A COMPUTER-BASED ACCOUNTABILITY SYSTEM (PHASE I) FOR SPECIAL NUCLEAR MATERIALS AT ARGONNE-WEST

by

R. S. Ingermanson* and A. E. Proctor

Argonne-West Division

May 1982

*Department of Physics, University of California, Berkeley
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FLOWCHARTS</td>
<td>4</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>5</td>
</tr>
<tr>
<td>I. INTRODUCTION</td>
<td>5</td>
</tr>
<tr>
<td>II. DEVELOPMENT OF ACCOUNTABILITY SYSTEM, PHASE I</td>
<td>6</td>
</tr>
<tr>
<td>A. Accountability Program Concepts</td>
<td>6</td>
</tr>
<tr>
<td>B. Applications of an Indexing and Retrieval System</td>
<td>8</td>
</tr>
<tr>
<td>1. Batch and Item Considerations</td>
<td>8</td>
</tr>
<tr>
<td>2. An Example: Capsule C309</td>
<td>9</td>
</tr>
<tr>
<td>C. Software Design</td>
<td>10</td>
</tr>
<tr>
<td>D. Program Operations</td>
<td>14</td>
</tr>
<tr>
<td>E. File Descriptions</td>
<td>15</td>
</tr>
<tr>
<td>III. SUMMARY</td>
<td>19</td>
</tr>
<tr>
<td>APPENDIX A - PROGRAM FLOWCHARTS</td>
<td>21</td>
</tr>
<tr>
<td>Data Storage and Retrieval</td>
<td>21</td>
</tr>
<tr>
<td>1. Program Summary</td>
<td>21</td>
</tr>
<tr>
<td>2. Flowchart Conventions</td>
<td>22</td>
</tr>
<tr>
<td>3. Documentation of Subroutines</td>
<td>25</td>
</tr>
<tr>
<td>4. Program Descriptions Flowcharts</td>
<td>25</td>
</tr>
<tr>
<td>APPENDIX B - INITIALIZATION PROCEDURE</td>
<td>108</td>
</tr>
<tr>
<td>APPENDIX C - OPERATING ENVIRONMENT</td>
<td>112</td>
</tr>
<tr>
<td>APPENDIX D - PROGRAM DEVELOPMENT EFFORT</td>
<td>113</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>113</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>114</td>
</tr>
<tr>
<td>LIST OF FLOWCHARTS</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Data Storage and Retrieval Program</td>
<td>27</td>
</tr>
<tr>
<td>GetANumber</td>
<td>29</td>
</tr>
<tr>
<td>Accept</td>
<td>31</td>
</tr>
<tr>
<td>YesOrNo</td>
<td>33</td>
</tr>
<tr>
<td>MatchIsotopes</td>
<td>35</td>
</tr>
<tr>
<td>Unpack</td>
<td>37</td>
</tr>
<tr>
<td>GetListfile</td>
<td>39</td>
</tr>
<tr>
<td>ReadBucket</td>
<td>41</td>
</tr>
<tr>
<td>FileRecord</td>
<td>43</td>
</tr>
<tr>
<td>Copy</td>
<td>45</td>
</tr>
<tr>
<td>Compare</td>
<td>47</td>
</tr>
<tr>
<td>Locate</td>
<td>51</td>
</tr>
<tr>
<td>FindOpen</td>
<td>55</td>
</tr>
<tr>
<td>Store</td>
<td>57</td>
</tr>
<tr>
<td>Retrieve</td>
<td>59</td>
</tr>
<tr>
<td>WriteRecord</td>
<td>61</td>
</tr>
<tr>
<td>GetBatch</td>
<td>63</td>
</tr>
<tr>
<td>GetSerial</td>
<td>65</td>
</tr>
<tr>
<td>BigRemove</td>
<td>67</td>
</tr>
<tr>
<td>SmallRemove</td>
<td>69</td>
</tr>
<tr>
<td>BigCompress</td>
<td>71</td>
</tr>
<tr>
<td>SmallCompress</td>
<td>73</td>
</tr>
<tr>
<td>BigSqueeze</td>
<td>75</td>
</tr>
<tr>
<td>SmallSqueeze</td>
<td>77</td>
</tr>
<tr>
<td>GetIsotopes</td>
<td>79</td>
</tr>
<tr>
<td>GetDescription</td>
<td>81</td>
</tr>
<tr>
<td>BitPack</td>
<td>83</td>
</tr>
<tr>
<td>DatePack</td>
<td>85</td>
</tr>
<tr>
<td>GetCharacteristics</td>
<td>89</td>
</tr>
<tr>
<td>GetInfo</td>
<td>91</td>
</tr>
<tr>
<td>PrintList</td>
<td>93</td>
</tr>
<tr>
<td>Update</td>
<td>95</td>
</tr>
<tr>
<td>Delete</td>
<td>97</td>
</tr>
<tr>
<td>Modify</td>
<td>99</td>
</tr>
<tr>
<td>Mix</td>
<td>101</td>
</tr>
<tr>
<td>HashBatch</td>
<td>103</td>
</tr>
<tr>
<td>HashDesc</td>
<td>105</td>
</tr>
<tr>
<td>HashSerial</td>
<td>107</td>
</tr>
<tr>
<td>InitFiles</td>
<td>110</td>
</tr>
<tr>
<td>Bucket</td>
<td>111</td>
</tr>
</tbody>
</table>
A COMPUTER-BASED ACCOUNTABILITY SYSTEM (PHASE I)
FOR SPECIAL NUCLEAR MATERIALS AT ARGONNE-WEST

by

R. S. Ingermanson and A. E. Proctor

ABSTRACT

An automated accountability system for special nuclear materials (SNM) is under development at Argonne National Laboratory-West. Phase I of the development effort has established the following basic features of the system: A unique file organization allows rapid updating or retrieval of the status of various SNM, based on "batch numbers," storage location, serial number, or other attributes. Access to the program is controlled by an interactive user interface that can be easily understood by operators who have had no prior background in electronic data processing. Extensive use of structured programming techniques make the software package easy to understand and to modify for specific applications. All routines are written in FORTRAN.

I. INTRODUCTION

Department of Energy requirements for safeguarding special nuclear materials (SNM) require that a "near-real time" accountability system be established to track SNM throughout its usage in a nuclear facility. The accountability procedures in use at Argonne National Laboratory-West (ANL-W) are based on a manual "CARDEX" filing system containing data on shipping, transfer, and storage. These methods, which are normally satisfactory, are now being strained by the volume of information which must be stored, the need to manually retrieve the data necessary to generate each transfer document, and the usual problems encountered when up-to-date information must be supplied for transfers of thousands of items.
To overcome these problems, development of an automated accountability system capable of rapidly storing and recalling data on large numbers of items was begun in 1980. Ideally, this new system would perform all the functions of the manual system more quickly and with less effort than is now required. Since the system would be used by staff with little or no computer background, a "friendly" user interface was mandatory. Most potential users already had access to a CRT terminal which could be used for interactive access to this system. When necessary, printed output would be generated at either the computer-room printer or a remote printer located in one of the material-handling areas. Lastly, the software would need to be in a modular format, so that the system could be updated without rewriting large sections of code.

Phase I of the development effort has established the basic features of such a system, as described below. Phase II will involve modifying the interactive part of the software to make it readily usable by people who are not trained in programming. It will further involve writing a user's manual and putting the system into operation at ANL-W.

II. DEVELOPMENT OF ACCOUNTABILITY SYSTEM, PHASE I

A. Accountability Program Concepts

Our objectives in developing accountability software were: (a) to design an efficient data storage and retrieval system, (b) to write interactive user-oriented routines to manipulate the stored data, and (c) to produce document-printing software and programs to meet the needs of specific areas, types of transfers, etc. (This approach is the opposite of that taken when document printing is considered most important.) The information-storage programs are the universal routines; these are called by all "user" programs whether those that accept the input data or those that print the output. From the operator's standpoint, the ease with which data can be entered and retrieved is the primary benefit of using the automated system.
We found it necessary to deviate from the "batch" oriented software prevalent in many accountability systems. Instead, we provide data storage for individual items. This approach obviously increases storage file sizes, but greatly improves the system's handling of individual transfers.

The SNM automated accountability program is an "item-based" system, which is compatible with the existing accountability procedures. Information concerning an identifiable can, plate, or capsule is stored for each item in storage or use. Important attributes, such as mass, serial number, and location, are included with each item entry. These attributes need not be the same for any two items, and may be changed (updated) as the item is used in various ANL-W programs. An item-based system conveniently covers most SNM handling operations. We had noted that the most common identifiers used during SNM accountability were the "batch numbers," 16-character numbers indicating such information as where the item originated, and transfer-document numbers, which were assigned as needed. From the material requestor's viewpoint, however, item serial numbers and fissile-material mass were most important. We concluded that an automated accountability system must provide for sophisticated retrieval based on a variety of item attributes, as well as storage of large quantities of information.

We accomplished retrieval of both item information and batch information by using (a) a large item file, based on a "hash" storage algorithm using batch numbers; and (b) a separate "characteristic" file, which is searched sequentially. An entry into the characteristic file automatically leads to the item file by use of a simple pointer calculation. Preliminary testing has shown that this arrangement is an effective method for data handling when extensive use of "secondary keys" (i.e., item characteristics) is required. The original accounting documents utilize primary-key (batch-number) searching; while later updating would most likely use a secondary-key search to locate individual serial numbers, specific isotopes, etc. (See Ref. 1 for further information on search methods.)
The user interface was constructed around the familiar interactive "menu" format in which the user is prompted to select a command from a list. The command list is displayed automatically, or in response to the user's request (HELP command). Once the user has selected a command, the interactive main program calls a functional subroutine to execute the command. The subroutine has commands and prompts of its own; the user's response to these determines what is done. The results of each operation are displayed at the user's terminal and printed either locally or on the computer-room printer.

At present the automated accountability system is completely general and a user can perform any available operation. Minimal password access is provided. As a result, the system is very adaptable, which is desirable at this point. Special restrictions may be easily added to the interactive main program without altering the functional software. Subsequent versions of this program will include initial password access, restrictions on operations for some users, and storage-area restrictions.

B. Applications of an Indexing and Retrieval System

1. Batch and Item Considerations

All items and groups of items stored in the present accountability system are identified by an "internal identification number," often referred to as a "batch" number. This number consists of four parts: the first three digits are derived from the DOE project number associated with the material. A three-digit "material-type" code is added to indicate the predominant isotope present. An internally generated "batch" number and a "breakout" number (a subdivision of a batch) are added. This complete batch identification, recorded in the SNM files, is used when items are transferred at ANL-W.

Transfers between ANL-W and other facilities reference SNM on a batch basis. The recipient of such materials, however, is interested in specific items, such as "PuO₂/UO₂ capsule No. 39," or "fuel element
AB12," or "can containing 50 g $^{235}$U metal turnings." A single batch usually contains many items, each having a different name or serial number, and possibly a unique application in some particular program. Internal accountability procedures have been adapted to the need for extensive item-batch cross-referencing; however, the manual methods currently used are time consuming.

A second problem arises when the items initially received as a single batch are separated during storage, use, and (ultimately) disposal. It is probable that several storage locations will contain items from a single batch. Other items from the same batch may have been previously built into reactor subassemblies for irradiation, placed in Experimental Breeder Reactor II (EBR-II), and later reassembled into another subassembly, until, finally, they are returned to the experimenter for disposal. Thus, there are two other identifiers of interest: subassembly (or storage drum, etc.), and the building or facility in which the item is currently stored.

