IRIS-SEIS Users Manual and Installation Guide

Version 1.3.0

D. Okaya, E. Karageorgi, and T.M. Daley

November 1990
DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.
IRIS-SEIS Users Manual
and Installation Guide

Version 1.3.0

Compatible with SIERRASEIS v1.3

David Okaya, Eleni Karageorgi, and Thomas M. Daley

SierraSeis Maintenance Center
Earth Sciences Division
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

November 1990

The SierraSeis Maintenance Center
is operated by the Earth Sciences Division
for Incorporated Research Institutions for Seismology

This work was supported by the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.
IRIS-SEIS Users Manual and Installation Guide

I  INTRODUCTION  1

II  SIERRASEIS ENVIRONMENT  3
   A  Structure of SIERRASEIS  8
   B  Structure of IRIS-SEIS  23
   C  Structure of LOCAL-SEIS  30

III  IRIS-SEIS  37
   A  Installation of IRIS-SEIS  37
   B  Running an IRIS-SEIS job  44
   C  Processors within IRIS-SEIS  47
   D  Stand-alone utility programs provided within IRIS-SEIS  57

IV  LOCAL-SEIS  65
   A  Installation of LOCAL-SEIS  65
   B  Running a LOCAL-SEIS job  71
   C  Adding a new processor to LOCAL-SEIS  74
   D  Creation of an initialization subroutine  77
   E  Creation of an execution subroutine  87

V  USER DEVELOPMENT WORK AREA  93
   A  Installation of template directory “local13/user1”  93
   B  Creation of user work area  98
   C  Developing a new processor in the work area  101

Appendix I  IRIS-SEIS manual pages  105

Appendix II  FTOCIO library  161
I - INTRODUCTION

The IRIS-SEIS seismological processing package

The IRIS-SEIS seismological processing package is a software extension to Sierra Geophysics' SIERRASEIS seismic data processing package. By using IRIS-SEIS, one can apply algorithms which are not provided within the original SIERRASEIS package. In addition, IRIS-SEIS provides an environment so that users may add locally developed algorithms into the IRIS-SEIS / SIERRASEIS package. Some knowledge of FORTRAN and UNIX is required to add routines into IRIS-SEIS.

IRIS-SEIS will not run as a stand-alone software package but is written to be integrated with SIERRASEIS. The user must have a valid SIERRASEIS license in order to properly install and run IRIS-SEIS. IRIS-SEIS as presented here is supported on SUN/UNIX-based systems; references to operating system commands or to installation command files are based on the UNIX operating system.

In addition to providing access to all original processing routines, IRIS-SEIS provides several capabilities which are not available in SIERRASEIS. Most important, IRIS-SEIS separates SIERRASEIS routines from IRIS routines so that the IRIS-added routines do not corrupt the SIERRASEIS routines or infrastructure. Functionality available within IRIS-SEIS are user-definable headers, import of external data, utility processors to fix trace headers, and additional processing algorithms.

Just as IRIS-SEIS is designed not to corrupt original SIERRASEIS libraries and executables, IRIS-SEIS provides an extension for users to add routines without adversely affecting IRIS-SEIS libraries and executables. This third tier ("LOCAL-SEIS") is designated for locally developed algorithms. Shell subroutines are provided within IRIS-SEIS to allow the user to create new processing routines; these can be copied and modified for inclusion into LOCAL-SEIS.

This reference manual provides information which is needed to optimally use IRIS-SEIS. The first of three sections discusses the SIERRASEIS environment which provides a reference into which IRIS-SEIS and "LOCAL-SEIS" resides. The second section provides information on IRIS-SEIS (e.g., capabilities, installation procedures, how to run a job, etc.). The third section provides similar information on LOCAL-SEIS. Within the appendices, user documentation on each IRIS-SEIS processor is given. An appendix describes a set of C-language I/O utility routines for FORTRAN code; several IRIS-SEIS processors use this library.

Library of IRIS-community processors

It's the intent of IRIS to assemble a user-developed library of processing algorithms which may be useful to various batch-mode seismological data processors within the IRIS community. If you create a new LOCAL-SEIS processor, please consider giving permission to IRIS to incorporate it into IRIS-SEIS. The processor will be included in the subsequent release of IRIS-SEIS to be distributed to all people eligible to receive IRIS-SEIS (i.e., have a valid SierraSeis license).

Processor routines of any kind (algorithms, utility, editing, trace header manipulations, etc.) will be accepted - whatever may be useful to IRIS members (given the wide range of types of seismological data which could be run through IRIS-SEIS). The SierraSeis Maintenance Center (SMC) will accept processor routines and will check the subroutines to ensure the code is compatible with the SierraSeis environment.
II - SIERRASEIS ENVIRONMENT

SIERRASEIS is designed to routinely handle data sets consisting of a few to a few hundred thousand seismograms. The package was originally created to process conventionally collected seismic reflection data using the seismic trace-to-seismic trace manipulation strategy used by essentially all the major data processing packages. In this strategy, seismograms are induced into a processing stream in the order in which they are stored. Trace selection and manipulation are based on a user-specified order of processing routines. This orderly approach to seismogram manipulation allows for the creation and execution of large-scale batch jobs using a common input reference scheme.

Although able to handle large data sets, SIERRASEIS is capable of manipulating individual or a small number of seismograms. While the seismograms cannot be manipulated in a screen-interactive approach, the data can be processed using small, fast jobs. The reference to "batch" mode processing does not imply slow-queue, low priority processing but to the non-interactive nature of the selection of processing steps. In this mode, one first chooses the processing steps to apply to a selected range of seismograms, then executes the processing steps to the selected data. The provision within SIERRASEIS of an environment to manipulate seismograms plus the availability of a number of processing and display algorithms compensates for the lack of screen-interactive capability (e.g., mouse or cross-hair selection of processing steps).

Trace-to-trace seismogram handling

Two major data handling schemes exist for processing seismic data: "file sequential" or "trace-interactive" order (Figure 1). Given a processing list of several steps, data can be processed in "file sequential" order by individually applying each step to all the data, producing intermediate output results which are used as the input for the next step. This scheme requires large I/O resources.

Within the "trace sequential" ("trace-to-trace" or "trace-by-trace") scheme, a seismic trace is processed through all the requested steps in the order the steps are specified. Upon completion, the next input trace is obtained and processed through all the requested steps. When a multi-trace processing step is reached, this step must accumulate sufficient input traces to satisfy the processing algorithm before releasing the seismograms to the rest of the processing stream.

Due to the major constraint of handling data stored on one or a number of magnetic tapes, major seismic processing packages are usually constructed to process seismic traces in trace-to-trace sequential order. In this form, the access of data is orderly (first through last stored traces) but loses the speed of random access capability of seismograms. Processing which requires irregular access usually necessitates the use of temporary, run-time disk files which are used for random access.

Seismic traces and trace headers

Within trace-to-trace processing, the identity of the seismograms is important. Identification or description parameters such as shot or CDP gather number, trace number within the gather, source-receiver offset, sample rate, and trace length are often required information for many processing algorithms. These information exist for each seismogram and are termed "trace headers" or "global variables" within processing packages. Proper trace header values and their corresponding seismogram are passed in tandem from step to processing step within most processing packages.

A processing job's trace flow is regulated (i.e., defined) by the user's selection of processing steps. The actual flow of traces through these steps depends on the seismograms' trace header values. An unsuccessful processing job is often due to traces incorrectly flowing (or not flowing) through portions of a sequence of processing steps.
Figure 1: Data Handling Schemes

File Sequential

- Input All Data
- Filter All Data
- Deconvolve All Data
- AGC All Data
- Display All Data

Trace Sequential

<table>
<thead>
<tr>
<th>Trace 1 Input</th>
<th>Trace 2 Input</th>
<th>Trace 3 Input</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter</td>
<td>Filter</td>
<td>Filter</td>
<td></td>
</tr>
<tr>
<td>Deconvolve</td>
<td>Deconvolve</td>
<td>Deconvolve</td>
<td></td>
</tr>
<tr>
<td>AGC</td>
<td>AGC</td>
<td>AGC</td>
<td></td>
</tr>
<tr>
<td>Display</td>
<td>Display</td>
<td>Display</td>
<td></td>
</tr>
</tbody>
</table>
From a user's standpoint, to ensure that a job will complete successfully, one must
- know the ranges of trace header values in the input data (range of shot numbers con­tained in the data, for example),
and
- understand the trace flow created by the sequence of processing steps.

**Trace flow and railroad cars**

An analogy for trace-to-trace processing is a railroad system between two localities (points A and B). A train composed of a number of train cars will pass, one car at a time, from point A to point B, traveling through a number of stops and sidings. The train cars are flexible enough to be used for various purposes.

At the front of each railroad car slots will be placed where panels with numbers or words on them can be inserted; these slots will be used to identify characteristics about the car itself. Some parameters may physically describe the car (length, height, weight, color), some may describe the car in relation to the train (train name, car number), and some may describe the contents of the car (passenger seats or cargo, number of passenger seats, number occupied, type of cargo, etc.).

In this analogy, the origination point is equivalent to the input data; the destination is the end result (display or output of data). Each train car is a seismic trace; the train is the data set comprising the entire set of traces. The I.D. slots are the trace headers which are used to store descriptors.

As grand master engineer, you plan a routing schedule for the train (Figure 2). For example, a passenger train composed of a specified number of cars of a certain physical description can travel, car-by-car, through a number of stops (processors). At stop 1 after the point of origin, passengers can disembark. At stop 2, each car can enter a roundhouse where workers remove the passenger seats and paint the car (updating the identifiers in the process). Stop 3 may be a loading plant where items are placed into the modified cars. At stop 4 cars can selectively be chosen so that those of an excessive weight can be partially unloaded. At the next stop, groups of cars are accumulated so that the cars can be reordered before being released (in car-to-car manner).

In addition to the direct route between points A and B, branches can be constructed. These branches can be constructed to allow selected cars to undergo extra modifications before being returned back in to the original route. Alternatively, a branch can have a dead-end termination so that an additional route can exist in parallel to the original route.

In a similar manner, seismic traces can be individually passed from one processing step to another. Trace amplitudes or header values can be analyzed or modified depending on each processing algorithm. Trace flow can created to be all-inclusive (all seismograms) for all or any of the algorithms; however, partial processing can be established using inclusion logic based on trace header information.
Figure 2: Car-by-car Routing Schedule
FIGURE 3A: Table of SIERRASEIS Directories

The following directories and files exist within the overall ..;/sierra directory.

<table>
<thead>
<tr>
<th>Directory/File</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGCONFIGN.DAT</td>
<td>Text file with definitions for run-time environment variables.</td>
</tr>
<tr>
<td>SGPROD.DAT</td>
<td>Text file listing licensed SIERRA products.</td>
</tr>
<tr>
<td>rasvue</td>
<td>Directory containing rasvue; might be named &quot;rasvue new&quot;.</td>
</tr>
<tr>
<td>slib15</td>
<td>Directory containing SierraLib and drivers for SIERRASEIS.</td>
</tr>
<tr>
<td>sseis13</td>
<td>Directory containing SIERRASEIS version 1.3 and contains the subdirectories listed in Figure 3B.</td>
</tr>
</tbody>
</table>

FIGURE 3B: Table of SIERRASEIS version 1.3 Subdirectories

The following directories are contained within the ..;/sierra/sseis13 directory.

<table>
<thead>
<tr>
<th>subdirectory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>bench</td>
<td>benchmark jobs and data for testing this version of SIERRASEIS.</td>
</tr>
<tr>
<td>driver</td>
<td>subroutines which define the infrastructural framework of SIERRASEIS.</td>
</tr>
<tr>
<td>extend</td>
<td>subdirectories containing source and object code for the extension packages (e.g., migration, deconvolution, f-k filtering, etc.).</td>
</tr>
<tr>
<td>inc</td>
<td>include files which act like common blocks for various subroutines.</td>
</tr>
<tr>
<td>install</td>
<td>shell and make files needed to build SIERRASEIS.</td>
</tr>
<tr>
<td>lib</td>
<td>archive libraries containing all object code for sseis13 subdirectories.</td>
</tr>
<tr>
<td>main</td>
<td>main-level routines for the executables crsras, gntabl, rasplot, ssinit, and ssexec.</td>
</tr>
<tr>
<td>plot</td>
<td>device-driver code needed to properly compile and link rasplot.</td>
</tr>
<tr>
<td>run</td>
<td>key files and executables for SIERRASEIS. The important executables are ssinit and ssexec.</td>
</tr>
<tr>
<td>seismic</td>
<td>source and object code for all processors in the basic package of SIERRASEIS.</td>
</tr>
<tr>
<td>util</td>
<td>utility subroutines which help define the SIERRASEIS infrastructure.</td>
</tr>
</tbody>
</table>
II-A. Structure of SIERRASEIS

1) Users viewpoint of SIERRASEIS

SIERRASEIS is a seismic data processing package which operates in trace-sequential format. The package will process a set of data based on a list of steps provided in a text file ("job listing" as defined in the SIERRASEIS manual). The list of steps within this file must obey a small number of rules regarding format and syntax.

To implement the desired job, the user must run SIERRASEIS in two phases: initialization and the execution. Within the first phase, the user-created text file ("job listing") is checked by SIERRASEIS for any syntax or typing mistakes. Improper or undefined processing steps or parameter values are usually caught at this point. If all input text appears valid, the computer resources for the execution phase of the job are computed and in some cases allocated. If errors are detected, the second, execution, phase will not run properly.

The actual data processing is performed within an execution phase. At the start of this phase all unallocated computer resources are obtained and input data are processed in trace-sequential format. Upon normal completion of the job (e.g., no run-time errors are encountered), SIERRASEIS cleans up after itself, closing all files and removing certain temporary work files.

The two phases are conducted using separate executable programs. For SIERRASEIS, ssinit (or a variation such as "ssinit13" for version 1.3) is the program to perform the job initialization. For the execution phase, the executable ssexec (or "ssexec13") is used.

Communication between the "init" and "exec" phases is performed via the use of a temporary file. This file has the SIERRASEIS file suffix ".CMB". The job description as interpreted by ssinit is listed in this file to be used by ssexec. Any resources (e.g., memory, temporary disk space) to be needed are also described within this file. If an error is detected during by ssinit, the ".CMB" file is not created.

Run-time documentation is provided within SIERRASEIS files whose suffixes are ".IPR" for ssinit and ".EPR" for ssexec. These files will contain run-time printouts, errors, and other information diagnostic to the processing job.

A more complete description about the use of SIERRASEIS can be found in Chapter 1 of the SIERRASEIS Basic Users Manual.

2) Organizational framework of SIERRASEIS

The files which compose SIERRASEIS are organized into several different directories. These directories as described in Figure 3 are maintained within the sseis13 directory within the overall sierra directory.

The driver and util directories contain subroutines which define the infrastructural framework of SIERRASEIS. These routines perform functions such as the definition of run-time parameters, the reservation of memory at execution time, and the passing of "init"-phase parameters to the "exec"-phase.

The seismic and extend directories contain source and object code which define the seismic processors available in the SIERRASEIS basic and extension packages. The lib directory contains the archive libraries composed of these object code. The libraries are separated into basic (sseis.a), extension (extend.a), array processor or simulator code (ap.a or apsim.a), and driver/utility code (SYSLIB.a).

The main directory contains main-level (or "driver-level") routines for the executables crsras, gntabl, rasplot, ssinit, and ssexec. crsras and rasplot are used for raster plotting. gntabl is a utility routine used by the shell file "makelk" (install directory). ssinit and ssexec are the primary SIERRASEIS executables.
The inc directory contains include files which are specified within various SIERRASEIS subroutines. These files act like common blocks through which information is passed between subroutines. The most important include files are SS.COM.INC and SS.COMC.INC which are used to define and store Global Common Variables. More on GCV's below.

The run directory contains key files and executables for SIERRASEIS. The important executables are ssinit and ssexec. The text file ERRORS.LST is used by the two executables whenever a run-time error occurs; this file contains the error messages which appear in .IPR or .EPR files.

Text file PROC.LIS in the run directory is an important file and represents the master list of available SIERRASEIS processors. The file also serves as a look up table which indicates for any processor name (e.g., /AGC) the corresponding initialization and execution phase subroutines which ssinit and ssexec need to implement. The executable gntabl in this directory translates PROC.LIS, creating two subroutines, "lkinit.f" and "lkexec.f" which are subsequently linked into ssinit and ssexec.

The install directory is composed of shell and make files which are needed to properly install or update SIERRASEIS. The files can be divided into two parts: configuration and construction. The configuration files are used to set the initial installation environment; the construction files are used to compile subroutines, create archive libraries from the object code, and link executables. Figure 4 contains a list of files and their specific functionality.

The plot directory contains device-driver code needed to properly compile and link "rasplot."

The bench directory contains benchmark jobs and data for testing this version of SIERRASEIS.

3) Structural and operational framework of SIERRASEIS

The seismic processing subroutines associated with the processors available in SIERRASEIS are linked to top-level "main" or "driver" routines to form the executables ssinit and ssexec. The "driver" routines are so named because they "drive" or regulate the trace-by-trace flow control of input data and call the requested processing subroutines in the appropriate order. These routines also regulate computer resources and ensure run-time documentation gets written to .IPR and .EPR files.

Sections below discuss the ssinit and ssexec drivers, global common variables (trace headers), the regulation of job control within an ssexec run, and the regulation of trace flow for multiple trace processors. These are concepts which should be understood to utilize SIERRASEIS to its maximum capability.

The ssinit driver: The initialization of a seismic processing job is conducted with the executable ssinit. This program will interpret a user's job listing, checking for errors and determining the computer resources needed by the execution routine (ssexec).

As Figure 5 illustrates, ssinit identifies processing steps within a job listing and interacts with a subroutine named "lkinit" which contains essentially a look-up table of processor names. These processor names are associated with the names of specific subroutines which obtain parameters from a job listing. Subroutine "lkinit" returns the name of the subroutine (in computer form, it actually returns the address of the subroutine with the given name). ssinit subsequently conducts a call to this subroutine. These subsidiary subroutines interpret parameters from the job listing and determines resources needed during run-time. This information is stored within the job's .CMB file for communication with ssexec. ssinit also stores the order in which processors should be applied.

As an example, suppose the job we wish to run contains the following information:

```
/JOB
/DIN
FILENAME 'TESTDATA'
```
Figure 4: Installation Shell and Make files for SIERRASEIS

The ../sierra/sseis13/install directory contains shell and make files needed to build SIERRASEIS. The files within this directory can be divided into two parts: configuration and construction. The configuration files are used to set the installation environment:

<table>
<thead>
<tr>
<th>file</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>install</td>
<td>sets installation environment (defines directory paths), creates a new &quot;makefile,&quot; and will even begin compilation.</td>
</tr>
<tr>
<td>installcfg</td>
<td>a template file for &quot;install.&quot;</td>
</tr>
<tr>
<td>protomakefile</td>
<td>a precursor of &quot;makefile,&quot; used in conjunction with &quot;installcfg&quot; by &quot;install.&quot;</td>
</tr>
<tr>
<td>makefile.old</td>
<td>an earlier version of &quot;makefile.&quot;</td>
</tr>
</tbody>
</table>

The construction files are used to build the SIERRASEIS executables:

<table>
<thead>
<tr>
<th>file</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>compall</td>
<td>compiles all source code in other SIERRASEIS directories. This shell file calls &quot;comptree.&quot;</td>
</tr>
<tr>
<td>comptree</td>
<td>a shell file which identifies and compiles all source code subroutines within a given directory.</td>
</tr>
<tr>
<td>liball</td>
<td>obtains all object code in other SIERRASEIS directories and creates archive libraries using the archive command. Net result are &quot;.a&quot; files in the lib directory: sseis.a, extend.a, ap.a or apsim.a, and SYSLIB.a. This shell file calls &quot;libtree.&quot;</td>
</tr>
<tr>
<td>libtree</td>
<td>a shell file which identifies and archives all object code within a given directory.</td>
</tr>
<tr>
<td>makefile</td>
<td>used with the make command to create ssinit and ssexec, the two executables which define SIERRASEIS.</td>
</tr>
<tr>
<td>makelk</td>
<td>a shell file which makes two key lookup subroutines which are necessary to make SIERRASEIS work properly. This shell uses executable gntabl and text file PROC.LIS to create files &quot;lkinit.f&quot; and &quot;lkexec.f.&quot;</td>
</tr>
<tr>
<td>ran.lib</td>
<td>a text file which denotes the time and date that the archive libraries were last reconstructed.</td>
</tr>
</tbody>
</table>
Figure 5: The SSINIT Driver

Driver

Allocate resources

get/PROCESSOR name from job listing

call LKINIT

obtain address of initialization subroutine for this processor

call processor initialization subroutine

interpret processing parameters; allocate resources; store in .CMB file

End of job listing?

N

Y

Shutdown

Allocate resources

call LKINIT

obtain address of initialization subroutine for this processor
ssinit initially obtains the first line of text within the job listing, searching for a call to a processor named /JOB. Once obtained, ssinit calls "lkinit" to identify the initialization subroutine associated with processor JOB. The name of this subroutine is returned from "lkinit" (JBJOBO, in this case) to the main-level routine whereby it is subsequently invoked. Subroutine JBJOBO will perform job initializations.

Upon completion of the subroutine call, ssinit will obtain "/DIN", the next processor within the job listing. Using "lkinit," the initialization subroutine DIN000 is identified. When DIN000 is invoked by ssinit, the parameters within the /DIN processor in the job listing are interpreted. This information is stored in a SIERRASEIS .CMB file for use by ssexec.

ssinit will then process in sequential order the rest of the job listing (/DISPLAY in the above example) by first obtaining the next processor name, calling "lkinit", then invoking the proper initialization subroutine. This procedure is repeated until either the "$EOJ" card is reached or there are no more text lines in the job listing.

The ssexec driver: If the initialization of the seismic job was without error, the execution or processing phase can begin. Upon commencement of the ssexec program, the contents of the SIERRASEIS .CMB file are retrieved. This information lists the order of processor names as selected by the user.

With behavior similar to the ssinit program, ssexec will call each processor in the order given. For each processor, subroutine "lkexec" is called. This subroutine will return for the processor the name of the actual subroutine which performs the data processing step as described by the processor.

For the first processor, /JOB, the subroutine name JBJOBA is retrieved and then the subroutine is called by ssexec. In the execution mode, this subroutine defines certain job-related variables and initializes (clears) other variables.

The following processor in the example as discussed above was /DIN. The subroutine name DINAAA is obtained from "lkexec" and invoked. At this point, a seismic trace is read from the specified disk file into a trace buffer. Control of the program is returned to ssexec which gets the name of the next processor (/DISPLAY), obtains its subroutine name from "lkexec" (i.e., DS0000), and calls the subroutine, passing as an argument the seismic trace. Subsequent processors in the user's list are called in a similar manner until the last processor is reached and called. At this point, ssexec returns to the top in order to cycle through the processor list with the next seismic trace as obtained by /DIN.

Figure 6 illustrates the operations within the execution driver ssexec. The operations are made one step more complicated in that trace flow must be monitored. For example, if a subroutine such as that for F-K filtering requires a gather of data, the execution subroutine will request more traces before filtering will take place. ssexec must recognize this request and backtrack in the job listing for more data rather than advance traces within the job stream. The mechanism to do this is via two run-time parameters ("global common variables) named KSTATE and KCMBCK which are described in more below.

