A method of sorting mailpieces comprising:
 introducing a plurality of mailpieces-to-be-sorted into a hopper means;
 conveying said mailpieces from said hopper means, past said reader means, and toward a plurality of bins;
 reading postage address destination information provided on said mailpieces-to-be-sorted using said reader means;
 35 classifying said mailpieces, according to predetermined criteria, into packages of mailpieces and in turn classifying packages of mailpieces, according to predetermined criteria, into sacks;
 using gate means to direct mailpieces into selected ones of said plurality of bins;
 whereby said classification whereby mailpieces classifiable in a sack but directed by reason of said criteria in a plurality of bins are gated into a plurality of physically adjacent bins without destroying the package classification of said mailpieces.
- 40 **8.** The method of claim 7, further comprising:
 generating labels for attachment to said sacks, said labels bearing information including the bins from which mailpieces should be extracted for inclusion in a sack.
- 45 **9.** The method of claim 8, wherein said labels also bear a sack number.
- 50 **10.** The method of claim 7, wherein upon conveying said mailpieces on an initial reading pass through the sorter, said mailpieces are associated into a plurality of groups by gating the mailpieces of each group into a set of bins corresponding to their group, said sets of bins having a plurality of bins being comprised of physically adjacent bins of the sorter, and wherein an indication is provided regarding the bins which contain each group of mailpieces.
- 55 **11.** The method of claim 10, wherein said mailpieces are associated into groups for the purpose of subsequently feeding each group separately into said hopper means for a subsequent pass through said sorter.
- 12.** The method of claim 11, wherein said mailpieces are associated into groups by gating each of the

mailpieces of each group into an associated set of bins, each of said groups being assigned a group number and each bin having a bin number, and whereby, as the group numbers monotonically increase during said assignment, the bin numbers included in the associated set also monotonically increase.

