C.5

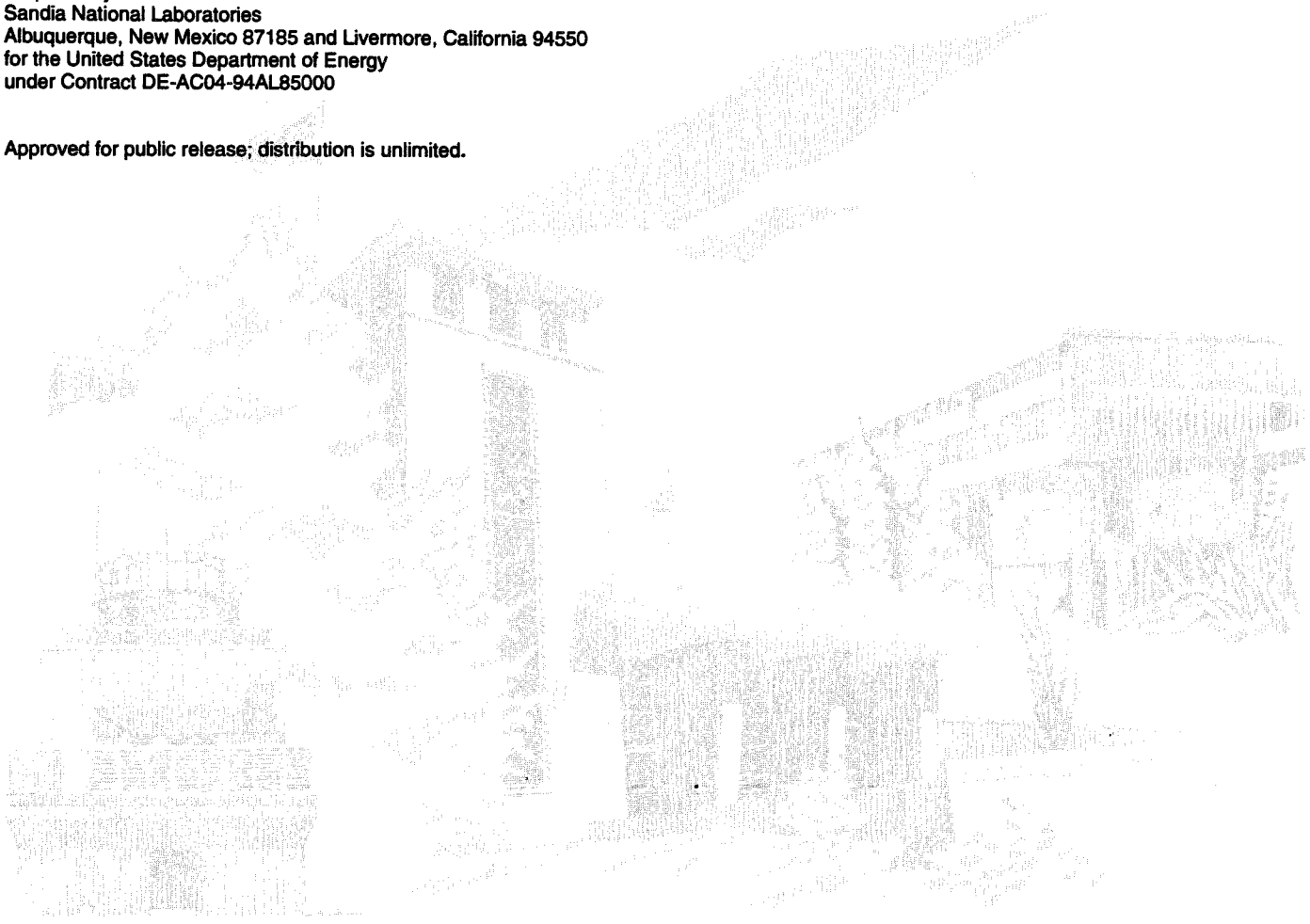
SANDIA REPORT

SAND96–0166 • UC–706
Unlimited Release
Printed January 1996

Storage and Retrieval of Nuclear Test Data

Samuel D. Stearns

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550
for the United States Department of Energy
under Contract DE-AC04-94AL85000



SF2900Q(8-81)

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
PO Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
US Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

SAND96-0166
Unlimited Release
Printed January 1996

Distribution
Category UC-706

STORAGE AND RETRIEVAL OF NUCLEAR TEST DATA

Samuel D. Stearns
Materials Radiation Science Department
Sandia National Laboratories
Albuquerque, NM 87185

Abstract

This report is a part of the Test Information Program (TIP) at Sandia National Laboratories. It is an interim report, written primarily as an instruction and reference document to aid in current work on the project. It addresses some of the issues found in storing and retrieving data from nuclear field tests conducted over the past five decades, primarily instrumentation data recorded from tests at the Nevada Test Site.