Job sequence number and run-time file names:

SIERRASEIS run-time files are named using what SIERRASEIS terms the "job sequence number". This is a one to four character text string (characters and/or numbers) which is used to identify all run-time files. The user is asked by the ssinit routine for this text string and again by the ssexec routine. The "job sequence number" identifies all files related to a particular processing job.
Figure 6: The SSEXEC Driver

- **Driver**
  - Allocate resources, read .CMB information
  - get/PROCESSOR name from .CMB file
  - call **LKEXEC**
    - call processor execution subroutine
    - Check trace flow control (KSTATE/KCMBCK):
      a) send trace to next processor;
      b) last processor in list, goto first processor with next trace;
      c) multiple mode: release trace;
      d) multiple mode: get another.
      e) end-of-data: cleanup
  - receive incoming seismogram; either process or resume multiple trace condition (release trace or go back for more)
  - obtain address of execution subroutine for this processor
  - Release resources; and shutdown
The format of run-time job files associated with a processing job is:

SSabcdvv.SFX

where \textit{abcd} is a one to four alphanumeric character sequence (the "job sequence number"), \textit{vv} is a "version number" (most noticeable for raster plot files), and \textit{SFX} is a three character file suffix. The suffix names are:

<table>
<thead>
<tr>
<th>File/Suffix</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSabcd00.IPR</td>
<td>Output from \textit{ssinit} with any error statements</td>
</tr>
<tr>
<td>SSabcd00.CMB</td>
<td>Communication file between \textit{ssinit} and \textit{ssexec}</td>
</tr>
<tr>
<td>SSabcd00.CBR</td>
<td>Parameter file for processors used in \textit{ssexec}</td>
</tr>
<tr>
<td>SSabcd00.EPR</td>
<td>Output from \textit{ssexec} with run-time status and any errors</td>
</tr>
<tr>
<td>SSabcd00.RAN</td>
<td>Temporary scratch disk file used by \textit{ssexec}.</td>
</tr>
<tr>
<td>SSabcd00.RAS</td>
<td>Raster plot file</td>
</tr>
</tbody>
</table>

For the above files, \textit{abcd} is the "job sequence number" one uses during execution of the \textit{ssexec} program.

Global common variables (trace headers): Seismic trace headers are an integral part of trace-by-trace data processing. These scalar values (a) identify individual seismic traces (i.e., shot or CDP gather number, trace number within gather), (b) provide descriptive information as to data dimensionality (i.e., sample rate, trace length, #traces/gather), (c) provide geometry information once calculated (i.e., midpoint X-Y coordinates, source-receiver offset). Within SIERRASEIS, these trace headers also give information regarding job status and trace flow.

Trace headers can be considered to be elements of an array of values for which there is one array for each seismic trace. When a data trace enters the trace flow of a processing job, an array of trace headers essentially tags along with the data trace. The trace header information is thus passed along with the seismic data.

For a seismic data processing package such as Cogniseis DISCO, a true array of trace headers exists for each trace and both the data and header arrays are passed from one execution processing subroutine to another. For processing packages such as SIERRASEIS or MERLIN SKS, the array of trace headers is structurally defined as a FORTRAN common block. In this latter construction, the common block can be accessed by any subroutine (processing or driver-level); header values are available at any time.

For SIERRASEIS, the common block which contains the trace headers is named "$SSCOM.INC$" in the "$inc$" directory of SIERRASEIS. Since a common block construction is used, the variables within the common block are referred to as "global common variables" or GCV’s.

Within the $SSCOM.INC$ common block, GCV’s are grouped into four types of variables based on FORTRAN declaration: integer, real, CHARACTER*8 and CHARACTER*4. The common block defines (allocates space for) more than SIERRASEIS actually requires, leaving room for expansion (IRIS-SEIS takes advantage of this feature):

<table>
<thead>
<tr>
<th>FORTRAN declaration</th>
<th>SIERRASEIS uses</th>
<th>available</th>
<th>array name for programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>134</td>
<td>200</td>
<td>SSCMAI</td>
</tr>
<tr>
<td>REAL</td>
<td>37</td>
<td>100</td>
<td>SSCMAR</td>
</tr>
<tr>
<td>CHAR*8</td>
<td>10</td>
<td>42</td>
<td>SSCMA8</td>
</tr>
<tr>
<td>CHAR*4</td>
<td>1</td>
<td>21</td>
<td>SSCMA4</td>
</tr>
</tbody>
</table>

The number of time samples in each seismogram is an integer number and is stored in location \textit{SSCMAI}(63); within common block \textit{SSCOM.INC}, the value of \textit{SSCMAI}(63) represents the trace length.
Using a form of equivalence, a list of variable names is associated with each type of common block information. For example, the number of samples per seismogram, SSCMAI(63), is equivalenced to the variable KNSAMP; either reference within a processing subroutine will yield the proper number of samples. The shot number, KSHOT, and the CDP number, KCDP, are the same as SSCMAI(76) and SSCMAI(15). The real-valued sample rate, SR, is stored in SSCMAR(28). The reference of GCV's by name rather than "array(index)" is for convenience both as a user and as a programmer.

The include file SSCOM.INC is accessed during execution of both ssinit and ssexec. The global common variables (GCV's) contained within the include file are used to describe the seismic data (trace headers, data dimensions), describe the processing job (line number, job name), and regulate the job control (trace flow). GCV values within the include file are initialized (cleared) prior to calling any processor subroutines. During execution of ssinit, the values of GCV's which are set or reset by any processor subroutine are "passed" or are "available" to the next processor subroutine.

When a seismic trace is entered into the processing stream by use of /IN, /DIN, /DMX, or any of the synthetic processors within ssexec, GCV's are filled into the include file. These values are treated as trace headers and are essentially passed from subroutine to processor subroutine with the seismic data. In certain instances, values which were defined in the initialization phase of a processor may be retrieved from the SIERRASEIS .CMB file by the processor's execution phase subroutine.

The include file (common block) "SSCOMC.INC" contains the names of the individual GCV's in the order in which they are defined in SSCOM.INC. SSCOMC.INC contains four sets of CHARACTER*6 arrays, one for each array defined in the .INC file.

<table>
<thead>
<tr>
<th>Include file:</th>
<th>SSCOM.INC</th>
<th>SSCOMC.INC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable declaration:</td>
<td>INT*4</td>
<td>CHAR*6</td>
</tr>
<tr>
<td>63rd variable:</td>
<td>INT*4</td>
<td>SSMAI(63)</td>
</tr>
<tr>
<td>Contents:</td>
<td>#samples</td>
<td>#samples</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;KNSAMP&quot;</td>
</tr>
</tbody>
</table>

By scanning the arrays in SSCOMC.INC for the character string "KNSAMP", we can obtain the location (index) where the corresponding value of KNSAMP is stored in SSCOM.INC.

Job control (a GCV named KSTATE): The trace-by-trace flow of seismic data within ssexec is described by the value of a GCV named KSTATE. The state of the trace flow determines whether a seismic trace is passed from one processor to the next (trace-by-trace) or has encountered a multiple-trace processor (f-k filtering, for example). A multiple-trace processor must first accumulate into memory or a temporary disk file ("RAN" file) a specified number of seismic traces. Once accumulated, the data are processed by some algorithm, then released in either trace-by-trace or multiple-release form. The value of KSTATE indicates to ssexec which of these trace operations to conduct.

<table>
<thead>
<tr>
<th>Value</th>
<th>KSTATE within a processor subroutine:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>no data passed out (multiple trace accumulation)</td>
</tr>
<tr>
<td>2</td>
<td>data in place &amp; releasable (trace-by-trace or multiple release)</td>
</tr>
<tr>
<td>3</td>
<td>end of input reached (no more incoming data)</td>
</tr>
<tr>
<td>4</td>
<td>no incoming trace in buffer but end of data not reached (multiple release flag)</td>
</tr>
</tbody>
</table>

In a job of simple processing trace flow (e.g., input, gain, bandpass filter, output), all processors work trace-by-trace so that KSTATE has value 2 until the last trace is read. After the last trace is processed, KSTATE will equal 3 and will trigger job shutdown and cleanup.

A subroutine is schematically structured in such a manner:
SUBROUTINE HALVE(TRACE)
DIMENSION TRACE(1)

IF(KSTATE.EQ.3)RETURN

DO I=1,KNSAMP
   TRACE(I)=TRACE(I)/2.
ENDDO
KSTATE=2
RETURN
END

A trace is passed into HALVE, divided by two for all trace samples (1 through KNSAMP), and then returned via the RETURN statement. The seismic data in the TRACE buffer is passed to the next processor by being returned from this subroutine and having a KSTATE=2 (trace-by-trace flow).

Upon first entry to this subroutine, the incoming KSTATE status is checked in order to see if it is equal to 3; if so, the job is in a state of "End-of-Data" and no processing should occur.

Note that it is within the subroutine code that the behavior of the processor is determined (e.g., the programmer has to define and set the various KSTATE states according to the desired trace flow behavior).

Multiple trace flow (KSTATE & KCMBCK): For those processor subroutines which require more than one data trace for their algorithms, a second GCV named KCMBCK is used in conjunction with KSTATE to regulate the accumulation and release of the data (keeping in mind that all trace action is performed one trace at a time). The subroutine must accumulate via programming statements the proper number of traces prior to application of the processing algorithm. Upon completion, proper trace flow must be constructed to release the appropriate number of traces. The programmer must remember that the subroutine is re-entered and exited each time a trace is accumulated or released.

<table>
<thead>
<tr>
<th>KSTATE</th>
<th>KCMBCK</th>
<th>action upon leaving subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>not used</td>
<td>accumulate traces (go back for more; no release)</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>trace-by-trace (incoming trace is released)</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>release stored traces (return here for next trace)</td>
</tr>
<tr>
<td>3</td>
<td>not used</td>
<td>end of data (no more data for entire job)</td>
</tr>
<tr>
<td>4</td>
<td>(1)</td>
<td>ssexec sets before returning to a routine which had set KCMBCK=1. The subroutine should reset to 1 or 2.</td>
</tr>
</tbody>
</table>

ssexec regulates trace flow based on these two GCV’s. KSTATE is used to determine upon leaving a processing subroutine whether to continue to the next processor with the current data trace (KSTATE=2) or to go back to the previous processor to look for an additional trace to receive (KSTATE=1). If the end of the input data has been reached, KSTATE=3 is used as a flag to indicate that no processing is done and subroutine cleanup should take place. A KSTATE=4 is a flag denoting "in transit" while ssexec searches back up the list of processing routines for the next available data trace.

KCMBCK is used by ssexec in backward searching mode to identify whether a processing subroutine is a mid-stream source of seismic traces. When a multiple-trace processor has completed its computation, it must release its traces in trace-by-trace mode. KCMBCK=1 flags for ssexec that this routine still has traces to release. ssexec in this case will not return to the top of the processor list but will stop at this routine to get its next trace. Only those processors which follow this routine in the job listing will receive the data.
**ssexec** will search backwards one processor at a time. **KCMBCK=0** indicates the current processing routine is not a source of data and the next processor back should be examined. This checking is conducted until a source routine is found (or the first, input, processor at the top of the list is found).

The states of **KSTATE** and **KCMBCK** must be set, checked, and updated by the programmer within each processing subroutine. For a multiple-trace algorithm, the programmer must first use these two GCV's to receive sufficient traces. When enough are accumulated, the GCV's must be reset according to the nature of the outgoing traces (multiple release or trace-by-trace if all the input traces are compressed into a single output trace). In addition, since the subroutine is re-entered for each trace operation, he or she must keep track of the status of mid-stream action (when only part of the data is accumulated or released). An example of multiple trace flow can be found in subroutines EXMUL0.F and EXMULX.F within the IRIS-SEIS processor source code directory (\sierra\iris13\seismic).

**Example of multiple trace flow:** Figure 7 illustrates the relationship between trace flow and the values of **KSTATE** and **KCMBCK**. In this example, a simple 5-processor job is used: input of seismic traces, application of normal moveout, F-K filtering, AGC gain, and plotting (/IN, /NMO, /FK, /AGC, and /DISPLAY, respectively). For brevity, the F-K filtering will be conducted using gathers composed of three traces (also, the actual name of the **SIERRASEIS** processor is "/FKFILT").

To get the entire job rolling, /IN will release the first trace into the processing flow. /NMO will apply normal moveout (using trace-by-trace flow). Upon entry into /FK, the seismic trace is placed into temporary storage. Since insufficient traces are held for the filtering, /FK will set KSTATE-KCMBCK (= 1-0) to return above for additional data. **ssexec** checks the processor immediately above /FK to see if it is a local source of data. Since /NMO is a trace-by-trace processor, **ssexec** moves back one processor. Since /IN is the first processor, **ssexec** will call the proper /IN subroutine and obtain the second data trace.

Similar to the first trace, the second trace is passed through /NMO and into /FK. Since /FK still does not have a "full" gather (3 traces), it sets trace flow to obtain an additional trace.

The third trace which is released by /IN and processed by /NMO is received by /FK. Since a full set of traces is now stored within /FK, the F-K filtering is performed. At this point, three traces are available for release to the rest of the processing flow. **KSTATE** and **KCMBCK** are reset to 2 and 1 to denote that /FK is now a local source of traces.

The first seismic trace can now be released to subsequent processors. This trace will be processed by /AGC and /DISPLAY in normal trace-by-trace action. When the last processor is finished with the trace, **ssexec** backtracks to identify the nearest source for the next trace. With a **KCMBCK=1**, the subroutine for /FK will be re-entered. This subroutine will then retrieve the second data trace for subsequent processing.

The third data trace retrieved from /FK will deplete the held traces in this routine. Since /FK has no more data, its KSTATE-KCMBCK status will be reset to 2-0 indicated that it is ready for a new gather (ready and waiting in trace-by-trace mode). After the third released trace reaches the end of the processing flow, **ssexec** will scan back for more data. With this new KSTATE-KCMBCK state, /FK will be ignored as a source of data; **ssexec** will end up back at the top at /IN.

Traces 4-6 will repeat the processing flow encountered by traces 1-3. Since it is "empty" when trace 4 is passed into it, /FK will reaccumulate data, resetting its KSTATE-KCMBCK to 1-0 (similar to when trace 1 entered /FK).

When no more data is available to /IN, **KSTATE** is set to equal three. With this GCV value, each processing subroutine will either clean up any resources used, complete its process (close a plot file, for example), or simply exit without any processing. Upon reaching the last processor in the job, **ssexec** will perform final clean up, run-time printouts, and will terminate.
Figure 7. Example of trace flow and KSTATE-KCMBCK values

Job listing (order of processors) is:
/IN
/NMO
/FK
/AGC
/DISPLAY

In this example, gathers of three traces will be f-k filtered. All other routines are single trace (trace-by-trace) processors. In the below table, the values of KSTATE-KCMBCK are given.

<table>
<thead>
<tr>
<th>trace</th>
<th>trace operation</th>
<th>/IN</th>
<th>/NMO</th>
<th>/FK</th>
<th>/AGC</th>
<th>/DISPLAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>/IN: enter stream</td>
<td>release</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>/NMO:</td>
<td>release</td>
<td>2-0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>enter /FK:</td>
<td>release</td>
<td>2-0</td>
<td>2-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>need more, reset GCV's,</td>
<td>release</td>
<td>2-0</td>
<td>1-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>leave /FK and go back</td>
<td>release</td>
<td>2-0</td>
<td>1-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>check /NMO KCMBCK=1?</td>
<td>release</td>
<td>2-0</td>
<td>1-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>check one more back (/IN)</td>
<td>release</td>
<td>2-0</td>
<td>1-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>second</td>
<td>/IN: release 2nd trace</td>
<td>release</td>
<td>2-0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>/NMO:</td>
<td>release</td>
<td>2-0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>enter /FK:</td>
<td>release</td>
<td>2-0</td>
<td>2-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>need more, reset GCV's,</td>
<td>release</td>
<td>2-0</td>
<td>1-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>leave /FK and go back</td>
<td>release</td>
<td>2-0</td>
<td>1-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>check /NMO KCMBCK=1?</td>
<td>release</td>
<td>2-0</td>
<td>1-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>check one more back (/IN)</td>
<td>release</td>
<td>2-0</td>
<td>1-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>third</td>
<td>/IN: release 3rd trace</td>
<td>release</td>
<td>2-0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>/NMO:</td>
<td>release</td>
<td>2-0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>enter /FK:</td>
<td>release</td>
<td>2-0</td>
<td>2-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sufficient: process</td>
<td>release</td>
<td>2-0</td>
<td>0-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>first</td>
<td>reset GCV's, release trace 1</td>
<td>[release 2-0]</td>
<td>2-1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>/AGC:</td>
<td>[release 2-0]</td>
<td>2-1</td>
<td>2-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>/DISPLAY:</td>
<td>[release 2-0]</td>
<td>2-1</td>
<td>2-0</td>
<td>2-0</td>
<td></td>
</tr>
<tr>
<td>second</td>
<td>/AGC: source of next trace?</td>
<td>[release 2-0]</td>
<td>2-1</td>
<td>2-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>/FK: source of next trace?</td>
<td>[release 2-0]</td>
<td>2-1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>release 2nd trace</td>
<td>[release 2-0]</td>
<td>2-1</td>
<td>2-0</td>
<td>2-0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/AGC:</td>
<td>[release 2-0]</td>
<td>2-1</td>
<td>2-0</td>
<td>2-0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/DISPLAY:</td>
<td>[release 2-0]</td>
<td>2-1</td>
<td>2-0</td>
<td>2-0</td>
<td></td>
</tr>
<tr>
<td>third</td>
<td>/AGC: source of next trace?</td>
<td>[release 2-0]</td>
<td>2-1</td>
<td>2-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>/FK: source of next trace?</td>
<td>[release 2-0]</td>
<td>2-1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>release 3rd trace (last)</td>
<td>[release 2-0]</td>
<td>2-0</td>
<td>2-0</td>
<td>2-0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/AGC:</td>
<td>[release 2-0]</td>
<td>2-0</td>
<td>2-0</td>
<td>2-0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/DISPLAY:</td>
<td>[release 2-0]</td>
<td>2-0</td>
<td>2-0</td>
<td>2-0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/AGC: source of next trace?</td>
<td>[release 2-0]</td>
<td>2-0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>/FK: source of next trace?</td>
<td>[release 2-0]</td>
<td>2-0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>/NMO: source of next trace?</td>
<td>[release 2-0]</td>
<td>2-0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>/IN: source of next trace?</td>
<td>[release 2-0]</td>
<td>2-0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fourth</td>
<td>/IN: enter stream</td>
<td>release</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[like traces 1-3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trace</td>
<td>Trace Operation</td>
<td>IN</td>
<td>NMO</td>
<td>FK</td>
<td>AGC</td>
<td>Display</td>
</tr>
<tr>
<td>-------</td>
<td>------------------------------------------</td>
<td>-----</td>
<td>-----</td>
<td>---------</td>
<td>-----</td>
<td>---------</td>
</tr>
<tr>
<td>Last</td>
<td>/FK: Release 3rd trace (last)</td>
<td>[2-0]</td>
<td></td>
<td>2-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>/AGC:</td>
<td>[2-0]</td>
<td></td>
<td>2-0</td>
<td>2-0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/DISPLAY:</td>
<td>[2-0]</td>
<td></td>
<td>2-0</td>
<td>2-0</td>
<td>2-0</td>
</tr>
<tr>
<td></td>
<td>/AGC: source of next trace?</td>
<td>[2-0]</td>
<td></td>
<td>2-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>/FK: source of next trace?</td>
<td>[2-0]</td>
<td></td>
<td>2-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>/NMO: source of next trace?</td>
<td>[2-0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>/IN: source of next trace?</td>
<td>[2-0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No more</td>
<td>/IN: End of data (KSTATE=3)</td>
<td>3-0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>/NMO: cleanup/no processing</td>
<td>3-0</td>
<td>3-0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>/FK: cleanup/no processing</td>
<td>3-0</td>
<td>3-0</td>
<td></td>
<td>3-0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/AGC: cleanup/no processing</td>
<td>3-0</td>
<td>3-0</td>
<td>3-0</td>
<td>3-0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/DISPLAY: close plot</td>
<td>3-0</td>
<td>3-0</td>
<td>3-0</td>
<td>3-0</td>
<td>3-0</td>
</tr>
</tbody>
</table>

END OF JOB
4) Writing a pair of subroutines for a /PROCESSOR

Each processor must have an initialization and an execution subroutine. Within these subroutines, one must keep track of global common variables, trace flow, and any needed memory and disk resources. Communication between the initialization and execution subroutines must be constructed properly with the aid of utility subroutines provided within SIERRASEIS. Issues relevant to programming of SIERRASEIS-compatible subroutines are discussed in Section IV.

A naming convention for SIERRASEIS-compatible processors exists. For processor names, up to eight alphabetic characters can be used; no numbers are acceptable. The name "/TWODIMEN" is valid; the name "/2DIMEN" is not. The initialization and execution subroutines can have names up to 6 alphanumeric characters in length (due to size limits in the PROC.LIS file and due, in part, to limits in certain other computer operating system naming conventions). Again, more on this will be discussed in Section IV.

5) The PROC.LIS file and subroutines ssinit.F and ssexec.F: master list of SIERRASEIS processors

The lookup subroutines lkinit.F and lkexec.F serve important roles in the functionality of executables ssinit and ssexec. These two subroutines equate processor names with other subroutines so that the proper subroutine is called at the appropriate time. The lookup subroutines must exist prior to final compilation/link of the executables.

A text file named "PROC.LIS" (located in the .../sseis13/run directory) is the master list in which processor names and their corresponding initialization and execution subroutines are listed. A portion of this file is excerpted below:

<table>
<thead>
<tr>
<th>DIN</th>
<th>DI0000</th>
<th>DINAAA</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISPLAY</td>
<td>DS0000</td>
<td>DSEXEA</td>
</tr>
<tr>
<td>DMX</td>
<td>DM0000</td>
<td>DMEXEA</td>
</tr>
<tr>
<td>DOUT</td>
<td>DO0000</td>
<td>DOUTAA</td>
</tr>
<tr>
<td>EXP</td>
<td>XPONTO</td>
<td>XPONTA</td>
</tr>
<tr>
<td>EXPGAIN</td>
<td>EX0000</td>
<td>EXPGAA</td>
</tr>
<tr>
<td>FILTPAN</td>
<td>FI0000</td>
<td>FILPNA</td>
</tr>
<tr>
<td>FSUM</td>
<td>FL0000</td>
<td>FLSUMA</td>
</tr>
<tr>
<td>GATHER</td>
<td>GATHRO</td>
<td>GATHRA</td>
</tr>
</tbody>
</table>

The first column contains processor names (i.e., DISPLAY). The second and third columns contain initialization and execution processor names (DS0000 and DSEXEA, respectively). ssinit calls subroutine DS0000 in order to digest /DISPLAY parameters. When live traces are to be plotted during the execution phase, ssexec passes the data into DSEXEA which creates the raster plot. Due to internal SIERRASEIS constraints, the processor names are no more than eight characters in length and the subroutine names are no more than six characters.

Prior to building the executables ssinit and ssexec during installation, the programmer must construct the needed subroutines lkinit.F and lkexec.F with the use of a stand-alone program named gntabl. This utility program separates the initialization and execution subroutine names from the PROC.LIS file and places them into the separate lookup subroutines. Once compiled, the object code lkinit.o and lkexec.o are used to build ssinit and ssexec. The shell file makelk located in the .../sseis13/install directory will run gntabl and compile the lookup subroutines.