5

10

15

20

25

30

35

40

45

50

55

FIG. 1

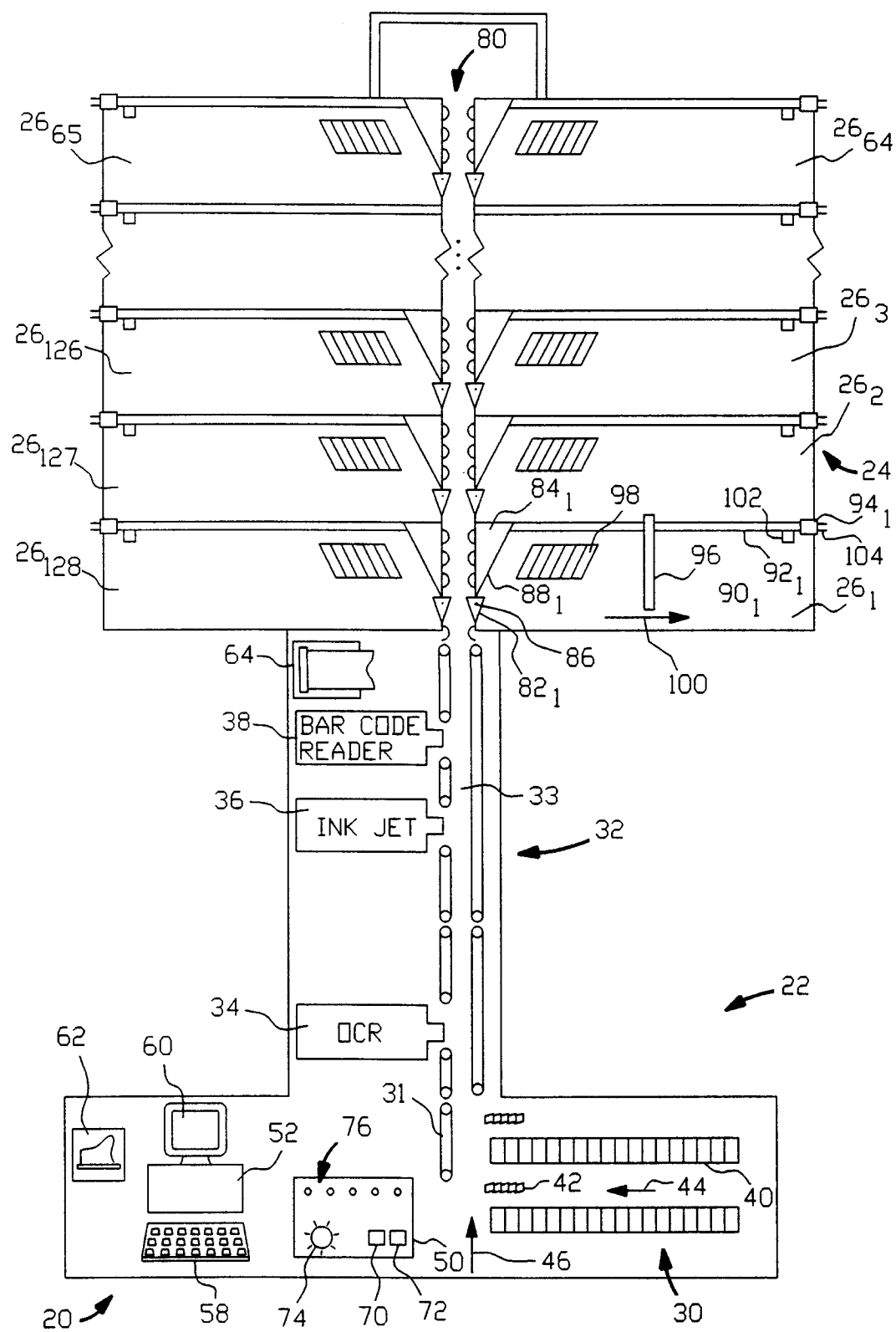


FIG. 2

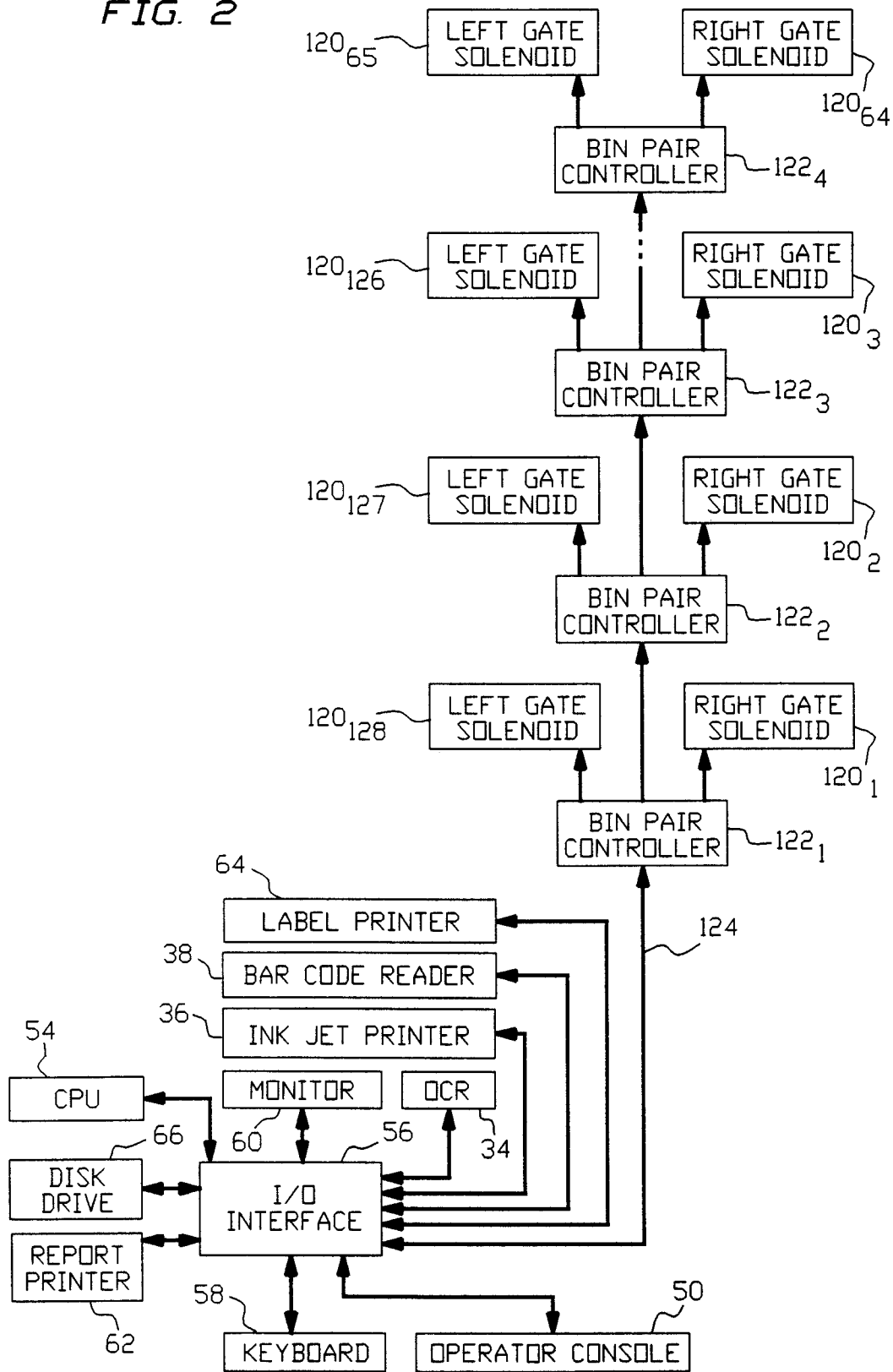


FIG. 3

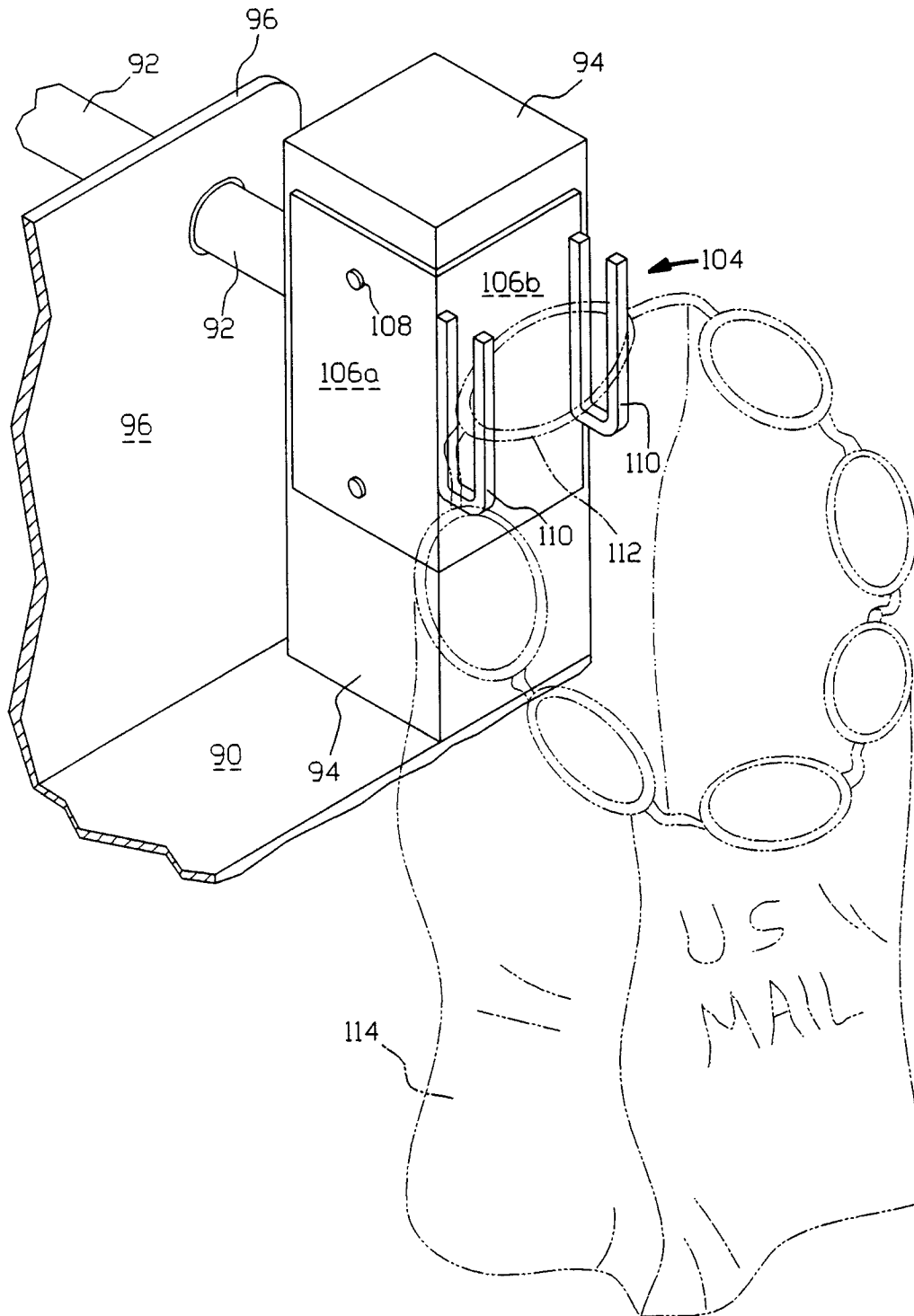


FIG. 4

FIG. 4A	FIG. 4B	FIG. 4C	FIG. 4D1	FIG. 4D2	FIG. 4E	FIG. 4F	FIG. 4G	FIG. 4H	FIG. 4I	FIG. 4J1	FIG. 4J2	FIG. 4K	FIG. 4L	FIG. 4M1	FIG. 4M2	FIG. 4N
------------	------------	------------	-------------	-------------	------------	------------	------------	------------	------------	-------------	-------------	------------	------------	-------------	-------------	------------

