A major purpose of the Technical Information Center is to provide the broadest dissemination possible of information contained in DOE's Research and Development Reports to business, industry, the academic community, and federal, state and local governments.

Although a small portion of this report is not reproducible, it is being made available to expedite the availability of information on the research discussed herein.

LA-UR -89-2242

LA-UR--89-2242

DE89 015240

TITLE SNM ACCOUNTING SYSTEMS - DBASE VERSUS C

AUTHOR(S) R. C. Bearse and R. M. Tisinger

SUBMITTED TO 30TH Annual Meeting of the Institute of Nuclear Materials Management, Orlando, July 9-12, 1989

DISCLAIMER

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

SNM ACCOUNTING SYSTEMS - dBASE VERSUS C*

R. C. Bearse,** R. M. Tisinger, and J. S. Ballmann
Safeguards System Group
Los Alamos National Laboratory
Los Alamos, NM 87545

ABSTRACT

The Fuel Manufacturing Facility (FMF) at Argonne National Laboratories-West (ANL-W) in Idaho Falls accomplishes its internal special nuclear material accounting with a PC-based DYnamic Material ACcounting (PC/DYMAC) system developed as a collaboration between FMF and Los Alamos National Laboratory staff members. This system comprises four computers communicating via floppy disks containing transfer information. The accounting software was written in dBASE† and compiled under Clipper.†† The decision was made to network the computers and to speed the accounting process. Moreover, it was decided to extend the collaboration to Sandia National Laboratory staff and to incorporate their recently developed CAMUS and WATCH systems to automate data input and to provide a measure of material control. The current version of the code is being translated into the C language. The implications of such a change will be discussed.

INTRODUCTION

The Fuel Manufacturing Facility (FMF) at Argonne National Laboratories-West (ANL-W) in Idaho Falls accomplishes its internal special nuclear material (SNM) accounting with a PC-based DYnamic Material ACcounting (PC/DYMAC) system developed as a collaboration between FMF and Los Alamos National Laboratory staff members.[1] This system comprises four computers communicating via floppy disks containing transfer information. The computers are IBM/PC-ATs,§ each with 2 MB of memory and a 40-MB hard disk. The PC/DOS* operating system runs the accounting software, which is a Clipper-compiled version of the dBASE code.

The original system design was dictated by circumstances. The facility was then spread over several buildings that could not be linked by wire. Transactions were posted by carrying the information from one computer to another via floppy disks. Subsequently ANL-W moved the FMF and all its operations into a single building and has at least doubled its throughput. Because all operations are now in a single building, the need for transferring data via floppy disks is obviated. The increase in throughput has exceeded the ability of the dBase code to perform quickly enough to keep up. Careful time scheduling is necessary to synchronize production and accounting. Thus, a new system was called for.

The decision was made to network the computers and to speed the accounting programs. Moreover, it was decided to extend the collaboration to Sandia National Laboratory staff and to incorporate their recently developed CAMUS and WATCH systems[2] for automatic data input and a measure of material control. (CAMUS allows barcode data entry of information for transactions, and the WATCH system sends an RF alarm if an unattended item is disturbed without permission.)

SOFTWARE AND HARDWARE

We believed that the XENIX** operating system (a variant of UNIX†) would facilitate the intercomputer communication and allow the simultaneous operation of several processes. We originally sought a XENIX-based version of dBASE and, after some research, determined that SCO FoxBASE+†† would fit our needs. Use of this system minimized

*This work is supported by the U.S. Department of Energy, Office of Safeguards and Security.

**Collaborator from the University of Kansas.

†Trademark of Ashton-Tate, Torrance, CA 90502.

††Trademark of Nantucket, Inc., Culver City, CA 90230.

§Registered trademark of the International Business Machines Corporation.

*Trademark of Microsoft Corporation.

**Trademark of Santa Cruz Operations, Inc. (SCO), Santa Cruz, CA 95061-9969.

†Registered trademark of Bell Laboratories.

††The Xenix version is available through SCO.

changes to the original PC/DYMAC code and provided a small speed improvement over the original. Moreover, in principle, the accounting system could be run as one process among many. The original system included a communications package, a transaction poster, a transaction builder, and a CAMUS interface, which ran as concurrent, separate processes.