A schematic representation of the lookup subroutines is shown on the next page:
SUBROUTINE LKINIT(name, sub)
CHARACTER*8 name, sub
IF(name.EQ. 'DIN') sub=DIO000
IF(name.EQ. 'DISPLAY') sub=DS0000
IF(name.EQ. 'DMX') sub=DM0000
IF(name.EQ. 'DOUT') sub=DO0000
IF(name.EQ. 'EXP') sub=XP0000
IF(name.EQ. 'EXPGAIN') sub=EX0000
IF(name.EQ. 'FSUM') sub=FL0000
IF(name.EQ. 'GATHER') sub=GATH00
SUBROUTINE LKEEXEC(name, sub)
CHARACTER*8 name, sub
IF(name.EQ. 'DIN') sub=DINAAA
IF(name.EQ. 'DISPLAY') sub=DSEXEA
IF(name.EQ. 'DMX') sub=DMEXEA
IF(name.EQ. 'DOUT') sub=DOUTAA
IF(name.EQ. 'EXP') sub=XPONTA
IF(name.EQ. 'EXPGAIN') sub=EXPGA
IF(name.EQ. 'FILTPAN') sub=FILPNA
IF(name.EQ. 'FSUM') sub=FLSUMA
IF(name.EQ. 'GATHER') sub=GATRA

In actuality, these subroutines return not the processor subroutine name, but the integer-valued address of the subroutine. For internal SIERRASEIS reasons, having the address rather than the subroutine name simplifies the call to the routine.

This information about the PROC.LIS file and the processor subroutine names is important for debugging a troublesome processing job. If the programmer is unable to either decipher an error message or understand why a processor subroutine function as it does, he or she may have to look at the processor subroutine's source code. To determine where the error is, the programmer should scan the PROC.LIS file for the name of the appropriate "init" or "exec" subroutine. Fatal error messages are called by a utility subroutine named "MPEROR"; calls to this subroutine must be identified. If the second argument is the error message number received, then the error is located in this section of code; for example:

IF(user_vel.EQ.0) CALL MPEROR("NMO", 13).

MPEROR will print error message #13 associated with /NMO as located in the ERRORS.LST text file in .../sseis13/run. The programmer can try to decipher why the IF decision failed; this will often explain why the job is bombing.

Upper case / Lower case Characters: SIERRASEIS/IRIS-SEIS and UNIX

Because it is constructed to run on many different computer environments, SIERRASEIS is written to always use upper-case characters. Any keyboard input into SIERRASEIS is automatically converted to upper-case. For some operating systems (VAX VMS, for example), this is not a problem. For UNIX versions of SIERRASEIS, the user must be aware of this behavior.

Information entered into the keyboard can be done in either upper- or lower-case, knowing that all entries are converted to upper-case. If the information entered is a file or device name, then SIERRASEIS will look for the upper-case name of the file or device. As a result, in UNIX systems, all SIERRASEIS filenames must be in upper-case. This is not necessarily a problem, but is sometimes an annoyance.

UNIX device names are by default in lower case and are not easily convertible to upper case. Because of this, upper-case symbolic links must be used to point to the (lower-case) device names. Thus, for example, when a tape mount request arises, even if the answer is /dev/rmt16 (in lower case), SIERRASEIS will not find the tape drive; /DEV/RMT16 (upper case) will not exist. A symbolic link, such as TAPE16, which points to /dev/rmt16 will make the tape drive accessible to SIERRASEIS. These need to exist locally within the directory in which the job is being executed. The user can respond to a tape request with either 'TAPE16' or 'tape 16' knowing that SIERRASEIS will find the symbolic link and access the proper device. [Alternatively, he or she could also create a root level directory named /DEV which contains only symbolic links such as RMT16 which point to the actual devices. This removes the need to have local symbolic links].
This upper-case/lower-case behavior exists within IRIS-SEIS. Because of this, the user can run the command line shells using lower-case characters knowing that IRIS-SEIS will access the upper-case files.

Filenames for the '/DIN', '/DOUT', and IRIS-SEIS processors /UNIXIO, and /MONITOR processors are not subject to upper-case conversion. These names are entered within a job listing and are not entered via the keyboard during execution. The filenames (and paths) can be either upper or lowercase.

It should be possible to have a run-time parameter which specifies whether SIERRASEIS/IRIS-SEIS should automatically convert to upper case or should leave characters in whatever case is entered. This could easily be a job parameter defined within the /JOB processor at the beginning of a job listing. The structural changes which would have to be made are sprinkled throughout SIERRASEIS, however; this is an improvement which Sierra personnel would need to implement.
II-B. Structure of IRIS-SEIS

The desire or need to add additional processing capability to SIERRASEIS can arise quickly in an academic environment. While additions can be made directly to the SIERRASEIS package, the danger arises of inadvertent programming errors corrupting the functionality of the original software. IRIS-SEIS is a programming structural addition to SIERRASEIS which allows for the separation but integration of locally developed processors and SIERRASEIS routines.

IRIS-SEIS is divided into two parts. The first part provides already-developed routines distributed as a package of IRIS processors. These processors are hopefully the beginning of a community-developed set which will be distributed through IRIS to those with SIERRASEIS licenses. These routines are checked to ensure they are fully compatible with SIERRASEIS prior to installation into IRIS-SEIS. IRIS-SEIS executables are named irisinit and irisexec and are used instead of ssinit and ssexec.

The second part of IRIS-SEIS is a development environment for those who wish to create new processors. In this environment (termed "LOCAL-SEIS"), executable programs similar to ssinit and ssexec will allow the use of locally-developed, IRIS, and SIERRASEIS processors within the same processing job.

Within the IRIS-SEIS construction, source and object code, link libraries, and utility programs and make files are housed in a self-contained directory named iris13 which is installed at the same level as SIERRASEIS; i.e., the same level as directory sseis13 within the overall sierra directory. The construction of the IRIS-SEIS executables, irisinit and irisexec, will incorporate link libraries containing SIERRASEIS basic and extension object code, and utility and driver routines in addition to IRIS-SEIS object code.

The full capability of SIERRASEIS is available within IRIS-SEIS. A job listing which contains only SIERRASEIS processors can be run using SIERRASEIS or IRIS-SEIS. However, a job listing which contains IRIS-SEIS processors will not run under SIERRASEIS.

The sections here describe the structural and organization framework of IRIS-SEIS. For installation procedures and explanations on running seismic processing jobs with IRIS-SEIS, see section III.

Directory/File Structure of IRIS-SEIS in relation to SIERRASEIS

IRIS-SEIS resides at the same directory level as SIERRASEIS:

```
../sierra/
  sseis13/
    extend/
    inc/
    install/
    lib/ - sseis.a, extend.a
    main/ - ssinit.o, ssexec.o
    run/ - ssinit, ssexec, PROC.LIS, lkinit.o, lkexec.o
  seismic/

iris13/
  inc/
  install/
  lib/ - irisseis.a
  main/ - irisinit.o, irisexec.o, (loclkinit.o, loclkexec.o)
  run/ - irisinit, irisexec, PROCIRIS.LIS, irislkinit.o,
       irislkexec.o, lkinit.o, lkexec.o
  seismic/
```

The files lkinit.o and lkexec.o in iris13/run differ from those in sseis13/run in that they recognize IRIS-SEIS calls (as explained below).
The executable `irisinit` is constructed by linking `iris13/main/irisinit.o` with `irlkinit.o` and `lkinit.o` in `iris13/run` and with libraries `irisseis.a`, `sseis.a`, `extend.a` and SierraLib libraries. The executable `irisexec` is formed by linking `iris13/main/irisexec.o` with `irlkexec.o` and `lkexec.o` in `iris13/run` and with libraries `irisseis.a`, `sseis.a`, `extend.a` and SierraLib libraries.

The files `loclkinit.o` and `loclkexec.o` in `iris/main` are dummy routines within IRIS-SEIS. These (empty) routines are replaces with live routines in LOCAL-SEIS.

**Structural and Operational framework of IRIS-SEIS**

IRIS-SEIS functions in a manner identical to SIERRASEIS. Since IRIS-SEIS is an addition to SIERRASEIS, the structures of both are the same in infrastructure, organization, and installation. The run-time environment of both packages are also the same. From a user's point-of-view, one does not need to understand the differences as described below in order to use IRIS-SEIS.

The IRIS-SEIS initialization and execution programs interpret a list of processing statements (i.e. the "job listing") using the same approach as that used by the SIERRASEIS executables. The initialization executable `irisinit` interprets a job listing and with the help of a lookup subroutine calls the correct processor subroutines. The execution program `irisexec` conducts the data processing by invoking the requested subroutines while regulating the proper trace flow control.

**Primary difference between the two packages: behavior of lookup subroutines**

The modifications to the pre-existing SIERRASEIS structural framework which were required to implement IRIS-SEIS were quite few in number. The primary modification is in the behavior of the driver routines `ssinit.F` and `ssexec.F` with their lookup subroutines `lkinit.F` and `lkexec.F`. These modifications are essentially transparent to users; installation/update utility programs and shell files implement the modifications. Users run the resulting executables without need to account for the structural differences.

As explained in section II-A, the driver routines use these lookup subroutines to identify for a given processor the actual subroutine which is to be invoked in order to implement the processing step. Within SIERRASEIS, a processor whose name is not defined within these lookup subroutines is considered undefined (and fatal error messages are invoked).

The definition of a new processor can be done by simply adding the names of the processors and the appropriate application subroutines to these lookup subroutines. The new processor would then be available to the driver routines via the lookup subroutines.

A schematic representation of this change is as follows:

<table>
<thead>
<tr>
<th>old lkinit.F</th>
<th>new lkinit.F</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBROUTINE LKINIT(name,sub)</td>
<td>SUBROUTINE LKINIT(name,sub)</td>
</tr>
<tr>
<td>CHARACTER*8 name,sub</td>
<td>CHARACTER*8 name,sub</td>
</tr>
<tr>
<td>IF(name.EQ.'DIN') sub=D10000</td>
<td>IF(name.EQ.'DIN') sub=D10000</td>
</tr>
<tr>
<td>IF(name.EQ.'DISPLAY') sub=DS0000</td>
<td>IF(name.EQ.'DISPLAY') sub=DS0000</td>
</tr>
<tr>
<td>IF(name.EQ.'DMX') sub=DM0000</td>
<td>IF(name.EQ.'DMX') sub=DM0000</td>
</tr>
<tr>
<td>IF(name.EQ.'EXPGAIN') sub=EX0000</td>
<td>IF(name.EQ.'EXPGAIN') sub=EX0000</td>
</tr>
<tr>
<td>IF(name.EQ.'GATHER') sub=GATHRO</td>
<td>IF(name.EQ.'NEWNAME') sub=NEWNA0</td>
</tr>
<tr>
<td>RETURN</td>
<td>RETURN</td>
</tr>
<tr>
<td>END</td>
<td>END</td>
</tr>
</tbody>
</table>

SIERRASEIS and added processors become intermixed in various ways within this approach. Certain text lookup files and link libraries will contain original SIERRASEIS and new names and subroutines, allowing for the possibility of adverse effects by incorrect non-SIERRASEIS routines.
IRIS-SEIS structurally separates SIERRASEIS and IRIS-SEIS subroutines (as will be explained in the following section) and intentionally preserves as much of the original SIERRASEIS structural framework as is possible.

The primary modification to SIERRASEIS is in the contents of the lookup subroutines lkinit.F and lkexec.F. With the use of modified SIERRASEIS utility programs, the original SIERRASEIS lookup subroutines can be constructed to call a second set of lookup subroutines if needed. For example, when called by the initialization driver (ssinit or more correctly, irisinit) to get the subroutine name of a non-SIERRASEIS processor, lkinit.F will not find the name in the list of original SIERRASEIS names. Since not successful, lkinit.F will call a subsidiary but similar subroutine named irislkinit.F which will contain the names of IRIS processors and subroutines.

Within this latter construction, all IRIS-SEIS related subroutines (source and object code) can be physically separated from SIERRASEIS files, creating a clear division between the original and added software. This separation reduces the risks imposed by incorrect programming within the added routines.

A schematic example of the new construction is as follows:

<table>
<thead>
<tr>
<th>new lkinit.F</th>
<th>irislkinit.F</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBROUTINE LKINIT(name,sub) CHARACTER*8 name,sub</td>
<td>SUBROUTINE IRISLKINIT(name,sub) CHARACTER*8 name,sub</td>
</tr>
<tr>
<td>IF(name.EQ. 'DIN') sub=D10000</td>
<td>IF(name.EQ. 'NEWNAME') sub=NEWNA0</td>
</tr>
<tr>
<td>IF(name.EQ. 'DISPLAY') sub=DS0000</td>
<td>IF(name.EQ. 'PROCTWO') sub=PROC20</td>
</tr>
<tr>
<td>IF(name.EQ. 'DMX') sub=DM0000</td>
<td>IF(name.EQ. 'ANOTHER') sub=ANOTH0</td>
</tr>
<tr>
<td>IF(name.EQ. 'EXPGAIN') sub=EX0000</td>
<td>RETURN</td>
</tr>
<tr>
<td>IF(name.EQ. 'GATHER') sub=GATHR0</td>
<td>END</td>
</tr>
<tr>
<td>IF(sub.EQ.NULL) CALL IRLSKINIT(name,sub)</td>
<td></td>
</tr>
<tr>
<td>RETURN</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

These lookup subroutines actually return the address of the named subroutines and not the names themselves.

The separation of IRIS-SEIS files from SIERRASEIS files requires the need for a parallel infrastructure similar to that of original SIERRASEIS. Separate archive libraries and installation make and shell files exist. This duplication is used during the installation and updating of IRIS-SEIS.

The stand-alone executable gntabl uses file "PROC.LIS" to create subroutines lkinit.F and lkexec.F. Similarly, a stand-alone executable irsgntabl uses file "PROCIRIS.LIS" to create subroutines irislkinit.F and irislkexec.F. The purpose and structure of file "PROCIRIS.LIS" is identical to that of "PROC.LIS" in that it is a master list of available routines added beyond those available in SIERRASEIS. The relationship between processor name (/PROCESSOR) and the initialization and execution subroutines is described here.

To allow for the lkxxxx subroutines to call irislkxxxx subroutines, a modification in executable gntabl was installed. Thus, gntabl, lkinit.F, and lkexec.F in ../iris13/run are modified versions from the files with the same names in ../sseis13/run.

While there are structural differences from a programming standpoint, the appearance and run-time behavior of IRIS-SEIS is no different than those of SIERRASEIS. Users of IRIS-SEIS do not need to understand the structural differences as described above.
Organizational framework

The several subdirectories containing IRIS-SEIS within the iris13 directory are similar in organization as those which contain SIERRASEIS. These subdirectories do not duplicate files which exist within SIERRASEIS but contain only those files which are needed to define IRIS-SEIS. Directories are listed in Figure 8.

The inc directory contains include files, most of which are the same as those found in .../sseis13/inc. The important file here is "IRISCOMC.INC" which is a modification of SSCOMC.INC to allow for user-defined GCV's (trace headers).

The install directories contain shell and make files needed to install or update IRIS-SEIS. The purpose of each file is listed in Figure 9.

Within the main directory, files "irisinit.F" and "irisexec.F" are the "driver"-level routines for IRIS-SEIS and will become irisinit and irisexec upon compilation. These routines are modified from "ssinit.F" and "ssexec.F" to allow for the addition of IRIS-SEIS functionality.

The run directory contains the executables irisinit and irisexec and files needed to construct the lookup routines. Two sets of lookup routines are needed. gntabl and "PROC.LIS" will create "lkinit.F" and "lkexec.F". irisgntabl and "PROCIRIS.LIS" will create "irislkinit.F" and "irislkexec.F".

The files gntabl, "lkinit.F" and "lkexec.F" are modified from those found in .../sseis13/run. The changes here allow the routines to recognize the IRIS-SEIS files "irislkinit.F" and "irislkexec.F".

The seismic directory contains source and object code for the processors available from IRIS for IRIS-SEIS. Useful subroutines to examine are: "EASYIN.F" and "EASYEX.F" which illustrate the structure of the most basic of processors (/EASYADD); "EXTRC0.F" and "EXTRCX.F" which illustrate a processor (/EXTRCTR) with parameters; and "EXMUL0.F" and "EXMULX.F" which form the processor /EXMULTRC, an example of a multiple trace processor.

The utility directory contains source and executables for stand-alone programs which provide functionality in concert with IRIS-SEIS. For example, tapeoper when used with processors /INOPER or /OUTOPER will provide the user with external tape mounting capability; a reply to a tape request can be given from a terminal different than from where a job is being run. This allows for tape drive sharing and batch job tape mounting. A full list of utility functionality is given in section III-D.

User-definable trace headers (GCV's)

IRIS-SEIS takes advantage of certain constructions within SIERRASEIS in order to allow for new trace headers (GCV's) to be defined and manipulated. The new headers are defined to look like addition SIERRASEIS headers so that they are compatible with many aspects of the original package. Due to limitations in SIERRASEIS, however, certain operations cannot be performed with user-defined headers (trace sorting, for example). In most cases, there are run-time ways to circumvent the limitations.

The new headers are defined by using empty slots within arrays which exist within the common block SSCOM.INC. The locations of these slots are identified by the definition of the header name within arrays in the common block IRISCOMC.INC (which was modified from the original common block SSCOMC.INC). When a new header name appears in an array in IRISCOMC.INC, the index (location) in the array is the location within a parallel array in SSSCOMC.INC which contains the actual value of the header. IRIS-SEIS contains processors and programming utility subroutines to define and bookkeep these additions.
FIGURE 8: Table of IRIS-SEIS version 1.3 Subdirectories

The following directories are contained within the ../sierra/iris13 directory.

<table>
<thead>
<tr>
<th>subdirectory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>bench</td>
<td>benchmark jobs and data for testing this version of IRIS-SEIS.</td>
</tr>
<tr>
<td>inc</td>
<td>include files which act like common blocks for various subroutines.</td>
</tr>
<tr>
<td>install</td>
<td>shell and make files needed to build IRIS-SEIS.</td>
</tr>
<tr>
<td>lib</td>
<td>archive libraries containing all object code for iris13 subdirectories.</td>
</tr>
<tr>
<td>main</td>
<td>main-level routines for the executables gntabl, irisgntabl, irisinit, and irisexec.</td>
</tr>
<tr>
<td>run</td>
<td>key files and executables for IRIS-SEIS. The important executables are irisinit and irisexec.</td>
</tr>
<tr>
<td>seismic</td>
<td>source and object code for all processors in IRIS-SEIS. Includes the FTOCIO package for FORTRAN-compatible C-language I/O routines.</td>
</tr>
<tr>
<td>utility</td>
<td>utility stand-alone programs which are compatible with IRIS-SEIS.</td>
</tr>
</tbody>
</table>
Figure 9: Installation Shell and Make files for IRIS-SEIS

The ..../sierra/iris13/install directory contains shell and make files needed to build IRIS-SEIS. The files within this directory can be divided into two parts: configuration and construction. The configuration files are used to set the installation environment:

<table>
<thead>
<tr>
<th>file</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>install</td>
<td>sets installation environment (defines directory paths), creates a new</td>
</tr>
<tr>
<td></td>
<td>&quot;makefile,&quot; and will even begin compilation.</td>
</tr>
<tr>
<td>installcfg</td>
<td>a template file for &quot;install.&quot;</td>
</tr>
<tr>
<td>protomakefile</td>
<td>a precursor of &quot;makefile,&quot; used in conjunction with &quot;installcfg&quot; by</td>
</tr>
<tr>
<td></td>
<td>&quot;install.&quot;</td>
</tr>
<tr>
<td>makefile.old</td>
<td>an earlier version of &quot;makefile.&quot;</td>
</tr>
</tbody>
</table>

The construction files are used to build the IRIS-SEIS executables:

<table>
<thead>
<tr>
<th>file</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>compall</td>
<td>compiles all source code in other IRIS-SEIS directories. This shell file</td>
</tr>
<tr>
<td></td>
<td>calls &quot;comptree.&quot;</td>
</tr>
<tr>
<td>compone</td>
<td>compiles a single .F subroutine to generate an object file. Proper</td>
</tr>
<tr>
<td></td>
<td>compilation flags are set within this shell file.</td>
</tr>
<tr>
<td>compmtree</td>
<td>a shell file which identifies and compiles all source code subroutines</td>
</tr>
<tr>
<td></td>
<td>within a given directory.</td>
</tr>
<tr>
<td>liball</td>
<td>obtains all object code in other IRIS-SEIS directories and creates</td>
</tr>
<tr>
<td></td>
<td>archive libraries using the archive command. Net result are &quot;.a&quot; files in</td>
</tr>
</tbody>
</table>
|           | the lib directory: irisseis.a and ftocio.a. This shell file calls "libtree."
| libtree   | a shell file which identifies and archives all object code within a given |
|           | directory.                                                               |
| makefile  | used with the make command to create irisinit and irisexec, the two      |
|           | executables which define IRIS-SEIS.                                      |
| makelk    | a shell file which makes the four key lookup subroutines which are        |
|           | necessary to make IRIS-SEIS work properly. This shell uses executable     |
|           | gntabl and text file PROC.LIS to create files "lkinit.f" and "lkexec.f."
|           | Executable irisgntabl and text file PROCIRIS.LIS then create files        |
|           | "irislkinit.f" and "irislkexec.f." Note that PROC.LIS here is the same as |
|           | that in ../sseis13/run/PROC.LIS. However, gntabl is a modified version of |
|           | that in ../sseis13/run.                                                   |
| ran.lib    | a text file which denotes the time and date that the archive libraries    |
|           | were last reconstructed.                                                  |
Functionality within IRIS-SEIS

Additional functionality is available within IRIS-SEIS. The ability to import or export data diskfiles from outside SIERRASEIS is provided. A tape operator system is provided so that tape mount requests can be fulfilled from various terminals or by other users; sharing of tape drivers by simultaneous jobs is possible. Trace monitoring and trace flow modification processors are available. Several trace header correction routines can be accessed in order to fix incorrectly defined headers.

These and additional functionality available within IRIS-SEIS are described in section III.
II-C. Structure of LOCAL-SEIS

The provision of a third level of processing subroutine definition and execution exists within IRIS-SEIS to allow for the addition of user-created processors. This structural extension allows users to add routines to IRIS-SEIS, essentially creating a local version able to mix the new routines with processors from SIERRASEIS and IRIS-SEIS without corrupting either of the packages.

LOCAL-SEIS, as named here, is the platform in which processing routines can be developed and tested. The added routines must be designed to conform to the SIERRASEIS environment (naming conventions, variable definition and usage, subroutine design, etc.). Provided that new routines are constructed properly, LOCAL-SEIS can provide a flexible or tailored processing environment depending on the added local applications.

In order to be compatible with SIERRASEIS, the LOCAL-SEIS construction and organization are very similar to those used to define IRIS-SEIS. If a user understands the structure of SIERRASEIS and IRIS-SEIS, he or she will understand the format of LOCAL-SEIS.

The intention here is that routines which are developed within LOCAL-SEIS can be forwarded to IRIS' SIERRASEIS Maintenance Center for inclusion into IRIS-SEIS. It is the hope that the IRIS SIERRASEIS users community will develop and share useful processing routines.