FIG. 4A

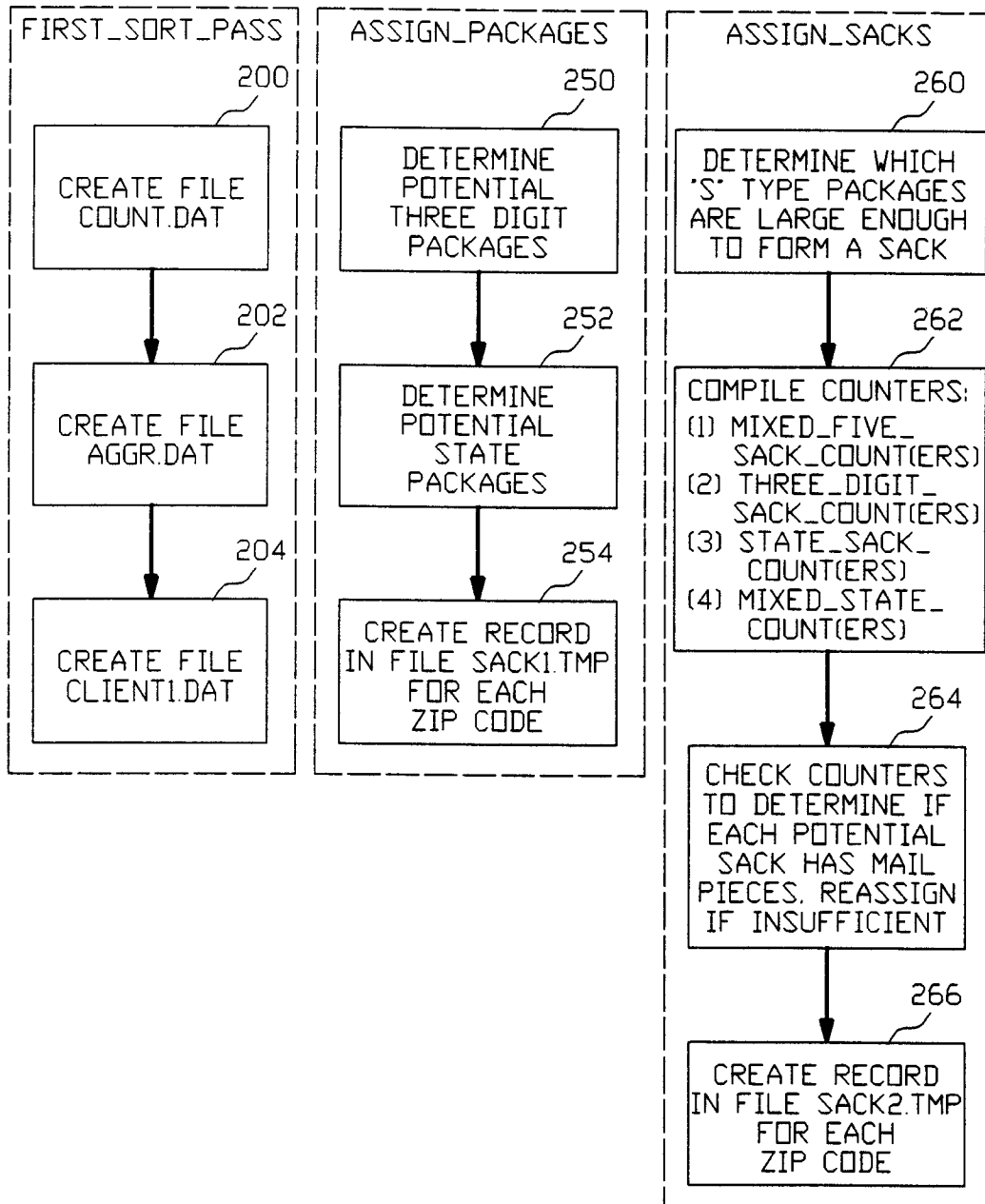


FIG. 4B

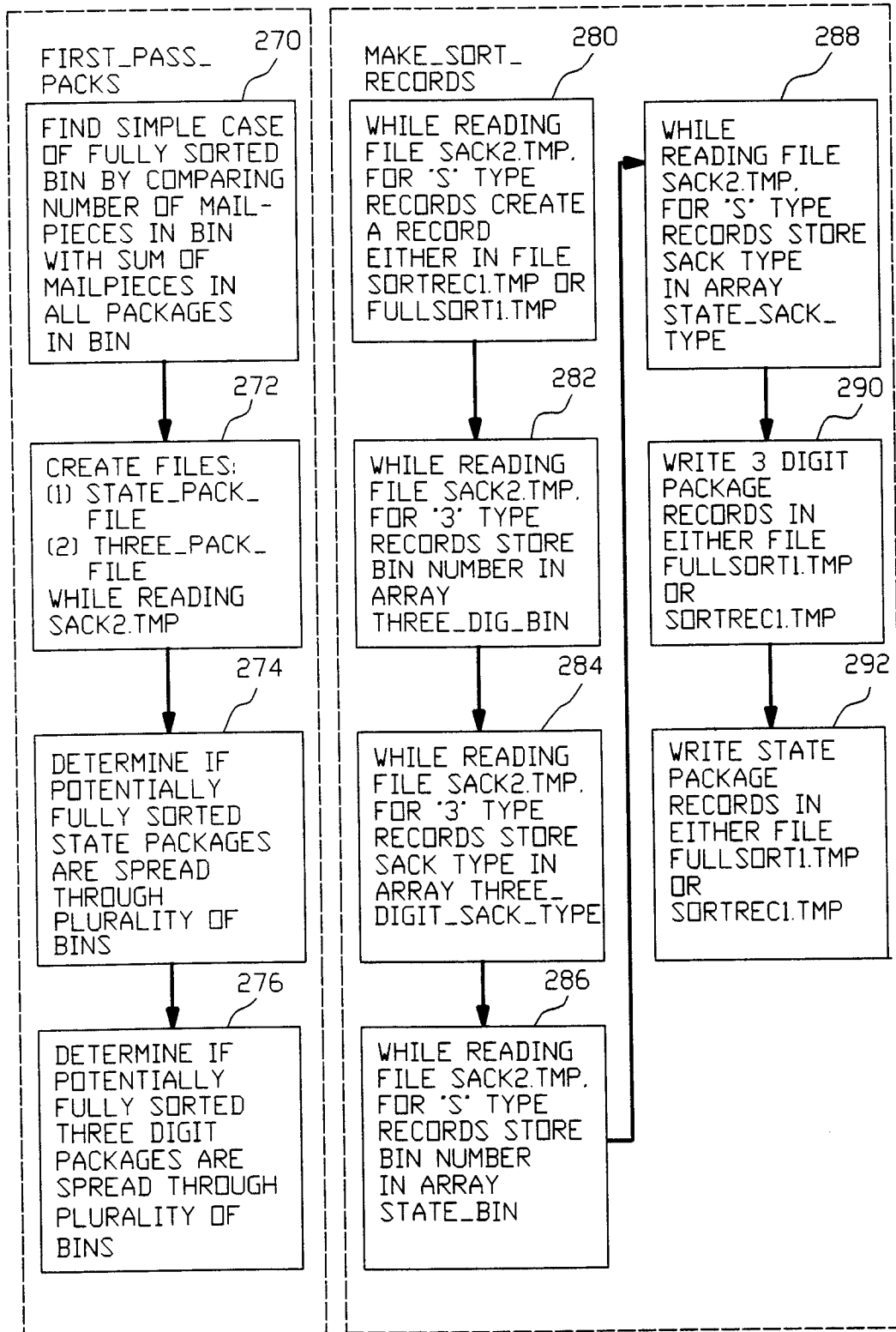


FIG. 4C

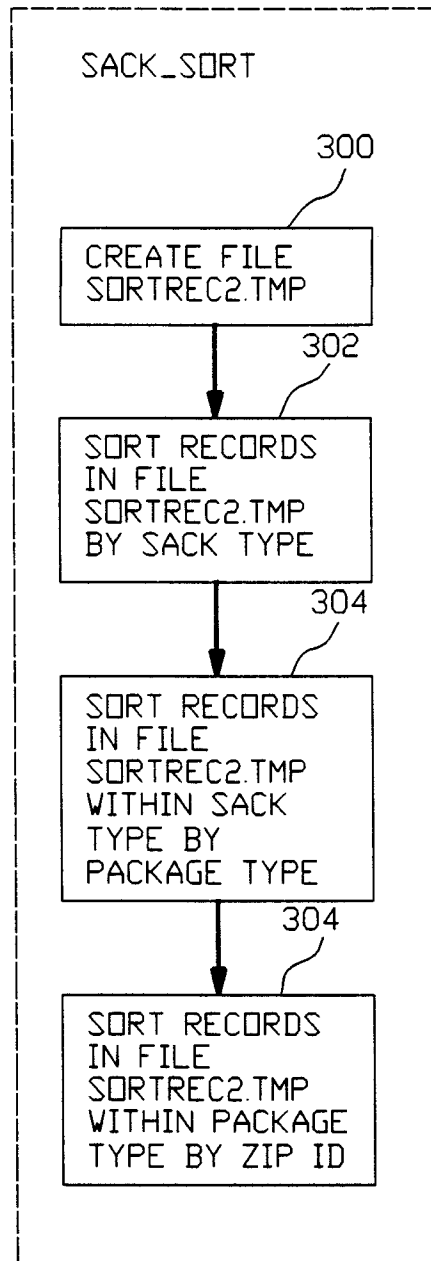


FIG. 4D(1)

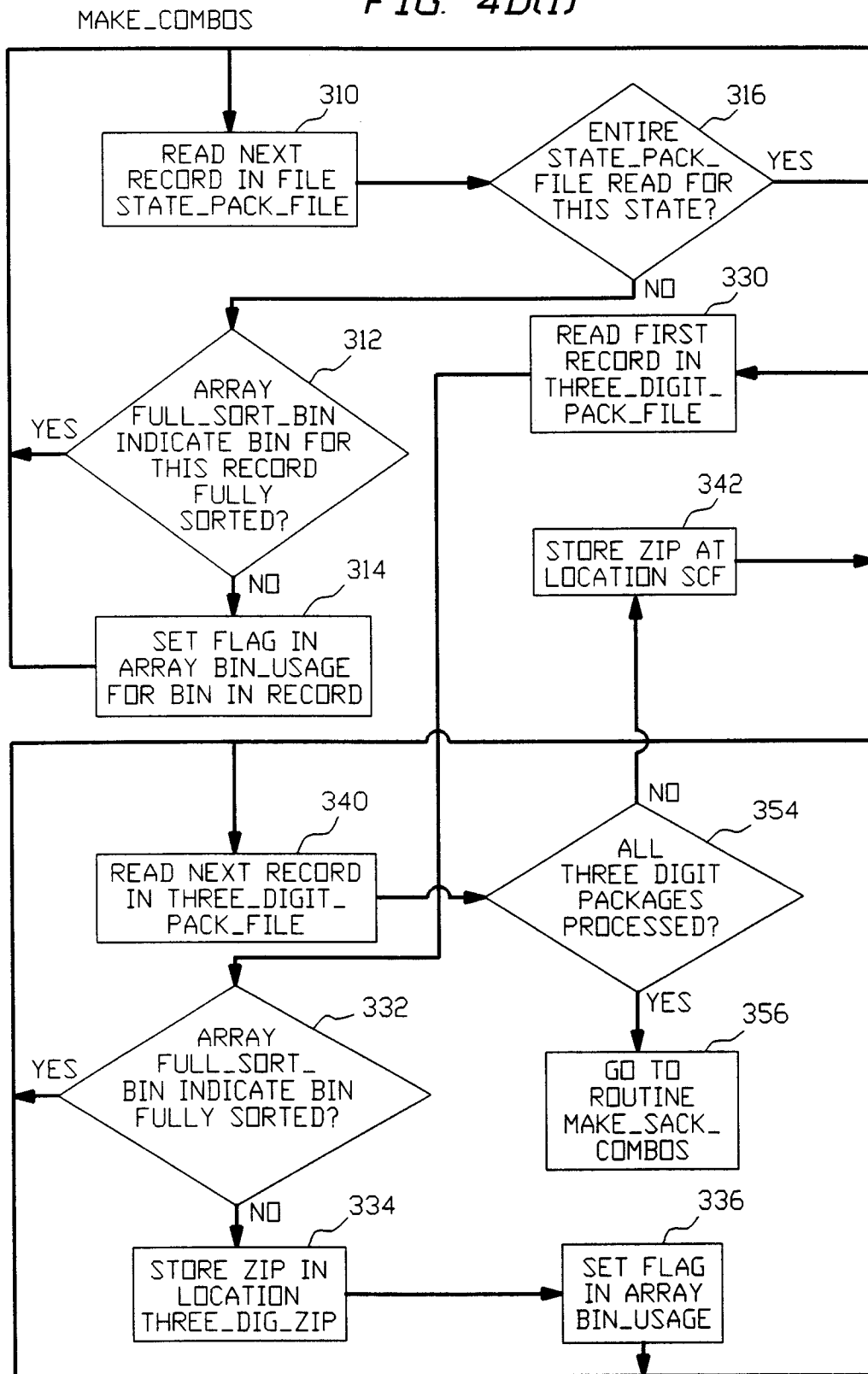


FIG. 4D(2)