First, the *TIP data unit* for storing and transporting TIP data is described. The data in the TIP data unit is typically recorded in a universal medium such as the portable optical or magnetic disk, or the tape cassette. Each TIP data unit is portable, and is also self-contained in the sense that it includes a set of related test data files, along with complete instructions and software for retrieval of the data by an unknown user, possibly on an unknown platform.

Secondly, we describe the use of current software for compressing and decompressing waveform data, for authenticating and checking for errors in data files, and for processing files to be used on foreign platforms.

Preface

This is a report on the storage and retrieval of nuclear test data, which, since the cessation of testing by the United States, has become an irreplaceable national archive. We are concerned here with some of the details of the Test Information Program (TIP), which has the goal of preserving the test results for future users who will do unknown and unpredictable things with the data, and will, no doubt, insist that the data be reliable.

We wish to acknowledge the source of the test data, which has become a legacy of the thousands of scientists and technicians, at Sandia Labs and elsewhere, who have dedicated their careers to the field test program. We leave the questions of why we expended so much effort to develop and test nuclear weapons, and of whether we are likely to do so again in the near future, as separate issues. We simply note that the cost of the data that we now possess, in terms of lives and dollars, certainly implies that a reasonable amount of effort be expended to preserve it.

Concerning the software described in this report, we also acknowledge with thanks the efforts of others who contributed to the research on waveform compression that led to the development of our present compression and decompression routines. We are especially grateful to Prof. D.M. Etter and Jonathan Haines at the University of Colorado, to Prof. N. Magotra and his students at the University of New Mexico, and to Prof. W.B. Mikhael and his students at the University of Central Florida.

Also, we wish to thank P. L. Nelson and B. C. Bedeaux in the Nevada Operations Department at Sandia Labs for managing and leading the TIP project, as well as J. Pearcey, R. M. Clancy, and T.S. Caldwell for managing and processing the data.

Contents

Figures

(blank page)

STORAGE AND RETRIEVAL OF NUCLEAR TEST DATA

Introduction

This report is an element of the Sandia Labs Test Information Program (TIP). It has two purposes. The first is to describe in simple and general terms the *TIP data unit*, which is proposed to be a basic unit in which TIP data is stored and transported from one location to another. The TIP data unit is described in the next section.

The second purpose is to introduce and describe software that we are now using for efficient and reliable storage and retrieval of test waveform data. The software consists of program modules used for compression and decompression of waveform data, as well as program modules for attaching cyclic redundancy checks to compressed data files, and program modules for decoding the redundancy checks, checking for errors, and authenticating the compressed data.

The overall goal is to preserve waveform and other test data which, in the presence of a moratorium on testing, forms an important national archive. Detailed digital instrumentation data is presently available from dozens of underground tests conducted over several decades, with yields ranging from well over a megaton down to very low levels. Future research may pose new questions on effects, vulnerability, and other design aspects -- questions that require a look at the details of these tests. If and when testing is resumed, the present data from large numbers of tests already conducted will form an important basis for comparison with new results.

To achieve this overall goal, we would like to be able to predict a storage medium that will be convenient for future users. Our current candidate for the medium with the greatest viability is the PC-DOS-compatible, compact disk read-only memory (CD-ROM). Currently the CD-ROM is the medium of choice in the entertainment industry. It is sold in computer, music, discount, and even grocery stores, and has made its way into the average American home; hence we believe that personal computers with CD-ROM readers are here to stay for the foreseeable future. Track formats may change, and storage capacities are likely to increase, but the CD-ROM in its basic form should be around for a while.

The correct choice of the storage medium, or even of the PC-DOS platform, is not critical. The data can always be copied to a new platform or to a new medium. What is critical, we believe, is the arrangement of test data into *data units* that are self-sufficient and self-explanatory -- units that depend as little as possible on specific computing equipment and do not depend at all on software external to the data unit itself. Hence we arrive at the concept described in the next section.

The Data Unit

We do not wish to propose here a detailed, all-purpose data storage format. There is no such thing, and experience has shown that any attempt in this direction leads to immediate revision, followed soon by obsolescence. Instead, we are proposing only a simple and obvious concept which, if followed, will help to keep the test data available to future investigators who do not have access to the original field test personnel, or to expository data other than that in the data unit itself.