Directory/File Structure of LOCAL-SEIS in relation to SIERRASEIS and IRIS-SEIS

As LOCAL-SEIS is similar in form to IRIS-SEIS, the major subdirectory housing LOCAL-SEIS resides at the same directory level as the IRIS-SEIS directory. Figure 10 illustrates the relationship between the LOCAL-SEIS directory (.. ./sierra/local13), the IRIS-SEIS directory (.. ./sierra/iris13), and SIERRASEIS (.. ./sierra/sseisl3).

Structural and Operational framework of LOCAL-SEIS

LOCAL-SEIS has the same relationship to IRIS-SEIS that IRIS-SEIS has to SIERRASEIS. In this construction, when the driver-level executable, localinit, is asked to apply a processor, it first looks to see if the processor is provided within SIERRASEIS by using lookup subroutine lkinit.F. If not found, subroutine irislkinit.F is invoked to see if the processor is provided in IRIS-SEIS. If still not found, the subroutine loclkinit.F is called to see if the processor is defined as a local user addition. Since all SIERRASEIS, IRIS-SEIS, and local processors are linked to create the executable localinit, if the processor is defined at any of the three stages, the appropriate subroutine can be invoked. The same behavior is true for the execution executable localexec.

Once processing subroutine names are determined, the behavior of the drivers localinit and localexec are identical to those for ssinit, ssexec, irislinit, and irisexec. The initialization drivers all check user job listings and determine run-time resources. The execution drivers all control the (normally) trace-by-trace processing of seismic data.

Each local processor has an initialization and an execution subroutine. These subroutines must conform to the SIERRASEIS environment; i.e., they must be constructed in such a manner as to be able to be inserted directly into SIERRASEIS without causing fatal run-time behavior.

Trace flow behavior for localinit and localexec is the same as that found in SIERRASEIS and IRIS-SEIS. The trace-by-trace processing scheme is still the framework about which subroutines must be constructed; multiple-trace action must be coded into individual subroutines.
FIGURE 10: Directory/File Structure of LOCAL-SEIS

The structural relationship between the directories for LOCAL-SEIS, IRIS-SEIS, and SIERRASEIS are described below. As can be seen, the subdirectories are organized in similar manner.

```
.../sierra/
  sseis13/
    extend/           ssinit.o, ssexec.o
    inc/             sseis.a, extend.a
    install/          main/          ssinit, ssexec, PROC.LIS, lkinit.o, lkexec.o
    lib/                     run/           ssinit.o, ssexec.o
    main/         seismic/  ...
.../iris13/
  inc/                      irisinit.o, irisexec.o
  install/                   main/          irisininit, irisexec, PROCIRIS.LIS, irislkinit.o, irislkexec.o,
  lib/                      run/           lkinit.o, lkexec.o
  main/         seismic/     ...
.../local13/
  install/                   lib/            locseis.a
  main/                      main/           localinit.o, localexec.o
  run/                       run/           localinit, localexec, PROCLOC.IRIS, loclkinit.o, loclkexec.o
  seismic/                   ...
```

Note 1: "lkinit.o" and "lkexec.o" in .../iris13/run are not the same as those files with the same name in .../sseis13/run.

To create localinit, one must link the following:

```
f77 localinit.o loclkinit.o irislkinit.o lkinit.o locseis.a irisseis.a sseis.a extend.a (+other SIERRA.a's) -o localinit.
```

localexec is made using the following:

```
f77 localexec.o loclkexec.o irislkexec.o lkexec.o locseis.a irisseis.a sseis.a extend.a (+other SIERRA.a's) -o localexec.
```
Text file PROCLOC.LIS and lookup subroutines

The lookup subroutines lockinit.F and loclkexec.F are constructed using a stand-alone executable locgntabl and the text file PROCLOC.LIS. This text file is the master list of locally added routines and also provides the names of the initialization and execution subroutines. The executable locgntabl interprets the text file to create the lookup subroutines.

<table>
<thead>
<tr>
<th>file</th>
<th>SIERRASEIS</th>
<th>IRIS-SEIS</th>
<th>LOCAL-SEIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>master list</td>
<td>PROC.LIS</td>
<td>PROCIRIS.LIS</td>
<td>PROCLOC.LIS</td>
</tr>
<tr>
<td>executable:</td>
<td>gntabl</td>
<td>ir伊斯gntabl</td>
<td>locgntabl</td>
</tr>
<tr>
<td>output:</td>
<td>lkinit.F</td>
<td>irislkinit.F</td>
<td>loclkinit.F</td>
</tr>
<tr>
<td>initialization</td>
<td>lkexec.F</td>
<td>irislkexec.F</td>
<td>loclkexec.F</td>
</tr>
</tbody>
</table>

IRIS-SEIS has in the directory ../iris13/run its own version of gntabl, ir伊斯gntabl, lkinit.o, lkexec.o, irislkinit.o, and irislkexec.o. LOCAL-SEIS uses the object code of these files to link with its own files, loclkinit.o and loclkexec.o, which are located in ../local13/run.

Organization framework of LOCAL-SEIS

Since locally developed processors are added to LOCAL-SEIS, the programmer must become quite familiar with the sub-directory structure and file organization of the ../local13 directory. The sub-directory structure is very similar to that found in ../iris13; see Figure 11. As many files from IRIS-SEIS are used as is possible; only the files in LOCAL-SEIS are duplicated if necessary. Full procedures for the addition of a new processor are given in section IV-C.

The main directory contains the source and object code for the driver-level routines localinit and localexec. These do not need to be modified in order to add a processor. The program "locgntabl.F" should not be modified, either.

The lib directory contains an archive library of processor object code from the seismic directory. A shell file within the install directory will update the locseis.a file.

The run directory contains needed object code (lookup subroutines) and final executables localinit and localexec.

The file PROCLOC.LIS should be modified just once for each new processor added to LOCAL-SEIS. Prior to making the executables, PROCLOC.LIS should be edited to include the name of the new processor and the names of the corresponding initialization and execution subroutines. These names must conform to naming conventions as defined within SIERRASEIS. Once these names are entered, they should not be revised whenever the subroutines in the seismic directory are modified. These entries should be changed only if the names of the processor or subroutines are changed.

The initialization and execution subroutines should be placed into directory seismic. The object code can be created by running within this directory the shell file "compone" found in the install directory. A typical compile command may look like:

(prompt) ../install/compone routine.F

The shell file "compone" contains numerous compilation flags and options. The include file compilation flag is used within this shell - see a listing of "compone". The path listing in the command above illustrates the location of the shell relative to subroutines within the seismic directory. Once error-free object files are constructed, the proper installation should continue from the directory install.

The install directory contains nearly all the shell and make files required to install or update LOCAL-SEIS.
FIGURE 11: Table of LOCAL-SEIS version 1.3 Subdirectories

The following directories are contained within the ../sierra/local13 directory.

<table>
<thead>
<tr>
<th>subdirectory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>install</td>
<td>shell and make files needed to build LOCAL-SEIS.</td>
</tr>
<tr>
<td>lib</td>
<td>archive library containing all object code for local13 subdirectories.</td>
</tr>
<tr>
<td>main</td>
<td>main-level routines for the executables locgntabl, localinit, and localexec.</td>
</tr>
<tr>
<td>run</td>
<td>key files and executables for LOCAL-SEIS. The important executables are localinit and localexec. The PROCLOC.LIS file should have an entry for each added processor with the names of their corresponding initialization and execution subroutines.</td>
</tr>
<tr>
<td>seismic</td>
<td>source and object code for all processors in LOCAL-SEIS. Locally-produced subroutines should be placed here.</td>
</tr>
<tr>
<td>user1</td>
<td>a development directory for a user. This directory is a work area where one may develop a processor prior to insertion into LOCAL-SEIS. The directory contains subdirectories similar to those found in ../local13.</td>
</tr>
</tbody>
</table>

Note: No inc directory is present; LOCAL-SEIS uses the include files from IRIS-SEIS (../iris13/inc).
User Development area

For some sites, testing and adding processors into LOCAL-SEIS will be sufficient. For a multi-user locality where different persons are developing routines, LOCAL-SEIS may best serve as the working, net 'local' version. In this mode, new processors are best developed outside so that the local version is not accidentally corrupted. Within local13/, a subdirectory is provided whereby a user can develop subroutines prior to inclusion into LOCAL-SEIS.

The local13/userl directory contains the files and subdirectories which define a development environment for individual users. This directory should be copied for each person who wishes to create SIERRA-SEIS - compatible processing subroutines. The relationship between the user development directory and LOCAL-SEIS is analogous to the relationship between LOCAL-SEIS and IRIS-SEIS or between IRIS-SEIS and SIERRA-SEIS. Essentially, the userl directory is a structural framework whereby processors can be added but will (should) not corrupt its host LOCAL-SEIS.

If a user is familiar (by now) with the structure of SIERRASEIS, IRIS-SEIS, and LOCAL-SEIS, he or she will be familiar with the userl directory. Figure 12 illustrates the directory structure of user. Five directories are defined: install which contains make and shell files to make test executables userinit and userexec; lib which contains a development archive library; main which contains subroutines which do not need any modification; run which contains object code needed for proper compilation and the test executables userinit and userexec; and seismic which is where development subroutines can be located.

The steps required in order to add a new test routine are described in section IV. Briefly, these steps are:
(a) add new processor and subroutine names to text file
  ../local13/userl/run/PROCUSER.LIS. Example: TESTGAIN and TGAIN0, TGAINX.
(b) write subroutines tgain0.F and tgainX.F.
(c) compile subroutines:
    (%%) ..install/compone tgain0.F
    (%%) ..install/compone tgainX.F
(d) make archive library: run shell liball in ../userl/install.
(e) make lookup subroutines: run shell makelk in ../userl/install.
(f) make executables:
    (%%) make userinit; make userexec

To modify and reinstall a pre-existing subroutine, steps b, c, d, and f must be used.
FIGURE 12: Directory/File Structure of USER DEVELOPMENT work area

The structural relationship between the directories for the user development work area (user1), LOCAL-SEIS, IRIS-SEIS, and SIERRASEIS are described below. As can be seen, subdirectories are organized in similar manner. Compilation of development subroutines involves linking object code from several directories.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>.../sierra/</td>
<td>sseis13/ extend/ inc/ install/ lib/ main/ run/ seismic/ sseis.a, extend.a</td>
</tr>
<tr>
<td></td>
<td>iris13/ inc/ install/ lib/ main/ run/ seismic/ irisiseis.a irisinit.o, irisexec.o irisinit, irisexec, PROCIRIS.LIS, PROC.LIS, irislkinit.o, irislkexec.o lkinit.o, lkexec.o</td>
</tr>
<tr>
<td></td>
<td>local13/ install/ lib/ main/ run/ seismic/ locseis.a localinit.o, localexec.o localinit, localexec, PROCLOC.IRIS, loclkinit.o, loclkexec.o</td>
</tr>
<tr>
<td></td>
<td>user1/ install/ - make files lib/ - userseis.a main/ - userinit.o, userexec.o run/ - userinit, userexec, PROCUSER.LIS, userlkinit.o, userlkexec.o seismic - subroutine.F, subroutine.o</td>
</tr>
</tbody>
</table>

To create userinit, one must link the following:

\[
\text{f77 userinit.o userlkinit.o loclkinit.o irislkinit.o lkinit.o userseis.a locseis.a irisseis.a } \\
\text{sseis.a extend.a (+other SIERRA.a's) -o userinit.}
\]

userexec is made using the following:

\[
\text{f77 userexec.o userlkinit.o loclkexec.o irislkexec.o lkexec.o locseis.a irisseis.a sseis.a } \\
\text{extend.a (+other SIERRA.a's) -o userexec.}
\]

Creation of the executables as stated above will produce working versions able to mix the development processors with those available in SIERRASEIS, IRIS-SEIS, and LOCAL-SEIS.

An alternative approach to creation of the executables is to only specify those subroutines needed for a particular test job. The executables will be much smaller in size and will take less time to compile/link.
III - IRIS-SEIS

In this section, information is provided on how to install IRIS-SEIS, how to run a job with IRIS-SEIS, what capabilities are provided by the IRIS-SEIS processors, and what utility programs are provided which are compatible with IRIS-SEIS processors.

A complete set of manual pages for IRIS-SEIS processors is provided in APPENDIX ONE.

III-A. Installation of IRIS-SEIS

This chapter presents a step-by-step installation procedure for IRIS-SEIS. It is not necessary to use the "SIERRA" account used for installation of SIERRASEIS, however, write-permission is required within the main SIERRA directory (./sierra).

For the installation and make files, the SIERRA directory is assumed to be placed into the root-level directory named usr; the sierra directory is defined as /usr/sierra. Given this definition for all needed paths, the installation may be made easier by creating a symbolic link of either /sierra within /usr to point to the directory containing sierra or /usr/sierra pointing to the subdirectories within sierra. However, by editing a few files, the user can explicitly give the directory where sierra/ is located if it is not desired to define a root-level symbolic link.

The installation procedures below discuss a quick and a complete approach to installation of IRIS-SEIS. The quick approach can be used as needed object code and executables are already provided in compiled/linked form within the distribution tar tape. These files can be used provided the host CPU is compatible with a SUN 4 (SPARC) with OS 4.0.3. If the host CPU is not compatible, then the full installation procedures should be used.

The installation of LOCAL-SEIS is covered in section IV-A.

Quick Installation of IRIS-SEIS

To install IRIS-SEIS using the quick steps, the following steps are required:

1. extract IRIS-SEIS from the distribution tar tape,
2. update the SGCONFIG.DAT file,
3. either:
   (A) create a symbolic link /usr/sierra or sierra within /usr to point to the overall SIERRA directory, or
   (B) make full pathname changes to the following files (#lines to change):
       -iris13/install/Makefile: three lines
       -iris13/install/Makefile.old: three lines
       -iris13/install/compone: one line
       -iris13/inc/IRISESSEIS.INC: one line
       -bin13/sh.link: all lines
4. recompile iris13/seismic/inoutoper.dir/oper_opcom.F,
5. run shell iris13/install/liball,
6. use the makefile to create executables irisinit and irisexec,
7. run shell file iris13/utility/makeall,
8. fix symbolic links in ./sierra/bin13.
(1) Extract IRIS-SEIS from distribution tape

The normal installation of SIERRASEIS will create a directory structure consisting of a main Sierra directory ("sierra") in which subdirectories named slib15, sseisl3, and rasvue (or rasvue new) will be present. In general, there should also be two text files present: SGCONFIG.DAT and SGPROD.DAT.

Change the working directory to the SIERRASEIS directory where these subdirectories and files are present (e.g., .../sierra/).

Use the tar command to extract all files, e.g.:

(%t) tar xvf /dev/rmt16

The proper tape drive device name must be used. When the tar is completed, three new subdirectories should exist: iris13, local13, and bin13. The directory for IRIS-SEIS should use about 15 Mbytes of disk; the directory for LOCAL-SEIS will require about 8 Mbytes. The directory bin13 should be very small (<10K).

(2) Update the SGCONFIG.DAT file

The SGCONFIG.DAT file defines some run-time environment variables which are needed for SIERRASEIS to run. Since IRIS-SEIS is constructed as similar to SIERRASEIS as is possible, a few extra environment variables need to be defined within this file. A backup copy of this file must be made before the implementation of the following changes.

The original SGCONFIG.DAT file may contain as an example the following lines (assuming RASVUE is installed too):

```
DEFAULT_product /usr/sierra/
DEFAULT_SSERRORS /usr/sierra/sseis13/run/
DEFAULT_PLTCFG /usr/sierra/sseis13/plot/PLTCFG.V36INCH
!
!----------------------------------------
!
RASVUE definitions (e.g,:)
!
DEFAULT_MENU /usr/sierra/rasvue_new/
!
```

Add the following lines (note the trailing "/"):

```
DEFAULT_IRISERR /usr/sierra/iris13/run/
DEFAULT_LOCERR /usr/sierra/local13/run/
```

If the use of the symbolic link for /usr/sierra is chosen, then the above pathnames should be kept. If instead of the symbolic link the full pathnames are used, then the correct path must be substituted for /usr/sierra. The path given for DEFAULT_IRISERR should be the directory which contains the file IRISERR.LST. The path given for DEFAULT_LOCERR should be the directory which contains LOCERR.LST.

No change should be made to the SGPROD.DAT file.

(3) Directory pathnames: symbolic links or explicit definition

Several installation and update make/shell files reference object code or include files in order to properly build the IRIS-SEIS executables. The correct locations of these files must be stated. The installation/make files have used the definition /usr/sierra as the directory which contains sseis13, iris13, and local13.

A symbolic link can be used to point this directory definition to the true location where the user's sierra/ directory resides. Within the /usr root-level directory, a symbolic link named sierra should be created. For example:

```
(%) ln -s /home/seis/sierra sierra
```

This symbolic link will equate /usr/sierra with the directory location /home/seis/sierra.
Alternatively, if the symbolic link is not used, the directory paths should be given explicitly by the user. Changes to the following files must be made:

<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
</tr>
</thead>
<tbody>
<tr>
<td>iris13/install/makefile</td>
<td>ROOT DIR = /usr/sierra/iris13</td>
</tr>
<tr>
<td></td>
<td>RUN_DIR = /usr/sierra/iris13/run</td>
</tr>
<tr>
<td></td>
<td>LK_DIR = /usr/sierra/iris13/run</td>
</tr>
<tr>
<td>iris13/install/makefile.old</td>
<td>ROOT_DIR = /usr/sierra/iris13</td>
</tr>
<tr>
<td></td>
<td>RUN_DIR = /usr/sierra/iris13/run</td>
</tr>
<tr>
<td></td>
<td>LK_DIR = /usr/sierra/iris13/run</td>
</tr>
<tr>
<td>iris13/install/compone</td>
<td>INC_DIR = /usr/sierra/iris13/inc</td>
</tr>
<tr>
<td>iris13/inc/IRISSEIS.INC</td>
<td>data tapeophysname '/[path]/TAPEOP.HIS'</td>
</tr>
<tr>
<td>bin13/sh.link</td>
<td>[all paths of /usr/sierra...]</td>
</tr>
</tbody>
</table>

All references to /usr/sierra should be modified to the true location of the user's sierra/ directory. When these changes are made, the new pathnames should point to the true locations of the directories or files.

(4) **Recompile one subroutine: oper_opcom.F**

One subroutine will need to be recompiled if the symbolic link /usr/sierra is not used:

(%) cd ../iris13/seismic/inoutoper.dir

(%). ./install/compone oper_opcom.F

An object code file should be created. The use of '../..' is very important here - the iris13/install directory is referenced using the relative position of the subdirectories. This subroutine, oper_opcom.F, is used for the tape operator/external tape mounting system which is a feature within IRIS-SEIS.

When this routine is recompiled along with all other subroutines, then the steps below should be followed:

(%) cd ../iris13/install

(%) compall

The compall shell will recompile all source code. This process will take a while.

(5) **Make the IRISSEIS.a archive library**

Now that the subroutine object code is constructed, the appropriate archive library is needed in order to compile/link. The user should change directory to the install directory if he or she is not already there:

(%) cd ../iris13/install

(%) liball

Running the liball shell will create or update the archive library irisseis.a. This shell will also create the library of FORTRAN-to-C I/O functions (ftocio.a).

(6) **Create IRIS-SEIS executables**

Create the IRIS-SEIS executables irisinit and irisexec using the makefile:

(%) make irisinit

and

(%) make irisexec.

The executables will be placed into ../iris13/run.
(7) Make utility programs

Several utility programs which are compatible with IRIS-SEIS reside in the directory 
../iris13/utility. These can be compiled using the following:

(%) cd ../iris13/utility
(%) makeall.

The shell file makeall will compile several programs by invoking the compilation shell 
Fcomp and maketapeoper.

(8) Fix symbolic links in ../sierra/bin13

The directory bin13 is provided for convenience to point to the various directories where 
executables reside. Rather than place the several directories into one's PATH environment 
variable or create aliases for each executable, one may place just the ../sierra/bin13 directory 
in the PATH environment variable list. Within this directory, symbolic links for all IRIS- 
SEIS (and SIERRASEIS) executables point to the proper executables. This directory 
simplifies management of executable locations and definitions.

To define the proper symbolic links, it is essential that correct path definitions were 
entered into the file ../sierra/bin13/sh.link as specified in step (1) above. Then:

(%) cd ../sierra/bin13
(%) sh.link

The shell file sh.link will create the proper symbolic links. The pathnames can be verified 
by fully listing the directory contents ('ls -l') or by running an executable program.

This directory must be placed into the user's PATH environment variable listing (in the 
.login or .cshrc file).

---

Full Installation of IRIS-SEIS

To conduct a full installation of IRIS-SEIS, the following steps are needed:

1. extract IRIS-SEIS from the distribution tar tape,
2. update the SGCONFIG.DAT file,
3. either:
   A) create a symbolic link /usr/sierra or sierra within /usr to point to the 
      overall SIERRA directory, or
   B) make full pathname changes to the following files (#lines to change):
      *iris13/install/makefile: three lines
      *iris13/install/makefile.old: three lines
      *iris13/install/compone: one line
      *iris13/inc/IRISSEIS.INC: one line
      *bin13/sh.link: all lines
4. compile all source code
5. create archive libraries using shell iris13/install/liball,
6. run the 'make' option within the install shell in iris13/install
7. run shell makelk in iris13/install
8. use the makefile to create executables irisinit and irisexec,
9. run shell file iris13/utility/makeall,
10. fix symbolic links in ../sierra/bin13.

1. Extract IRIS-SEIS from distribution tape

The normal installation of SIERRASEIS will create a directory structure consisting of a 
main Sierra directory ("sierra") in which subdirectories named s1ib15, sseis13, and rasvue (or 
rasvue new) will be present. In general, there should also be two text files present: 
SGCONFIG.DAT and SGPROD.DAT.
The working directory should be the SIERRASEIS directory where these subdirectories and files are present (e.g., .../sierra/).

Use the `tar` command to extract all files, e.g.;

```
(%) tar xvf /dev/rmt16
```

The proper tape drive device name should be used. When the `tar` is completed, three new subdirectories should be created: `iris13`, `local13`, and `bin13`. The directory for IRIS-SEIS should use about 15 Mbytes of disk; the directory for LOCAL-SEIS will require about 8 Mbytes. The directory `bin13` should be very small (<10K).

(2) Update the SGCONFIG.DAT file

The SGCONFIG.DAT file defines some run-time environment variables which are needed for SIERRASEIS to run. Since IRIS-SEIS is constructed as similar to SIERRASEIS as is possible, a few extra environment variables need to be defined within this file. A backup copy of this file should be made before the implementation of the changes below:

The original SGCONFIG.DAT file may contain as an example the following lines (assuming RASVUE is installed too):

```
DEFAULT_product /usr/sierra/
DEFAULT_SSERRORS /usr/sierra/sseis13/run/
DEFAULT_PLTCFG /usr/sierra/sseis13/plot/PLTCFG.V36INCH
```

```
!----------------------------------------
! RASVUE definitions (e.g.,):
DEFAULT_MENU /usr/sierra/rasvue_new/
```

Add the following lines (note the trailing "/"):

```
DEFAULT_IRISERR /usr/sierra/iris13/run/
DEFAULT_LOCERR /usr/sierra/local13/run/
```

If the use the symbolic link for `/usr/sierra` is chosen, then the above pathnames should be kept. If instead of the symbolic link the full pathnames are used, then the correct path must be substituted for `/usr/sierra`. The path given for DEFAULT_IRISERR should be the directory which contains the file `IRISERR.LST`. The path given for DEFAULT_LOCERR should be the directory which contains `LOCERR.LST`.