An IBM PS/2 Model 80 running an 80386 processor at 20 MHz was procured to replace the central computer. The FoxBASE+ system worked effectively on that computer and improved the speed two- to three-fold over the Clipper-based system currently in use. Running on the IBM ATs, however, speed was not enhanced overall, and the process swapping necessary in the new design caused disconcerting data entry delays that threatened the usefulness of the final product. Most importantly, it was clear that there would be no reserve capacity to accommodate increased manufacturing productivity.

We decided to rewrite the code in the C language* using the FairCom** software package c-tree as the file handler. Time constraints did not allow a detailed study of available file handlers. One of us had contact with University of Kansas programmers† who had had success with c-tree for their application. Their experience clearly indicated that the FairCom product was efficient, reliable, and suited to our task.

EXPERIENCE GAINED

File Handler

The c-tree product and its companion report generator r-tree have worked flawlessly. c-tree can work in each of three modes: as a stand-alone file handler, as a multi-user file handler, and as a file server. We chose the first approach because the manual suggested that the necessary overhead for multi-user and file-server modes would significantly slow the design of the system. Our design is such that access to the inventory file is through the same singly-threaded modules, so the simpler access procedure is acceptable.

The c-tree and r-tree codes are warranted to be portable. They are delivered as C-language source code. Installation instructions are included for UNIX, XENIX, PC/MS-DOS and VMS.†† For the XENIX system, a distinction is made for installation on 286 and 386 versions.

We have found the c-tree package to be absolutely bug-free; it is more than fast enough for our purpose; it has capacity (such as multi-user and variable length record capabilities) that we have not begun to explore.

The report writer is easy to use once it is understood. A compiled driver is part of the main code and calls report scripts that are maintained as ASCII text files. The driver interprets these files and generates the report. Thus, it is possible to change reports or design new ones without having to recompile. The only drawback so far is the lack of a way to program the script file to ask the user to enter data. Presently, we are forced to modify the report driver to accomplish this. This may simply be caused by an incomplete understanding of the r-tree package.

A d-tree package completes the set. It appears to be a development package that allows the user to "easily" produce database-type applications. The effort required to understand the other packages has not encouraged us to explore d-tree further at this time.

C-CODE VERSUS dBASE

The C language has much to recommend it. It is a far older language than many care to recognize (its direct predecessors are older than FORTRAN). The adoption of C does not require FORTRAN programmers to relearn completely solutions to programming problems as would LISP or even INGRES.* C is basically a linear, block-structured, function-based language like FORTRAN 77 and Pascal.

When the recoding was begun, we had little experience with XENIX and essentially none with C. Even with that limited background, translation of the basic system took less than 300 person-hours. The program at that point consisted of about 130 routines, each varying in size from 10 lines of code to several hundred; not counting declarations in "include" files, there were more than 4000 lines.

The second phase of the coding consisted of finishing all the necessary transaction modules, various utilities, and generators for the 23 reports used at FMF on a regular basis. Because we had learned much by then concerning the C language and XENIX, we saw that a significant restructuring of the entire system would be necessary. In particular, we improved the screen handlers and data entry subroutines, and incorporated a mouse interface into the system. Finally, we developed, tested, and added the communication packages to the system. The code now consists of more than 200 subroutines and 12 report scripts and is now roughly 10 000 lines long. Our total investment is nearly 1000 person-hours.

Our experience to date has illumined several differences between dBASE and C language codes

*We used the SCO C-language compiler.

**FairCom, Columbia, MO 65203.

†Data supplied by David Mannering, University of Kansas.

††Trademark of Digital Equipment Corporation, Maynard, MA 01754-9903.

*Trademark of Relational Technology, Inc., Berkeley, CA 94705.

that are worth discussing here. Moreover, our choice of C was based on widely held assumptions about the C language, some of which now seem justified, others of which need more study.

Speed

At the start of debugging, we compared the speeds of the system now operating at ANL-W and the recoded one. A set of 100 transactions, each requiring a rewrite of the inventory file, takes about 500 seconds on the old system, but only 20 seconds on the new system. This 25-fold speed enhancement is even more impressive when it is remembered that the old code is compiled under Clipper, which is invariably faster than interpreted dBASE code.

Portability

One of the highly touted advantages of C is its portability. It is supposed to be easy to move code from one computer to another. Compilers are available on a wide variety of machines, and the highly portable UNIX operating system is largely coded in C. We are not yet able to testify to the portability, although we hope to port it to VMS soon.