Our concept of the data unit is shown in Fig. 1. Its main feature is its stand-alone capability, that is, its independence of any current hardware or software or procedures other

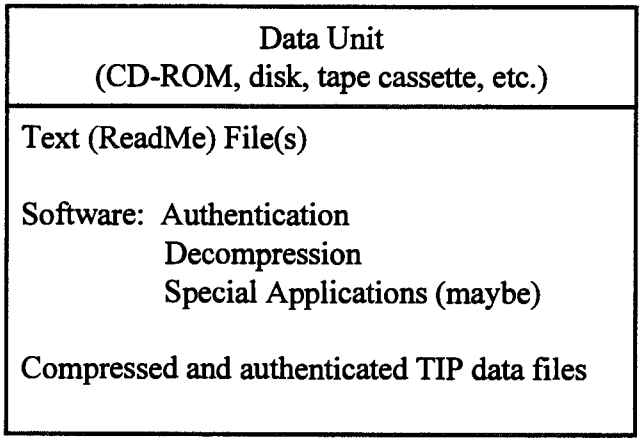


Figure 1. Contents of the proposed data unit.

than the current platform and operating system. We assume that at least one of the platforms used will be the personal computer (PC) running the well-known and currently standard Disk Operating System (DOS). Thus far, new versions of DOS have always been backward-compatible, and PC-DOS systems exist almost everywhere.

Each data unit must contain one or more ASCII text (ReadMe) files that can be easily read or printed. These files contain all necessary information on the disk contents, history and reliability of the data, descriptions of directory and file structures, instructions for authenticating and decompressing the data, etc. In short, the ReadMe files should contain all the information needed by an uninformed user who wishes to examine the data without outside help. We must assume that the user has no information about the data unit contents, other than the information stored in the data unit itself.

There may be more than one ReadMe file. For example, a short ASCII ReadMe file could give instructions for decompressing a second file which, when decompressed, becomes a second and much longer ASCII text file that provides complete instructions on the data unit.

As shown in Fig. 1, we are assuming that the data itself is compressed in order to allow a maximum amount of data to fit on the storage unit. We are also assuming that the data is authenticated to ensure the detection of any errors in the stored data. Compression and authentication are discussed in the following sections. In order to be self-sufficient, the data unit must contain all software necessary to authenticate (check for errors) and decompress the data into formatted ASCII files, the latter being described in the ReadMe files. Such software is of course valid only for a given platform, and must be recompiled if the data files are moved to another medium and platform. For example, the authentication and decompression programs could be executable files designed to run under the DOS or Windows operating systems on a PC.

The data unit could also contain special application programs, also in the form of executable files designed to run on the specified platform. These might be files designed to plot the decompressed data files, or to process the decompressed data in some way likely to be useful to a future user of the data.

The data files are compressed and authenticated in order to maximize the use of the storage medium, and to put as much or all of the data together into a single data unit. They must be checked for errors using the authentication software and converted to ASCII files using the decompression software before they become useful.

In summary, with the files just described, the data unit becomes a complete and self-contained volume of data files, complete with all instructions and software necessary for extracting individual waveforms, images, or other data.

Compression of Waveform Data

A typical TIP waveform data file (Fig. 1) consists of a short series of text records that identify the file and its properties, followed by a long series of data records, each containing one or more samples from a digitized waveform. If the text portion of the file is relatively short, we usually copy the text into the data unit file without compression. Then we compress the waveform data and append it to the text, thus creating a complete TIP data file.

To illustrate the current Sandia waveform compression process, an example of a digitized waveform is shown in Fig. 2. Suppose first that the data in this waveform is stored in an

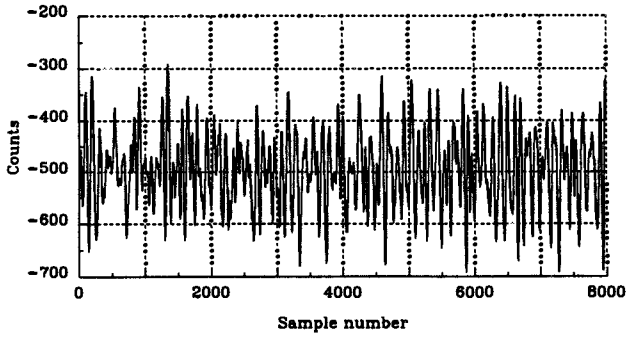


Figure 2. Portion of a digitized waveform.