Specific information concerning the item's fissile-material content is carried with each item: "element," such as uranium, thorium, etc.; "element mass," i.e., the total mass of any selected element; and "isotope mass," the mass of fissile isotope present. This information is used for criticality safety calculations and to comply with various DOE safeguards requirements.

2. An Example: Capsule C309

Let us look at the record-keeping associated with a typical EBR-II test capsule. Consider a hypothetical uranium-plutonium oxide capsule, C309. Assume that the capsule was received at ANL-W on May 13, 1981. The capsule was part of a larger batch shipment of 10 capsules (numbered C301-C310), each of which contained 50 g total uranium ($33 \text{ g } ^{235}\text{U}$), and 46 g total plutonium ($29 \text{ g } ^{239}\text{Pu}$). The 10 capsules were noted on shipping document No. 10014 and initially entered in the computerized inventory system by "JGW." Later, JGW's assistant, "EDX," updated the
file to show that capsule C309 was stored in drum 32, also checking the total contents of drum 32 to ensure that the fissile content was within the nuclear safety storage limits. This latter operation was done by retrieving a list of every item in drum 32. Later, capsule C309 was requested by an experimental group for fabrication into subassembly No. 504. JDW entered this transaction, including the current transfer document number.

After irradiation, capsule C309 was transferred to a hot-cell facility for examination. It was necessary to update the mass values to account for burnup and to change the capsule's location, again including the current transfer-document number. This was done by representative "ABC" in the hot-cell area. The previous subassembly-number and storage-location attributes were also deleted. When the in-cell examination was completed, capsule C309 was returned to the original shipper. The entry for C309 was then deleted from the ANL-W accountability records.

The above process required two off-site transfers, two internal transfers, and several updates. Without an automated file system, each storage area and user would keep individual records of item locations, batches, and item serials. Cross-referencing is cumbersome. With an automated system, information on all of the above categories can be found rapidly and with minimal user effort.

C. Software Design

The primary problem to be solved was to find a way to rapidly retrieve information by several types of indexing. Information may be desired on the following general indices:

<table>
<thead>
<tr>
<th>Index Type</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch number</td>
<td>Obtain a list of all items in a single batch.</td>
</tr>
<tr>
<td>Serial number</td>
<td>Obtain information on a single item with a given serial number.</td>
</tr>
<tr>
<td>Isotopic content</td>
<td>Obtain a list of all items containing an arbitrary set of isotopes.</td>
</tr>
</tbody>
</table>
Subassembly number - Obtain a list of all items belonging to a given subassembly.

User - Obtain a list of all items in the possession of a given user of special materials.

Date - Obtain a list of all items that were received in a given month and year.

Clearly, it is wasteful to store all the information in six different files. We decided to index complete information by batch number in a very large disk file called the batch file. Smaller files could be created to serve as pointers to the main file, for the other indices. Note that the serial number is unique to each item, while the other four indices are very broad categories. This prompted us to create two small files: the serial file, for serial numbers, and the characteristic file, for the other four indices.

The information stored in the batch file is as follows: batch number, serial number, subassembly number, isotope names and masses (up to eight such mass/data groups), the receipt document number, the most recent document number, the user of the material, and the initials of the person who put the information in file and when.

With each batch, a description is included, which applies to the batch as a whole. To avoid duplication, we decided to place the description in a separate file, the description file, indexed by batch number. This file has also been useful for maintaining information on how many items are in the batch.

All of the files mentioned so far are random-access disk files, in which information can be updated or deleted at will. To provide a measure of security (against carelessness as well as sabotage) a fifth file was created. This file, the archives file, is a sequential file to be saved on magnetic tape. It contains a record of all information ever stored in the batch file, and all changes or deletions ever made. Each record
contains a few words to indicate whether the record was being stored, modified, or deleted, and who was responsible, the date, and the address in the batch file.

In order to make retrieval as fast as possible, we decided to store and retrieve data using "hash" functions. The idea behind the hash function is simple. The computer performs some operation on the "key" -- the batch or serial number -- which generates an address in the file at which to store the data. Note that the batch number is 16 digits long, i.e., about 10 orders of magnitude greater than our disk capacity. The hash function must obviously do a lot of condensing. Also note that the serial number may contain numbers or letters. The hash function should be able to handle both.

It can be shown that even in a small collection of keys, the probability is high that two or more will hash to the same number. There are many methods of handling such "collisions." The method we chose was to divide the file (in our imagination) into a large number of "buckets," each containing several records. The hash function is used to assign a bucket to each key. The buckets are made small enough so that each can be treated as a miniature sequential file.

Normally, the hash function is very similar to a pseudo-random number generator. Assuming that an arbitrary key has an equal probability of hashing to any bucket, the expected number of records in an arbitrary bucket obeys a Poisson distribution. By choosing the number of buckets carefully, the probability that any bucket will ever be filled can be made very small. For those few buckets which do fill up, a general overflow area is provided at the end of the file. This consists of a few buckets of the same size as those in the main file.

It is useful to know how many records are in a given bucket. We have adopted the convention that the 64th word in each bucket should contain the number of records in the bucket. All random-access files
have been designed to have a little unused space. The file-handling
routines are set up to read or write 64-word records. Hence, by reading
only one record, the number of records in the bucket can be determined.
The 64th word of each record is unused, in all files.

As mentioned before, a batch may have up to several thousand items.
Obviously, these cannot all fit into buckets of any reasonable size.
Therefore, an item number is appended to the batch number of each item in
the batch file. This number is unique within a batch. The batch number
and item number are hashed together. This spreads a batch out more
evenly among the buckets in the batch file. For a given batch, the most
recently assigned item number is saved in the description file, where it
is referred to as "totalitems."

The serial file contains records with only five words of data. It
is well to note the difference between true record size and logical
record size. The true record size is the number of words necessary for a
single item, in this case, five words. The logical record size is always
64 words. In the source code, "recsize" always refers to the true record
size, "bucksize" always refers to the number of logical records in the
bucket. For the serial file, each bucket is one logical record, containing
12 true records plus four unused words at the end.

There are three different hash functions. For the description file,
the batch number is hashed. For the serial file, the serial number is
hashed. For the batch file, the batch number and item number are simul-
taneously hashed.

The characteristic file is handled differently. Whenever data are
stored, updated, or deleted in the batch file, data are stored, updated,
or deleted in the characteristic file. There exists a one-to-one corres-
pondence between each record and each bucket in the two files. This
enables us to conduct a brute-force search of the characteristic file
which is effectively a brute-force search of the batch file. If we find
an interesting item in the characteristic file, we can get the complete information at the same record and bucket in the batch file. Since the true record size for the characteristic file is only eight words (two of which are unused), a brute-force search can be made much quicker. Note that a high probability exists of finding at least one successful "match" for each record of the characteristic file because the characteristic categories are rather broad. A sequential search of the characteristic file is as efficient as other search methods, and considerably simpler than most.

D. Program Operations

There are four essential operations we would like to perform on our data. We would like to store information, retrieve it, update it, and eventually delete it. In addition, the program must be easy to learn to use, should check for mistakes by the user, and should have the capacity to be extended easily.

In order to make the program as simple as possible for the user to learn, it is set up in a prompt-response format. Several subroutines exist solely to obtain some particular type of data from the user, with appropriate error checking. These include Accept, Getanumber, Getdescription, Getinfo, Getisotopes, Getlistfile, and Yesorno.

The majority of the subroutines are general-purpose utility routines to perform a single task. Some of these are much more general than necessary, with an eye to future extensions. The utility routines are of two types: simple-processing and input/output routines.

Simple-processing routines perform central-processor-bound tasks that are often needed. These routines include BitPack, Compare, Copy, DatePack, HashBatch, HashDesc, HashSerial, MatchIsotopes, Mix, and Unpack. Like the data-obtaining subroutines, they are usually short, straightforward, and obvious.
Input/output routines involve either file manipulation or printing output to the user. These include BigCompress, BigRemove, BigSqueeze, FileRecord, FindOpen, Locate, PrintList, ReadBucket, SmallCompress, SmallRemove, SmallSqueeze, and WriteRecord. A few of these are trivial. Most are carefully glued together and may fall apart if modified indiscriminately.

The remainder of the subroutines are of the put-it-all-together variety. Much of their donkey work is done by the utility routines, so they are fairly straightforward, although inclined to be long-winded. They include Store, Retrieve, GetBatch, GetSerial, GetCharacteristics, Update, Modify, and Delete. They generally contain some user interaction, some processing, and some simple file handling. They are less dangerous to alter than the utility routines, but still fairly perilous.

A more complete survey of these routines is given in Appendix A.

E. File Descriptions

The accountability system requires five disk-resident data files: batch, description, serial, characteristic, and archive files. The organization of each of these files is described in the following tables. The "Current File Allocations" are those used for a demonstration version of this program.

**Batch File**

Logical Unit No.: 500
Bucket Size: 24 Records
Record Size: 128 Bytes
Record Format:
<table>
<thead>
<tr>
<th>Field (bytes)</th>
<th>Information</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-16</td>
<td>Batch number</td>
<td>ASCII**</td>
</tr>
<tr>
<td>17-20</td>
<td>Item number</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>21-28</td>
<td>Serial number</td>
<td>ASCII</td>
</tr>
<tr>
<td>29-34</td>
<td>Subassembly number</td>
<td>ASCII</td>
</tr>
<tr>
<td>35-50</td>
<td>Element abbreviations</td>
<td>ASCII</td>
</tr>
<tr>
<td>51-82</td>
<td>Masses</td>
<td>REAL*4</td>
</tr>
<tr>
<td>83-90</td>
<td>Receipt document number</td>
<td>ASCII</td>
</tr>
<tr>
<td>91-98</td>
<td>Latest document number</td>
<td>ASCII</td>
</tr>
<tr>
<td>99-104</td>
<td>Date of last change</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td>105-108</td>
<td>Initials of last person</td>
<td>ASCII</td>
</tr>
<tr>
<td>109-112</td>
<td>User of the material</td>
<td>ASCII</td>
</tr>
</tbody>
</table>

**Description File**

Logical Unit No.: 600
Bucket Size: 24 Records
Record Size: 128 Bytes
Record Format:

<table>
<thead>
<tr>
<th>Field (bytes)</th>
<th>Information</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-16</td>
<td>Batch number</td>
<td>ASCII</td>
</tr>
<tr>
<td>17-20</td>
<td>Total items</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>21-24</td>
<td>Actual items</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>25-104</td>
<td>Description</td>
<td>ASCII</td>
</tr>
</tbody>
</table>

Serial File

Logical Unit No.: 700
Bucket Size: 128 Words
Record Size: 10 Bytes
Record Format:

<table>
<thead>
<tr>
<th>Field (Bytes)</th>
<th>Information</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-8</td>
<td>Serial number</td>
<td>ASCII</td>
</tr>
<tr>
<td>9-10</td>
<td>Bucket number on the Batch file</td>
<td>INTEGER*2</td>
</tr>
</tbody>
</table>

Characteristic File

Logical Unit No.: 800
Bucket Size: 384 Words
Record Size: 16 Bytes
Record Format:

<table>
<thead>
<tr>
<th>Field (Bytes)</th>
<th>Information</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-4</td>
<td>Isotopic content</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>5-6</td>
<td>User, CAN-coded**</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td>7-10</td>
<td>Storage location or subassembly number, CAN-coded</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>11-12</td>
<td>Date of original entry</td>
<td>INTEGER*2</td>
</tr>
</tbody>
</table>

Archives File

Logical Unit No.: 900
Record Size: 64 words
Record Format:

**Compressed alphanumeric (a means to fit three characters into two bytes).
<table>
<thead>
<tr>
<th>Field (bytes)</th>
<th>Information</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-16</td>
<td>Batch number</td>
<td>ASCII</td>
</tr>
<tr>
<td>17-20</td>
<td>Item number</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>21-28</td>
<td>Serial number</td>
<td>ASCII</td>
</tr>
<tr>
<td>29-34</td>
<td>Subassembly number</td>
<td>ASCII</td>
</tr>
<tr>
<td>35-40</td>
<td>Element abbreviations</td>
<td>ASCII</td>
</tr>
<tr>
<td>51-82</td>
<td>Masses</td>
<td>REAL*4</td>
</tr>
<tr>
<td>83-90</td>
<td>Receipt document number</td>
<td>ASCII</td>
</tr>
<tr>
<td>91-98</td>
<td>Latest document number</td>
<td>ASCII</td>
</tr>
<tr>
<td>99-104</td>
<td>Date of archive storage</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td>105-108</td>
<td>Initials of person who caused archive storage</td>
<td>ASCII</td>
</tr>
<tr>
<td>109-112</td>
<td>User of the material</td>
<td>ASCII</td>
</tr>
<tr>
<td>113-114</td>
<td>Action taken: 'S' = Stored</td>
<td>ASCII</td>
</tr>
<tr>
<td></td>
<td>'D' = Deleted</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'M' = Modified</td>
<td></td>
</tr>
<tr>
<td>115-116</td>
<td>Bucket number</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td>117-118</td>
<td>Record number</td>
<td>INTEGER*2</td>
</tr>
</tbody>
</table>

**Current File Allocations**

<table>
<thead>
<tr>
<th>File Name</th>
<th>Logical Unit No.</th>
<th>Bucket Size</th>
<th>No. of Buckets</th>
<th>Overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAT</td>
<td>500</td>
<td>24 records</td>
<td>80</td>
<td>61</td>
</tr>
<tr>
<td>DES</td>
<td>600</td>
<td>24 records</td>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>SER</td>
<td>700</td>
<td>128 bytes</td>
<td>240</td>
<td>151</td>
</tr>
<tr>
<td></td>
<td></td>
<td>= 1 record</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHR</td>
<td>800</td>
<td>384 bytes</td>
<td>80</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td></td>
<td>= 3 records</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARC</td>
<td>900</td>
<td>10 tracks (6144 bytes) allocated as a sequential file</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
III. SUMMARY

The following basic features of the computer-based accountability system have proven to be a sound approach to our accountability needs:

- Storage and retrieval of both item and batch information.
- Use of a "characteristic" file to facilitate retrieval of items with specific attributes such as isotopes present, location, responsible individual, etc.
- Use of "hash" functions, which condense batch and item numbers to generate storage addresses.
- Use of prompt-response subroutines to facilitate training of operators in input/output operations.
- Operation on a multiuser minicomputer; all routines coded in FORTRAN language.

The basic features as developed in Phase I will be retained and improved in Phase II, in which the system will go into operation at ANL-W. Phase II will include refinement of the interactive subroutines to make them easier to use by persons untrained in programming. The full implementation of the system is expected to greatly increase the speed and ease of accounting for SNM at ANL-W.
APPENDIX A

Program Flowcharts

Appendix A contains program descriptions and flowcharts for each program and subroutine used in the inventory system. The material is intended to give readers a general overview of the system.

Those who want more detail or are seriously considering the installation of such a system should refer to the program listings, a summary of which is given below. These program listed are available on magnetic tape from the National Energy Software Center, Argonne, Illinois.

Data Storage and Retrieval

1. Program Summary

Accept (Array, MaxLength, TrueLength, MinLength)
BigCompress (Filenum, Overflow, Bucksize, Hash, CharAlso)
BigRemove (Filenum, Array, Bucknum, Bucksize, Record)
BigSqueeze (Filenum, FirstOverflow, LastOverflow, Bucksize)
BitPack (Pointer, Doubleword, Groupnumber)
Compare (First, Second, Length, Match)
Copy (From, To, Length)
DatePack (Date, Result)
Delete (Date, Initials)
Data Storage and Retrieval Program
FileRecord (Filenum, Array, Bucknum, Bucksize, Recnum)
FindOpen (Filenum, Bucknum, Bucksize, MaxRecords, Array, Overflow, Result)
GetANumber (IntgrPart, RealPart, Least, Greatest, NormalSize)
GetBatch
GetCharacteristics
GetDescription (Description)
GetInfo (Sixwords, Prefix)
GetIsotopes (Element, Pointer, Groupnumber, MaxGroups)
GetListfile (Listfile)
GetSerial
HashBatch (BatchItem, Bucket)
HashDesc (Batchnumber, Bucket)
HashSerial (Serialnumber, Bucket)
Locate (Key, Length, Position, Array, NextRecord, Recsize, Filenum, Overflow, Bucksize, Result, Bucket)
MatchIsotopes (Element, Pointer, Groupnumber)
Mix (Key, Length, Result)
Modify (BatArray, Bucket, Record, Date, Initials)
PrintList (List, Last, Listfile)
ReadBucket (Filenum, Bucknum, Bucksize, Array)
Retrieve
SmallCompress (Filenum, Overflow, Bucksize, Hash, Recsize)
SmallRemove (Filenum, Array, Bucknum, Bucksize, Record, Recsize)
SmallSqueeze (Filenum, FirstOverflow, LastOverflow, Bucksize, Recsize)
Store (Date, Initials)
Unpack (Input, Output, Length)
Update (Date, Initials)
WriteRecord (Filenum, Array)
YesOrNo (Response)

2. Flowchart Conventions

A one-page flowchart accompanies each subroutine description. Since the subroutines range in length from about five lines to a few hundred lines, the detail shown in the flowcharts varies widely.

To reduce the ill effects of this variation, we have attempted to keep the more difficult subroutines small. The longer subroutines generally contain a lot of user interaction. In addition, we have attempted to make our comments in the description correspond to the various sections of the flowcharts.

In the main, our flowcharts follow the concepts of structured programming. Since FORTRAN is not easily forced into the structured programming
mold, there is an occasional divergence between the program listings and the flowcharts.

The elements used in a flowchart include box shapes, parameter specifications, and statement forms. A brief explanation of each of these conventions is in order.

a. **Box Shapes** - Flowcharts contain only four types of boxes:
   - **Ellipse** - Reserved for BEGIN, END, or RETURN.
   - **Rectangle** - Contains a sequence of nonbranching statements.
   - **Oval** - Contains a decision statement of some type.
   - **Half Oval** - Contains a loop statement, usually a DO loop or its equivalent.

b. **Parameter Specifications** - Each subroutine has a list of parameters in the upper-left corner of the flowchart. This list has the format:

   Parameters
   Parameter 1 - Type [*Size] $[\frac{V}{A}]$
   .
   .
   .
   Parameter N - Type [*Size] $[\frac{V}{A}]$

Type is an abbreviation of the data type, according to the scheme:

I ≡ Integer
R ≡ Real
C ≡ Complex
L ≡ Logical
Size refers to the number of bytes allocated to the parameter, and is always 2, 4, 6, or 8. If size is omitted, 4 is implied.

The optional letter after size indicates special circumstances:

V = Value (The parameter can be passed by value.)
A = Array (The parameter is an array.)

c. Statement Forms - The flowchart statements have several forms, based on the degree of precision and the amount of condensation required.

FORTRAN I/O statements are listed directly. These include file-handling statements, implied DO statements, and occasionally a format statement.

Data on files are often accessed via "direct access" read-and-write statements. The form of these statements is:

READ (U 'R)
WRITE (U 'R)

where U is the file name and R is the record number within file U.

Assignment statements and loop statements follow a PASCAL-like format. Assignment statements take the form "variable: = expression." Loops have the form "FOR index: = first UNTIL last [STEP stepsize]." The PASCAL FOR statement is superior to the FORTRAN DO statement. In FORTRAN, if last < first, the loop is iterated once anyway. In PASCAL, it would not be iterated at all. Since all programs are written in FORTRAN, most of the DO loops have IF statements preceding them to guard against this danger. However, in general, the flowcharts don't show this.

Subroutine calls may include a CALL statement; sometimes only the subroutine name is used. Some of the parameters may be included, if they don't clutter things too much.
Two statements from "assembler" are used: INC and DEC mean increment and decrement. For example, "INC variable" means "variable = variable + 1."

When all else fails, the flowcharts resort to a statement in English. This usually occurs in decision boxes and in the large subroutines, where the one-page size limit becomes restrictive.

3. Documentation of Subroutines

Each subroutine has one or more pages with information under each of the following five headings:

Subroutine: Subroutine name [(parameter 1, ..., parameter N)]

Type List: Parameter 1 - Type * Size [VAR]
  .
  .
  .
  Parameter N - Type * Size [VAR]

Description: An explanation of what the subroutine does and how it does it.

Details: Interesting or unusual data structures, explanation of strange or difficult parts of the subroutine, defense of poor programming practices, etc.

Pitfalls: Anything that has caused us trouble or that might conceivably cause trouble is listed here.

4. Program Descriptions and Flowcharts

The remainder of Appendix A contains the program descriptions and flowcharts for the inventory system. Where possible, a description and its corresponding flowchart are shown on facing pages.
Data Storage and Retrieval Program

The inventory system's various functions are coded as subroutines. The Data Storage and Retrieval Program provides a framework for the functional subroutines. It includes a preliminary section for user identification and initial user dialog. Once the user has entered the requisite information, the program loops through a section that prompts the user to enter a command, receives the command, interprets the command, and calls the appropriate subroutine. Several types of inventory manipulation, such as entering data or updating, may be performed without restarting the program. The QUIT command exits the program.
DATA STORAGE AND RETRIEVAL PROGRAM

BEGIN

DEFINE FILES.
GET TIME & DATE.
ACCEPT INITIALS.

STORE, RETRIEVE,
UPDATE, DELETE,
HELP, OR QUIT?

CALL STORE
CALL RETRIEVE
CALL UPDATE
CALL DELETE
WRITE HELP

END
Subroutine: GetANumber (IntgrPart, RealPart, Least, Greatest, NormalSize)

Type List: IntgrPart I * 4
RealPart R * 4
Least R * 4 V
Greatest R * 4 V
NormalSize R * 4 V

Description: This subroutine reads a number from the terminal. It is assumed that the user has already been prompted so he knows what to input. If the number entered is outside of the closed interval between Least and Greatest, the user is told to enter a number between these limits, and control returns to the start of the subroutine. If the number is within these limits, but greater than NormalSize, the user is asked to verify it. If the number is within the limits and less than or equal to NormalSize, it is returned to the calling procedure. Since sometimes an integer is needed and sometimes a real number, both types are returned, as IntgrPart and RealPart respectively. The calling procedure should specify one of these as a dummy parameter.

Details: None.