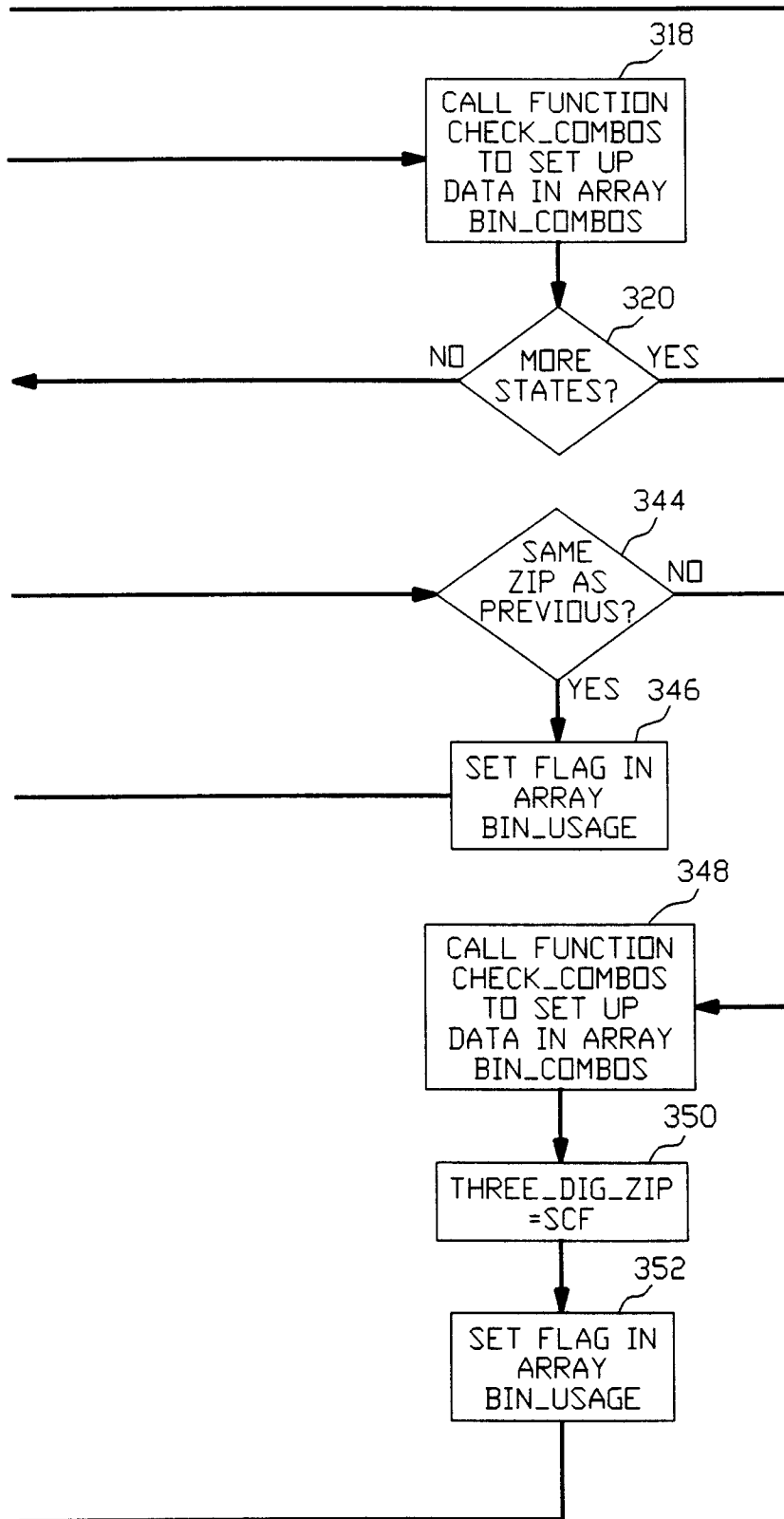


FIG. 4E

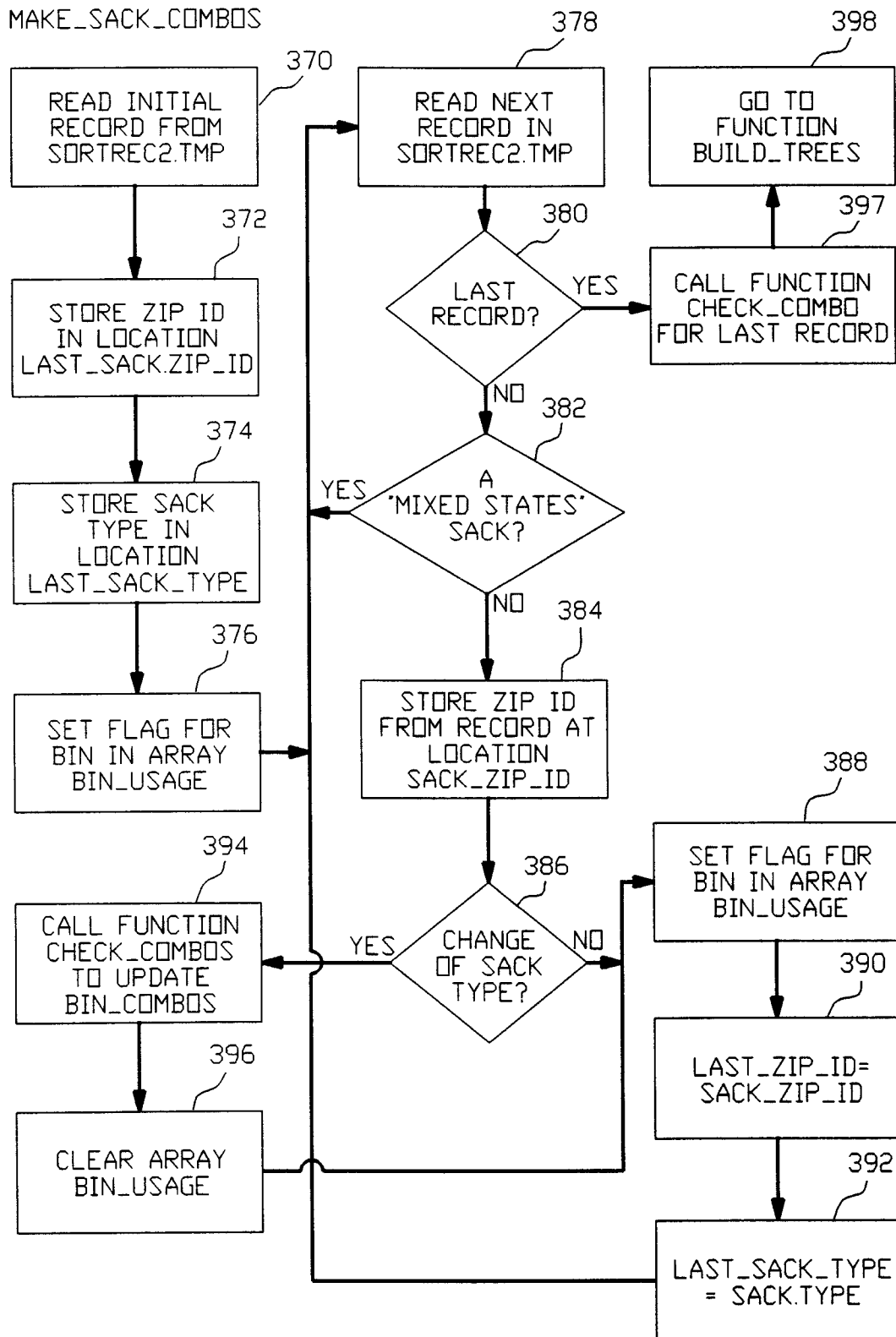


FIG. 4F

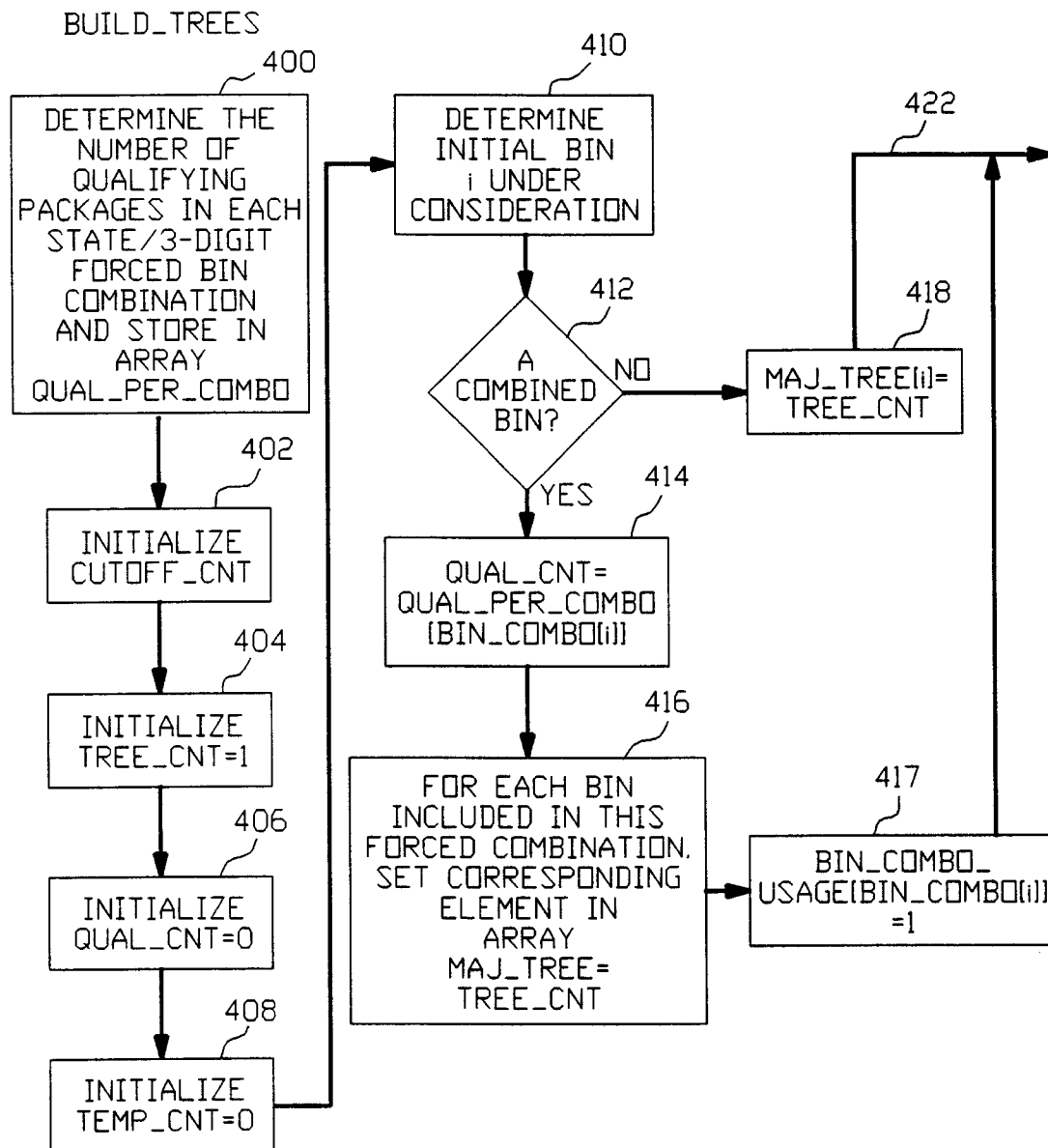