ASCII data file named WAVEFORM.ASC, and suppose we wish to compress this file. A portion of WAVEFORM.ASC shown in Fig. 3. The file contains initial text lines plus 800 lines of integer samples, ten to a line, for a total of 8000 samples.

```
TEXT: WAVEFORM IDENTIFICATION, FORMAT, PARAMETERS, SCALE FACTORS, UNITS, ETC.
DATA:
-000531 -000526 -000521 -000515 -000511 -000504 -000496 -000490 -000484 -000484
-000478 -000471 -000473 -000471 -000468 -000466 -000461 -000458 -000458 -000457
-000459 -000461 -000459 -000463 -000470 -000468 -000469 -000471 -000475 -000480
-000481 -000483 -000486 -000488 -000492 -000495 -000499 -000504 -000510 -000517
-000525 -000528 -000530 -000536 -000544 -000549 -000547 -000547 -000551 -000554
-000555 -000561 -000563 -000565 -000565 -000566 -000564 -000561 -000563 -000563
-000563 -000563 -000558 -000551 -000549 -000547 -000541 -000535 -000534 -000529
-000522 -000514 -000504 -000497 -000492 -000484 -000477 -000470 -000461 -000454
-000441 -000432 -000423 -000414 -000410 -000402 -000394 -000389 -000380 -000375
-000371 -000366 -000360 -000356 -000356 -000352 -000351 -000351 -000345 -000347
-000349 -000350 -000354 -000352 -000354 -000360 -000363 -000366 -000369 -000377
-000384 -000389 -000394 -000399 -000405 -000409 -000417 -000422 -000428 -000437
-000442 -000449 -000457 -000464 -000473 -000482 -000493 -000504 -000512 -000519
-000528 -000536 -000547 -000558 -000564 -000573 -000585 -000590 -000597 -000606
-000612 -000619 -000626 -000632 -000635 -000637 -000641 -000644 -000645 -000648
-000648 -000647 -000653 -000647 -000637 -000634 -000631 -000630 -000624 -000615
-000609 -000601 -000591 -000583 -000573 -000560 -000550 -000542 -000533 -000521
-000507 -000494 -000478 -000468 -000455 -000442 -000436 -000422 -000410 -000400
-000390 -000382 -000375 -000368 -000363 -000354 -000347 -000344 -000337 -000334
-000329 -000325 -000324 -000319 -000315 -000315 -000315 -000314 -000314 -000318
... etc.
```

Figure 3. Beginning of the WAVEFORM.ASC file containing the data plotted in Fig. 2.

A Fortran program, PROGRAM 1, to compress the data in the WAVEFORM.ASC file is shown in Fig. 4. The code is simple to follow, but a few remarks are helpful. First, the effect of the program is to create the compressed data unit file, WAVEFORM.CMP. The latter consists of text data followed by eight frames, each with K=1000 samples. In the *do 2* loop, the frames are encoded one by one into an *ic* vector and then written to the output file. The number of bytes, *Nc*, is written ahead of each frame so the frame can be read by the decompression program described later.

Encoding, that is, compression, is accomplished by the *encod6* routine, which is part of the Sandia Fortran compression library, CMPLIB3.FOR. The latter is described elsewhere [1]. It involves waveform compression algorithms developed in a Sandia Labs LDRD project. Here we treat the routines in CMPLIB3 as black boxes, and only illustrate their use in compression and decompression software. The arguments in *encod6* are obvious except for the last argument (0), which tells *encod6* not to print any messages during execution.

The compressed data unit file, WAVEFORM.CMP, is written in binary form by PROGRAM 1. Thus we have created a platform-dependent data unit, that is, a data unit that must be decompressed on a similar PC platform using MS-DOS. To make the data unit decompressible on a foreign platform, we would need to write WAVEFORM.CMP as a formatted ASCII file. This process is discussed under "Platform Independence".

```
c-PROGRAM 1: Compress the WAVEFORM.ASC file with 8000 samples.
      parameter(K=1000,N=5000)
      integer ix(0:K-1)
      character*1 ic(0:N-1)
      character buf*80
c-Open the ASCII and compressed files.
      open(1,file='WAVEFORM.ASC',form='formatted')
      open(2,file='WAVEFORM.CMP',form='binary')
c-Write Nv=3 for version 3 of CMPLIB and No. frames Nff=8000/K.
      write(2) 3,8000/K
c-Copy the text data.
    1 read(1,'(a)') buf
      length=len_trim(buf)
      write(2) length,buf(1:length)
      if(buf(1:4).ne.'DATA') go to 1
c-Compress and write the waveform data, K samples at a time.
      do 2 j=0,8000/K-1
        read(1,'(9(i7,1x),i7)') ix
        call encod6(ix,K,N,Nc,ic,0)
        write(2) Nc,(ic(jj),jj=0,Nc-1)
    2 continue
      end
```

Figure 4. PROGRAM 1 compresses WAVEFORM.ASC into WAVEFORM.CMP.