Pitfalls: If the last three parameters are passed by value, they should have a decimal point so the FORTRAN compiler will know they are real.
SUBROUTINE GET A NUMBER

PARAMETERS
Intgpart - I
Realpart - R
Least - RV
Greatest - RV
NormalSize - RV

BEGIN
READ REALPART

LEAST ≤ REALPART ≤ GREATEST?

Y
WRITE ERROR MESSAGE

REALPART > NORMAL SIZE?

N

DID USER MAKE MISTAKE?

N

INTGPART: = REALPART

RETURN
Subroutine: Accept (Array, MaxLength, TrueLength, MinLength)

Type List:

- Array - I A
- MaxLength - I V
- TrueLength - I V
- MinLength - I V

Description: This subroutine reads a string from the terminal. It is assumed that the user has already been prompted for input. Initially, MaxLength bytes are read into a buffer. All ASCII blanks, dashes, and commas are removed from the string. If the resulting string has fewer than MinLength bytes, the user is told to input a longer string, and control returns to the start of the subroutine. Otherwise, the string is packed into Array, two bytes per word, and either truncated or padded with ASCII blanks to make it exactly TrueLength bytes in length.

Details: None.

Pitfalls: Observe that all three length parameters refer to bytes, not words. An odd number is acceptable for any of these, but an even number of bytes is always returned.
PARAMETERS
Array - 1x2A
MaxLength - IV
TrueLength - IV
MinLength - IV

SUBROUTINE ACCEPT

BEGIN
INITIALIZE
READ INPUT
REMOVE BLANKS, DASHES, & COMMAS AND PLACE EVERYTHING ELSE IN COPY
WRITE ERROR MESSAGE

INSUFFICIENT INPUT?
Y
N
PACK COPY INTO ARRAY, 2 BYTES/WORD

ARRAY FULL?
Y
N
BLANK REMAINDER OF ARRAY

RETURN
Subroutine: YesorNo (Response)

Type List: Response - I * 2

Description: This subroutine reads one character from the terminal. It is assumed that the user has already been asked a question that requires either a yes or a no answer. If the character is either Y or N, it is returned in the left byte of Response. Otherwise, the user is told to answer either yes or no, and control returns to the start of the subroutine.

Details: None.

Pitfalls: None known.
SUBROUTINE YES OR NO

PARAMETERS

Response - 1^2

BEGIN

READ RESPONSE

RESPONSE = 'Y' ?
OR
RESPONSE = 'N' ?

WRITE ERROR MESSAGE

RETURN
Subroutine: MatchIsotopes (Elements, Pointer, GroupNumber)

Type List:
- Elements - I * 2 A
- Pointer - I * 2 A
- GroupNumber - I * 4 V

Description: This subroutine matches the two-byte abbreviations that are already in the array Elements with two sets of standardized abbreviations of the various isotopes. There are 18 isotopes. If the i-th word in Element matches the j-th word of either abbreviation list, then the i-th word in Pointer is set equal to j. If no match is found, the i-th word in Pointer is set to 19. The subroutine matches the first GroupNumber words in Element. (All the others are blank).

Details: The standard abbreviations are defined as follows:

<table>
<thead>
<tr>
<th>Mnemonic Code</th>
<th>DOE Code</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>DU</td>
<td>10</td>
<td>Depleted Uranium</td>
</tr>
<tr>
<td>EU</td>
<td>20</td>
<td>Enriched Uranium</td>
</tr>
<tr>
<td>NU</td>
<td>81</td>
<td>Normal Uranium</td>
</tr>
<tr>
<td>PU</td>
<td>50</td>
<td>Plutonium 239-241</td>
</tr>
<tr>
<td>P8</td>
<td>83</td>
<td>Plutonium 238</td>
</tr>
<tr>
<td>P2</td>
<td>40</td>
<td>Plutonium 242</td>
</tr>
<tr>
<td>A1</td>
<td>44</td>
<td>Americium 241</td>
</tr>
<tr>
<td>A3</td>
<td>45</td>
<td>Americium 243</td>
</tr>
<tr>
<td>CU</td>
<td>46</td>
<td>Curium</td>
</tr>
<tr>
<td>BK</td>
<td>47</td>
<td>Berkelium</td>
</tr>
<tr>
<td>CF</td>
<td>48</td>
<td>Californium</td>
</tr>
<tr>
<td>LI</td>
<td>60</td>
<td>Lithium 6</td>
</tr>
<tr>
<td>U3</td>
<td>70</td>
<td>Uranium 233</td>
</tr>
<tr>
<td>NP</td>
<td>82</td>
<td>Neptunium</td>
</tr>
<tr>
<td>D2</td>
<td>86</td>
<td>Deuterium</td>
</tr>
<tr>
<td>TR</td>
<td>87</td>
<td>Tritium</td>
</tr>
<tr>
<td>TH</td>
<td>88</td>
<td>Thorium</td>
</tr>
<tr>
<td>UC</td>
<td>89</td>
<td>Uranium in Cascade</td>
</tr>
</tbody>
</table>

Pitfalls: Note that Pointer is an Integer*2 array, although it contains numeric, rather than alphabetic, data.
PARAMETERS
Element - I=2A
Pointer - I=4A
GroupNumber - IV

BEGIN
FOR I := 1 UNTIL GROUPNUMBER
TEST ELEMENT := ELEMENT (I)
POINTER (I) := 19
FOR J := 1 UNTIL 18
TEST ELEMENT := ABBREVIATION (J)?
or TEST ELEMENT := DOE CODE (J)?
IF Y THEN
POINTER (I) := J
J := 19
RETURN
Subroutine: Unpack (Input, Output, Length)

Type List:  
- Input: I * 2 A  
- Output: I * 2 A  
- Length: I * 4 V

Description: It is assumed that Input contains ASCII data, packed two bytes per word. This subroutine unpacks this into the Output array, one byte per word (the left byte). The right byte of each word of Output receives an ASCII blank. "Length" is the number of bytes to be unpacked. This subroutine is particularly useful for printing blocks containing an odd number of bytes, using the FORTRAN A1 format.

Details: Two system library subroutines were used here:  
- INTLEN - is a MAX IV operating-system routine which conditions other system subroutines to accept two- or four-byte integers. (See Ref. 2, Chap. 2, p. 82.)  
- ISHFT - is used to shift words left 8 bits. Note that 8 is passed by reference as an Integer*2 parameter. ISHFT requires both parameters to be of the same size -- either both Integer*2 or both Integer*4. See Ref. 2, Chap. 3, p. 61.

Pitfalls: Note that Length is in bytes, not words.
PARAMETERS
Input - 1=2A
Output - 1=2A
Length - IV

SUBROUTINE UNPACK

BEGIN
CALL INTLEN (.FALSE.)
FOR BYTE:= 1
UNTIL LENGTH
STEP 2
WORD:= (BYTE+1)/2
OUTPUT(BYTE):=INPUT (WORD)
OUTPUT (BYTE+1):=ISHFT (INPUT (WORD),8)
RETURN
Subroutine: GetListFile (ListFile)

Type List: ListFile - I * 4

Description: This subroutine asks the user which output device he wants and reads the reply. The user must reply either LO (the user's terminal) or LP (the system line printer), causing ListFile to be set to 2 or 6 respectively. Any other reply causes a message to be printed, listing the possible replies; and control returns to the beginning of the subroutine.

Details: None.

Pitfalls: Before program execution, the file assignments should be made as follows: $ASS 2 = LO 6 = LP.
SUBROUTINE GET LIST FILE

PARAMETERS
ListFile - 1

BEGIN
PROMPT USER READ ANSWER

IF ANSWER = 'LO' THEN
    LISTFILE = 2
ELSE IF ANSWER = 'LP' THEN
    LISTFILE = 6
ELSE
    WRITE ERROR MESSAGE
ENDIF
RETURN
**Subroutine:** ReadBucket (Filenum, Bucknum, Bucksize, Array)

**Type List:**
- Filenum - I * 4 V
- Bucknum - I * 4 V
- Bucksize - I * 4 V
- Array - R * 8 A (see details)

**Description:** This subroutine reads bucket number Bucknum from file number Filenum into Array. The bucket is assumed to be Bucksize 64-word records in length.

**Details:** Array is declared to be Real*8 because certain test programs indicate that reading from the disk is several times faster for this type of array. The FORTRAN compiler does no type checking for parameters. Hence, it is perfectly admissible for the calling procedure to pass an Integer*2 array.

**Pitfalls:** It is important to remember that Bucksize is measured in 64-word records, not in true records (which would be much larger for 8-word records or 5-word records).
SUBROUTINE READ BUCKET

PARAMETERS

Filenum - IV
Bucknum - IV
Bucksize - IV
Array - R*8A

BEGIN

OFFSET: = (BUCKNUM - 1) * BUCKSIZE

FOR RECORD: = 1 UNTIL BUCKSIZE

RECNUM: = OFFSET + RECORD
FIRST: = (RECORD - 1) * 16 + 1
LAST: = RECORD * 16
READ (FILENUM 'RECORD)
ARRAY (1), I=FIRST, LAST)

RETURN
Subroutine: FileRecord (Filenum, Array, Bucknum, Bucksize, Recnum)

Type List: Filenum - I * 4 V
Array - R * 8 A (see details)
Bucknum - I * 4 V
Bucksize - I * 4 V
Recnum - I * 4 V

Description: It is assumed that Array contains a 64-word record. This subroutine writes the record onto record number Recnum of bucket number Bucknum of file number Filenum. Each bucket is assumed to contain Bucksize 64-word records (irrespective of the number of true records).

Details: Array is declared to be Real*8 for faster I/O. Normally, the calling procedure passes an Integer*2 array, since the FORTRAN compiler does not check the parameter types.

Pitfalls: Note that Bucksize is measured in 64-word records. Also, the subroutine assumes that the record to be filed is in the first 64 words of Array. It is sometimes necessary to pass an index along with Array in order to write a record onto the file from the middle of a large table.
SUBROUTINE FILE RECORD

PARAMETERS
Filenum - IV
Array - R*8A
Bucknum - IV
Bucksiz - IV
Recnum - IV

BEGIN

POINTER:=(BUCKNUM-1)*BUCKSIZE + RECNUM

WRITE (FILENUM 'POINTER)
(ARRAY(I), I=1, 16)

RETURN
Subroutine: Copy (From, To, Length)

Type List: From - I * 2 A (see details)
            To    - I * 2 A (see details)
            Length   - I * 4 V

Description: This subroutine copies a block of Length words from one point in memory to another. Words are copied from the block beginning at the address of From to the block beginning at the address of To.

Details: From and To are declared to be of type Integer*2. However, this subroutine is not restricted to copying this type of array. The FORTRAN compiler does no type checking on parameters. Hence, the subroutine can be used to transfer exact bit patterns from one type of variable to another. For example, a Real*8 variable can be put into any four adjacent words of an Integer*2 array (or vice versa). This can eliminate the need for FORTRAN equivalence statements.

Pitfalls: If bytes are being copied, keep in mind that Length is measured in words, not bytes.
SUBROUTINE COPY

PARAMETERS
From - 1.2A
To   - 1.2A
Length - IV

BEGIN
FOR WORD := 1 UNTIL LENGTH
TO (WORD):= FROM (WORD)
RETURN
Subroutine: Compare (First, Second, Length, Match)

Type List:  
First - I * 2 A  
Second - I * 2 A  
Length - I * 4 V  
Match - L * 4

Description: This subroutine compares two arrays, each containing Length words. One array begins at the address of First, the other at the address of Second. If the two arrays are equal over their entire length, then Match is returned as .True. Otherwise, it is returned as .False.