No change should be made to the SGPROD.DAT file.

(3) Directory pathnames: symbolic links or explicit definition

Several installation and update make/shell files reference object code or include files in order to properly build the IRIS-SEIS executables. The correct locations of these files must be stated. The installation/make files have used the definition `/usr/sierra` as the directory which contains `sseis13`, `iris13`, and `local13`.

A symbolic link can be used to point this directory definition to the true location where the user’s `sierra` directory resides. Within the `/usr` root-level directory, create a symbolic link named `sierra`. For example:

```
(%) ln -s /home/seis/sierra sierra
```

This symbolic link will equate `/usr/sierra` with the directory location `/home/seis/sierra`.

Alternatively, if the symbolic link is not used, the directory paths should be given explicitly by the user. Changes to the following files must be made:

<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
</tr>
</thead>
<tbody>
<tr>
<td>iris13/install/makefile</td>
<td>ROOT_DIR = /usr/sierra/iris13</td>
</tr>
<tr>
<td></td>
<td>RUN_DIR = /usr/sierra/iris13/run</td>
</tr>
<tr>
<td></td>
<td>LK_DIR = /usr/sierra/iris13/run</td>
</tr>
</tbody>
</table>
(4) Recompile all source code

To recompile all IRIS-SEIS source code, do the following:

(%) cd /iris13/install

(%) compall

The `compall` shell will recompile all source code. This process will take a while.

(5) Make the IRISSEIS.a archive library

Now that all subroutine object code are constructed, the appropriate archive library is needed in order to compile/link. Within the install directory:

(%) liball

Running the `liball` shell will create or update the archive library `irisseis.a`. This shell will also create the library of FORTRAN-to-C I/O functions (`ftocio.a`).

(6) Run 'make' option in install shell

At this point, nearly all the necessary object code will exist. The missing subroutines are the lookup subroutines used by the IRIS-SEIS drivers. These subroutines are created by running stand-alone programs `ssgntabl` and `irisgntabl` (as discussed in section II A-C). Before the creation of the lookup tables, it is necessary that the stand-alone executables exist. This can be done in two different ways.

The first approach is using the `install` shell. Within

(%) `install`

If the user gets the UNIX operating system command `install`, then his or her path is set so that the current directory is not searched first. This can be avoided by renaming the `install` shell:

(%) `mv install install.sh`

(%) `install.sh`

When the screen menu appears, the 'make' option should be selected. The shell file will proceed to create executables `gntabl` and `irisgntabl`.

The second approach may be easier. Within the `iris13/install` directory, run:

(%) `make ssgntabl`

and

(%) `make irisgntabl`


Since the stand-alone executables exist from step (6), the lookup subroutines can be created. Within the `iris13/install` directory, the programmer should run the shell file `makelk`:

(%) `makelk`

This shell will run the stand-alone programs `ssgntabl` and `irisgntabl`, using the text files
.../iris13/run/PROC.LIS and .../iris13/run/PROCIRIS.LIS as input. The SIERRASEIS banner should flash on the screen for each executable run.

From the execution of these programs, the output will be the lookup subroutines. These subroutines will also be compiled to produce object code. The source and object code of these routines will be placed into the .../iris13/run directory.

(8) Create IRIS-SEIS executables

Create the IRIS-SEIS executables irisinit and irisexec using the makefile located in ...

.../iris13/install:

(%) make irisinit

and

(%) make irisexec.

The executables will be placed into ...

.../iris13/run.

(9) Make utility programs

Several utility programs which are compatible with IRIS-SEIS reside in the directory ...

.../iris13/utility. These can be compiled using the following:

(%) cd ...

.../iris13/utility

(%) makeall.

The shell file makeall will compile several programs by invoking the compilation shell Fcomp and maketapeoper.

(10) Fix symbolic links in ...

.../bin13

The directory bin13 is provided for convenience to point to the various directories where executables reside. Rather than place the several directories into a PATH environment variable or create aliases for each executable, the user may place just the ...

.../sierra/bin13 directory in the PATH environment variable list. Within this directory, symbolic links for all IRIS-SEIS (and SIERRASEIS) executables point to the proper executables. This directory simplifies management of executable locations and definitions.

To define the proper symbolic links, it must be ensured that correct path definitions were entered into the file ...

.../sierra/bin13/sh.link as specified in step (1) above. Then:

(%) cd ...

.../sierra/bin13

(%) sh.link

The shell file sh.link will create the proper symbolic links. The pathnames can be verified by a complete listing of the directory contents ('ls -l') or by running an executable program.

The programmer must ensure that this directory is placed into his or her PATH environment variable listing (in the .login or .cshrc file).

Verification of Installation

The best test of a successful installation is to run an IRIS-SEIS job. The executables can be tested in two parts. First, a job which calls only SIERRASEIS processors (a plot job, for example) and is known to run correctly within SIERRASEIS can be used. Same results should be produced by using IRIS-SEIS. Second, an IRIS-SEIS processor can be inserted into this job stream, then run with IRIS-SEIS. If the processor is called correctly, then no runtime errors should occur.

The next section (III-B) describes how to run IRIS-SEIS jobs.
The run-time environments of SIERRASEIS and IRIS-SEIS are very similar due to the structural similarity between SIERRASEIS and IRIS-SEIS. Since IRIS-SEIS is in actuality an extension of SIERRASEIS, the behavior of IRIS-SEIS is the same as that for SIERRASEIS. Because of this, IRIS-SEIS is downward-compatible; that is, a job which will run in SIERRASEIS will run in IRIS-SEIS (but a job which runs in IRIS-SEIS may not necessarily run within SIERRASEIS).

To run a job within IRIS-SEIS, the user follows the approach needed to run a job in SIERRASEIS:
(a) construct a job listing. The calls to processors and the list of processing parameters must be syntactically correct.
(b) check the job listing by using the initialization executable, irisinit.
(c) run the job by using the execution executable, irisexec.

The format of the job listing must conform to the rules set forth for SIERRASEIS job listings. IRIS-SEIS processors can be called at any point within a job listing. These processors can be treated as if they were SIERRASEIS processors.

The executables irisinit and irisexec are run as if they were the executables ssinit and ssexec. irisinit interprets a job listing and determines run-time conditions. irisexec conducts the actual processing.

The trace-by-trace data handling scheme is adhered to by IRIS-SEIS. This scheme is identical in behavior to that used by SIERRASEIS. The contents of trace headers (GCV's) should be understood in order to allow for proper trace flow to occur.

Shells to simplify the initialization and execution steps: IRISCHECK and IRISRUN

The run-time behavior of the SIERRASEIS and IRIS-SEIS driver executables is based on interactive entry of job information. In this mode, the job listing name and job-execution parameters are entered via a console keyboard at run-time.

This behavior is not always optimum, particularly when a series of jobs to be run does not need monitoring. IRIS-SEIS provides a few shell files which allow users to run an IRIS-SEIS job using command-line entry of parameters and to run jobs from within shell file listings.

The normal commencement of an IRIS-SEIS or SIERRASEIS job will require from the user four keyboard entries for the initialization step and (at least) two entries for the execution step. IRIS-SEIS shell files IRISCHECK and IRISRUN are constructed to allow the user to run a job by providing command line parameters rather than to interactively enter parameters.

For example, the initialization step irisinit can be replaced with the command IRISCHECK:

<table>
<thead>
<tr>
<th>irisinit: query:</th>
<th>user response:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(%)</td>
<td>irisinit</td>
</tr>
<tr>
<td>SierraSeis banner</td>
<td></td>
</tr>
<tr>
<td>input file name</td>
<td>job listing name (e.g., FILTER.DAT)</td>
</tr>
<tr>
<td>job sequence number</td>
<td>CODE (e.g., FILT)</td>
</tr>
<tr>
<td>version number</td>
<td>&lt;return&gt; to default</td>
</tr>
<tr>
<td>run-time options</td>
<td>&lt;return&gt; to default</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IRISCHECK: query:</th>
<th>user response:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(%)</td>
<td>IRISCHECK FILTER.DAT FILT</td>
</tr>
</tbody>
</table>

By providing run-time information using command line parameters needed by the shell file IRISCHECK, the interactive query system required by irisinit can be bypassed. The need to
enter the first <return> after the initial SIERRA-SEIS banner can also be bypassed. This shell assumes that the user will wish to select default values for the version number and run-time options.

An additional advantage is that this command line approach can be easily placed into 'background' execution. This replaces the steps of interactive parameter entry followed by an interrupt and the issuance of the 'background' UNIX command:

**IRISCHECK FILTER.DAT FILT &**

as opposed to the entire sequence of:

```
(%) irisinit
[SierraSeis banner] <return>
Input file name: FILTER.DAT
job sequence number: FILT
version number: <return> [to default]
run-time options: <return> [to default]
'Z' [to interrupt]
(%) bg [to place in background]
```

The execution phase can be run using a similar command-line shell:

```
irisexec:
query:           user response:
(%)             irisexec
SierraSeis banner <return>
job sequence number CODE (e.g., FILT)
version number <return> to default
```

```
IRISRUN: query:           user response:
(%)             IRISRUN FILT
```

The commencement of a processing job is simple using the command-line approach. Again, the process can be easily placed into 'background' using this approach.

The use of IRISRUN does not take subsequent interactive queries into account. For example, answers to tape drive requests are not placed on the command line. The IRISRUN shell will not allow for tape requests and will most likely abort.

Two ways to circumvent the above problem for background execution exist. The first approach is to create a copy of IRISRUN and insert the correct tape request responses into the shell file prior to running the job. Alternatively, one can use the external tape request option provided within IRIS-SEIS to remove all tape requests from requiring interactive keyboard response (explained in more detail in section III-C).

**Two additional run-time shells: irischeck and irisrun**

The SIERRA-SEIS banner displayed at the beginning of an IRIS-SEIS job causes the elimination of whatever console screen information existed on the screen. The user can bypass this behavior by using the shell files irischeck and irisrun. These shells are used in a manner identical to IRISCHECK and IRISRUN but no information should be sent to the user's console screen. These shells redirect screen information to temporary files (which are removed upon completion of the job process).

For the above examples, the job listing FILTER.DAT can be run using the two lines:

```
(%) irischeck FILTER.DAT FILT
(%) irisrun FILT &
```

The screen will not be erased by the SIERRA-SEIS banner (nor does the user have to wait for it to finish writing).
The shell files IRISCHECK, IRISRUN, irischeck, and irisrun are located in the
.../iris13/run directory. Symbolic link pointers exist within /.../bin13 so that if this latter
directory is already in the programmer's PATH environment variable list (i.e. irisinit or
irisexec can already be accessed), he or she should be able to use these shells.

Summary of IRIS-SEIS executables and shells:

<table>
<thead>
<tr>
<th>name</th>
<th>parameter entry</th>
<th>screen behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>irisinit</td>
<td>interactive keyboard</td>
<td>banner clears screen and queries printed</td>
</tr>
<tr>
<td>irisexec</td>
<td>interactive keyboard</td>
<td>banner clears screen and queries printed</td>
</tr>
<tr>
<td>IRISCHECK</td>
<td>command line</td>
<td>banner clears screen and queries printed</td>
</tr>
<tr>
<td>IRISRUN</td>
<td>command line</td>
<td>banner clears screen and queries printed</td>
</tr>
<tr>
<td>irischeck</td>
<td>command line</td>
<td>nothing to screen</td>
</tr>
<tr>
<td>irisrun</td>
<td>command line</td>
<td>nothing to screen</td>
</tr>
</tbody>
</table>
III-C. Processors within IRIS-SEIS

The processors available within IRIS-SEIS can be divided into five types: data processing, utility processors related to the seismic data, trace header (GCV) manipulation, utility processors related to the processing job, and modifications of SIERRASEIS processors (Figure 13). These processors are provided to add extra capability to those available in SIERRASEIS. Several capabilities provided in IRIS-SEIS are described below followed by descriptions of processors and their functionality. Manual pages for each processor are provided in Appendix One.

Important functionality within IRIS-SEIS includes: structural examples of simple and complex processors (/EASYADD, /EXTRCTR, and EXMULTRC), external job monitoring (/MONITOR), dynamic trace header definition (/DEFHEAD, /HEADEQL, /HEADLIST, /HDMINMAX), irregular data entry (/GATHCNTR, /TRCOUNTR, /INTRLEAV), physical omission of traces (/OMIT, /OMITTRAC), and an external tape mount request/response system (/INOPER, /OUTOPER).

Job utility processors

Print total number of GCV is data: /COMVARPR

A small utility processor will indicate the total number of GCV's which are used for the current processing job. SIERRASEIS/IRIS-SEIS jobs customarily have a minimum number of trace headers which are defined. One can determine if additional, user-defined GCV's are used given the output from this processor.

Normally there are 137 integer, 71 real, 10 character*8, and 1 character*4 GCV's used within SIERRASEIS. Any total exceeding these indicates that user-defined headers exist.

A major role that this processor plays is that it is essentially a very minor-impact processor which can be inserted into a list of processors. Due to programming oddities within processor subroutines, occasionally a sequence of processors may result in a syntactical error in the initialization stage. For some reason, inserting essentially a dummy or null routine into the middle of the sequence can break up a bad sequence and remove the error.

Example of a very simple trace-by-trace processor: /EASYADD

The processor /EASYADD is an example of a processor which works in trace-by-trace mode. An algorithm which operates strictly on individual seismograms can be inserted into LOCAL-SEIS using the initialization and execution subroutines associated with this processor. The /EASYADD subroutines are provided as examples which one can be copied and modified.

The EASYADD processor assumes that a given algorithm will be essentially hardwired into the processor; no user-choice parameters are to be defined (see /EXTRCTR for examples of how to define user parameters).

The subroutines for EASYADD should not be modified. Rather, a copy of each subroutine, EASYIN.F and EASYEX.F in subroutine directory ../local13/seismic or into a user development-level subdirectory within ../local13. The new processor should have its own processor name and should be defined in the ../local13/run/PROCLOC.LIS text file. Normal installation and update procedures can be followed (see section IV-C).

Example of a trace-by-trace processor with user parameters: /EXTRCTR

The definition and use of user-choice parameters makes a processor generically more useful. Rather than having hardwired algorithm parameters for special case processing, a processor can be applied to various data sets if algorithm parameters can be selected within a job listing. Unlike /EASYADD, the processor /EXTRCTR is an example of a trace-by-trace processor which has several parameters defined for user selection.
FIGURE 13: Processors Within IRIS-SEIS

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job Utility</td>
<td>/COMVARPR</td>
<td>prints total number of GCV values</td>
</tr>
<tr>
<td></td>
<td>/EASYADD</td>
<td>example of simple trc-trc processor</td>
</tr>
<tr>
<td></td>
<td>/EXTRCTRC</td>
<td>example of trc in/trc out processor</td>
</tr>
<tr>
<td></td>
<td>/EXMULTRC</td>
<td>example of multiple trace processor</td>
</tr>
<tr>
<td></td>
<td>/MONITOR</td>
<td>* monitor job by printing trace GCV's values</td>
</tr>
<tr>
<td></td>
<td>/NUMTRFLO</td>
<td># of traces passing this point of trace flow</td>
</tr>
<tr>
<td></td>
<td>/UNIXIO</td>
<td>read/write UNIX binary file with/without GCV's)</td>
</tr>
<tr>
<td>Headers</td>
<td>/DEFHEAD</td>
<td>define new GCV/trace header</td>
</tr>
<tr>
<td></td>
<td>/FLOTDAT</td>
<td>calculate floating datum statics</td>
</tr>
<tr>
<td></td>
<td>/GATHCNTR</td>
<td>set KSHOT when KFLDFN changes</td>
</tr>
<tr>
<td></td>
<td>/HEADEQL</td>
<td>set any GCV value w/ scalar math option</td>
</tr>
<tr>
<td></td>
<td>/HEADLIST</td>
<td>print values of any 6 specified GCV's</td>
</tr>
<tr>
<td></td>
<td>/HDMINMAX</td>
<td>get GCV min/max values within data</td>
</tr>
<tr>
<td></td>
<td>/TRCOUNTR</td>
<td>fix trace header keyed on gather number</td>
</tr>
<tr>
<td>Data Utility</td>
<td>/AMPDUMP</td>
<td>dump amplitudes; like /TRDUMP</td>
</tr>
<tr>
<td></td>
<td>/INTRLEAV</td>
<td>interleave traces and/or pad gathers</td>
</tr>
<tr>
<td></td>
<td>/NULENGTH</td>
<td>changes trace length</td>
</tr>
<tr>
<td></td>
<td>/OMIT</td>
<td>omit traces based on GCV ranges</td>
</tr>
<tr>
<td></td>
<td>/OMITTRAC</td>
<td>omit traces by GATHER/TRACE</td>
</tr>
<tr>
<td></td>
<td>/RMSAMP</td>
<td>print RMS-amplitudes for traces/sections</td>
</tr>
<tr>
<td>Processing</td>
<td>/AIRWVMUT</td>
<td>automatic air wave mute using RANGE</td>
</tr>
<tr>
<td></td>
<td>/AMPMATH</td>
<td>applies scalar math to trace amplitudes</td>
</tr>
<tr>
<td></td>
<td>/BULKSHFT</td>
<td>integral sample static bulk shifts</td>
</tr>
<tr>
<td></td>
<td>/CLIPIT</td>
<td>clips amplitudes above a specified threshold</td>
</tr>
<tr>
<td></td>
<td>/DIPFIL</td>
<td>time domain Butterworth dip filter</td>
</tr>
<tr>
<td></td>
<td>/EXCORTPR</td>
<td>data taper for extended correlation</td>
</tr>
<tr>
<td></td>
<td>/FILTER</td>
<td>simple FFT bandpass (four corners of trapezoid)</td>
</tr>
<tr>
<td></td>
<td>/OKAGC</td>
<td>more effective AGC than /AGC</td>
</tr>
<tr>
<td></td>
<td>/ROT</td>
<td>rotates three component data</td>
</tr>
<tr>
<td></td>
<td>/TRTAPER</td>
<td>Tapers end of zero parts (dropouts) of trace</td>
</tr>
<tr>
<td></td>
<td>/XCORR</td>
<td>simple X-corr using VECLIB</td>
</tr>
</tbody>
</table>

Modified SIERRASEIS

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job Utility</td>
<td>/INOPER</td>
<td>* /IN with console response</td>
</tr>
<tr>
<td></td>
<td>/OUTOPER</td>
<td>* /OUT with console response</td>
</tr>
<tr>
<td></td>
<td>/VAXIN</td>
<td>/IN with VAX header byte swapping</td>
</tr>
<tr>
<td>Data Utility</td>
<td>/PEAKVAL</td>
<td>prints trace peak amplitudes</td>
</tr>
<tr>
<td>Processing</td>
<td>/FIGURE</td>
<td>/DISPLAY for figures (no side labels)</td>
</tr>
</tbody>
</table>

Note: Column U denotes the processor interacts with an external, stand-alone utility program.
These user-choice parameters are defined within the initialization subroutine using SIERRASEIS utility subroutines DRTABL and DRPRMS. DRTABL defines valid parameter names and what types of and how many values can be given (i.e., integer or real values, character strings). DRPRMS will retrieve parameter names and values from a job listing. The software code within the initialization subroutine must be structured to interpret the values which are retrieved. The subroutine also must store these parameters using utility subroutines (DRSAVE) in order to transfer them to the execution subroutine.

Example of a multiple-trace processor: /EXMULTRC

For algorithms which require more than one seismogram, multiple-trace data flow is required within the processor's execution subroutine in order to temporarily store, process, and release the seismograms. This internal trace flow is required since the multiple-trace behavior must be compatible with the overall trace-by-trace data handling scheme. EXMULTRC presents within its execution subroutine the programming code required to make the multiple-trace handling work.

The accumulation/release aspect of multiple-trace flow is described in section II-A. The processor must use global common variables KSTATE and KCMHCK to regulate when to return for more data or when to release data. /EXMULTRC is designed so that one can use the structural framework of the execution subroutine, only needing to insert the multiple-trace algorithm.

Job monitoring: /MONITOR

The provisions within SIERRASEIS to monitor the progress of a job are to either use the '/PRMODCOM' processor or to estimate the status from the size of run-time output files (growth of .EPR or .RAS files). The MONITOR option of the '/PRMODCOM' processor will send the values of specified GCV's to the monitor from which the job was run. One can limit the number of monitor prints by using the trace flow INCLUDE limiters; however, there is no way to turn off this behavior once the job is started.

The progress of a job can be judged by the growth of .EPR or .RAS files. With experience, a user can estimate for the processors in the job listing what size output to expect for either the run-time execution phase printout (.EPR file) or for plot raster files (.RAS). These estimates are approximate.

The /MONITOR processor allows a user to selectively examine the progress of a job. When using this processor, a user specifies two GCV's in which to identify seismic data; for example, shot gather and trace number (KSHOT and KTRC), or CDP gather and offset (KCDP and RANGE). The processor will update the active trace's values of these GCV's in a local temporary file.

A stand-alone utility program named monitor will examine the GCV values in the local file. When an updated set of values are detected, monitor will display to the terminal screen the GCV values. This program will continue to display the GCV values of each active trace passing through /MONITOR until either the job has completed or the user interrupts or kills the monitor process.

The default local filename for both /MONITOR and monitor is 'MONITOR.TMP'. The following lines will use this file:

<table>
<thead>
<tr>
<th>job listing: /MONITOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>GATHER 'KSHOT' TRACE 'KTRC'</td>
</tr>
<tr>
<td>external: (% monitor</td>
</tr>
</tbody>
</table>

One can select a different local filename if 'MONITOR.TMP' is already in use:

<table>
<thead>
<tr>
<th>job listing: /MONITOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILENAME 'MONIT.local'</td>
</tr>
<tr>
<td>GATHER 'KSHOT' TRACE 'KTRC'</td>
</tr>
</tbody>
</table>
In this case, the user specifies the filename as a command line argument for `monitor`. Note that for SIERRASEIS/IRIS-SEIS, the filenames can use lower case characters.

**Number of data seismograms passing through trace flow:** /NUMTRFLO

There are times when the knowledge of how many seismic traces will pass through a specific point in a processing job's trace flow structure is useful. For example, in order to efficiently store a set of CDP gathers onto disk, one may wish to know how many total traces are stored in the range of CDP gathers.

Processor `/NUMTRFLO` will count the number of traces which flow through the processor. The total number will be printed into the .EPR file at the end of the processing job.

**Import and export of binary data files:** /UNIXIO

The need to transport data in and out of SIERRASEIS arises quickly. Intermediate processing steps which exist as stand-alone programs may be applied - one must be able to export SIERRASEIS data, then import the externally processed results. Other data which one may wish to process with SIERRASEIS may have originated outside of the package and may not be readily available in SEGY format. IRIS-SEIS provides a disk I/O processor named /UNIXIO which allows for the import and export of binary data disk files.