As a matter of interest, we note that even the short waveform in Fig. 2 can be significantly compressed by the process in Fig. 4. The capability of *encod6* to compress the waveform is shown by the following three file sizes:

```
Original:   WAVEFORM.ASC   64,886 bytes
Zipped:     WAVEFORM.ZIP   15,971 bytes
Encod6:     WAVEFORM.CMP    4,185 bytes
```

The compression produced by *encod6* results in almost a 16:1 reduction of the original ASCII file size, and is also significantly beyond the compression achieved by the *zip* process in this case. The *zip* process uses a powerful but general coding technique, whereas *encod6* is designed specifically for waveform data. The theory and techniques implemented in *encod6* are described in the references [1-3]

Decompression of Waveform Data

Decompression of waveform data amounts to reversing the compression process. We begin with a compressed data unit file and produce the corresponding ASCII data file that

can be read or processed by one wishing to use the data. An example of decompression is shown in PROGRAM 2 in Fig. 5, which decompresses WAVEFORM.CMP and produces WAVEFORM.DEC, the latter being the decompressed ASCII file. In this case, PROGRAM 2 exactly reverses the effect of PROGRAM 1 so that WAVEFORM.DEC and WAVEFORM.ASC are identical files.

```
c-PROGRAM 2: Decompress the WAVEFORM.CMP file.
      parameter(N=5000)
      integer ix(0:N-1)
      character*1 ic(0:N-1)
      character buf*80
c-Open files.  Read Nv=version and Nff=# frames.
      open(1,file='WAVEFORM.CMP',form='binary')
      open(2,file='WAVEFORM.DEC',form='formatted')
      read(1) Nv,Nff
c-Copy the text data to the output file.
    1 read(1) Nb,buf(1:Nb)
      write(2,'(a)') buf(1:Nb)
      if(buf(1:4).ne.'DATA') go to 1
c-Decompress and write the data, one frame at a time.
      do 2 j=0,Nff-1
        read(1) Nc, (ic(jj),jj=0,Nc-1)
        call decod6(ic,N,K,ix,0)
        write(2,'(9(i7.6,1x),i7.6)') (ix(jj),jj=0,K-1)
    2 continue
      end
```

Figure 5. PROGRAM 2 decompresses WAVEFORM.CMP into WAVEFORM.DEC.

In PROGRAM 2 the integer sample and character byte arrays, *ix* and *ic* respectively, must be dimensioned large enough to accommodate the incoming data. This in itself is one reason the decompression software must be made a part of the data unit, so these dimensions can be set at the time the data is compressed. The *decod6* routine is the decoding routine that reverses the effect of *encod6* and changes an *ic* vector into an *ix* vector. The output data is written in ASCII to produce the WAVEFORM.DEC file.

Authentication

We currently provide the means for checking the authenticity of TIP data files by using cyclic error detecting codes. The routines used for this purpose are in a library called CRCLIB.FOR. Typically, within each TIP data file, we authenticate separately each compressed data frame having *K* data samples. Thus, in PROGRAM 1 for example, we would authenticate the *ic* vector after each call to *encod6*, and then write the authenticated

version of *ic* to unit 2, as shown in Fig. 6.

The purpose of authentication is to provide means for checking to see that the compressed waveform data, when it is read from the data unit, has not been altered. We are providing only for error *detection*, and not for error *correction*, because the latter would add too much redundant data to the data unit. Instead of correcting errors, we prefer simply to duplicate the entire data unit many times, and store the duplicates in different places. This procedure not only eliminates the need for error correcting codes, but also takes care of the case in which the physical data unit is damaged or lost.

The authentication routines in CRCLIB.FOR are described in a separate SAND report [4]. Essentially there are two pairs of routines, each for attaching cyclic check bytes when the data unit is written and then checking for errors later, when the data is read. The names of the two pairs of routines are {*encodc, decodc*} and {*encodl, decodl*}. The first pair is used for encoding and decoding a single vector of bytes, and the second pair is used primarily for encoding and decoding a sequence of bytes that is so long that it must be partitioned into a sequence of two or more vectors.

An example of the use of *encodc* to authenticate each successive data frame in PROGRAM 1 is shown in Fig. 6. The arguments passed to *encodc* are *ic*, the byte vector to

```
c-PROGRAM 1: Compress the WAVEFORM.ASC file with 8000 samples.
      parameter(K=1000,N=5000)
      integer ix(0:K-1)
      character*1 ic(0:N-1)
      character buf*80
c-Open the ASCII and compressed files.
      open(1,file='WAVEFORM.ASC',form='formatted')
      open(2,file='WAVEFORM.CMP',form='binary')
c-Write Nv=3 for version 3 of CMPLIB and No. frames Nff=8000/K.
      write(2) 3,8000/K
c-Copy the text data.
    1 read(1,'(a)') buf
      length=len_trim(buf)
      write(2) length,buf(1:length)
      if(buf(1:4).ne.'DATA') go to 1
c-Compress and write the waveform data, K samples at a time.
      do 2 j=0,8000/K-1
        read(1,'(9(i7,1x),i7)') ix
        call encod6(ix,K,N,Nc,ic,0)
        call encodc(ic,Nc,4,ic(Nc))
        Nc=Nc+4
        write(2) Nc,(ic(jj),jj=0,Nc-1)
    2 continue
      end
```