We did find, however, what the experts already knew--that portability is not automatic.[3] We were shocked to discover that the first version of our code, which ran well on an 80286 processor, would not run on an 80386. We finally determined that the problem was, as usual, ours. Integers on the two machines had different byte lengths, and we were not carefully controlling our declarations to ensure compatability. By revising the code and carefully adopting standards that we believe are now machine independent, we seem to have solved this problem. We are not, however, ready to move the code to a completely different architecture such as VMS.

Record and Field Sizes

In dBASE, numbers are stored as ASCII strings. Thus, a 10-digit integer is stored in 10 bytes as is a 9-digit floating point number. In C, integers can be stored as 2 or 4 bytes and floating numbers as 4 or 8 bytes. Four-byte real numbers will accurately hold values of seven significant figures. We have chosen to use 4 bytes for storing floating point assay results, but use 8 bytes when performing calculations.

Thus, we can compress the size of an inventory record from 916 bytes in the dBASE version to only 260 bytes in the C version. This not only saves space but affords some speed improvement.

Indexed Variables

dBASE does not naturally allow the use of indexed variables. C provides two equivalent methods of accessing arrays. One can use the form var[j], which represents the jth element of the array var (j starts at zero) or the pointer method *(var+j). In dBASE it is possible to achieve the equivalent of indexed variables by creating variable names from the index itself, but this technique is not natural and is difficult to follow. Compilers, such as Clipper, obviate this problem by providing subscripting as an extension to the dBASE language.

Structures

The C language allows structures. Users of Pascal will recognize immediately the concept and the syntax. A structure is defined by declaring the elements (fields) that compose it. An element within a structure can then be referred to as "struct.element," or in pointer notation, as "struct->element." Structures may be nested. We have taken advantage of this by defining a structure "amt," which represents each chemical and isotopic assay of an item. We use this structure in two larger structures: the inventory and the transaction. An inventory record in our system consists of demographic information (such as where the object is and how it relates to others) and quantitative information including its chemical analysis. The analysis is in the substructure "inventory.amt." To reference the uranium amount, one need only write "invent.amt.uran." Similarly, the transaction is made up of three parts--an xn part representing the change, and two inventory amounts: the source and destination inventory records. The fact that the amt substructures in each of these substructures are the same provides some powerful advantages as will be shown below. Structure and substructure addresses and values can be passed at will between subroutines and other program parts providing a significant economy of code and hence improving program lucidity.

Subprograms

Although dBASE only tolerates subprograms, C encourages them. With structures and pointers, it is possible to write very powerful modules to manipulate data entities and to call them succinctly from the calling program. This succinctness makes the code easier to understand. Care in the design of these functions and copious internal error checking can be a powerful aid to rapid, accurate development of code.

Close Relationship with the Operating System

The C language was used in developing the XENIX operating system; thus, its use permits interactions with the operating system that would not be natural or fast with other languages. dBASE is particularly awkward in its interactions with the operating system.

Because much of the XENIX operating system is itself written in C, it presents a well-defined and predictable interface to C code. We have found that opening and closing of files is at least ten times faster than with dBASE. We have not bothered to implement indexed files that maintain allowed values for fields, such as product descriptions and locations, because there is no perceptible delay in system operation without them.

Generic Modules

The Holy Grail of SNM-accounting-system writers has been the generic accounting system. We have been involved with the development of at least three systems hoped to be generic, and each fell far short of the goal. C may provide, however, a way to develop a significant number of modules that would be generic and portable. A system specific to a particular plant could be built more cheaply using some of these modules rather than starting from scratch. We will discuss one such module in detail. The example we have chosen also points out some of the other features of C that have been indicated above.

Figures 1-3 show fragments of C code. The first file is part of an "include" file that contains the declaration of an "amountform" ("amt") structure, which contains the assay amount. Figure 2 shows the first code that was written to add one amount record to another. The very basis of an accounting system is crediting and debiting. Adding one "amt" record to another is clearly generic. This is the routine that credits an inventory record with a transferred amount. It is quite clear to the average programmer, be he or she FORTRAN or Pascal adept. Although the crediting process is generic, the fields chosen here are not, and thus it would be necessary to rewrite the code shown in Fig. 2 each time different elements are chosen.