FIG. 4G

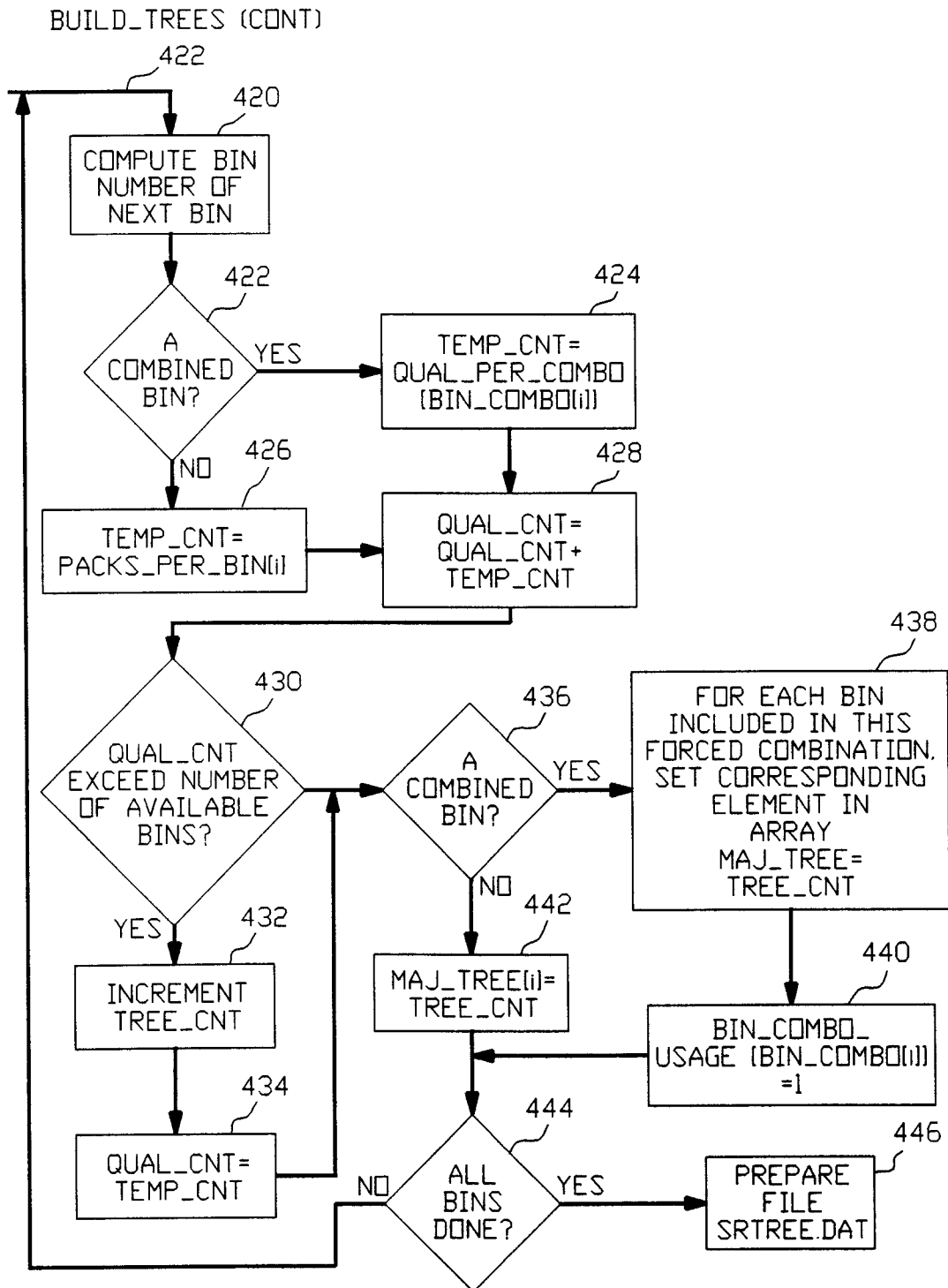


FIG. 4H

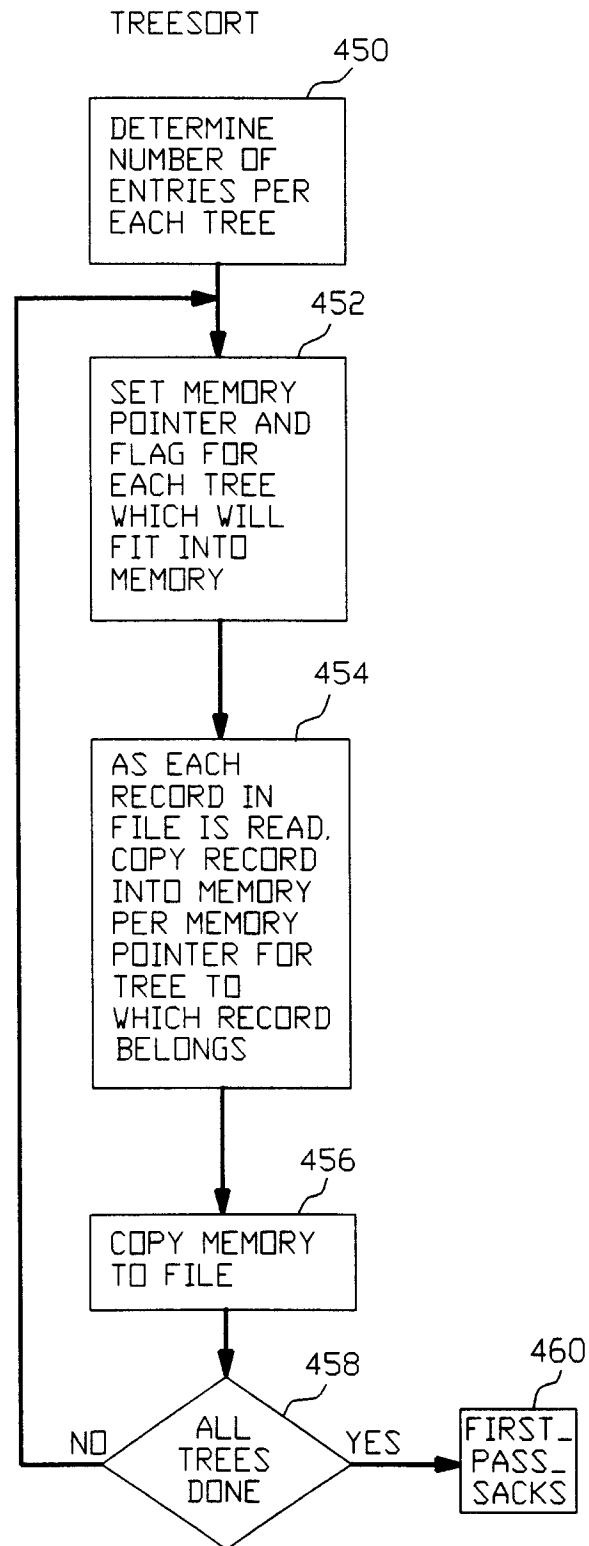


FIG. 4I

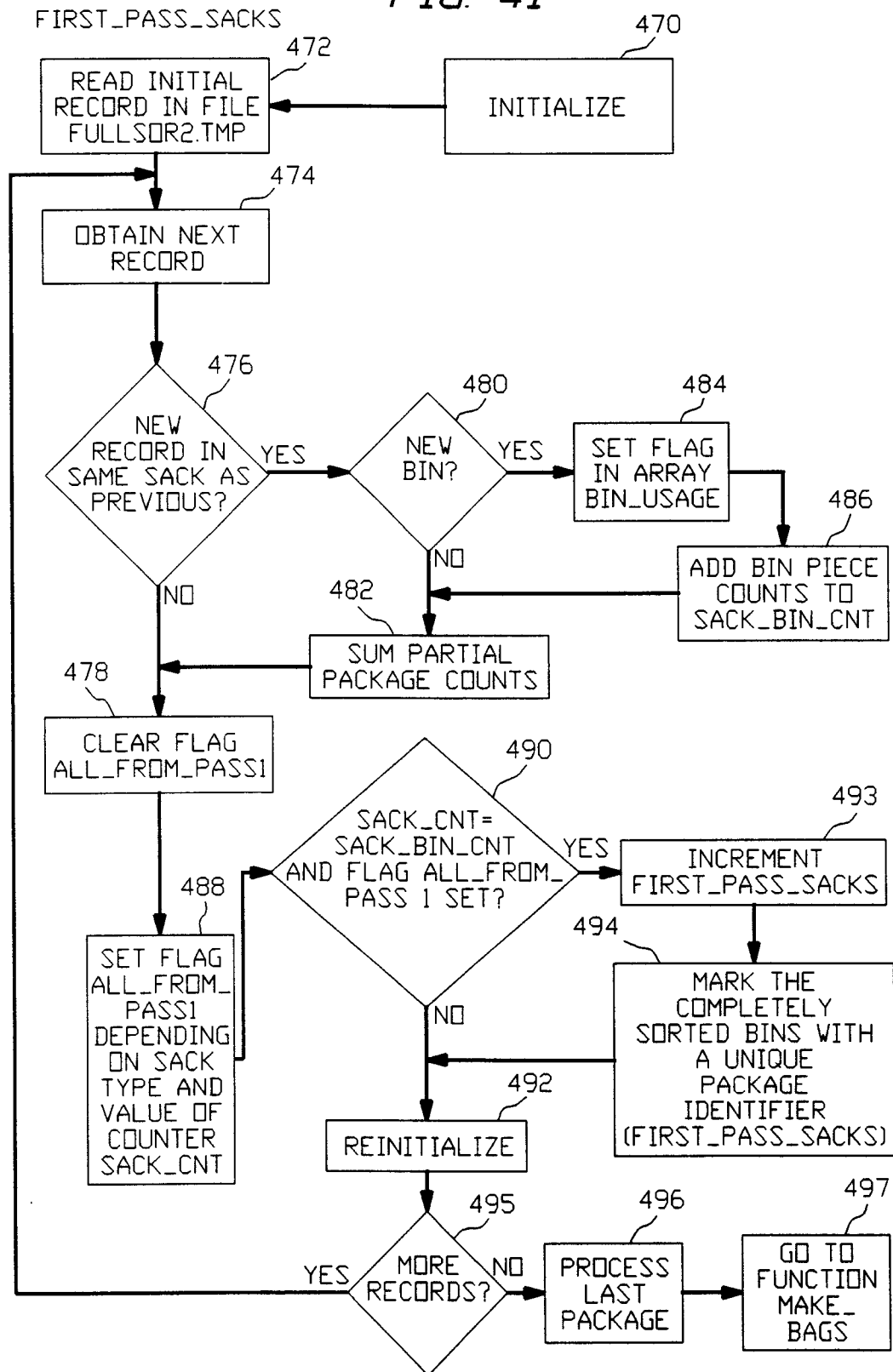