Figure 6. Modification of PROGRAM 1 to produce authenticated waveform data.

be authenticated, *Nc*, the number of bytes in *ic*, 4, indicating that four check bytes are to be computed, and *ic(Nc)*, the location in which to begin storing the four check bytes. After *encodc* is executed, *Nc* is increased by four to indicate the new length of the *ic* vector, which now ends with the four check bytes. Thus, the waveform data is authenticated simply by adding two lines to PROGRAM 1. Since there are four check bytes per frame and eight frames in this case, the authentication process causes the overall size of the WAVEFORM.CMP file to increase by 32 bytes.

An example of the use of *decodc* to decode and check the authenticity of each successive data frame in PROGRAM 2 is shown in Fig. 7. In this case the arguments sent to *decodc* are *ic*, the stored byte vector, *Nd*, the number of bytes in *ic*, 4, the number of check bytes

```
c-PROGRAM 2: Decompress the WAVEFORM.CMP file.
      parameter(N=5000)
      integer ix(0:N-1)
      character*1 ic(0:N-1)
      character buf*80
c-Open ASCII files.  Read Nv=version and Nff=# frames.
      open(1,file='WAVEFORM.CHK',form='binary')
      open(2,file='WAVEFORM.DEC',form='formatted')
      read(1) Nv,Nff
c-Copy the text data to the output file.
    1 read(1) Nb,buf(1:Nb)
      write(2,'(a)') buf(1:Nb)
      if(buf(1:4).ne.'DATA') go to 1
c-Decompress and write the data, one frame at a time.
      do 2 j=0,Nff-1
        read(1) Nd, (ic(jj),jj=0,Nd-1)
        call decodc(ic,Nd,4,ierror)
        if(ierror.ne.0) stop 'Error detected in decoded data.'
        call decod6(ic,N,K,ix,0)
        write(2,'(9(i7.6,1x),i7.6)') (ix(jj),jj=0,K-1)
    2 continue
      end
```

Figure 7. Modification of PROGRAM 2 to check authenticity of waveform data.

originally appended when *encodc* was executed, and *ierror*, an integer error indicator. If *ierror* is not zero, the program stops and indicates a decoding error. If *ierror* is zero, the program proceeds to call *decod6* as before. We note that *Nc*, although not used explicitly, is equal to *Nd*-4, which is the length of the *ic* vector without the four check bytes.

The error detecting capabilities of *encodc* and *decodc* are described in reference [4]. We note here, however, that the capabilities are sufficient for use in the TIP data unit. For example, with four check bytes per frame, if there is a burst of error bits of any length less than 33 anywhere in the frame, the error is detected with probability 1.0. If there is an error

burst longer than 32 bits, the error is detected with probability $1-2.3\cdot10^{-10}$. Thus, a user can have reasonable confidence in the accuracy of the waveform data.

Platform Independence

A TIP data file is usually written to the data unit as it is in PROGRAM 1, that is, as a sequence of character (byte) records. These records are written in the *binary* form in PROGRAM 1, thus making the TIP data file and hence the entire data unit platform-dependent. Whenever it is used, the data unit must be processed and decompressed on the same or a similar platform, using a compatible operating system. Once the ASCII file, WAVEFORM.ASC, has been reconstructed, it may of course be used on another platform.

To make the data unit independent of its platform, one must do two things, both of which are risky. First, the TIP data files must be written as *formatted* files instead of binary files. Secondly, source code for authentication and decompression must be included in the data unit, in order to allow executable files to be compiled on the unknown platform. We will discuss these things in order.

If the TIP data files are written by a Fortran program using the *A* edit descriptor, each record within a file becomes a sequence of bytes with appropriate terminating bytes. Since the compression routines use all 256 8-bit ASCII combinations, any given record may contain one or more control characters. These may in turn cause the record to be read incorrectly when it is read using the *A* edit descriptor. To remove this difficulty, a pair of routines named {*expand, shrink*} is included in CRCLIB.FOR. These routines have the effect of expanding a TIP data file to remove certain control characters before the file is written to the data unit, and then shrinking the file back to its original form after the data file has been successfully read from the data unit.