Details: Although this subroutine has the same versatility as Copy, it is almost exclusively used for Integer*2 arrays. Note that either array can be passed with an index.

Pitfalls: Note that Length is measured in words, not bytes.
SUBROUTINE COMPARE

PARAMETERS
First - 1=2A
Second - 1=2A
Length - IV
Match - L

BEGIN
MATCH:=.TRUE.
FOR WORD:= 1 UNTIL LENGTH
FIRST(WORD)= SECOND(WORD)?
Y
MATCH:= FALSE
WORD:= LENGTH + 1
N
RETURN
Subroutine: Locate (Key, Length, Position, Array, NextRecord, Recsize, Filenum, Overflow, Bucksize, Result, Bucket)

Type List:  
Key - I * 2 A  
Length - I * 4 V  
Position - I * 4 V  
Array - I * 2 A  
NextRecord - I * 4  
Recsize - I * 4 V  
Filenum - I * 4 V  
Overflow - I * 4  
Bucksize - I * 4 V  
Result - I * 4  
Bucket - I * 4

Description: The purpose of this subroutine is to locate all occurrences of the array Key in the file Filenum. Key will be found either in bucket number Bucket or in a bucket greater than or equal to Overflow or in both or in neither. Key is an array of Length words which is supposed to begin at word number Position in a record of Recsize words. The subroutine begins by reading bucket number Bucket into Array from file Filenum. The bucket is known to be Bucksize 64-word records in length. Starting with record number NextRecord, Array is searched record by record until Key is found or until the bucket is exhausted. If Key is not found, one of two things will happen. If the bucket was completely full, a new bucket is read from Overflow and the subroutine begins its search anew. If the bucket was only partially full, then the Key is not on file, so Result is returned as zero. On the other hand, if Key was found, then Result is set to the record number where it was
found, Bucket is set to the bucket number where it was found, and NextRecord is set to Result + 1. Control then returns to the calling procedure.

Details:

This subroutine is designed to be called repeatedly, until Result is returned as zero. This is necessary if Key can occur more than once in a given file. Otherwise it need be called only once. Note that information is passed back to the calling procedure through five different parameters. Furthermore, the calling procedure passes information to the subroutine through three of these. This provides great opportunity for errors. Handling of all five parameters is explained below.

Array - The current bucket being examined is returned in Array. If a nonzero Result is returned, the calling procedure can easily find the appropriate record in Array for whatever processing it wants to do.

NextRecord - The first time Locate is called to find a particular Key, Next Record should be set to 1. On each succeeding call, it should be passed without alteration. This enables the subroutine to find several occurrences of Key in the same bucket.

Overflow - On the first call of Locate, Overflow should be set to the first bucket in the overflow area. This variable is incremented each time the subroutine reads a new overflow bucket. On succeeding calls, Overflow should be passed without alteration.
Result - The record in which Key was found is returned as Result. The calling procedure is responsible for finding this record in Array and processing it.

Bucket - On the first call of Locate, Bucket should be set to the first bucket number to be searched. This will usually be the result of a hash function. If the subroutine is forced to read a new bucket, Bucket will be modified to reflect the new bucket number. On succeeding calls of Locate, Bucket should be passed unchanged.

Pitfalls: Firstly, with 11 parameters, it is easy to leave one out. Make sure all are present in the call statement, in proper order, and of the proper type.

Secondly, the information in Details, above, should be followed carefully. To summarize: Before the first call, initialize Bucket, Overflow, and Next Record. On succeeding calls, leave them unaltered.

Thirdly, note that Length and Position refer to words, not bytes.

Fourthly, note that Recsize refers to the true size of the records (e.g., 5, 8, or 64). Conversely, Bucksize refers to the number of 64-word records per bucket (e.g., 1, 3, or 24).
SUBROUTINE LOCATE

PARAMETERS
Key - I^2A
Length - IV
Position - IV
Array - I^2A
NextRecord - 1
Recsize - IV
Filenum - IV
Overflow - I
Bucksze - IV
Result - 1
Bucket - I

BEGIN

RESULT: = 0
MAX RECORDS: = (BUCKSIZE*64)/RECSIZE
READ BUCKET (FILENUM, BUCKET . . .)

LAST RECORD: = ARRAY (64)

FOR RECORD: = NEXT RECORD
UNTIL LAST RECORD

POINT: = (RECORD-1)*
RECSIZE + POSITION
COMPARE (KEY, ARRAY (POINT),
LENGTH, MATCH)

MATCH?

RESULT: = RECORD
NEXT RECORD: = RECORD+1
RECORD: = LAST RECORD+1

RESULT > 0?
OR
LAST RECORD < MAX RECORDS?

READ BUCKET (FILENUM, OVERFLOW . . .)
BUCKET: = OVERFLOW
INC OVERFLOW
NEXT RECORD: = 1

RETURN
Subroutine: FindOpen (Filenum, Bucknum, Bucksize, MaxRecords, Array, Overflow, Result)

Type List:  
- Filenum: I * 4V  
- Bucknum: I * 4  
- Bucksize: I * 4V  
- MaxRecords: I * 4V  
- Array: I * 2A  
- Overflow: I * 4V  
- Result: I * 4

Description: The purpose of this subroutine is to find the next open record in which to store information on file Filenum. The subroutine begins by reading bucket number Bucknum, which is assumed to be Bucksize 64-word records in length, from file Filenum into Array. MaxRecords is the number of true records (e.g., the number of 5-word, 8-word, or 64-word records) which the bucket can contain. It is known that the 64th word of every bucket in every file contains the number of true records actually in the bucket. If this is less than MaxRecords, another record can be put into the bucket, so Result is returned as Array(64) + 1. Otherwise, bucket number Overflow is read into Array, Overflow is incremented, and the process is repeated. On return, Bucknum contains the number of the open bucket and Result contains the number of the open record.

Details: Note that information is passed in both directions through Bucknum. Generally, before calling FindOpen, Bucknum will be calculated by a hash function. On return, it may contain a different number, indicating the bucket to which the next record should be copied. Also, note that the contents of the open bucket are
contained in Array. Hence, Result is a redundant parameter, since it can be calculated as Array(64) + 1. This was tolerated because it makes the calling procedure more readable.

**Pitfalls:**

With seven parameters, one is easily omitted, especially since Bucksize and MaxRecords are often the same quantity (when the true record size is 64 words).

The calling procedure is responsible for setting Array(64) = Result and recopying the first 64-word record back to the file after completion of processing of Array.
PARAMETERS
Filenum - IV
Bucknum - IV
Bucksize - IV
MaxRecords - IV
Array - 1*2A
Overflow - I
Result - I

SUBROUTINE FIND OPEN

BEGIN

CALL READBUCKET (FILENUM, BUCKNUM, BUCKSIZE, ARRAY)
LAST RECORD := ARRAY (64)

LAST RECORD = MAX RECORDS?
N

RESULT := LAST RECORD + 1

RETURN
Subroutine: Store (Date, Initials)

Type List:
- Date - I * 2A
- Initials - I * 2A

Description: This subroutine is responsible for collecting information about an entire batch from the user and placing it in various files. It is an unusually long procedure because of all the I/O -- prompts, verifications, and file handling. "Date" is an array containing the current month, day, and year in words 1, 2, and 3 respectively. "Initials" contains the three initials of the terminal user, for tracing purposes. The user is first asked for general information, applicable to the entire batch. Then he is asked for specific information about each item (serial number and masses of isotopes) within a loop.

Details: Consult the documentation on File Descriptions (Subsection II.E) for the exact format of the various records in the different files.

Pitfalls: None known.
SUBROUTINE STORE

PARAMETERS

Date - I*2A
Initials - I*2A

BEGIN

ACCEPT BATCH #

IS THIS BATCH ON THE DESCRIPTION FILE?

OUTPUT DESCRIPTION
GET DESCRIPTION FIND AN OPEN RECORD ON THE DESCRIPTION FILE

ACCEPT DOCUMENT #
ACCEPT SUBASSEMBLY #

EVERYTHING OK?

GET ISOTOPES
ACCEPT USER
GET # OF ITEMS
ASSEMBLE BATCH AND CHAR INFO
COPY DESCRIPTION BACK TO FILE

FOR ITEM: = FIRST UNTIL TOTAL ITEMS

ACCEPT SERIAL #
GET MASSES

FIND AN OPEN RECORD ON BATCH AND SERIAL FILES
ASSEMBLE BATCH AND SERIAL INFO
WRITE BATCH, SERIAL, AND CHAR INFO TO FILES

RETURN
Subroutine: Retrieve

Type List: None.

Description: This subroutine initializes the 64-word record Last-Write in the Common area labeled History, and asks the user what type of retrieval he wants. The various options are:

- **Batch** - Retrieve an entire batch, referenced by batch number.
- **Serial** - Retrieve a single item referenced by serial number.
- **Char** - Retrieve all items on file with some arbitrary set of characteristics.
- **Help** - Print a list of user options similar to this list.
- **Quit** - Exit subroutine, return to main program.

Pitfalls: None.
Subroutine: WriteRecord (Filenum, Array)

Type List: Filenum - I * 4 V
Array - I * 2 A

Description: This subroutine extracts the useful information from Array and writes it in a readable fashion on the output file Filenum. It is assumed that Array contains a single 64-word record read from the batch file.

Details: Often, this subroutine is called several times in succession to write consecutive records from the same batch. Several lines are required to print complete information on a single item. Since much of this information is invariant throughout the batch, it is unnecessary to print all of it for each item. By printing the general information once and only the item-specific information thereafter, a lot of paper can be saved and the jittery-eye syndrome can be avoided. To determine whether to write all of the information available, some historical data must be kept. This is the purpose of the common array LastWrite. It contains the last record that was written in full. If Array agrees with LastWrite in all essential fields (batch number, subassembly number, elements, and user), then only the serial number and mass data are written. Otherwise, all data are written and Array is copied to LastWrite.

Pitfalls: None known.
SUBROUTINE WRITE RECORD

PARAMETERS
Filenum - IV
Array - 1*2A

BEGIN

COPY MASSES FROM ARRAY
CALCULATE GROUPNUMBER

IS IT TIME FOR A NEW PAGE?

Y

FORM FEED
WRITE PAGE #
INC PAGE #

N

IS THIS RECORD MUCH LIKE THE LAST ONE?

Y

WRITE ALL DATA EXCEPT MASSES
UPDATE LINE #

N

WRITE MASSES
UPDATE LINE #

RETURN
Subroutine: GetBatch

Type List: None.

Description: This subroutine retrieves information about all or part of an entire batch. The user is prompted for the desired batch number, which is then looked up on the description file. This provides information on the existence of the batch, number of items in the batch, and a description of the batch. The user is asked for the desired range of items and the output file (terminal or line printer). Items are then looked up one after another on the batch file and printed using WriteRecord.

Details: When an item is stored, it is assigned an item number, unique within its batch. This number is located directly after the batch number on the record in the batch file. The batch number and item number are combined to determine the bucket in which the item is placed. This allows essentially infinite batches, without straining the bucket capacities or overflow area.

Pitfalls: None known.
SUBROUTINE GET BATCH

BEGIN

ACCEPT BATCH #

Y IS THIS BATCH ON FILE?