Contrast this with Fig. 3. Here the amount substructure consists of a contiguous sequential series of 4-byte segments, each segment representing a floating point number. To change the meaning of these fields, it is only necessary to change the declarations in the "include" file. The "sizeof" operator automatically determines the size of the amount substructure and allows the code to cut off the scan of the 4-byte segments at the appropriate place. This code can be made independent of the choice of 4- or 8-byte numbers so long as all the elements of "amt" are of the same type, contiguous, and sequential.

Virtues of dBASE

dBASE has some obvious advantages over C. It can be used interactively--a powerful advantage when debugging or attempting to develop a strategy to accomplish a computing task. Files can be manipulated ad hoc, aiding in revising field meanings and correcting errors. dBASE is certainly more user friendly and easier to learn.

CONCLUSIONS

The C language has some serious drawbacks. Its compilers are very tolerant, making it possible to legally write code that manipulates other code far away in space and time from itself. We often found bugs arising from the addition of a new piece of code that were not due to the new code but to a third piece of code that was writing into an unintended location. The new code simply provided a place for the old code to make its malignant effect manifest.

```
/* STRUCTURE FOR AMOUNT TYPE RECORDS */
/* INCLUDE FILE RGSTRC.H */

typedef float FLOAT;
typedef char CHAR;
typedef unsigned short USHORT;
typedef long LONG;

struct amountform {
     FLOAT netamt;
     FLOAT alloy;
     FLOAT uran;
     FLOAT u 234;
     FLOAT u 235;
     FLOAT u 236;
     FLOAT u 238;
     FLOAT pu;
     FLOAT pu iso;
     FLOAT pu 239;
     FLOAT pu 240;
     FLOAT pu 241;
     FLOAT pu 242;
     FLOAT du;
     FLOAT du 235;
     FLOAT zr;
     FLOAT ta;
     FLOAT si;
     FLOAT mo;
     FLOAT ru;
     FLOAT rh;
     FLOAT pd;
     FLOAT other;
};
/*STRUCTURE FOR INVENTORY TYPE RECORDS */
struct inventform {
     CHAR  type;
     LONG  serial no;
     LONG  col no;
     LONG  supcol no;
     USHORT col count;
     USHORT batch seq;
     USHORT slug no;
     USHORT position;
     USHORT year;
     CHAR  batch no[RGSZbatch no];
     CHAR  jacket no[RGSZjacket no];
     CHAR  prod desc[RGSZprod desc];
     CHAR  spm no[RGSZspm no];
     CHAR  tran no[RGSZtran no];
     CHAR  status;
     CHAR  xn tag[RGSZxn tag];
     CHAR  room[RGSZroom];
     CHAR  zone[RGSZzone];
     CHAR  container[RGSZcontainer];
     CHAR  remarks[RGSZremarks];
     CHAR  mescod;
     CHAR  collection[RGSZcollection];
     FLOAT length;
     FLOAT avg diam;
     FLOAT min diam;
     FLOAT max diam;
     FLOAT density;
};
```

Fig. 1.
Part of the include file that declares the inventory structure.

```
# include "rgstrc.h"

VOID increment(origp, changep)
struct amountform *origp;
struct amountform *changep;
{
     origp->netamt  += changep->netamt;
     origp->alloy   += changep->alloy;
     origp->uran    += changep->uran;
     origp->u 234   += changep->u 234;
     origp->u 235   += changep->u 235;
     origp->u 236   += changep->u 236;
     origp->u 238   += changep->u 238;
     origp->du      += changep->du;
     origp->du 235  += changep->du 235;
     origp->pu      += changep->pu;
     origp->pu 239  += changep->pu 239;
     origp->pu 240  += changep->pu 240;
     origp->pu 241  += changep->pu 241;
     origp->pu 242  += changep->pu 242;
     origp->pu iso  += changep->pu iso;
     origp->mo      += changep->mo;
     origp->pd      += changep->pd;
     origp->rh      += changep->rh;
     origp->ru      += changep->ru;
     origp->si      += changep->si;
     origp->ta      += changep->ta;
     origp->zr      += changep->zr;
     origp->other   += changep->other;
}
```

Fig. 2.
The original program to increment the fields kept at ANL-W. The routine is passed two pointers—a pointer to an inventory record and a pointer to a transaction record. The inventory fields that are the part holding the amounts are incremented by the amount in the corresponding fields in the xn part of the transaction record. The += syntax means "add the right hand side to the left hand side and store the result in the left hand side."