Detailed instructions for using *expand* and *shrink* may be found in the CRCLIB listing; however, the routines are easy to use, as shown in Figs. 8 and 9. Figure 8 is another iteration of PROGRAM 1, this time producing a platform-independent, formatted data file named WAVEFORM.PI. Several lines of the code are altered to produce the formatted file. A second array, *icx*, is declared to hold the expanded *ic* vector produced by *expand*. Also, as shown, all of the output statements are altered to write formatted data. The first argument in the calling sequence to *expand* is *ic*, the byte vector to be expanded. The second argument is *Nc*+4, the length of the *ic* vector after the check bytes have been added by *encodc*. The third and fourth arguments are *icx*, the output (expanded) vector and its dimension, *N*. The fifth argument is 2, which indicates to *expand* that the linefeed, *char*(10), carriage return, *char*(13), and form feed, *char*(12), are to be expanded to other codes in the *icx* vector. The final argument, *Ncx*, is the length of the *icx* vector produced by *expand*.

The removal of the three control bytes (linefeed, carriage return, and form feed) is, as far as we know, sufficient to allow any version of Fortran to read data records containing the *icx* vectors successfully. Other allowable values of the fifth argument in the *expand* calling sequence are described in the CRCLIB listing. In the present case, with just the three control bytes removed, the expected expansion factor, that is, *E*[*Ncx/Nc*], is 1.016. Thus, the penalty

```
c-PROGRAM 1: Compress the WAVEFORM.ASC file with 8000 samples.
      parameter(K=1000,N=5000)
      integer ix(0:K-1)
      character*1 ic(0:N-1),icx(0:N-1)
      character buf*80
c-Open the ASCII and compressed files.
      open(1,file='WAVEFORM.ASC',form='formatted')
      open(2,file='WAVEFORM.PI',form='formatted')
c-Write Nv=3 for version 3 of CMPLIB and No. frames Nff=8000/K.
      write(2,'(i1,i5)') 3,8000/K
c-Copy the text data.
    1 read(1,'(a)') buf
      length=len_trim(buf)
      write(2,'(i3,a)') length,buf(1:length)
      if(buf(1:4).ne.'DATA') go to 1
c-Compress and write the waveform data, K samples at a time.
      do 2 j=0,8000/K-1
        read(1,'(9(i7,1x),i7)') ix
        call encod6(ix,K,N,Nc,ic,0)
        call encodc(ic,Nc,4,ic(Nc))
        call expand(ic,Nc+4,icx,N,2,Ncx)
        write(2,'(i5,5000a1)') Ncx,(icx(jj),jj=0,Ncx-1)
    2 continue
      end
```

Figure 8. Further modification of PROGRAM 1 to produce platform-independent output.

for platform-independence is approximately a 1.6% increase in the storage requirement.

The use of *shrink* to reverse the effect of *expand* is illustrated in Fig. 9, where PROGRAM 2 is again modified to read the formatted data produced in Figs. 8 and then shrink each *ic* vector back to its original form before passing it on to be decoded. Note that *shrink* processes the *ic* vector in place, and does not require a second output vector. The second vector is not required because shrink is able to collapse the *ic* vector progressively onto itself as it moves along. The calling sequence for *shrink* is therefore simple. We specify the input byte vector, *ic*, and its expanded length, *Ncx*, and the routine computes *Nd*, the length of the shrunken version of *ic* containing the control bytes removed by *expand*.

In addition to formatted data files, platform independence also implies a capability for the execution of all of the software in Fig. 1 on a foreign platform. Thus, the data unit must contain source files instead of (or in addition to) execute files for the software, and these must be capable of being compiled later on the foreign platform. There is some risk involved in this process, since there is no absolute guarantee that the software will execute as originally intended. In this respect, we have noted that software written in Fortran-77 is more likely than ANSI-C software to produce consistent results on different platforms.

```
c-PROGRAM 2: Decompress the WAVEFORM.CMP file.
      parameter(N=5000)
      integer ix(0:N-1)
      character*1 ic(0:N-1)
      character buf*80
c-Open files.  Read Nv=version and Nff=# frames.
      open(1,file='WAVEFORM.PI',form='formatted')
      open(2,file='WAVEFORM.DEC',form='formatted')
      read(1,'(i1,i5)') Nv,Nff
c-Copy the text data to the output file.
    1 read(1,'(i3,a)') Nb,buf(1:Nb)
      write(2,'(a)') buf(1:Nb)
      if(buf(1:4).ne.'DATA') go to 1
c-Decompress and write the data, one frame at a time.
      do 2 j=0,Nff-1
        read(1,'(i5,5000a1)') Ncx, (ic(jj),jj=0,Ncx-1)
        call shrink(ic,Ncx,Nd)
        call decodc(ic,Nd,4,ierror)
        if(ierror.ne.0) stop 'Error detected in decoded data.'
        call decod6(ic,N,K,ix,0)
        write(2,'(9(i7.6,1x),i7.6)') (ix(jj),jj=0,K-1)
    2 continue
      end
```

Figure 9. Further modification of PROGRAM 2 to process platform-independent input.