PRINT DESCRIPTION AND NUMBER OF ITEMS
GET RANGE OF ITEMS TO OUTPUT
GET LIST FILE

FOR ITEM: = FIRST UNTIL LAST

LOCATE THIS ITEM ON THE BATCH FILE

Y FIND ANYTHING?

WRITE IT TO LIST FILE

WRITE "MISSING"

RETURN

N WRITE "FINISHED"

WRITE ERROR MESSAGE
Subroutine: GetSerial

Type List: None.

Description: The purpose of this subroutine is to look up information on a single item, referenced by serial number. It is slightly more complex than the related subroutine GetBatch, because of the remote possibility that more than one item may exist on file with the same serial number. The idea is to repeatedly try to locate the serial number until a result of zero is returned. Each time we find the serial number in the serial file, we obtain the bucket number where the item is to be found in the batch file. We then try to locate the serial number in the batch file and print the information using WriteRecord. Again, it is necessary to repeatedly call Locate until the result is returned as zero, because of the infinitesimal chance that two items may exist in the same bucket in the batch file with the same serial number.

Details: Note that this subroutine jumps back and forth between the batch and serial files, trying to locate the serial number in each of them. Since the parameters for Locate are rather volatile, the subroutine uses two independent sets.
SUBROUTINE GET SERIAL

BEGIN

ACCEPT SERIAL #

LOCATE THIS SERIAL # ON THE SERIAL FILE

FIND ANYTHING? N

Y

LOCATE THIS SERIAL # ON THE BATCH FILE

FIND ANYTHING? N

Y

WRITE THE RECORD

WRITE "FINISHED"

RETURN
**Subroutine:**  
BigRemove (Filenum, Array, Bucknum, Bucksize, Record)

**Type List:**
- Filenum - I*4 V  
- Array - I*2 A  
- Bucknum - I*4 V  
- Bucksize - I*4 V  
- Record - I*4 V

**Description:** The purpose of this subroutine is to remove a 64-word record from file Filenum and perform the necessary bookkeeping functions. It is assumed that the bucket has already been read into Array. It is known that Array(64) contains the number of records in the bucket. The last record in the bucket is copied to record number Record of bucket number Bucknum, effectively deleting the old record. Array(64) is then decremented and the first record in Array is copied to the file to reflect the change in the number of records.

**Details:** None.

**Pitfalls:** This subroutine should be used only for files having a true record size of 64 words. Files with smaller true record sizes should be processed with the slower SmallRemove subroutine.
SUBROUTINE BIG REMOVE

PARAMETERS

Filenum - TV
Array - I*2A
Bucknum - TV
Bucksize - TV
Record - TV

BEGIN

LAST RECORD: = ARRAY (64)
POINTER: = (LAST RECORD -1)*64 + 1

Y RECORD = 1? N

ARRAY (POINTER + 63): = LAST RECORD -1
ARRAY (64): = LAST RECORD -1
FILE RECORD (FILENUM, ARRAY, . . . 1)

FILE RECORD (FILENUM, ARRAY (POINTER), . . . RECORD)

RETURN
Subroutine: SmallRemove (Filenum, Array, Bucknum, Bucksize, Record, Recsize)

Type List:
- Filenum - I * 4 V
- Array - I * 2 A
- Bucknum - I * 4 V
- Bucksize - I * 4 V
- Record - I * 4 V
- Recsize - I * 4 V

Description: This subroutine is used to remove a small (e.g., 5-word or 8-word) record from file Filenum. The record size is specified by Recsize. It is assumed that the bucket has already been read into Array. The last record is copied to record number Record in Array, Array(64) is decremented, and the entire Array is copied to bucket number Bucknum. Bucksize is the number of 64-word records in the bucket.

Details: None.

Pitfalls: This subroutine should be used only for files with record sizes less than 64 words.
SUBROUTINE SMALLREMOVE

PARAMETERS

Filenum - IV
Array - 1*2A
Bucknum - IV
Bucksize - IV
Record - IV
Recsize - IV

BEGIN

LAST RECORD: = ARRAY (64)
FROMPOINT: = (LAST RECORD -1) * RECsize +1
TOPOINT: = (RECORD -1) * RECSIZE +1
COPY (ARRAY (FROMPOINT), ARRAY (TOPOINT), RECSIZE)
ARRAY (64): = LAST RECORD -1

FOR I: = 1
UNTIL LAST RECORD

POINTER: = (I-1) * 64 +1
FILE RECORD (FILENUM, ARRAY (POINTER),
BUCKNUM, BUCKSIZE, I)

RETURN
**Subroutine:** BigCompress (Filenum, Overflow, Bucksize, Hash, Charalso)

**Type List:**
- Filenum - I * 4 V
- Overflow - I * 4 V
- Bucksize - I * 4 V
- Hash - Subroutine
- Charalso - L * 4 V

**Description:** This subroutine searches the overflow area for records that could be placed in the main file, because of vacancies created by Delete. Buckets each contain Bucksize 64-word records. The overflow area begins with bucket number Overflow on file number Filenum. "Hash" is the hash function to be used. If Charalso is .True., then anything done to the main file is done also to the characteristic file.

**Details:** This subroutine can be used on either the batch file or the description file. It hashes the first several words of each record, and checks the resulting bucket number to see if it is full. If so, the record is removed from overflow and put into the bucket. Otherwise, nothing happens. If this subroutine is used on the batch file, Charalso should be passed as .True. Otherwise it should be .False. This is an inelegant gimmick to get around the fact that the characteristic file has no identifying field to hash.

**Pitfalls:** None known.
SUBROUTINE BIG COMPRESS

PARAMETERS
Filenum - IV
Overflow - IV
Bucksize - IV
Hash - Subroutine
Char also - LV

BEGIN
OVER BUCKET: = OVERFLOW

READ BUCKET (FILENUM . . . )
IF CHAR ALSO, READ BUCKET (CHAR FILENUM . . . )

FINISHED?
Y
N

FOR RECORD: = 1
UNTIL LAST RECORD

FIND THE BUCKET TO WHICH THIS RECORD BELONGS

IS IT FULL?
Y
N

TAKE THE RECORD OUT OF OVERFLOW AND PUT IT INTO THIS BUCKET. REREAD THE OVERFLOW BUCKET.
IF CHAR ALSO, DO THE SAME FOR THE CHAR FILE.

DEC RECORD
DEC LAST RECORD

INC OVERBUCKET

BIG SQUEEZE (FILENUM . . . )
IF CHAR ALSO, SMALL SQUEEZE (CHAR FILENUM . . . )

RETURN
Subroutine: SmallCompress (Filenum, Overflow, Bucksize, Hash, Recsize)

Type List:
- Filenum: $\mathbb{I} \times 4V$
- Overflow: $\mathbb{I} \times 4V$
- Bucksize: $\mathbb{I} \times 4V$
- Hash: Subroutine
- Recsize: $\mathbb{I} \times 4V$

Description: This subroutine does the same thing as the BigCompress subroutine, except that it is designed for files with true record sizes smaller than 64 words. Recsize is the number of words per record.

Details: Note that the characteristic file is taken care of by BigCompress. Hence, this subroutine is used only for the serial file.

Pitfalls: None known.
SUBROUTINE SMALL COMPRESS

PARAMETERS

Filenum - IV
Overflow - IV
Bucksize - IV
Hash - Subroutine
Resize - IV

BEGIN

OVERBUCKET: = OVERFLOW

READ BUCKET ( . . . OVERBUCKET . . . )

FINISHED?

Y

N

FOR RECORD: = 1 UNTIL LAST RECORD

READ THE BUCKET TO WHICH THIS RECORD BELONGS

IS IT FULL?

Y

N

TAKE THE RECORD OUT OF OVERFLOW AND PUT IT INTO THIS BUCKET

DEC RECORD
DEC LAST RECORD

INC OVERBUCKET

SMALL SQUEEZE (FILENUM . . . )

RETURN
Subroutine: BigSqueeze (Filenum, FirstOverflow, LastOverflow, Bucksize)

Type List:  
- Filenum - I * 4 V  
- FirstOverflow - I * 4 V  
- LastOverflow - I * 4 V  
- Bucksize - I * 4 V

Description: After file number Filenum has been compressed, the overflow area between FirstOverflow and LastOverflow may have some buckets with fewer than Bucksize records. BigSqueeze fixes this situation by copying the overflow area into itself, making sure that each copied bucket is full. After copying, all succeeding buckets have their first record altered to indicate an empty bucket.

Details: As each record is copied, the 64th word is set to Bucksize. This ensures that the first record of each bucket indicates a full bucket. When all records have been copied, the last bucket will probably not be full. The first record of this bucket is then changed to indicate the actual number of records in the bucket.

Pitfalls: None known.
SUBROUTINE BIG SQUEEZE

PARAMETERS

Filenum - IV
FirstOverflow - IV
LastOverflow - IV
Bucksize - IV

BEGIN

COPY BUCKET: = FIRST OVERFLOW
COPY RECORD: = 1

FOR BUCKET: = FIRST OVERFLOW UNTIL LAST OVERFLOW

READ BUCKET ( . . . BUCKET . . . )

FOR I: = 1 UNTIL LAST RECORD

COPY FROM RECORD #I IN BUCKET TO RECORD # COPY RECORD IN COPY BUCKET.
INC COPY RECORD.

IS COPY BUCKET FULL?

Y

N

COPY RECORD: = 1
INC COPY BUCKET

REWRI TE RECORD #I OF COPY BUCKET TO INDICATE THE # OF RECORDS IN THE BUCKET

FOR I: = COPY BUCKET +1 UNTIL LAST OVERFLOW

REWRI TE RECORD #I OF BUCKET #I TO INDICATE THAT IT IS EMPTY

RETURN
Subroutine: SmallSqueeze (Filenum, FirstOverflow, LastOverflow, Bucksize, Recsize)

Type List:  
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filenum</td>
<td>- I * 4 V</td>
</tr>
<tr>
<td>FirstOverflow</td>
<td>- I * 4 V</td>
</tr>
<tr>
<td>LastOverflow</td>
<td>- I * 4 V</td>
</tr>
<tr>
<td>Bucksize</td>
<td>- I * 4 V</td>
</tr>
<tr>
<td>Recsize</td>
<td>- I * 4 V</td>
</tr>
</tbody>
</table>

Description: This subroutine does the same thing as BigSqueeze does, for files having a true record size of Recsize words.

Details: Instead of records being copied directly to the file, they are copied to an array. When the array contains Bucksize records, the entire bucket is copied to file at once.

Pitfalls: None known.
SUBROUTINE SMALL SQUEEZE

PARAMETERS

Filenum - IV
FirstOverflow - IV
LastOverflow - IV
Bucksize - IV
Recsize - IV

BEGIN
COPY BUCKET: = FIRST OVERFLOW
COPY RECORD: = 1

FOR BUCKET: = FIRST OVERFLOW UNTIL LAST OVERFLOW

READ BUCKET (. . . BUCKET . . . ARRAY 1)

FOR I: = 1 UNTIL LAST RECORD

COPY RECORD #1 IN ARRAY 1 TO RECORD # COPY RECORD IN ARRAY 2.
INC COPY RECORD

IS ARRAY 2 FULL?

Y
WRITE ARRAY 2 INTO BUCKET # COPY BUCKET.
COPY RECORD: = 1
INC COPY BUCKET.