/UNIXIO is designed to read or write seismograms stored in either unformatted 32-bit floating point or 16-bit integers. The processor will use data written such that the last sample of a seismogram is followed immediately by the first sample of the next seismogram. This format is compatible with C-language write statements; FORTRAN unformatted write statements often carry 4-byte record headers which are not compatible with this processor.

UNIXIO will read seismograms of uniform trace length (KNSAMP, the number of samples per seismogram). Trace headers (GCV's) will be created so that proper trace flow can take place.

As an additional option in /UNIXIO, the trace headers (GCV's) for each trace can be stored. In this form, user-defined headers will also be kept for future processing. External programs which will act upon disk files which contain trace header information will need to skip both a file header and each of the trace headers in order to access the seismograms. A brief description of this file format is described in the manual pages for /UNIXIO.

Large blocks of continuously recorded data can be easily viewed with IRIS-SEIS by using /UNIXIO. This processor will read the data, breaking the continuous data into shorter, uniform length segments which can be plotted. The net display will be akin to a drum recording display where the end of one seismogram wraps around to continue with the beginning of the next seismogram.

By convention, data output from UNIXIO with or without GCV's is stored in ".UIO" format. This format name is used to differentiate between this format and the format used in SIERRASEIS ".DIO" files. The two formats are not compatible. The ".UIO" suffix is not automatically appended to disk file names but must be explicitly stated using the FILENAME parameter. Although any (or no) suffix can be used, the convention of using the ".UIO" suffix helps keep track of data files in this format.

**Trace header manipulation processors**

Definition and manipulation of dynamic trace headers (GCV's): /DEFHEAD, /HEADEQL, /HEADLIST, /HDMINMAX

The current version of SIERRASEIS (v1.3) has no provision for user definition of new trace headers (GCV's). IRIS-SEIS takes advantage of portions of the structural framework of SIERRASEIS to allow for this capability. New GCV's can be defined either within
processing subroutines or by using a processor written to define new headers.

The processor `/DEFHEAD` will allow a user to define a new GCV during the course of a processing job. The new GCV can be designated to hold either an integer or floating point value or can contain a character string of four or eight characters. The contents of the header can be filled by using SIERRASEIS’s `/PRMDCOM` processor or with `/HEADEQL`. This latter routine is used to set a GCV value based on another GCV; scalar math can be applied to the incoming GCV value.

Similar to the `/PRMDCOM` routine in SIERRASEIS, the processor `/HEADLIST` can be used to print the values of up to six GCV’s for each trace which passes into this processor. `/HDMINMAX` will examine the values of a specified GCV for all traces which pass through the processor, printing at the completion of the job the minimum and maximum values which were encountered.

All the above IRIS-SEIS processors will work with original SIERRASEIS GCV’s or with user-defined routines. Certain other SIERRASEIS processors will accept user-defined routines; however most SIERRASEIS processors will not recognize the new ones. For example, '/GATHER' will not sort on any header; the '/AUX' and INCLUDE systems won’t accept new names either. Certain SIERRASEIS routines can be made to accept the new GCV’s by equating the values of the new GCV with acceptable SIERRASEIS GCV’s. For example, either of the GCV’s KUSRCM and RUSRCM within SIERRASEIS can be used as a 'user GCV'. The contents of these GCV’s can be set equal to a user-defined GCV (using `/HEADEQL`, for example). The INCLUDE system will recognize the KUSRCM/RUSRCM header. See the /IN, /AUX, and INCLUDE processors in the SIERRASEIS BASIC manual.

The tape and disk I/O routines do not have a provision for the explicit storage of user-defined headers. One can bypass this by placing user-defined values into GCV’s which '/OUT' or '/DOUT' will write to tape/disk. The user must be careful in this case not to overwrite GCV values which may be needed in subsequent runs. The values can be recovered in later jobs by redefining the new headers and extracting their values from the transfer SIERRASEIS GCV’s. The UNIX binary disk file I/O capability provided by IRIS-SEIS (/UNIXIO) will allow for the preservation of the user-defined headers (see below).

Any GCV header (SIERRASEIS or user-defined) can be accessed within subroutines. A programmer can make provisions for run-time parameter selection of GCV’s using subroutines available within the IRIS-SEIS library. Specifically, subroutine `irisDEFHED.F` is used to define a GCV. Subroutine `irisGETHED.F` will obtain the contents of a particular GCV. These subroutines can be found in `.../iris13/seismic`.

**Calculate floating datum statics: /FLOTDAT**

The statics algorithms provided in the '/GEOMETRY' do not have a provision for floating datum statics. `/FLOTDAT` will compute static elevation corrections based on the elevation of the CDP location:

\[
STAT_{rec} = (ELEV_{CDP} - ELEV_{rec}) / V_{datum}
\]

and

\[
STAT_{shot} = (ELEV_{CDP} - ELEV_{shot}) / V_{datum}
\]

The net static is the sum of the receiver and elevation statics. The static shift is applied using the '/STATIC' processor.

The elevation of the CDP location is stored in global common variable KCDPDT. The receiver elevation is in KGPEL, the shot elevation is in KSPEL.

Other schemes for floating datum statics exist and can be easily implemented. One should examine the set of geometry GCV’s which are available (see Appendix 2 of the SIERRASEIS Basic manual).
Header fix of irregular data incorrectly counted by IN: GATHCNTR, TRCOUNTR, INTRLEAV

The 'IN' processor was written to accept uniform data; that is, an integral number of gather containing a regular number of traces per gather. When data is read in using this processor (from SEGY tapes, for example), the gather and trace numbers are incremented by counters. For uniform data (i.e., roll-along multichannel reflection data), this is not a problem. When irregular data or a partial gather is entered, however, the counters do not update properly. As a result, all subsequent gather/trace numbers will not identify the correct seismograms.

Within the DATA parameter in the 'IN' processor, one specifies the number of seismograms per input gather. This value, GCV 'KNTR', is used to determine the dimensionality of the gathers. If the data is read by 'IN' in 'shot gather' mode (data types 1 or 5), then all gathers are expected to have KNTR traces per gather. Rather than have the counting keyed on a pre-defined GCV (KFLDFN, the field file number, for instance), the traces are simply counted (1 to KNTR; next gather, 1 to KNTR...). The GCV counters, KSHOT and KTRC, as a result are quite uniform.

The erroneous counting will mis-identify traces, leading to erroneous trace flow and processing. As an example, data consisting of 3 shots will be examined, with 6 traces per gather (KNTR=6) with the second gather incomplete, having only three traces (traces 4, 5, and 6 are physically missing):

<table>
<thead>
<tr>
<th>incoming tape</th>
<th>tape KFLDFN</th>
<th>tape KFLDTN</th>
<th>actual KSHOT</th>
<th>actual KTRC</th>
<th>counted KSHOT</th>
<th>counted KTRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>101</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>101</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>101</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>101</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>101</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>102</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>102</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>102</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>103</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>103</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>103</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>103</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>103</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>103</td>
<td>6</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Because the 'IN' processor will count traces, it will not see that a partial gather was encountered. In this example, the trace flow for subsequent processors will involve two full gathers followed by one incomplete gather rather than the correct pattern of full, partial, and full gathers. This can be a major mistake, particularly if geometry is applied to these misnumbered traces.

If this incorrect numbering is encountered, the IRIS-SEIS processors GATHCNTR and TRCOUNTR can be used to properly renumber the traces. GATHCNTR will renumber the gather number KSHOT, keying on changes in the field file number KFLDFN. TRCOUNTR will resequence the trace numbers within a gather (KTRC for KSHOT, for example), restarting at one when a new gather number is encountered.

In the above example, GATHCNTR will resequence the KSHOT numbers based on values of KFLDFN. When the KFLDFN number changes, so will the KSHOT number (i.e., at incoming traces 7 and 10 when the KSHOT number changes to 2 and 3, respectively).
Based on the updated KSHOT gather numbering, /TRCOUNTR will resequence the trace numbers (KTRC in this case). Since after /GATHCNTR, KSHOT switches from values of 2 to 3 at incoming trace 10, the trace numbering for KTRC will restart at one. Incoming traces 10 through 15 will have, as a result, KTRC numbers of 1 through 6. The net result of using these two processors is to have properly indexed (KSHOT, KTRC) data. The subsequent processors will apply the correct trace flow.

Irregularly sized gathers can be made uniform using the processor /INTRLEAV. This processor can detect when partial gathers are encountered and will pad the gathers so that a full number of traces exist for each gather. The resulting data will be uniformly-dimensioned data.

Seismic data utility processors

Print seismogram amplitudes: /AMPDUMP

Similar to SIERRASEIS processor /TRDUMP, processor /AMPDUMP will print seismogram sample amplitudes. The amplitudes can be printed in FORTRAN write() formats F15.5 (floating point), I15 (integer), or E15.7 (exponential). The traces are identified by using the values of two GCV's; one can specify any two headers.

Change trace length (number of trace samples): /NULENGTH

One can easily change the number of samples per seismogram by changing the GCV value of KNSAMP, the number of samples per seismogram. /NULENGTH will change the KNSAMP value for both the initialization and execution stages. If the new trace length is shorter than that for the incoming trace, the trace is simply truncated. If the new trace length is greater than the incoming trace, the extra, trailing samples are zeroed.

The use of /NULENGTH is equivalent to the double use of /PRMODCOM:

/NULENGTH NEWTIME 6000.

is the same as

/PRMODCOM MODIFY KNSAMP SET 1501
/PRMODCOM MODIFY KNSAMP SET 1501 INITPR

This example assumes the sample rate is 4 msec. The first call to /PRMODCOM will reset the GCV KNSAMP during the execution phase. The behavior of /PRMODCOM requires that the call be repeated for the initialization phase.

Physical omission of data traces: /OMIT, /OMITTRAC

A data set’s uniform dimensionality is a characteristic which SIERRASEIS handles easily. A number of gathers with a fixed number of traces per gather can be entered and processed by SIERRASEIS with no difficulty. Irregular data can present trace flow problems unless attention is paid to the trace labeling as defined by trace header GCV's (see section on irregular data above).

Due to a preference to preserve data uniformity, when traces are edited (zeroed for example), GCV headers are set to indicate processors should skip the traces. The traces are still physically present, however, and occupy positions within the trace flow.

IRIS-SEIS provides processors which will physically remove traces from the trace flow. These processors, /OMIT and /OMITTRAC will reduce the number of traces which will enter into the processors which follow. These routines should be used carefully as they change the uniformity of data dimensions (number of gathers by number of traces per gather). Certain hidden GCV values which describe the total number of traces in a data set are not updated to the new value (as either processor does not know a priori how many traces will actually be removed; this is a function of exactly which traces are input into the job stream). The use of either of these processors can reduce the amount of time needed to run a job by
reducing the amount of traces which pass through the trace flow.

The processor /OMIT can be used to remove blocks of data. The blocks of data can be defined as all seismograms that have a certain GCV value or that fall within a range of GCV values. For individual trace referencing, the processor /OMITTRAC should be used. This latter processor uses two GCV's to identify traces to drop.

Print seismogram RMS amplitudes: /RMSAMP

The processor /RMSAMP will compute the RMS-amplitude for each trace and will print the running RMS-amplitude for all traces up to the current trace. RMS-amplitudes are computed using either all (positive and negative) amplitudes or just positive amplitudes. Traces are identified using any two GCV's. As an added option, the user can print absolute amplitudes rather than RMS-amplitudes.

Seismic data processing processors

Automatic air wave mute: /AIRWVMUT

The processor /AIRWVMUT will apply a surgical mute to remove air waves. The processor determines the zone to mute based on the source-receiver offset (GCV 'RANGE') and the speed of sound in air. GCV RANGE values must be defined either by applying geometry or by explicitly defining RANGE. The width of the air wave mute can be set as run-time parameters.

Apply scalar math to seismogram amplitudes: /AMPMATH

/AMPMATH will modify seismogram amplitudes by applying scalar math to each sample amplitude. The scalar math is applied as:

\[ \text{OUT}(t) = (\text{MULT}_1 \times \text{IN}(t) + \text{ADD}_1) \times \text{MULT}_2 + \text{ADD}_2 \]

Each trace amplitude is first multiplied by \( \text{MULT}_1 \), then added to \( \text{ADD}_1 \). The net result is then multiplied by \( \text{MULT}_2 \) and added to \( \text{ADD}_2 \).

Optionally, the scalar math can be applied as:

\[ \text{OUT}(t) = (\text{IN}(t) \div \text{DIV}_1 + \text{ADD}_1) \div \text{DIV}_2 + \text{ADD}_2 \]

Here, each trace amplitude is first divided by \( \text{DIV}_1 \), then added to \( \text{ADD}_1 \). The net result is then divided by \( \text{DIV}_2 \) and added to \( \text{ADD}_2 \).

The default values of the scalar coefficients are such that if no coefficients are modified, the output trace is simply the input trace.

Integral sample static shifts: /BULKSHFT

/BULKSHFT will apply static shifts to data traces involving whole (integral) sample shifts. By using only integral sample shifts, the statics can be applied quickly as no amplitude interpolation needs to be applied.

Reset excessive amplitudes to a user-specified level: /CLIPIT

Similar to a despiking routine, /CLIPIT will detect amplitudes above a user-specified threshold. Amplitudes beyond this threshold are reset to the threshold value.

Spatial-domain Butterworth dipfiltering: /DIPFIL

/DIPFIL will apply an (x-t) domain Butterworth dip filter to data gathers. The algorithm is based on an algorithm by Hale and Claerbout (GEOPHYSICS, v. 48, 1033-1038, 1985). The symmetric-dip filter can preserve or reject dips within the specified dip range.
Prepare uncorrelated traces for extended correlation: /EXCORTPR

Uncorrelated seismic traces can be prepared for extended correlation using /EXCORTPR. This processor will cosine-taper the end of the original data, then zero pad the data to a length necessary to produce the desired extend-correlated data. The cross-correlation can be performed by the SIERRASEIS processor ’/VCORR’.

Simple frequency-domain bandpass filtering: /FILTER

/FILTER applies frequency-domain bandpass filtering to seismic data. This processor transforms each seismogram into the frequency domain to apply a box-car frequency filter. User parameters are the four corners (0%, 100%, 100%, 0%) to define the box-car. The ends of the filters (the 0-100% rise and 100-0% drop) are cosine-tapered. This routine is easier to use than ’/STVF’ but is not as quick.

An alternative automatic gain control processor: /OKAGC

The ’/AGC’ processor is sometimes not as ruthless in its amplitude balancing as might be expected. For these situations, the user can apply /OKAGC. This alternative AGC processor will quite strongly balance amplitudes within and across seismic traces.

Three component trace rotations: /ROT

Sets of three component data can be rotated into principal orthogonal directions using /ROT. The rotation is specified as user parameters. Data traces must be stored in triplets (the three components). Output traces are the three rotated seismograms.

Automatic taper of amplitude dropouts: /TRTAPER

Amplitude dropouts sometime exist within seismic data. These dropouts can cause adverse processing artifacts, particularly if they occur in uncorrelated seismic data. /TRTAPER is designed to scan trace amplitudes to search for amplitude dropouts. When detected, the ends of the dropouts are cosine-tapered to create more gentle transitions into the dropouts.

Cross Correlation using an external sweep: /XCORR

/XCORR will conduct cross-correlation of seismic traces using an external correlation operator. The external sweep is obtained from a disk file which should contain only the sweep amplitudes. All data passed into the /XCORR will be cross-correlated.

Modifications of SIERRASEIS processors

Two important processors here are the modified versions of ’/IN’ and ’/OUT’ which allow for external tape mount requests and replies. These processors, /INOPER and /OUTOPER are described below.

External tape mount requests and responses: /INOPER, /OUTOPER

When data is to be read from or written to tape, the ’/IN’ and ’/OUT’ processors within SIERRASEIS issue tape mount requests. These requests are sent directly to the console or monitor from where the job was run; the responses also must come from that monitor. This behavior is not a problem when the job is run from a terminal designated for (often long) processing jobs.

This behavior is not useful under certain conditions. When the tape drives are not located in the same room as the monitor, one must return to the monitor after mounting the tapes; to verify the tape mount (i.e., see the tape moving) or to fix an incorrect mount (i.e., to re-hit the on-line button), the user must go back to the room with the tape drives. To be able to reply to the tape request at a monitor near the tape drives (often the system console) would be useful.
If the monitor from where the job was started does not have multiple-window capability, the job essentially ties up the monitor for the duration of the job. Having the job in 'background' when a tape request arises will force the job to abort. An independent tape request system would not occupy the monitor, freeing it for other work. In certain system configurations, a login process will be terminated if the process is idle for a specified amount of time. Under this situation, the interactive nature of the processing job will be interrupted, causing the job to be lost during logout. Again, an independent tape request system would circumvent this problem.

IRIS-SEIS provides an external tape mount request and reply system which is directly compatible with the SIEERRASEIS framework. Using modified versions of the '/IN' and '/OUT' processors, IRIS-SEIS will send the tape requests not to the monitor from where the job was run but to a tape monitor file. This file will store both tape mount requests and tape mount replies. The modified I/O processors, named /INOPER and /OUTOPER, will pause upon issuance of a tape mount request to the tape monitor file. Upon sensing that a reply was returned into the file, the processors will interpret the reply and continue. The file requests and responses are identical to those which normally are sent to the monitor.

A stand-alone utility program tapeoper is used to examine the tape monitor file. This utility program will identify each request with a request number and will display this information. After mounting the appropriate tape, one can reply to the request using the utility program. The responses are given using the request number in order to identify the job which sent the request (several requests can be present at the same time).

The benefits of this approach are several. The user may respond to a tape mount request from not only the monitor from where the job was run but from any terminal. Since the responses are external to the processing job, the job can be run in UNIX 'background' or as a batch job. Monitor timeouts will not affect the ability to respond to the job nor will they abort the job. Other users (i.e., a tape operator) can reply to the tape requests. Since a job will pause until its tape request is met, one can share tape drives between concurrent jobs by alternating the jobs to which tape replies are made.

The processors /INOPER and /OUTOPER behave exactly in the same way as '/IN' and '/OUT' except for the tape requests/responses. The data selection (BI/BI parameters) and format translation (SEGY/HOUSE) behavior are identical. The processors are essentially interchangeable (except for the tape requests/responses); a tape written with /OUTOPER can be read back in with '/IN'.

Print trace peak amplitudes: /PEAKVAL

/PEAKVAL is similar to either /AMPDUMP or '/TRDUMP' in that it will display amplitudes contained in seismic traces. /PEAKVAL will print only peak (minimum and maximum) amplitudes.

Raster plots for figures: /FIGURE

The '/DISPLAY' processor can be used to generate raster plots to be used for either slide or journal figures. These plots often need to be reduced or fit into specified dimensions (i.e., 35 mm frame). For such reductions, having extra blank paper on the sides of the plot help in the photographic reduction by providing a uniform background on all sides. Plots created by '/DISPLAY' have a text header plotted on the left side of the plot.

/FIGURE is a modification of '/DISPLAY' which will not print the text header. Rather than cut the header off a plot created by '/DISPLAY', the raster file can be plotted by using /FIGURE. The plot parameters are the same for the two processors.
III-D. Stand-alone Utility programs provided within IRIS-SEIS

Stand-alone utility programs supplied with IRIS-SEIS can be divided into four groups of functionality: external tape request/response, monitoring of run-time progress of a job, examination of trace header of data values for exported binary disk files, and raster file manipulation.

A brief summary of the available utility programs follows. An explanation on how to use each program follows a description of its functionality.

Stand-alone Utility Programs

<table>
<thead>
<tr>
<th>External tape request/response:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tapeoper</strong></td>
</tr>
<tr>
<td><strong>tapeoperbells</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Job monitor:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>monitor</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary format disk files (.UIO format) and GCV values:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GCVminmaxI</strong></td>
</tr>
<tr>
<td><strong>GCVminmaxR</strong></td>
</tr>
<tr>
<td><strong>UIOdump</strong></td>
</tr>
<tr>
<td><strong>UIOlength</strong></td>
</tr>
<tr>
<td><strong>UIOminmax</strong></td>
</tr>
<tr>
<td><strong>UIOtruncate</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Raster file manipulation:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rscat</strong></td>
</tr>
<tr>
<td><strong>rasdump</strong></td>
</tr>
</tbody>
</table>

External tape request/response: tapeoper

The external tape request and response system allows users to receive and respond to tape request messages at monitors or consoles other than the one at which the processing job was begun (see section III-C). While inside a processing job, one must use the /INOPER or /OUTOPER processors in place of '/IN' or '/OUT'. These IRIS-SEIS processors communicate with the external program tapeoper to allow for external response to tape requests. tapeoper allows anyone to see what tape requests are still waiting for some action. Depending on tape allocation permissions, anyone can respond to the tape requests.

In order to use the tapeoper approach to tape requests, one must use /INOPER or /OUTOPER instead of '/IN' or '/OUT'. All uses of parameters are identical for /INOPER-'/IN' and /OUTPER-'/OUT'. Example:

/JO
/OUTOPER
FORMAT SEGY DENSITY HY
When the job is run through IRIS-SEIS, tape request messages will not appear on the terminal at which the job was run. Rather, messages must be retrieved by using the stand-alone program **tapeoper**. When this program is invoked, all active tape messages/requests will be displayed. At this point, one may mount a tape and respond to the appropriate request.

**tapeoper** will display communication messages which normally would be sent to the terminal. Each message will have an I.D. number along with the date, time, job sequence number, request status, and the text of the message. An example of information given by **tapeoper** is:

```
  ID#  DATE       TIME  KSEQ  STAT    MESSAGE
   ---  -----      -----  -----  -----    -----  
      502  1-APR-90  02:31:30  CPY   MESG  SierraSeis Input tape request
```

The message "SierraSeis Input tape request" was issued at 2:31:30 on April 1, 1990. The message I.D. number is 502; this number is important when a message is to be replied to. The job sequence number is "CPY", which implies that, when the job was run, the letters "CPY" was used by **irisexec** to identify the communication file created by **irisinit**. The STAT ("STATUS") column here shows that the text message is indeed just a message. If the message was an actual request that needed a response, the STAT value would be "REQU". Other possible states are "PEND" for pending (a request was replied to and is in the process of being enacted) and "DONE" for done (a request was successfully replied to).

To reply to a tape request from **INOPER**, the user must provide the tape drive and tape density as answers to two successive queries. The initial request for the tape drive may look like:

```
  ID#  DATE       TIME  KSEQ  STAT    MESSAGE
   ---  -----      -----  -----  -----    -----  
      502  1-APR-90  02:31:30  CPY   MESG  SierraSeis Input Tape Request
      503  1-APR-90  02:31:30  CPY   MESG  Please ready tape number DAT2 for Input
      504  1-APR-90  02:31:31  CPY   REQU  Enter tape drive [/dev/rmt8]
```

The subsequent request for the tape drive density will look like:

```
  505  1-APR-90  02:31:49  CPY  REQU  Enter tape density [1600].
```

Each time the current messages are displayed on the screen, a prompt follows to ask what the user wishes to do. He or she may list the messages again, reply to a request, delete a line from the list, clear all lines from the screen, get on-line help, or quit the program. The two-line prompt will look like:

**List Reply Del Clear Help Quit:**

```
command, [i5], [a8]>
```

and indicates that the command has been entered, and optionally a message I.D. number [i5 format] and a short text string [a8 format]. The command can be the full command or can be the first letter [in upper or lower case].