Summary

In this report we have introduced the TIP data unit as a vehicle for the storage and retrieval of nuclear test data. The TIP data unit is not a specific hardware or software configuration. It is instead a simple set of ingredients which, taken together, form a transportable, self-contained storage unit that can be opened and used by someone having no information outside of the data unit itself. The capability of the data unit to operate successfully depends on the quality and integrity of the files within the unit.

Also, we have described current software for compressing and decompressing waveform data, for authenticating and checking for errors in data files, and for processing files to be used on foreign platforms.

References

1. Stearns, S.D., *Final Report: Lossless Compression of Instrument Data*, Sandia National Laboratories SAND95-2470, November 1995.
2. Stearns, S.D., L.Z. Tan, and N. Magotra, "Lossless Compression of Waveform Data for Efficient Storage and Transmission", *IEEE Trans. on Geoscience and Remote Sensing*, vol. 31, no. 3, pp. 645-654, May 1993
3. Stearns, S.D., "Arithmetic Coding in Lossless Waveform Compression", *IEEE Trans. on Signal Processing, IEEE Trans. on Signal Processing*, vol. 43, no. 8, pp. 1874-1979, August 1995.
4. Stearns, S.D., *Authentication of Byte Sequences*, Sandia National Laboratories SAND91-1004, June 1991.
5. Sanders, M.L., *Description of Ground Motion Data Processing Codes, Vol. 1, 2,* and *3*. Sandia National Laboratories SAND87-1176, February 1988.

DISTRIBUTION:

1 Mr. William C. Anderson - MS P-915
Los Alamos National Laboratory
P.O. Box 1663
Los Alamos, NM 87545

1 Mr. William A. Bookless
B Div., L-035, Box 808
Lawrence Livermore Nat. Laboratory
Livermore, CA 94550

1 Prof. D.M. Etter
Electrical and Computer Engineering
University of Colorado
Boulder, CO 80309

1 Dr. C.R. Hutt
Albuquerque Seismological Lab.
Bldg. 10002, KAFB East
Albuquerque, NM 87115-5000

1 Prof. N. Magotra
Electrical and Computer Engineering
University of New Mexico
Albuquerque, NM 87131

1 Prof. W.B. Mikhael, Chair
Electrical and Computer Engineering
University of Central Florida
Orlando, FL 32816-2450

1 LCDR Jerry White
Deputy Dir. for Information Mgmnt.
HQ Defense Nuclear Agency - OFIM
6801 Telegraph Road
Alexandria, VA 22310-3398

1 Dept. of Energy
Nevada Operations Office
Attn: Stephen H. Leedom
P.O. Box 98518
Las Vegas, NV 89193-8518

1 Dept. of Energy
Attn: Ray Ferry, DP-12
19901 Germantown Road
Germantown, MD 20874

1 Mr. Frank Biggs
3515 Monte Vista NE
Albuquerque, NM 87106

1 Mr. Douglas C. Browne
3619 Horacio Ct. NE
Albuquerque, NM 87111

1 Mr. Rich McClean
2600 Yale Blvd SE
Albuquerque, NM 87106

1 Mr. Gary L. Ogle
4607 Larchmont NE
Albuquerque, NM 87111

1 Mr. Philip L. Wehrman
11600 Bellamah NE
Albuquerque, NM 87108

1	MS 0457	S.S. Rottler	5139
1	MS 0750	M.C. Walck	6116
1	MS 0980	G.R. Elliott	9225
1	MS 1170	P.L. Nelson, Jr.	9305
5	MS 1170	B.C. Bedeaux	9305
1	MS 1170	R.M. Clancy	9305
1	MS 1170	J. Pearcey	9305
1	MS 1159	M.A. Hedemann	9311
5	MS 1159	S.D. Stearns	9311
1	MS 1160	H.D. Garbin	9312
1	MS 1160	K.M. Glibert	9312
1	MS 1169	C.W. Cook	9322
1	MS 1169	T.S. Caldwell	9322
1	MS 1391	F.M. Raymond	9331
1	MS 1391	D.D. Thompson	9331
1	MS 0459	J.C. Hogan	14707

1	MS 9018	Central Technical Files	8523-2
5	MS 0899	Technical Library	4414
1	MS 0619	Print Media	12615
2	MS 0100	Document Processing for DOE/OSTI	7613-2