These potential problems can be kept under control by careful use of debugging devices when developing modules. A major aid is the "lint" program supplied with all C compilers. A program that compiles may generate warnings when it passes through lint. We have usually found that what appear to be innocuous warnings from lint are ignored at peril.

Even lint will not always catch errant pointers. C's pointer capability will allow you to write to areas of code or data that you should not. Unfortunately, there is no convenient way to eschew the use of pointers. We have found that a debugging tool developed by White[4] is a powerful method for catching such errant pointers. A side benefit is that, when in operation, it provides a convenient tracing method. This debugger is incorporated in a routine that dynamically allocates space. We maintain two space allocation routines: one that includes the debugger and one that does not. We need only relink (a matter of less than a minute on the PS/2 Model 80) to go from one to the other. The debugger does slow the system down noticeably, but not so much that it will inhibit alpha or beta testing.

```
# include "rgstrc.h"

VOID increment(origp, changep)
struct amountform *origp;       /* ptr to original record */
struct amountform *changep;     /* ptr to increment record */
{
     INT max;           /* number of floating point numbers in structure */
     INT k;             /* loop index */
     FLOAT * origf;     /* ptr to a floating point field */
     FLOAT * changef;   /* ptr to the field of the increment */

     /* get the pointer for the first floating variables */
     origf   = (FLOAT *)origp;
     changef = (FLOAT *)changep;

     /* determine the number of floating point numbers in structure */
     max = sizeof(struct amountform ) / sizeof(FLOAT);

     /* handle each one in turn */
     for (k = 0; k < max; k++) {

          /* increment the original number by the amount of change */
          *(origf++) += *(changef++);
     }
}
```

Fig. 3.
A generic program to increment floating point fields in a structure of floating fields. This code serves exactly the same purpose as the code in Fig. 2. Here, however the power (and complexity) of the pointer make the module generic. It will work with any list of items in the structure amt, so long as each is of type FLOAT. Moreover, FLOAT could be redefined as DOUBLE, LONG, or INT without changing the code in the subroutine. The only requirements are to change the definitions of FLOAT and the structure amountform and to recompile the subroutine increment. The (FLOAT *) notation casts the term to its immediate right into the variable type pointer-to-a-float. The notation ++ after the variable means "add one to the variable after the original value has been used."

Proper attention to the portability-enhancing features of C may indeed allow portable programs, but we have much to learn before we see why it would be superior to a carefully designed FORTRAN program. The use of pointers may make standard generic packages possible that could ease the task of building an accounting package.

The C language is well worth considering for building an accounting system, particularly a new one. If the program is properly designed, the programmers need not fear hardware upgrades. A brief examination of computer journals indicates a bewildering variety of C support products for every purpose and operating system.

The C language is not difficult to learn. FORTRAN or Pascal programmers, particularly the latter, will find the transition easy. The major learning barrier is the concept of indirect addressing (pointers), particularly because argument passing requires their use. Good books[5-6] and support groups are a must, and we have included some suggestions.

It has taken us 6 months to get to where we are now, and we have still much to learn about the potentials and problems of C. Plan for a learning curve if you assign FORTRAN or Pascal programmers to such a task. A seasoned C programmer as part of a team would prevent many false starts.

REFERENCES

1. R. C. Bearse, R. J. Thomas, S. P. Henslee, B. G. Jackson, D. B. Tracy, and D. M. Pace, "A Materials Accounting System for an IBM PC," Nucl. Mater. Manage. **XV**, 373-378 (1986).

2. S. Anthony Roybal, Stephen Ortiz, and S. Paul Henslee, "Demonstration Personnel and Material Tracking System at ANL-W," Nucl. Mater. Manage. **XVII**, 789-793 (1988).

3. The C Users Journal 7(1) (1989). (The entire issue is devoted to portability.)

4. Eric White, "Controlling the Malloc Heap," The C Users Journal 7(2), 45 (1989).

5. B. W. Kernighan and D. M. Ritchie, The C Programmer Language (Prentice Hall, Englewood Cliffs, New Jersey, 1988).

6. A. Koenig, C Traps and Pitfalls (Addison-Wesley Publishing Company, Reading, Massachusetts, 1988).

7. H. Schildt, C: The Complete Reference (Osborne McGraw Hill, Berkeley, California, 1988).

8. R. Ward, "Debugging C," Que Corporation, Carvel, Indiana (1988).

9. The C Users Journal, 2120 W 25th St., Lawrence, KS 66046.