FIG. 4J1

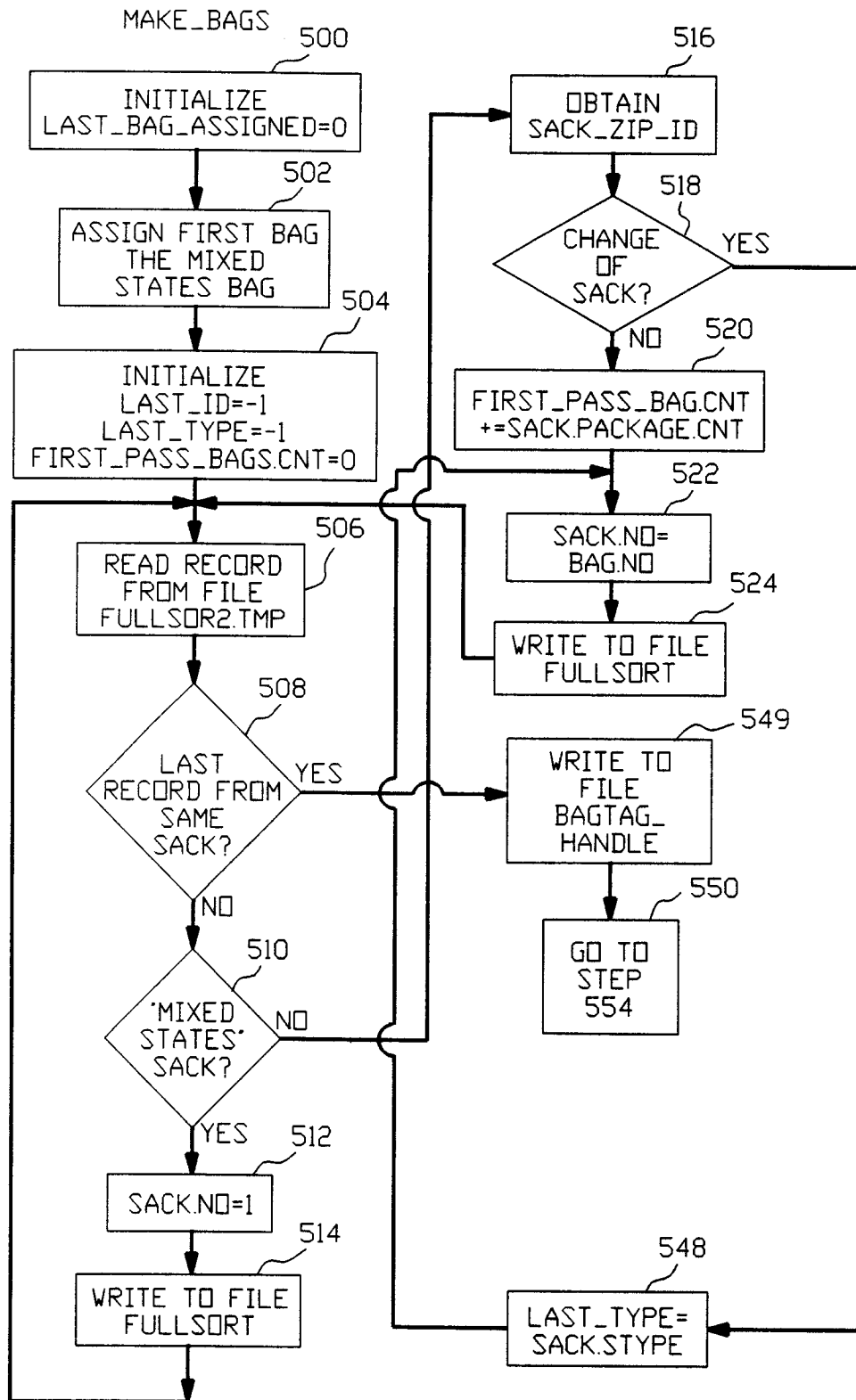


FIG. 4J2

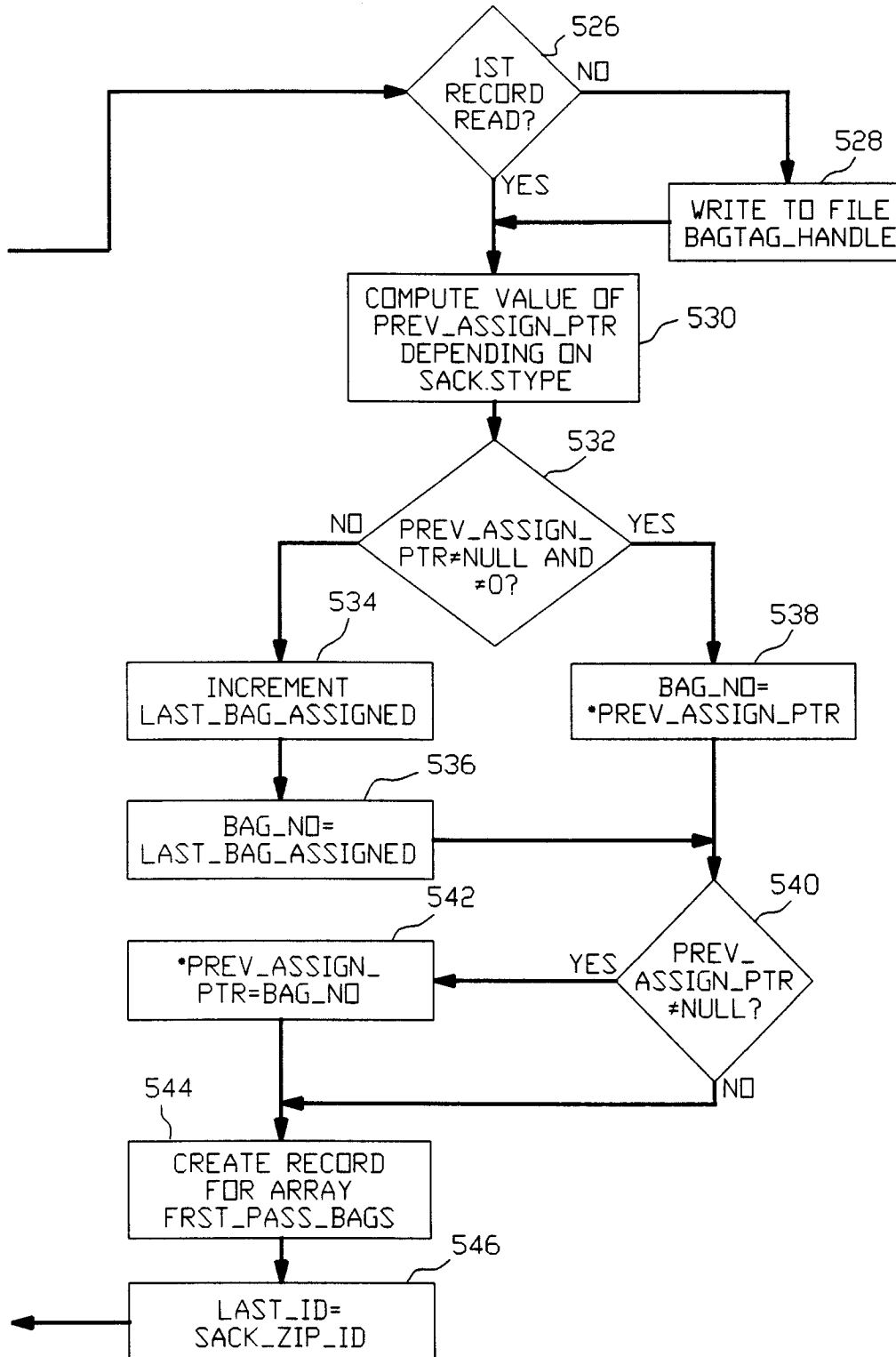


FIG. 4K

MAKE_BAGS (CONTINUED)