N
FOR I: = COPY BUCKET +1 UNTIL LAST OVERFLOW

WRITE ARRAY 2 INTO BUCKET # COPY BUCKET

REWRITE RECORD #1 OF BUCKET #1 TO INDICATE THAT IT IS EMPTY.

RETURN
Subroutine: GetIsotopes (Element, Pointer, Groupnumber, MaxGroups)

Type List:
- Element: I * 2 A
- Pointer: I * 2 A
- Groupnumber: I * 4
- MaxGroups: I * 4 V

Description: This subroutine obtains a list of isotopes from the user and checks to be sure no errors have been made. It is assumed that the calling procedure has already prompted the user for the number of isotopes. This subroutine reads this number, Groupnumber, from the terminal, making sure it is between zero and MaxGroups. If it is zero, the subroutine returns control to the calling procedure. Otherwise, the subroutine accepts Groupnumber abbreviations of isotopes from the user. It then prints a table showing the computer's definitions for the abbreviations that were input. If the user wants, he can make modifications to his list, until he is satisfied.

Details: None.

Pitfalls: Note that Pointer is an Integer*2 array, despite the fact that it contains numeric data.
SUBROUTINE GET ISOTOPES

PARAMETERS

Element - 1
Pointer - 1
GroupNumber - 1
MaxGroups - 1

BEGIN
GET GROUP NUMBER
DOES USER WANT A SYMBOL LIST?
Y
PRINT SYMBOL LIST
ACCEPT ELEMENTS
PRINT DEFINITION TABLE
EVERYTHING OK?
Y
RETURN
N
ACCEPT A MODIFICATION
N
RETURN
Subroutine: GetDescription (Description)

Type List: Description I * 2 A

Description: This subroutine obtains a description of a batch from the user. If the batch has no description, Description is returned filled with blanks. Otherwise, the user inputs a string from the terminal, describing the batch. If the string is longer than 80 bytes, the user is told to shorten it. Otherwise, the string is returned in the array Description.

Details: None.

Pitfalls: None known.
SUBROUTINE GET DESCRIPTION

PARAMETERS

Description - 1*2A

BEGIN

DOES THIS BATCH HAVE A DESCRIPTION?

Y

ACCEPT DESCRIPTION

Y

TOO MANY BYTES?

RETURN

N

FILL DESCRIPTION WITH BLANKS

N
Subroutine: BitPack (Pointer, DoubleWord, Groupnumber)

Type List:  
- Pointer - I*2A  
- DoubleWord - I*4  
- Groupnumber - I*4V

Description: This subroutine packs the information in the array Pointer into the integer DoubleWord. It is assumed that Pointer has previously been initialized by the subroutine MatchIsotopes, so that the first Groupnumber words each contain an integer between 1 and 19, indicating a particular isotope. Bitpack sets or clears the first 19 bits in DoubleWord to indicate the presence or absence respectively of the corresponding isotope. DoubleWord is to be stored on the characteristic file, where space is at a premium.

Details: This subroutine uses two system subroutines:
- **INTLEN:** Sets the system integer size to 4 bytes. See Ref. 2, Chap. 2, p. 82.
- **SETBF:** Sets a single bit in an integer to 1. See Ref. 2, Chap. 3, p. 62.

Note that both parameters must be passed to SETBF as either Integer*4 or Integer*2.

Pitfalls: Note that Pointer is an Integer*2 array.
SUBROUTINE BIT PACK

PARAMETERS

Pointer - 1*2A
DoubleWord - 1
GroupNumber - IV

BEGIN

DOUBLEWORD: = 0
CALL INTLEN (.TRUE.)

FOR I: = 1
UNTIL GROUPNUMBER

BIT: = POINTER (I)
CALL SETBF (DOUBLEWORD, BIT)

RETURN
Subroutine: DatePack (Date, Result)

Type List:

Date       -   I * 2 A
Result     -   I * 2

Description: This subroutine packs the month and year into the left and right bytes respectively of the single word Result. It is assumed that Date(1) contains a number between 1 and 12, indicating the month, and that Date(3) contains the A.D. year. This is the form in which the system subroutine TIMDAT returns the date.

Details: In order to get the year into 8 bits, 1900 is subtracted from it.

Pitfalls: None known.
SUBROUTINE DATE PACK

PARAMETERS
Date - 1*2A
Result - 1*2

BEGIN

MONTH: = DATE (1)
YEAR: = DATE (3) -1900
CALL INTLEN (.FALSE.)
RESULT: = ISHFT (MONTH, 8) + YEAR

RETURN
Subroutine: GetCharacteristics

Type List: None.

Description: This subroutine searches the characteristic file sequentially for all records having some user-defined set of characteristics. There are four fields of interest on each record in the characteristic file: isotopes, user, subassembly number, and date. There are three ways to conduct a search: inclusion, exclusion, or don't care. For example, we could specify a search for all items with user initials ABC or for all items except those with user ABC, or we could specify that the user not be looked at, at all. We can make one list of things that must be included and another list of things that must be excluded. If these lists overlap, the computer responds with an indignant error message. Otherwise, it begins reading buckets from the characteristic file and checking all nonempty records. Each record is checked to see that everything on the inclusion list is present and that nothing on the exclusion list is present. If so, the record number is saved in a list. When this list is full, the subroutine interrupts its search to generate some output. Each record in the list is looked up in the batch file and printed. The search is then resumed. This continues until the characteristic file is exhausted.

Details: The four fields on each record of the characteristic file have the following format:

- Isotopes - 32 bits. If a given bit is set, the corresponding isotope is present. If clear, it is absent.
User - One word. The three-byte user initials are compressed into a two-byte word using the system's CAN (compressed alphanumeric) library routines.

Subassembly number - Two words. The six-byte subassembly number is CAN-coded into two words.

Date - Two bytes. The left byte contains the month (1-12). The right contains the year. The year is the A.D. year minus 1900. For example, August 1980 is stored as 8, 80.

Pitfalls: None known.
SUBROUTINE GET CHARACTERISTICS

BEGIN

OBTAIN THINGS TO INCLUDE
OBTAIN THINGS TO EXCLUDE

IS THERE AN OVERLAP?

Y

GET LIST FILE
TEST BUCKET: = 1

READ BUCKET #
TEST BUCKET FROM
THE CHAR FILE

FINISHED?

NO

FOR RECORD: = 1
UNTIL LAST RECORD

IS THIS RECORD
OK TO PRINT?

NO

SAVE RECORD #
ON LIST

IS LIST BIG ENOUGH
TO PRINT?

NO

PRINT LIST

INC TEST BUCKET

Y

WRITE ERROR
MESSAGE

PRINTLIST

RETURN
Subroutine: GetInfo (Sixwords, Prefix)

Type List: Sixwords - I * 2 A
Prefix - I * 2 V

Description: This subroutine obtains the inclusion and exclusion lists from the user for use by GetCharacteristics. Prefix is generally passed by value, and it must be either IN or EX, to specify inclusion or exclusion respectively. The list is put into the array Sixwords.

Details: Sixwords has the same format as records from the characteristic file:
Words 1 and 2 - Isotopic data, bit-packed.
Word 3 - User, CAN-coded.
Words 4 and 5 - Subassembly number, CAN-coded.
Word 6 - Date of storage on file, byte-packed.
If a field is a don't care field, it is set to zeroes.

Pitfalls: Prefix can have only the two values IN or EX.
SUBROUTINE GET INFO

PARAMETERS
Sixwords - 1^2A
Prefix - 1^2V

BEGIN

FILL SIXWORDS WITH ZEROES

GET ISOTOPES (ELEMENTS, POINTER, GROUP NUMBER, MAX GROUPS)
BITPACK (POINTER, SIXWORDS, GROUP NUMBER)

ACCEPT USER CAN SIXWORDS (3)

ACCEPT SUBASSEMBLY # CAN SIXWORDS (4)

GET MONTH AND DATE
DATEPACK (DATE, SIXWORDS (6))

IS EVERYTHING OK?

RETURN
Subroutine: PrintList (List, Last, Listfile)

Type List:  
List       - I * 4 A  
Last      - I * 4 V  
Listfile  - I * 4 V

Description: This subroutine prints a set of records from the batch file to Listfile, in readable format. "List" is an array containing a series of record numbers to be printed. "Last" is the number of records to be printed.

Details: None.

Pitfalls: None known.
SUBROUTINE PRINT LIST

PARAMETERS
List - IA
Last - IV
ListFile IV

BEGIN
FOR RECORD: = 1 UNTIL LAST
READ (BAT FILENUM LIST (RECORD)) ARRAY.
WRITE RECORD (LIST FILE, ARRAY).
RETURN
Subroutine: Update (Date, Initials)

Type List: 
- Date - I * 2 A
- Initials - I * 2 A

Description: This subroutine enables the user to modify an arbitrary number of items in a batch. The user is asked for the batch number. If it is on file, he is asked for the number of items he wants to modify. Modifications are made within a loop. The user inputs a serial number. If this number is found, the subroutine Modify is called to make changes. Date and Initials specify the date of modification and the initials of the person who made it.

Details: Since serial number could conceivably not be unique, the subroutine Locate is called repeatedly, until a zero result is returned. Recall that the serial file does not provide an exact address for the item in the batch file. It provides only the bucket number to which the item was hashed. Therefore, we must search this bucket for a record with the correct batch and serial numbers. This is done by locating the batch number and comparing the serial number.

Pitfalls: None known.
SUBROUTINE UPDATE

BEGIN

PARAMETERS
Date - I*2A
Initials - I*2A

ACCEPT BATCH #

IS BATCH ON FILE?

WRITE DESCRIPTION WRITE ERROR MESSAGE

GET NEW DESCRIPTION IF NECESSARY
GET MODNUM

FOR I = 1 UNTIL MODNUM

ACCEPT SERIAL #

LOCATE SERIAL #

FIND ANYTHING?

Y

LOCATE BATCH #

FIND ANYTHING?

N

IS SERIAL # CORRECT?

N

Y

MODIFY

RETURN
Subroutine: Delete (Date, Initials)

Type List: Date \(- I \ast 2 \ A\)
Initials \(- I \ast 2 \ A\)

Description: This subroutine enables the user to delete an arbitrary number of items in a batch. The user is asked for the batch number. If it is on file, he is asked for the number of items he wants to delete. Deletions are made within a loop. The user inputs a serial number. If this number is found, the subroutine makes a note of the deletion on the archives file, along with date and initials and the bucket number and record number of the record. It then removes the appropriate records from the batch, serial, and characteristic files. After all items are deleted, the description file record is updated or removed, depending on whether the batch still has undeleted items.

Details: The means of finding records is the same as in Update. In fact, Delete and Update are so similar that the source code for Delete was cannibalized to write Update. Notice that each deletion creates space in the bucket for one record from the overflow area (assuming that the bucket was full). Until the file is compressed, all records in overflow belonging to this bucket are temporarily lost. If a very large number of items are being deleted, a bucket might go from the full state to the empty state. In this extreme situation, the file will be compressed accordingly.

Pitfalls: None known.
SUBROUTINE DELETE

PARAMETERS

Date - 1#2A
Initials - 1#2A

BEGIN

ACCEPT BATCH #

IS BATCH ON FILE?

Y

ACCEPT BATCH #

WRITE DESCRIPTION
GET DELNUM

FOR I: = 1
UNTIL DELNUM

ACCEPT SERIAL #

LOCATE SERIAL #

FIND ANYTHING?