<table>
<thead>
<tr>
<th>Command</th>
<th>Input</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>L or l</td>
<td>List current messages.</td>
</tr>
<tr>
<td></td>
<td>L, I.D.#</td>
<td>List from I.D.# to end.</td>
</tr>
<tr>
<td></td>
<td>l, I.D.#</td>
<td></td>
</tr>
<tr>
<td>Reply</td>
<td>R, I.D.#, text</td>
<td>Reply to a request with text string.</td>
</tr>
<tr>
<td></td>
<td>r, I.D.#, text</td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>D, I.D.#</td>
<td>Delete a message or request from current list.</td>
</tr>
<tr>
<td></td>
<td>d, I.D.#</td>
<td></td>
</tr>
<tr>
<td>Clear</td>
<td>C c</td>
<td>Clear all messages.</td>
</tr>
<tr>
<td></td>
<td>C, I.D.#</td>
<td>Clear all messages from start to I.D.#.</td>
</tr>
<tr>
<td></td>
<td>c, I.D.#</td>
<td></td>
</tr>
</tbody>
</table>
The LIST command will list all messages which are current. The list will always give all messages which are stored, from first stored to most recent, unless a subset is asked for.

The range of messages can be limited by specifying a message I.D. number; only those messages between this number and the most recent I.D. number will be printed. For example: 1,501 will print messages between #501 and the most recent.

To REPLY to a request, one must identify the request using the I.D.#, then provide the text answer. For example, the tape mount request as shown in the example above (request #504) can be given by r, 504, TAPE0. The answer TAPE0 is the same response as if the request was presented to the terminal; that is, in this case (UNIX operating systems) it is the symbolic link pointing to the appropriate tape drive. A request for the tape density could be answered with: r, 505, 6250.

When a reply text string is entered, the string is placed into a tape request history disk file which /INOPER and /OUTOPER uses to communicate requests. These IRIS-SEIS routines will retrieve the reply string, verify the response, and commence the I/O action. Within tapeoper, first a "PEND" status (reply is pending) will appear. If the response is valid, the status is switched to "DONE" (as found using a successive "LIST"). If the response is invalid, a new message will appear upon the next use of "LIST" with most likely a diagnostic message followed by a new request.

The tape request history disk file can contain only fifty lines of messages at any given time. Periodic purging of old messages should be conducted using the "CLEAR" command. Messages which are still openly requested or pending cannot be purged, however. These lines must be explicitly deleted prior to purging. The DELETE command will allow you to remove any line from the list of messages. After issuing a DELETE command, a verification prompt is received; the user must answer with an affirmative (Y or y) to indeed remove the line.

Once the status of all lines are either "MESG" or "DONE", the set of lines may be purged using the CLEAR command. If this command is selected with no I.D.# argument, then all lines are purged. If an I.D.# is provided, then only those lines from the first line up and including the given line number is purged.

The HELP command can be used to obtain a quick look at the input formats for these commands along with a short description of their functionality.

The QUIT command gracefully shuts down the tapeoper program. As an external disk file is used by this program, a proper closing of the file is needed in order to store the most current values to be contained within the file.

New external tape request/response messages: tapeoperbells

While waiting for a tape request to come up, the tapeoperbells can be used in foreground or background mode. This utility detects when new tape requests are logged. The monitor bell is rung until the program is interrupted. The user can then call tapeoper to examine the new tape request.

Execution is simple. In UNIX foreground:

```
(%) tapeoperbells
```

When the monitor bell (control-G) rings, interrupt the program. For background use, note the job PID number from the UNIX "ps" command. When the bell rings, kill the job using the UNIX "kill" command. Then use tapeoper to identify the new request.
Job monitor: monitor

The monitor program is used in conjunction with the IRIS-SEIS /MONITOR processor to examine the status of trace flow within a processing job. This functionality is similar to the use of the MONITOR option in '/PRMODCOM'. The latter option will print to the terminal screen values of selected GCV's for each seismic trace passing through the '/PRMODCOM' routine. Once started, however, this behavior cannot be turned off. The use of the /MONITOR processor with the stand-alone program monitor can be started and stopped as desired.

/MONITOR will place the values of two GCV's from the active seismic trace into a temporary file, updating the values as each new trace enters the processor. monitor will examine this file, printing the GCV values to the terminal screen each time a new set of values is detected. The stand-alone program will continue to display trace GCV values until it is interrupted.

By default, the temporary file is named MONITOR.TMP and will be located in the directory where the IRIS-SEIS job is run. A different temporary file name can be defined by using the "FILENAME" parameter within /MONITOR. When the monitor program is invoked, specify the same filename as a command line argument. For example, the IRIS-SEIS job description file may have the following call to /MONITOR:

```
/MONITOR
GATHER 'KCDP'
TRACE 'RANGE'
FILENAME 'MONIT.sort'
```

Since the temporary file that will be used is named "MONIT.sort", the user invokes monitor using this name:

```
(%) monitor MONIT.sort
```

In this example, the CDP and offset values for each trace that enters /MONITOR will be displayed for as long as monitor is executed.

UIO format files - Integer GCV minimum-maximum values: GCVminmaxI

The stand-alone utility program GCVminmaxI can be used to obtain the minimum/maximum values for up to four trace header (GCV) values within a disk file stored by /UNIXIO. The disk file must have the GCV headers stored ('WRITE' / 'USEGCV' options in /UNIXIO). While one could run an IRIS-SEIS job using the processor named /HDMINMAX, this utility program is faster to run.

The utility program is run using the following command line arguments:

```
(%) GCVminmaxI file nbypass ntraces 11 12 13 14 format.
```

file is the filename of the UIO-format datafile. nbypass is the number of traces to skip prior to computing minimum/maximum values. ntraces is the number of traces to use to compute the extrema. 11, 12, 13, and 14 are the four possible GCV headers in which the minimum/maximum values are computed (for each header). The values for these latter arguments are the index positions of the GCV's within the integer GCV common block SSCOM.INC. For example, the CDP number, "KCDP", is located at index 15 (the 15th integer GCV); the number of samples per trace, "KNSAMP", is located at index 63. The format argument is used to indicate whether the data trace values are stored in 32-bit floating point or 16-bit integers. format is 32 for real values and 16 for integers; the default value is 32.

As an example, the following line

```
(%) GCVminmaxI CDP7.UIO 0 200 15 26
```

will find the minimum/maximum values for GCV headers "KCDP" and "KFOLD" (GCV indices 15 and 26, respectively) for the first 200 traces in file CDP7.UIO. No traces are skipped at the beginning of the file. The data format is assumed to be 32-bit floating point.
UIO format files - Real GCV minimum-maximum values: GCVminmaxR

Similar to GCVminmaxI, GCVminmaxR can determine the minimum/maximum values for real trace headers (GCV values). The operation of GCVminmaxR is similar to that of its integer-format counterpart; the programs are run in the same manner.

As an example,

(%) GCVminmaxR CDP7.UIO 0 200 3 4 13 29 16 will find the minimum/maximum values for GCV headers "CDPX", "CDPY", "RANGE", and "STATIC" (real GCV indices 3, 4, 13, and 29, respectively) for the first 200 traces in file CDP7.UIO. No traces are skipped at the beginning of the file. The data format is explicitly stated to be 16-bit integers.

UIO format files - Dump key GCV values for requested traces: UIOdump

An easy way to identify the traces within a UIO-format file is with UIOdump. This program will list key identifying GCV values for each trace encountered. The headers whose values will be dumped are "KFLDFN", "KFLDTN", "KCDP", "KTRACE", "KSHOT", and "KTRC". Upon the initial open of the data file, the number of samples per trace ("KNSAMP") as stored in the trace headers will also be given.

The program is run using the following command line arguments:

(%) UIOdump filename nbypass ntraces format

where filename is the UIO-format data file, nbypass is the number of traces to initially skip, ntraces is the number of traces to read, and format is the data sample format. format is 32 if the data is 32-bit floating point samples and is 16 if the samples are 16-bit integers. format=32 is a default condition.

UIO format files - Data file format: UIOlengt

UIOlengt is a utility program which will give structural information regarding a UIO-format file. This information can be used to describe the dimensionality of the data file for either other utility programs or within IRIS-SEIS jobs. UIOlengt will provide information on number of samples per trace ("KNSAMP"), the total number of GCV's stored for the four format types (integer, real, character*8 and character*4), the number and names of any user-defined headers, and the total number of traces contained within the file.

UIOlengt is run with the following command line arguments:

(%) UIOlengt filename format

filename is the UIO-format data file. format is the format of data samples (=16 for 16-bit integers and =32 for 32-bit floating point). The default format value is 32; if the numbers are stored in 4-byte real values, no format argument needs to be given.

Upon execution, the number and names of headers are given along with the number of samples per trace. The program will determine the number of traces within the file by actually trying to read each trace. For large data files, there may be a pause between the initial listings of header and sample numbers and the number of traces within the file.

UIO format files - Trace amplitude minimum/maximum: UIOminmax

While GCVminmaxI and GCVminmaxR are stand-alone programs to determine header minimum/maximum values, UIOminmax can be used to determine amplitude extrema per seismic trace. This program can be used to determine amplitude ranges (for plot scaling, for example). Each trace is identified by printing six key header values along with the amplitude minimum and maximum. The six headers are "KFLDFN", "KFLDTN", "KCDP", "KTRACE", "KSHOT", and "KTRC".

UIOminmax is run in an identical manner as UIOdump:

(%) UIOminmax filename nbypass ntraces format

where filename is the UIO-format data file, nbypass is the number of traces to initially skip, ntraces is the number of traces to read, and format is the data sample format. format is 32 if the data is 32-bit floating point samples and is 16 if the samples are 16-bit integers.
format=32 is a default condition.

**UIO format files - Trace length truncation: UIOtruncate**

The length of data traces can be modified by running an IRIS-SEIS job using processors '/PRMODCOM' or /NULENGTH. For UIO-format data files, one can also accomplish this change using the utility program UIOtruncate. This stand-alone program will allow trace lengths to be shortened or lengthened (up to 20,000 samples/trace). This program runs faster than a SIERRA-SEIS or IRIS-SEIS job.

To modify trace lengths of UIO-format data, use:

```
(%) UIOtruncate infile outfile ntraces NewKNSAMP format
```

where `infile` is the incoming UIO-format file, `outfile` is the modified (output) file, `ntraces` is the number of traces to process, `NewKNSAMP` is the new number of samples per trace, and `format` is the data sample format (16 or 32; see above examples). One can obtain the original number of samples per trace or verify whether the output file is the proper size by using UIOlength.

**Raster file manipulation - Concatenate two .RAS files: rascat**

For large-width raster plotters, the act of plotting several narrow-width plots can consume a large amount of plotting paper. In order to save paper, the utility program rascat may be used. This program will "concatenate" two SIERRASEIS .RAS files into one larger .RAS file so that both original files will plot across the plotting paper rather than in consecutive manner. Several narrow plots can be "concatenated" into one mega-.RAS file with successive calls to rascat.

`rascat` will interpret the encoded form of two .RAS files in order to reconstruct the proper encoded form for the merged file. This is necessary to allow the raster driver (i.e., 'RASPLOT') to function properly. rascat can merge files of the same or variable physical dimensions. A small plot can be merged with a large plot; the resulting plot dimensions will always be large enough to encompass both plots.

To use `rascat`, one simply gives the two incoming plots and the output plot file name:

```
(%) rascat plot1.RAS plot2.RAS outplot.RAS
```

where `plot1.RAS` and `plot2.RAS` are the two incoming raster plots and `outplot.RAS` is the merged plot. To concatenate three or more plots together, successive calls to rascat must be executed where the output file from a previous call becomes one of the input files for the successive call. Keep in mind that when concatenating large raster files, the available disk space will diminish rapidly.

**Raster file manipulation - print descriptive parameters: rasdump**

`rasdump` is a utility program which will print descriptive parameters from a given SIERRASEIS raster file. For programming reasons, a raster file is structured in a more complex format than a pure raster bit map form of the desired plot. Rather, the raster bytes are divided into ~60K records which are written as sets of 512-byte blocks. Actual raster scans will map across these blocks.

The first 512-byte block of a .RAS file serves as a header block. The first 32 bytes are divided into eight 4-byte integers; `rasdump` will provide these values:

<table>
<thead>
<tr>
<th>bytes</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-4</td>
<td>Number raster records in plot</td>
</tr>
<tr>
<td>5-8</td>
<td>Number of bytes per record</td>
</tr>
<tr>
<td>9-12</td>
<td>Number of raster scans per record</td>
</tr>
<tr>
<td>13-16</td>
<td>Total number of raster scans in plot</td>
</tr>
<tr>
<td>17-20</td>
<td>Plot direction: 1=LTOR, 2=RTOL</td>
</tr>
<tr>
<td>21-24</td>
<td>Swap flag</td>
</tr>
<tr>
<td>25-28</td>
<td>Raster file type: 1=B&amp;W, 2=color</td>
</tr>
</tbody>
</table>
From these values, rasdump will compute the number of bytes per raster scan.

Programming Considerations: UIO-format files

Stand-alone programs can be encoded which are capable of reading or writing UIO-format files. These programs must properly account for the file and trace headers which are associated with the data traces. The stand-alone programs described above which manipulate UIO-format files use a library of subroutines which performs much of the header operations. This library is contained in the source code file UIOssubs. F in the ../iris13/utility directory. A description of the UIO format can be found in section IIIB and in the manual pages for /UNIXIO.

The library of UIO header operations include the following functionality:

<table>
<thead>
<tr>
<th>Subroutine name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>UIOreadhead</td>
<td>reads file header to obtain defined GCV's</td>
</tr>
<tr>
<td>UIOprinthead</td>
<td>prints to monitor screen GCV information</td>
</tr>
<tr>
<td>UIOcheck</td>
<td>checks if file I/O operation was successful</td>
</tr>
<tr>
<td>UIOwritehead</td>
<td>writes file header to output file</td>
</tr>
<tr>
<td>UIOgetKNSAMP</td>
<td>obtains KNSAMP from first seismic trace's GCV headers</td>
</tr>
<tr>
<td>UIOreadtrace</td>
<td>reads next seismic trace's headers and amplitudes</td>
</tr>
<tr>
<td>UIOgettrhead</td>
<td>performs actual trace amplitude read</td>
</tr>
<tr>
<td>UIOputtrace</td>
<td>performs actual trace header read</td>
</tr>
<tr>
<td>UIOwritetrace</td>
<td>writes next seismic trace's headers and amplitudes</td>
</tr>
<tr>
<td>UIOnuttrhead</td>
<td>performs actual trace amplitude write</td>
</tr>
</tbody>
</table>

The library of subroutines is written in FORTRAN but uses the FTOCIO library (APPENDIX B) to use C-language I/O for its file operations. As in C-language file I/O, one must bookkeep the current location of the file "pointer" in order to keep track of which portion of the file one is going to read from or write to.
IV - LOCAL-SEIS

LOCAL-SEIS is a platform where development of processors can take place. This platform provides subroutines and shell files which are needed to compile and link the new routines with SIERRASEIS and IRIS-SEIS. While a description of LOCAL-SEIS was presented in section II-C, the sections here present information on how to install, run jobs with, and add to LOCAL-SEIS.

IV-A. Installation of LOCAL-SEIS

This chapter presents a step-by-step installation procedure for LOCAL-SEIS. It is not necessary to use the "SIERRA" account used for installation of LOCAL-SEIS, however, write-permission is required within the main SIERRA directory (.../sierra).

The overall directory for LOCAL-SEIS is named local13 and exists at the same level as iris13 which contains IRIS-SEIS. Similar to IRIS-SEIS, the top-level directory which will contain both local13 and iris13 is defined as /usr/sierra in all LOCAL-SEIS installation and make files. If the sierra directory is located within /usr or a symbolic link sierra is created within /usr to point to the location of the sierra directory, no path name substitutions are needed. If the user had to change path name locations in make files during the installation of IRIS-SEIS, he or she is required to do the same for LOCAL-SEIS.

The installation procedures below discuss a quick and a full approach to installation of LOCAL-SEIS. The quick approach can be used because needed object code and executables are already provided in compiled/linked form within the distribution tar tape. These files can be used provided the host CPU is compatible with a SUN 4 (SPARC) with OS 4.0.3. If the host CPU is not compatible, then the full installation procedures must be used.

The following procedures are described with the assumption that IRIS-SEIS has been already installed (see section III-A).

Quick Installation of LOCAL-SEIS

To install LOCAL-SEIS using the quick steps, the user is required to perform the following steps:

1. Extract LOCAL-SEIS from the distribution tar tape,
2. either:
   - (A) if symbolic link /usr/sierra or sierra within /usr was created to point to the overall SIERRA directory, skip, or
   - (B) make full path name changes to the following files (#lines to change):
     - local13/install/makefile: three lines
     - local13/install/makefile.old: three lines
     - local13/install/compone: one line
3. Run shell local13/install/liball,
4. use the makefile to create executables localinit and localexec,
5. Fix symbolic links in .../sierra/bin13 if necessary.

(1) Extract LOCAL-SEIS from distribution tape

The initial installation procedure for IRIS-SEIS should have extracted the local13 directory from the distribution tape. The user should check to see if subdirectories exist within local13 and if various types of files exist within these subdirectories (see section II-C for a list of LOCAL-SEIS subdirectories and files).
If these files are not present, then they must be extracted from the distribution tape. Change the working directory to the overall SIERRASEIS directory (e.g., /.../sierra/).

Use the tar command to extract all files, e.g.;

(%). tar xvf /dev/rmt16 /local13

The proper tape drive device name must be used. When the tar is completed, an additional 8 Mbytes of files should be created.

(2) Directory pathnames: symbolic links or explicit definition

Several installation and update make/shell files reference object code or include files in order to properly build the LOCAL-SEIS executables. The correct locations of these files must be stated. The installation/make files have used the definition /usr/sierra as the directory which contains sseis13, iris13, and local13.

If IRIS-SEIS was installed using either the explicit path /usr/sierra or a symbolic link pointing to this path, then no action should be taken at this step. Either of these options which were used for IRIS-SEIS will automatically work for LOCAL-SEIS.

If the installation of IRIS-SEIS did not use the symbolic link or if the location of the overall sierra directory is not at /usr/sierra, then some directory paths must be given explicitly. Make the changes to the following files:

<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
</tr>
</thead>
<tbody>
<tr>
<td>local13/install/makefile</td>
<td>ROOT DIR = /usr/sierra/local13</td>
</tr>
<tr>
<td></td>
<td>RUN DIR = /usr/sierra/local13/run</td>
</tr>
<tr>
<td></td>
<td>LK DIR = /usr/sierra/local13/run</td>
</tr>
<tr>
<td>local13/install/makefile.old</td>
<td>ROOT DIR = /usr/sierra/local13</td>
</tr>
<tr>
<td></td>
<td>RUN DIR = /usr/sierra/local13/run</td>
</tr>
<tr>
<td></td>
<td>LK DIR = /usr/sierra/local13/run</td>
</tr>
</tbody>
</table>

All references to /usr/sierra should be modified to the true location of the user's sierra/ directory. When these changes are made, the new pathnames should point to the true locations of the directories or files.

No modification to file ../local13/install/compone should be made. The include flag used during the compilation of a LOCAL-SEIS subroutine is the same as that used for the compilation of an IRIS-SEIS subroutine within ../iris13/install/compone.

(3) Remake the LOCSEIS.a archive library

In order to remake ("archive" and "ranlib") the archive library of LOCAL-SEIS object code, the liball shell must be used. Change directory to the install directory:

(%). cd ../local13/install

Running the liball shell will create or update the archive library locseis.a.

(4) Create LOCAL-SEIS executables

Create the LOCAL-SEIS executables localinit and localexec using the makefile located in ../local13/install:

(%). make localinit

and

(%). make localexec.

The executables will be placed into ../local13/run.
(5) Fix symbolic links in .../sierra/bin13 if necessary

The directory bin13 is provided for convenience to point to the various directories where executables reside. Rather than place the several directories into the user's PATH environment variable or create aliases for each executable, one may place just the .../sierra/bin13 directory in the PATH environment variable list. Within this directory, symbolic links for all LOCAL-SEIS, IRIS-SEIS, and SIERRASEIS executables point to the proper executables. This directory simplifies management of executable locations and definitions.

The proper symbolic links should have been established during installation of IRIS-SEIS using the shell file .../sierra/bin13/sh.link. If symbolic links to localinit and localexec are missing, either rerun .../sierra/bin13/sh.link or define them explicitly:

(%) cd .../sierra/bin13
(%) ln -s /usr/sierra/local13/run/localinit localinit
and
(%) ln -s /usr/sierra/local13/run/localexec localexec

Use the correct path to point to .../local13/run (if not /usr/sierra).

Full Installation of LOCAL-SEIS

A full installation of LOCAL-SEIS includes several more steps beyond the quick installation:

1. extract LOCAL-SEIS from the distribution tar tape,
2. either:
   (A) if symbolic link /usr/sierra or sierra within /usr was created to point to the overall SIERRA directory, skip, or
   (B) make full pathname changes to the following files (#lines to change):
       -local13/install/makefile: three lines
       -local13/install/makefile.old: three lines
       -local13/install/compone: one line
3. compile all source code
4. create archive libraries using shell local13/install/liball,
5. run the 'make' option within the install shell in local13/install
6. run shell makelk in local13/install
7. use the makefile to create executables localinit and localexec,
8. fix symbolic links in .../sierra/bin13.

(1) Extract LOCAL-SEIS from distribution tape

The initial installation procedure for IRIS-SEIS should have extracted the local13 directory from the distribution tape. The programmer should check to see if subdirectories exist within local13 and if various types of files exist within these subdirectories (see section II-C for a list of LOCAL-SEIS subdirectories and files).

If these files are not present, then they must be extracted from the distribution tape. Change the working directory to the overall SIERRASEIS directory (e.g., .../sierra/).

Use the tar command to extract all files, e.g.;

(%) tar xvf /dev/rmt16 ./local13

Use the proper tape drive device name. When the tar is completed, an additional 8 Mbytes of files should be created.

(2) Directory pathnames: symbolic links or explicit definition

Several installation and update make/shell files reference object code or include files in order to properly build the LOCAL-SEIS executables. The correct locations of these files must be stated. The installation/make files have used the definition /usr/sierra as the
directory which contains sseis13, iris13, and local13.

If IRIS-SEIS was installed using either the explicit path /usr/sierra or a symbolic link pointing to this path, then no action should be taken at this step. Either of these options which were used for IRIS-SEIS will automatically work for LOCAL-SEIS.

If the installation of IRIS-SEIS did not use the symbolic link or if the location of the overall sierra directory is not at /usr/sierra, then some directory paths must be given explicitly. Make the changes to the following files:

<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
</tr>
</thead>
<tbody>
<tr>
<td>local13/install/makefile</td>
<td>ROOT_DIR = /usr/sierra/local13</td>
</tr>
<tr>
<td></td>
<td>RUN_DIR = /usr/sierra/local13/run</td>
</tr>
<tr>
<td></td>
<td>LK_DIR = /usr/sierra/local13/run</td>
</tr>
<tr>
<td>local13/install/makefile.old</td>
<td>ROOT_DIR = /usr/sierra/local13</td>
</tr>
<tr>
<td></td>
<td>RUN_DIR = /usr/sierra/local13/run</td>
</tr>
<tr>
<td></td>
<td>LK_DIR = /usr/sierra/local13/run</td>
</tr>
</tbody>
</table>

All references to /usr/sierra should be modified to the true location of user's sierra directory. When these changes are made, the new pathnames should point to the true locations of the directories or files.