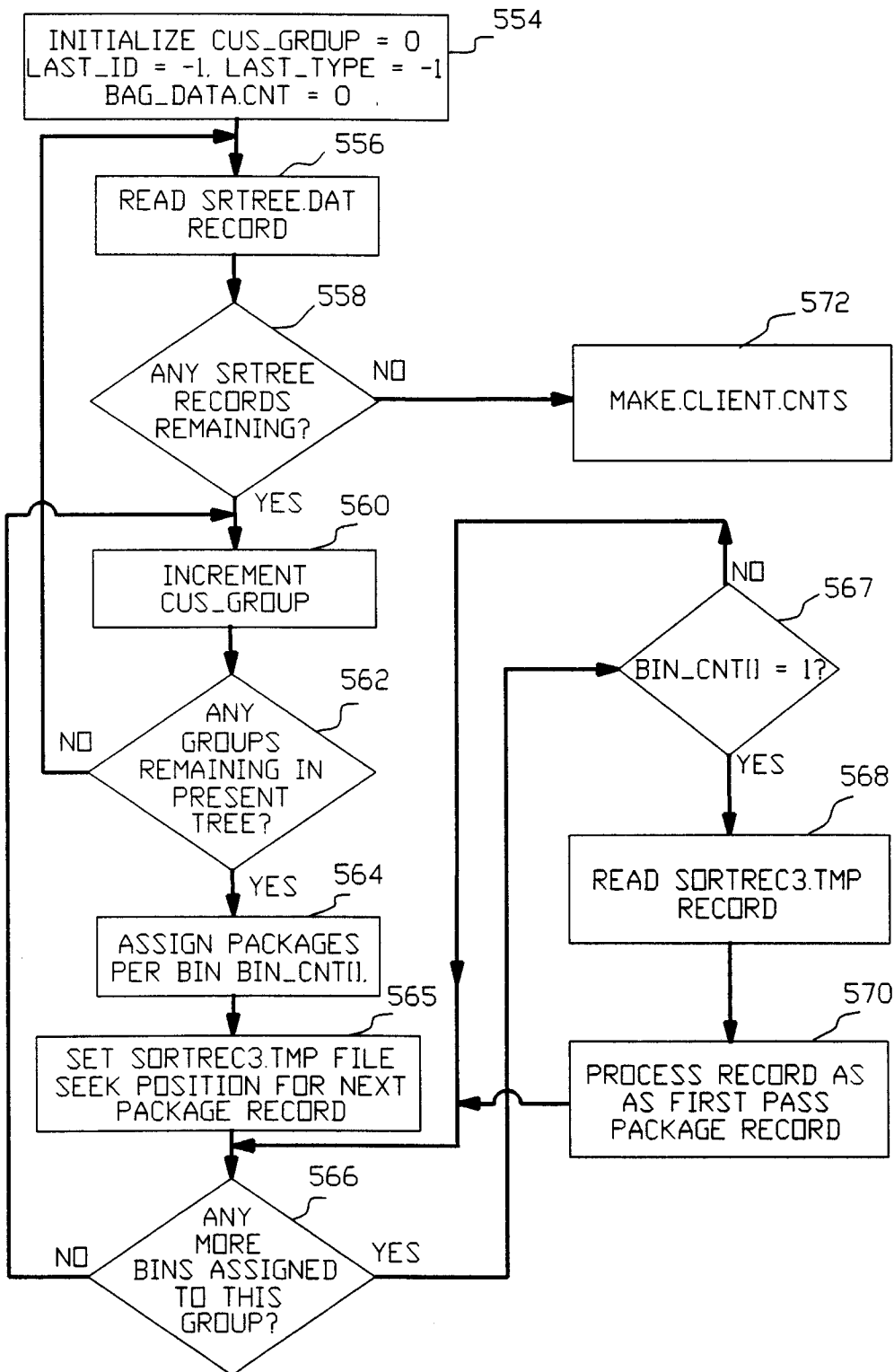


FIG. 4L

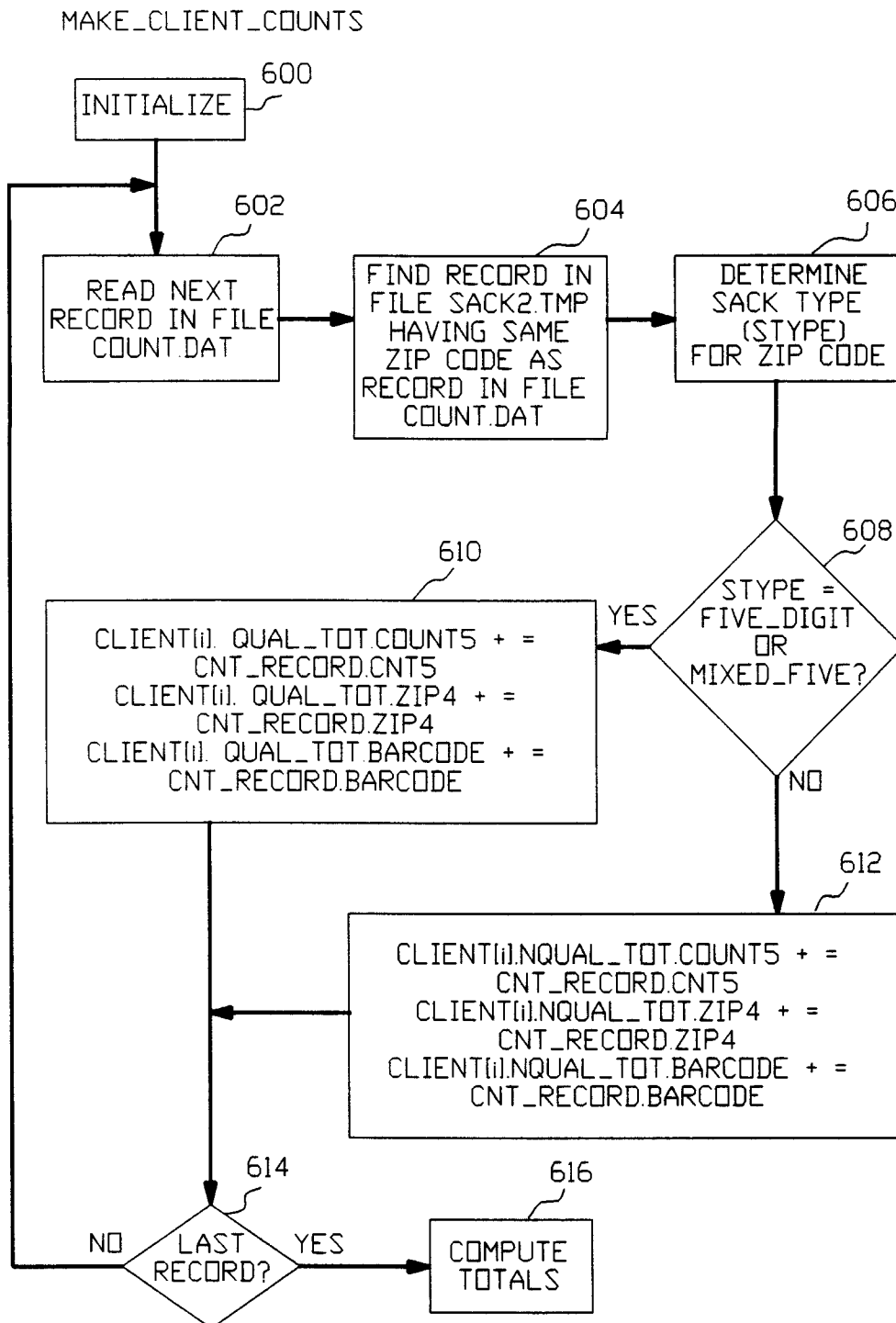


FIG. 4M1