N

LOCATE BATCH #

FIND ANYTHING?

N

IS SERIAL # CORRECT?

Y

SAVE ON ARCHIVES
REMOVE FROM BATCH FILE
REMOVE FROM CHAR FILE
REMOVE FROM SERIAL FILE
COMPRESS ANY EMPTY BUCKET

N

IF NECESSARY, REMOVE AND COMPRESS DESCRIPTION

COMPRESS BATCH FILE
COMPRESS SERIAL FILE

RETURN

N
Subroutine: Modify (BatArray, Bucket, Record, Date, Initials)

Type List:

- BatArray - I * 2 A
- Bucket - I * 4 V
- Record - I * 4 V
- Date - I * 2 A
- Initials - I * 2 A

Description: This subroutine obtains modifications to a batch record from the user. The record is contained in BatArray and is to be copied back to record number Record of bucket number Bucket of the batch file. In addition, any changes affecting the characteristic file must be taken care of, and the modified batch record must be saved on the archives file. Date and initials refer to the date of modification and the initials of the person responsible for it.

Details: It is a waste of time and archives space to write anything to the various files, unless the record was actually altered. The logical variable Altered indicates whether a change was in fact made.

Pitfalls: None known.
SUBROUTINE MODIFY

PARAMETERS
BatArray - 1*2A
Bucket - IV
Record - IV
Date - 1*2A
Initials - 1*2A

BEGIN
ALTERED: = FALSE
READ BUCKET (CHAR FILENUM . . .)

WRITE RECORD (2, BATARRAY)

MODIFY WHAT?

DOC, #
ISOTOPES
CHANGE DOC #
ALTERED: = TRUE

SUBASSEMB. #
CHANGE ELEMENTS
CHANGE MASSES
ALTERED: = TRUE

USER
CHANGE SUBASSEMB. #
ALTERED: = TRUE

HELP
WRITE HELP

CHANGE USER
ALTERED: = TRUE

ALTERED?
Y
WRITE RECORDS TO
BATCH FILE, CHAR
FILE, AND ARCHIVE

RETURN

QUIT
Subroutine: Mix (Key, Length, Result)

Type List: 

<table>
<thead>
<tr>
<th>Type</th>
<th>Key</th>
<th>Length</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>I * 2 A</td>
<td>I * 4 V</td>
<td>I * 4</td>
</tr>
</tbody>
</table>

Description: This subroutine converts an array, Key, of Length bytes, into a pseudorandom number, Result, between zero and ten million. Key usually contain ASCII data, but may also contain numeric data.

Details: Actually, Result is returned as an integer on the closed interval between zero and 9999990. According to Knuth, an adequate pseudorandom number R (X) is given by R (X) = (A*X + C) mod M. A, C, and M must be relatively prime. Knuth also provides tables of large primes. From these, I have chosen A = 2097143, C = 32749, and M = 9999991. Using these, the string of bytes in Key is converted to a set of pseudorandom numbers. These are combined into a single number, using a method similar to the above, which causes permutations of Key to provide different Results.

Pitfalls: Observe that Length refers to bytes, not words.

See: Ref. 1, pp 506-570.
SUBROUTINE MIX

PARAMETERS

Key - 182
Length - 14
Result - 1

BEGIN

MODULUS: = 999999
FACTOR: = 2097143
CONSTANT: = 32749
PRIME: = 47

RESULT: = 0
UNPACK (KEY, STRING, LENGTH)
CALL INTLEN (.FALSE.)

FOR BYTE: = 1
UNTIL LENGTH

NEXTBYTE: = ISHFT (STRING (BYTE), -8)
NEXTNUMBER: = MOD (FACTOR * NEXTBYTE + CONSTANT, MODULUS)
RESULT: = MOD (PRIME * RESULT + NEXTNUMBER MODULUS)

RETURN
Subroutine: HashBatch (BatchItem, Bucket)

Type List: BatchItem - I * 2 A
Bucket - I * 4

Description: This subroutine computes a hash function for use with the batch file. BatchItem is a 10-word array consisting of 16 bytes and one Integer*4 number. The bucket number assigned to BatchItem is returned as Bucket.

Details: The string BatchItem is converted to a pseudorandom number between zero and ten million, using the subroutine Mix. Then Bucket is set to a number between one and Filesize using the MOD function. Note that Filesize equals BatOverflow minus one.

Pitfalls: None known.
SUBROUTINE HASH BATCH

PARAMETERS

BatchItem - 1*2A
Bucket - 1

BEGIN

INITIALIZE FILESIZE
MIX (BATCH ITEM, 20, RESULT)
BUCKET: = MOD (RESULT, FILESIZE) +1

RETURN
**Subroutine:** HashDesc (Batchnumber, Bucket)

**Type List:**
- Batchnumber  I * 2 A
- Bucket        - I * 4

**Description:**
This subroutine computes a hash function for use with the description file. Batchnumber is a 16-byte array. The bucket number assigned to Batchnumber is returned as Bucket.

**Details:** Essentially the same as for HashBatch.

**Pitfalls:** None known.
SUBROUTINE HASH DESC

PARAMETERS

Batchnumber - 1*2A
Bucket - 1

BEGIN

INITIALIZE FILESIZE
MIX (BATCHNUMBER, 16, RESULT)
BUCKET: = MOD (RESULT, FILESIZE) +1

RETURN
Subroutine: HashSerial (Serialnumber, Bucket)

Type List: Serialnumber - I * 2 A  
Bucket - I * 4

Description: This subroutine computes a hash function for use with the serial file. Serialnumber is an 8-byte array. The bucket number assigned to Serialnumber is returned as Bucket.

Details: Essentially the same as for HashBatch.

Pitfalls: None known.
SUBROUTINE HASH SERIAL

PARAMETERS

Serialnumber - 1x2A
Bucket - 1

BEGIN

INITIALIZE FILESIZE
MIX (SERIALNUMBER, 8, RESULT)
BUCKET: = MOD (RESULT, FILESIZE) +1

RETURN
APPENDIX B

Initialization Procedure

1. For each file, decide the maximum number of items to be stored on the file. The same number should apply to the batch, serial, and characteristic files. A much smaller one applies to the description file.

2. Use the program Bucket to find, by trial and error, the number of buckets necessary for each file. The program first asks for the number of items. Use the number decided on in step 1. It then asks for the number of buckets. Make a guess. (A good first guess is the number of items divided by the bucket capacity.) Finally, the program asks for a random number. Give it a three- or four-digit number. After some thought, the program prints a table showing the distribution of items among the buckets. The left column is the Recordcount. The right column is the expected number of buckets with Recordcount records in them. Run the program several times, continually adjusting the number of buckets, until practically no buckets have a Recordcount greater than the bucket capacity (either 12 or 24).

3. Add one to the number of buckets obtained in step 2. This is the first overflow bucket. Go through the source code and change all the initializations of this overflow bucket. They have the form "xxxOVERFLOW = ....." where XXX is BAT, SER, DESC, or CHAR. Note that BATOVERFLOW and CHAROVERFLOW should be the same number.

4. Determine the sizes of all files. These should include a reasonable number of buckets in the overflow area. Change the DEFINE FILE statements in the main program and in the InitFiles program. Change the Filesize initializations in the hash subroutines. These take the form "FILESIZE = .....". The Filesize should be the number of buckets not in overflow. Create disk-resident files with the appropriate sizes. These should be "partitioned data sets" on MODCOMP systems.
5. Compile and execute the program InitFiles. It will write zeroes to the first 64 words of each bucket in each file. When it asks for the number of buckets, be sure to include those in overflow.

6. Recompile the entire data storage and retrieval program. It should now execute properly.
BEGIN
DEFINE FILES
FOR I := 1 UNTIL 16
BLANK (I) := 0
FOR FILENUM := 500 UNTIL 800 STEP 100
GET BUCKET SIZE
GET FILE SIZE
FOR RECORD := 1 UNTIL FILESIZE STEP BUCKETSIZwE
WRITE (FILENUM, RECORD)
(BLANK (I), I = 1, 16)
END
PROGRAM BUCKET

BEGIN

INITIALIZE TABLES
REQUEST USER'S INPUT

FOR ITEM = 1 UNTIL N

INSERT "ITEM" INTO RANDOM BUCKETS USING "HASH" FUNCTION

FOR BUCKET = 1 UNTIL M

BUILD DISTRIBUTION OF TOTAL ITEMS VS. NUMBER OF ITEMS PER BUCKET

DETERMINE NON-ZERO LIMITS OF DISTRIBUTION: FIRST, LAST

FOR I = FIRST UNTIL LAST

COMPUTE AVERAGE BUCKET OCCUpancy, TOTAL BUCKETS USED, STD. DEVIATION

PRINT RESULTS

END
APPENDIX C
Operating Environment

The inventory system was written for, and operates as, a MAX IV BATCH overlay program on a multiuser MODCOMP 7860 computer system. The FR5 FORTRAN compiler was used to compile all programs in this package. The FR5 compiler supports all standard FORTRAN IV features and some FORTRAN 77 ones as well. Generally the binary code for each subroutine was stored on a temporary library file. After full debugging, the subroutines were then link-edited with the Data Storage and Retrieval Program and vendor-supplied "system" routines. The edited program is stored in "quick-load" format using the Task/Overlay Cataloger. Except for use of two- and four-byte integers and certain MODCOMP Library subroutines, the coding should be compatible with most FORTRAN implementations.

The program size is less than 128 000 bytes. No attempt has been made to use the "segmentation" or "shared load module" features of the operating system, although the program lends itself readily to segmentation. Several disk files are used, a line printer is used for output listings, and a backup device to store archive files is needed. Operating-system details are given in Ref. 4.

No attempt was made to measure the program execution speed; we have noted, however, that the retrieval routines generally keep up with the terminal output (9600 baud) for the file sizes shown in the listings.

Those interested in installing this system on a non-MODCOMP computer should have minimal difficulty, provided that their FORTRAN compiler supports two- and four-byte integers, and that system library programs exist to return the current date, perform Boolean and shift operations, etc. Different file-handling routines may be needed; however, use "direct access" file routines which are straightforward "reads" and "writes" to a particular record in a specified file.
APPENDIX D

Program Development Effort

The Special Nuclear Materials accountability system was written by a summer student, R.S. Ingermanson, during June-August 1980 with the support of the Nuclear Engineering Institute of the Argonne Division of Educational Programs. The original concepts used in designing the system were developed by A. E. Proctor, Nondestructive Assay Laboratory Manager. User input and comments were provided by J. J. Winkel, Special Materials Representative. The overall effort to design, write, and debug the program was less than four-man months. Although some document printing and archive file routines are not in their final form, we believe that this system compares favorably to others requiring man-years of effort to complete.

The rapid completion of this package may be attributed to many factors. The most important of these are: an understanding of the desirable characteristics of an accountability system and the capabilities of the available computing equipment, the expertise and diligence of the programmer/analyst, and (because this was a student project) the elimination of the usual design, review, and documentation phases at the beginning of the project.

ACKNOWLEDGEMENTS

The authors acknowledge the background information and helpful suggestions from the ANL-W Special Materials Section, especially W. P. Keeney, Manager, Special Materials, W. R. Vroman, Nondestructive-assay Laboratory, and C. R. Garcia and J. J. Winkel, Special Materials Representatives.
REFERENCES