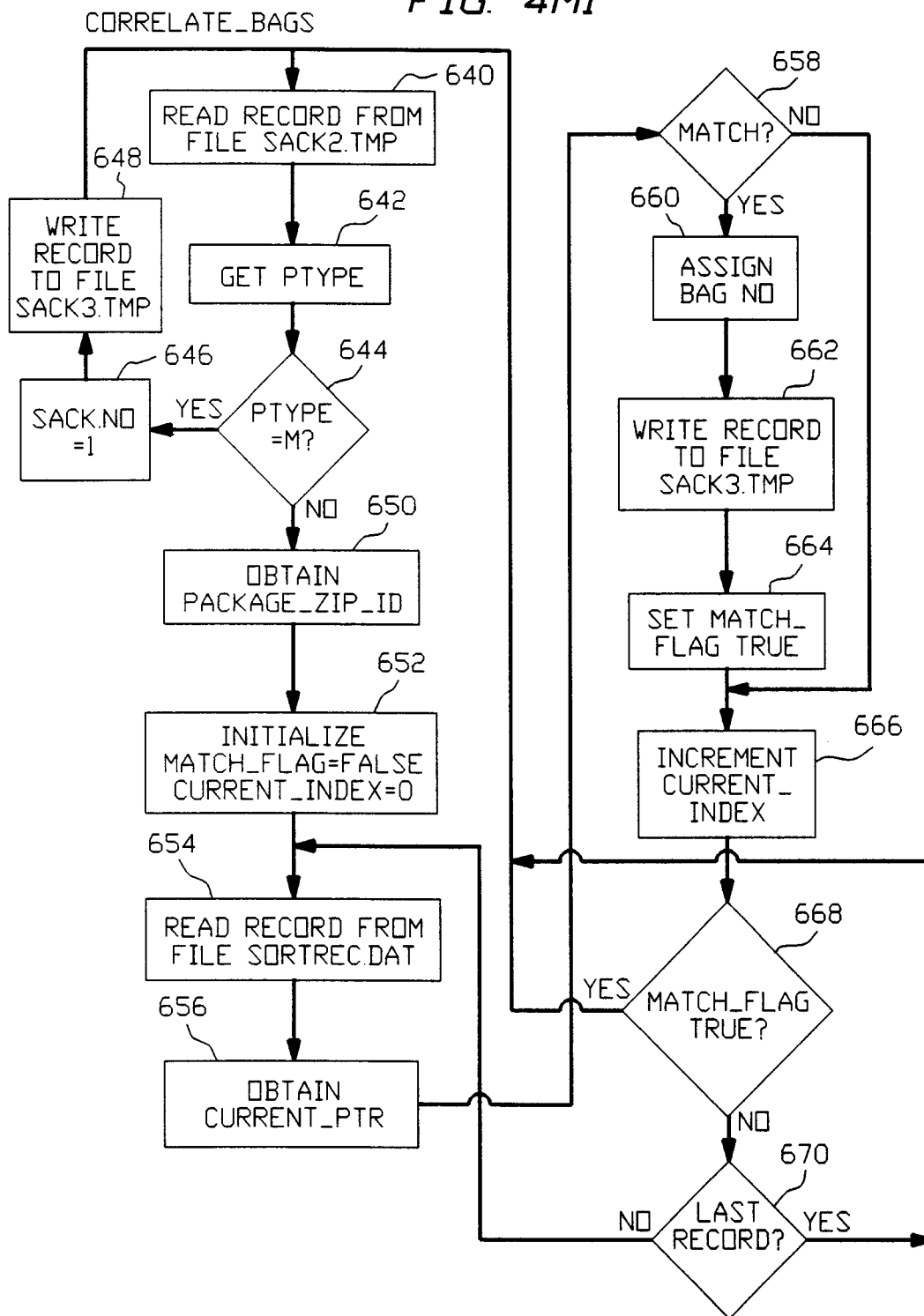


FIG. 4M2

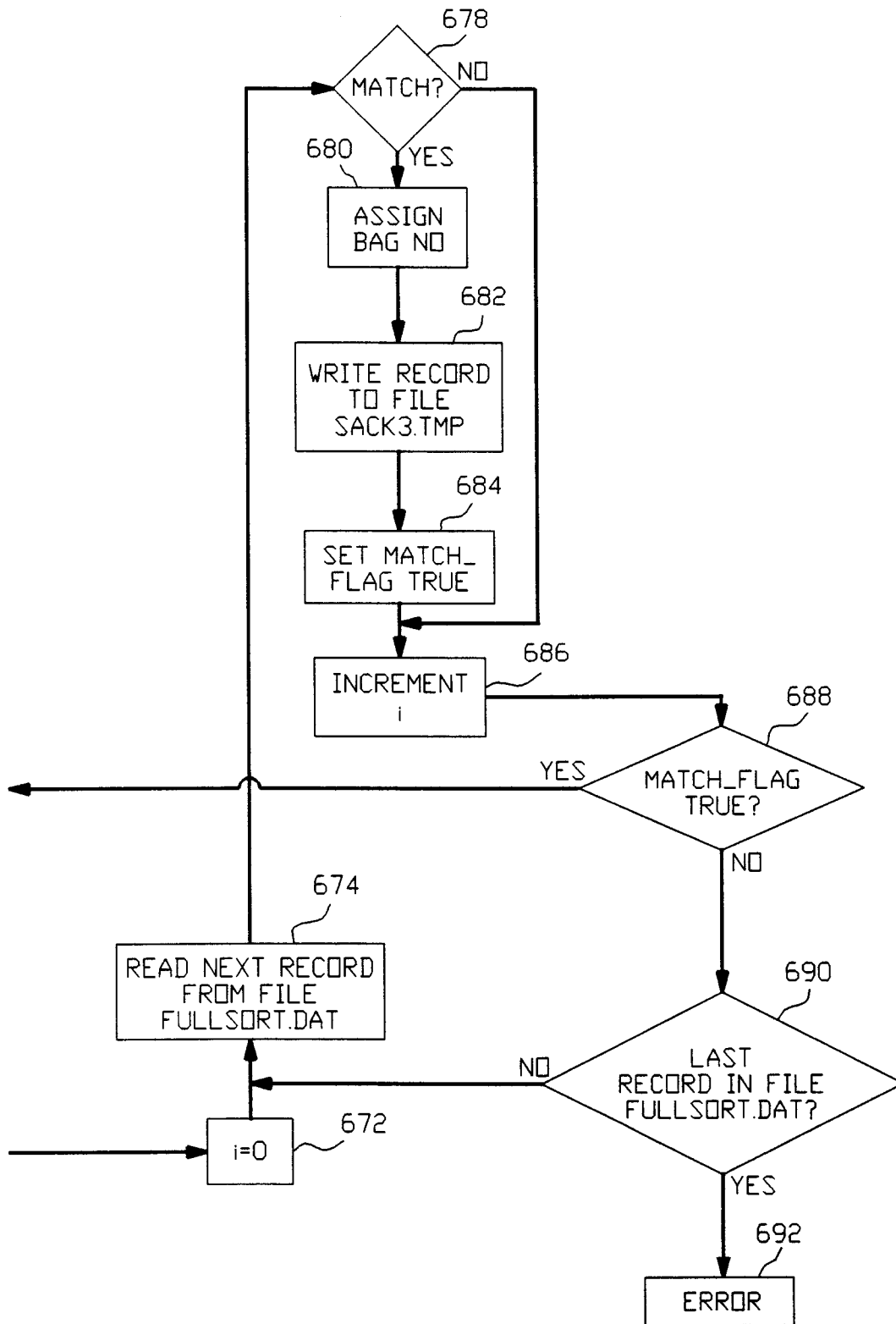


FIG. 4N