The file ..../local13/install/compone needs no modification. The include flag used during the compilation of a LOCAL-SEIS subroutine is the same as that used for the compilation of an IRIS-SEIS subroutine within ..../iris13/install/compone.

(3) Recompile all source code

To recompile all LOCAL-SEIS source code, do the following:

(%) cd ..../local13/install
(%) compall

The compall shell will recompile all source code. There are not very many subroutines which exist within the installation version of LOCAL-SEIS, so the recompilation should be quick.

(4) Make the LOCSEIS.a archive library

In order to remake ("archive" and "ranlib") the archive library of LOCAL-SEIS object code, use the liball shell. Change directory to the install directory:

(%) cd ..../local13/install
(%) liball

Running the liball shell will create or update the archive library locseis.a.

(5) Run 'make' option in install shell

At this point, nearly all the necessary object code will exist. The missing subroutines are the lookup subroutines used by the LOCAL-SEIS drivers. These subroutines are created by running the stand-alone program locgntabl (as discussed in section II A-C). Before the creation of the lookup tables, the programmer should verify that the stand-alone executable exists. This can be done in two different ways.

The first approach is using the install shell. Within

(%) install

If the UNIX operating system command install appears, then the path is set so that the current directory is not searched first. This can be bypassed by renaming the install shell:

(%) mv install install.sh
(%) install.sh
When the screen menu appears within the install shell, select the 'make' option. The shell file will proceed to create the executable locgntabl.

The second approach may be easier. Within the local13/install directory, run:

(%) make locgntabl

(6) Create lookup subroutines: loclkinit.F, loclkexec.F

Now that the stand-alone executable exists from step (5), the lookup subroutines can be created. Within the local13/install directory, run the shell file makelk:

(%) makelk

This shell will run the stand-alone program locgntabl, using the text file ../local13/run/PROCLOC.LIS as input. The SIERRASEIS banner will flash on the screen as the executable is run.

From the execution of these programs, the output will be the lookup subroutines. These subroutines will also be compiled to produce object code. The source and object code of these routines will be placed into the ../local13/run directory.

(7) Create LOCAL-SEIS executables

Create the LOCAL-SEIS executables localinit and localexec using the makefile located in ../local13/install:

(%) make localinit
and
(%) make localexec.

The executables will be placed into ../local13/run.

(8) Fix symbolic links in ../bin13

The directory bin13 is provided for convenience to point to the various directories where executables reside. Rather than place the several directories into the user’s PATH environment variable or create aliases for each executable, one may place just the ../sierra/bin13 directory in the PATH environment variable list. Within this directory, symbolic links for all LOCAL-SEIS, IRIS-SEIS, and SIERRASEIS executables point to the proper executables. This directory simplifies management of executable locations and definitions.

The proper symbolic links should have been established during installation of IRIS-SEIS using the shellfile ../sierra/bin13/sh.link. If symbolic links to localinit and localexec are missing, either rerun ../sierra/bin13/sh.link or define them explicitly:

(%) cd ../sierra/bin13
(%) ln -s /usr/sierra/local13/run/localinit localinit
and
(%) ln -s /usr/sierra/local13/run/localexec localexec

Use the correct path to point to ../local13/run (if not /usr/sierra).

(NOTE): SGCONFIG.DAT entry

The installation of IRIS-SEIS resulted in an additional two lines being placed into the SGCONFIG.DAT file located in ../sierra. One of the two lines is relevant to LOCAL-SEIS, if the following line is missing, add it in:

DEFAULT_LOCERR /usr/sierra/local13/run/

The full directory path should point to the run directory for LOCAL-SEIS (i.e., may not be exactly as stated above). If this line is missing from the SGCONFIG.DAT file, the programmer should check the installation of IRIS-SEIS for completeness.
Customization of Executable Names

If the user wishes to rename localinit and localexec, then he or she should modify the makefile in local13/install. Near the bottom of the makefile are the compile/link commands to constrain the executables. For the "-o" option, the name desired for the initialization/execution phase executables is shown in the following example:

\[
\begin{align*}
\text{f77 localinit.o} & \quad \text{f77 localinit.o} \\
[\text{libraries and flags}] & \quad [\text{libraries and flags}] \\
-o \text{localinit} & \quad -o \text{uscinit}
\end{align*}
\]

Within this example, the version of localinit at the University of Southern California is named uscinit.

A more complete name change requires modifications of the LOCAL-SEIS name for .IPR and .EPR listings. These changes can be performed with modifications in the subroutines "loc_drinpt.F" and "loc_drxtim.F" in local13/seismic/fr.

Verification of Installation

The best test of a successful installation is to run a LOCAL-SEIS job. The executables can be tested in two parts. First, a job which calls only SIERRASEIS processors is used (a plot job, for example) and is known to run correctly within SIERRASEIS. Same results should be produced by using LOCAL-SEIS. Second, an IRIS-SEIS processor should be inserted into this job stream, then run with LOCAL-SEIS. If the processor is called correctly, then no run-time errors should occur.

The next section (IV-B) describes how to run LOCAL-SEIS jobs.
IV-B. Running a LOCAL-SEIS job.

The run-time environment of LOCAL-SEIS is very similar to those of SIERRASEIS and IRIS-SEIS. The procedures to run a seismic job within LOCAL-SEIS are the same as for SIERRASEIS and IRIS-SEIS. LOCAL-SEIS is downward-compatible; that is, a job which will run in SIERRASEIS or IRIS-SEIS will run in LOCAL-SEIS. However, a job which runs in LOCAL-SEIS may not necessarily run within SIERRASEIS or IRIS-SEIS.

To run a job within LOCAL-SEIS, the user follows the approach needed to run a job in SIERRASEIS:

(a) construct a job listing. The calls to processors and the list of processing parameters must be syntactically correct.
(b) check the job listing by using the initialization executable, localinit.
(c) run the job by using the execution executable, localexc.

The format of the job listing must conform to the rules set forth for SIERRASEIS job listings. LOCAL-SEIS processors can be called at any point within a job listing. These processors can be treated as if they were SIERRASEIS processors.

The executables localinit and localexc are run as if they were the executables ssinit and ssexec. localinit interprets a job listing and determines run-time conditions. localexc conducts the actual processing.

The trace-by-trace data handling scheme is adhered to by LOCAL-SEIS. This scheme is identical in behavior to that used by SIERRASEIS. The contents of trace headers (GCV's) should be understood in order to allow for proper trace flow to occur.

Summary of LOCAL-SEIS executables and shells:

<table>
<thead>
<tr>
<th>name</th>
<th>parameter entry</th>
<th>screen behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>localinit</td>
<td>interactive keyboard</td>
<td>banner clears screen and queries printed</td>
</tr>
<tr>
<td>localexc</td>
<td>interactive keyboard</td>
<td>banner clears screen and queries printed</td>
</tr>
<tr>
<td>LOCALCHECK</td>
<td>command line</td>
<td>banner clears screen and queries printed</td>
</tr>
<tr>
<td>LOCALRUN</td>
<td>command line</td>
<td>banner clears screen and queries printed</td>
</tr>
<tr>
<td>localcheck</td>
<td>command line</td>
<td>nothing to screen</td>
</tr>
<tr>
<td>localrun</td>
<td>command line</td>
<td>nothing to screen</td>
</tr>
</tbody>
</table>

Shells to simplify the initialization and execution steps: LOCALCHECK and LOCALRUN

The run-time behavior of the initialization driver executable localinit is based on interactive entry of job information. In this mode, the job listing name and job-execution parameters are entered via a console keyboard at run-time.

This behavior is not always optimum, particularly when a series of jobs to be run does not need monitoring. LOCAL-SEIS provides a few shell files which allow users to run a LOCAL-SEIS job using command-line entry of parameters and to run jobs from within shell file listings. These shells are similar to a set of shells available within IRIS-SEIS.

The normal commencement of a LOCAL-SEIS, IRIS-SEIS or SIERRASEIS job will require from the user four keyboard entries for the initialization step and (at least) two entries for the execution step. LOCAL-SEIS shell files LOCALCHECK and LOCALRUN are constructed to allow the user to run a job by providing command line parameters rather than to interactively enter parameters.

For example, the initialization step localinit can be replaced with the command LOCALCHECK:
By providing run-time information using command line parameters needed by the shell file \texttt{LOCALCHECK}, the interactive query system required by \texttt{localinit} can be bypassed. One also bypasses the need to enter the first \texttt{<return>} after the initial SIERRA-SEIS banner. This shell assumes that the user will wish to select default values for the version number and run-time options.

An additional advantage is that this command line approach can be easily placed into 'background' execution. This replaces the steps of interactive parameter entry followed by an interrupt and the issuance of the 'background' UNIX command:

\texttt{LOCALCHECK FILTER.DAT FILT &}

as opposed to the entire sequence of:

\texttt{(\%)} \texttt{localinit}  
\texttt{[SierraSeis banner]} \texttt{<return>}  
\texttt{Input file name: FILTER.DAT}  
\texttt{job sequence number: FILT}  
\texttt{version number: <return> [to default]}  
\texttt{run-time options: <return> [to default]}  
\texttt{Z [to interrupt]}  
\texttt{(\%)} \texttt{bg [to place in background]}

The execution phase can be run using a similar command-line shell:

\begin{tabular}{|c|c|c|}
\hline
\textbf{localexec}: & \textbf{query}: & \textbf{user response}: \\
\hline
(\%) & \texttt{localexec}  
SierraSeis banner & <return>  
job sequence number & CODE (e.g., FILT)  
version number & <return> to default \\
\hline
\textbf{LOCALRUN}: & \textbf{query}: & \textbf{user response}: \\
\hline
(\%) & \texttt{LOCALRUN FILT} \\
\hline
\end{tabular}

The commencement of a processing job is simple using the command-line approach. Again, the process can be easily placed into 'background' using this approach.

The use of \texttt{LOCALRUN} does not take subsequent interactive queries into account. For example, answers to tape drive requests are not placed on the command line. The \texttt{LOCALRUN} shell will not allow for tape requests and will most likely abort.

Two ways to circumvent the above problem for background execution exist. The first approach is to create a copy of \texttt{LOCALRUN} and insert the correct tape request responses into the shell file prior to running the job. Alternatively, one can use the external tape request option provided within IRIS-SEIS to remove all tape requests from requiring interactive keyboard response (explained in more detail in section III-C).
Two additional run-time shells: localcheck and localrun

The SIERRA-SEIS banner displayed at the beginning of a LOCAL-SEIS job eliminated whatever console screen information existed on the screen. This behavior can be bypassed by using the shell files localcheck and localrun. These shells are used in a manner identical to LOCALCHECK and LOCALRUN but do not send any information to the console screen. These shells redirect screen information to temporary files (which are removed upon completion of the job process).

For the above examples, the job listing FILTER.DAT can be run using the two lines:

(%) localcheck FILTER.DAT FILT
(%) localrun FILT &

The screen will not be erased by the SIERRA-SEIS banner (it is not necessary to wait for it to finish writing).

The shell files LOCALCHECK, LOCALRUN, localcheck, and localrun are located in the .../local13/run directory. Symbolic link pointers exist within /.../bin13 so that if this latter directory is already in the user’s PATH environment variable list (i.e. localinit or localexec can already be accessed), these shells could be used.
IV-C. Adding a New Processor to LOCAL-SEIS

The addition of a new processor to LOCAL-SEIS requires seven steps. These steps involve construction of the processor initialization and execution subroutines along with slight modifications to the structure of LOCAL-SEIS. Shell files are provided within .../local13/install to help implement the modifications.

Briefly, the steps needed to add a new processor are:

1. Select new, unused names for the processor and the associated initialization and execution subroutines.
2. Add these names to the master PROCLOC.LIS file.
4. Create the source code for the initialization and execution subroutines.
5. Compile the initialization and execution subroutines.
6. Remake the LOCSEIS.a archive library.
7. Remake the LOCAL-SEIS executables.

1. Select new names for processor and associated subroutines.

Since LOCAL-SEIS accesses IRIS-SEIS and SIERRASEIS processors, a new LOCAL-SEIS processor must have a name distinct from all other processors. The programmer must scan the files ../sseis13/run/PROC.LIS and ../iris13/run/PROCIRIS.LIS to see if the new LOCAL-SEIS processor name is already used.

Naming convention rules for SIERRASEIS require that a new processor name is limited to one to eight alphabetic characters. No numerals can be used; "\/*/TIMESTWO" is a valid processor name while "\/*/TIMES2" is not. [NOTE: the up to eight characters of a processor name do not include the preceding "/"; the "/" character is used here to differentiate between processor names and names of other files].

The selection of a name should reflect the functionality of the processor. Usually up to an eight character name can be determined which is descriptive of the function of the processor. Overly generic names should be avoided (e.g., "/ADD" for a new stacking routine). Cryptic names should also be avoided (e.g., "/CVTWODM" for "constant velocity f-k (two-dimensional fft) migration").

The initialization subroutines have six-character names. These characters can be alphabetic or numerical; in general they end with a "0". Alternatively, "IN" or "INI" can be used as the last two or three characters.

The execution subroutines also have up to six alphanumeric characters. SIERRASEIS execution subroutines generally end with "A", while IRIS-SEIS routines may end with "X", "EX", or "EXE", depending on the number of characters needed to make the name unique.

While the initialization and execution subroutines are limited to six characters, second-level subroutines (those called by these two subroutines) can have any number of characters. Because of the number of other subroutines used within SIERRASEIS, the user may wish to prefix or suffix any subsidiary subroutines. A feature within UNIX FORTRAN allows for subroutine names of variable length (e.g., IRIS_GETHED(), for example).

2. Add new names to master PROCLOC.LIS file

The new names of the processor and the initialization and execution subroutines should be entered into the PROCLOC.LIS file within .../local13/run. The initial contents of PROCLOC.LIS is:

```
processor: init name: exec name:
LOCAL FILES
NULLMOD NULLM0 NULLMX
(MODIFIED IRISFILES
```
The null processors have empty subroutines which exist within .../local13/seismic.

Suppose a new processor is /FFTPLOT. The initialization and execution subroutines are named "ftpl0" and "ftplx", respectively. The PROCLOC.LIS file will look like:

<table>
<thead>
<tr>
<th>processor:</th>
<th>init name:</th>
<th>exec name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(LOCAL FILES)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NULLMOD</td>
<td>NULLM0</td>
<td>NULLMX</td>
</tr>
<tr>
<td>FFTPLOT</td>
<td>FFTPLO</td>
<td>FFTPXL</td>
</tr>
<tr>
<td>(MODIFIED IRISFILES)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NULLIRIS</td>
<td>NULLI0</td>
<td>NULLIX</td>
</tr>
<tr>
<td>(MODIFIED SIERRA)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NULLSS</td>
<td>NULLS0</td>
<td>NULLSX</td>
</tr>
</tbody>
</table>


The newly modified PROCLOC.LIS file is used to create the lookup subroutines "loclkinit.F" and "loclkexec.F". The executable locgntabl is used to create these subroutines. The shell file makelk located in .../local13/install can be used to invoke locgntabl:

(%) cd .../local13/install.
(%) makelk.

The shell file will place the lookup subroutines into .../local13/run and will also compile the subroutines in order to create object code.

4. Create the source code for initialization and execution subroutines.

The initialization and execution subroutines should be written according to rules as will be discussed in sections IV-D and IV-E. The format and structure of these subroutines should try to follow the structure of SIERRASEIS subroutines as much as possible; this ensures compatibility. Remember that at least two subroutines are required for each new processor.

Once completed, the subroutines should be placed into directory .../local13/seismic.

5. Compile the initialization and execution subroutines.

The initialization and execution subroutines can be compiled using the shell compone located in .../local13/install. For example, to compile subroutines "exampO.F" and "exampx.F":

(%) cd .../local13/seismic
(%) ./install/compone examp0.F
(%) ./install/compone exampx.F

Note the relative directory definition of file "compone" compared to the location of the source code files. Two new object files, examp0.o and exampx.o, will appear.

6. Remake the LOCSEIS.a archive library.

Once the new object code exists, the archive library LOCSEIS.a needs to be updated:

(%) cd .../local13/install.
(%) liball

The liball shell will re-archive the object code within .../local13/seismic and "ranlib" the archive library.
7. Remake the LOCAL-SEIS executables.

The LOCAL-SEIS executables, localinit and localexec can be recompiled using the make shell:

\[ (%) \text{ make localinit} \]

and

\[ (%) \text{ make localexec}. \]

The executables should be properly linked and constructed; they should appear in the \texttt{../local13/run} directory.

If an object code file is missing, return to the \texttt{../local13/seismic} directory to see if it was compiled correctly.

Revising a Processor in LOCAL-SEIS

If an error occurs in one of the subroutines related to a processor, the user may have to fix the code within the subroutine. Upon debugging and revision, the updated subroutine needs to be reinstalled into the proper executable.

To revise an executable, the following steps are needed:

1. Edit the (initialization or execution) subroutine.
2. Compile the subroutine
3. Remake the LOCSEIS.a archive library
4. Remake the appropriate LOCAL-SEIS executable (localinit or localexec).

1. Edit the (initialization or execution) subroutine.

If a mistake exists within the initialization or execution subroutine, the code should be debugged and revised. The updated subroutine should be placed within \texttt{/../local13/seismic}.

2. Compile the subroutine.

The revised subroutine should be recompiled:

\[ (%) \text{ cd} \texttt{../local13/seismic}. \]

\[ (%) \text{ ../install/compone subroutine.F} \]

Note the relative location of file "compone" relative to the location of the source code subroutine.

3. Remake the LOCSEIS.a archive library.

Once the revised object code exists, the archive library LOCSEIS.a needs to be updated:

\[ (%) \text{ cd} \texttt{../local13/install}. \]

\[ (%) \text{ liball} \]

The \texttt{liball} shell will re-archive the object code within \texttt{../local13/seismic} and "ranlib" the archive library.

4. Remake the appropriate LOCAL-SEIS executable (localinit or localexec).

If the revised subroutine was an initialization subroutine, then remake the initialization executable:

\[ (%) \text{ cd} \texttt{../local13/install}. \]

\[ (%) \text{ make localinit} \]

If the revised subroutine was an execution subroutine, then remake the execution executable:

\[ (%) \text{ cd} \texttt{../local13/install}. \]

\[ (%) \text{ make localexec} \]

Either of the two new executables will be placed within \texttt{../local13/run}. 

IV-D. Creation of an Initialization Subroutine

The initialization subroutine is the subroutine called during the initialization phase of a seismic processing job. The subroutine is called from within localinit and sets up the behavior of the processor for subsequent execution. This subroutine defines and retrieves user parameter values and conveys them to the execution phase via temporary communication files. The name of the subroutine is defined within the PROCLOC.LIS file as described in section IV-C.

The initialization subroutine is constructed to have six major parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Note:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. variable declarations</td>
<td>include files and defaults</td>
</tr>
<tr>
<td>2. parameter definition</td>
<td>call to DRTABL()</td>
</tr>
<tr>
<td>3. parameter retrieval and processing</td>
<td>call to DRPRMS()</td>
</tr>
<tr>
<td>4. printing to .IPR listing</td>
<td>WRITE(LO,*)</td>
</tr>
<tr>
<td>5. parameter transfer to EXEC subroutine</td>
<td>call to DRSAVE()</td>
</tr>
<tr>
<td>6. work space reservation: EXEC subroutine</td>
<td>call to DRRSRV(), DRGNCL()</td>
</tr>
</tbody>
</table>

Certain (and sometimes numerous) rules and regulations govern each part. Prior to constructing a subroutine, one should examine subroutines within .../sseis13/seismic and .../iris13/seismic for examples. In particular, subroutines for IRIS-SEIS processors /EASYADD, /EXTRACTRC, and /EXMULTRC should be examined. The SIERRASEIS Programmer’s Manual should also be consulted, particularly for detailed information on SIERRASEIS function-type subroutines.

The initialization subroutine is used to define the parameters which describe the options available for a processor. Parameter values are used to regulate how the processor is to treat seismic data. Based on the run-time behavior of the processor’s algorithm, a programmer can define available parameters and also regulate default values and how parameter values are to be interpreted and made available to the execution phase. Resources (work arrays or buffers) can be allocated by the INIT subroutine for use in the execution subroutine.

Key SIERRASEIS initialization subroutines

The construction of an initialization subroutine is structured around several key subroutines which are provided within SIERRASEIS. These subroutines are used in two major ways: parameter definition/retrieval and parameter transfer to EXEC subroutine.

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRTABL()</td>
<td>add a parameter into the table of available parameters.</td>
</tr>
<tr>
<td>DRPRMS()</td>
<td>retrieve the next parameter and associated arguments from the job listing.</td>
</tr>
<tr>
<td>DRSAVE()</td>
<td>communicate values to execution-phase subroutine.</td>
</tr>
<tr>
<td>DRRSRV()</td>
<td>reserve extra work space for execution-phase subroutine.</td>
</tr>
<tr>
<td>DRGNCL()</td>
<td>reserve seismic trace and work buffers for execution-phase subroutine.</td>
</tr>
</tbody>
</table>

The programmer must define within the subroutine all possible parameter options, even if only a subset may be selected within a seismic processing job at any one time. The definition of each parameter is conducted using subroutine DRTABL(). Programming statements must be given to retrieve and process each parameter. Subroutine DRPRMS() is used for parameter retrieval.

The transfer of parameter values from the initialization to the execution subroutines is conducted using subroutines DRSAVE(), DRRSRV(), and DRGNCL(). The transfer is performed via one or two temporary SIERRASEIS-format files (.CMB and .CBR files). The execution subroutine must be coded in a specific manner to receive the parameter values. The
format in which this is done is based on the execution subroutine argument list (e.g., FORTRAN statement "SUBROUTINE ABCDEF( arg1, arg2, arg3... )"). The order and size of arguments passed into the execution subroutine are defined by the initialization subroutine. This definition is the primary mechanism to pass parameter values between the initialization and execution subroutines.

**Schematic Initialization Subroutine**

A schematic subroutine shown in Figure 14 has the six major parts of an initialization subroutine. This subroutine defines three possible parameters that a user may choose and defines the code to process whatever choices are made (or keep default values). Print statements to the .IPR file allows the user to see how the processor has interpreted parameter values from the job listing. Finally, transfer of parameters to the execution subroutine is established.

The subroutine is named using no more than six characters with the last character(s) describing that it is an initialization subroutine. In this case, the subroutine name ends with an '0'; however it could end with "I", "IN", "INI", or any other similar description. This name is listed within the "PROCLOC.LIS" file in ../local13/run.

The first section is for variable declaration and default values. The include files are part of the SIERRASEIS infrastructure and allow the subroutine access to the common global variables (SSCOM.INC) and to the .IPR output unit number for the subsequent WRITE statements (SSCUNI.INC). The subroutine will make use of an array named "freq()". Default values for "freq()", "numtraces", and "ix" are defined. The variable "ix" is a trace counter which will be incremented within the execution subroutine. Initially, "ix"=0, indicating to the execution subroutine that it has been entered for the first time. Initial setup can be flagged to occur at this moment.

In the second section, the programmer has defined three possible parameters which the user may select for this processor. The user may override default values by specifying values after the parameters "TRACES", "FREQBEG", or "FREQEND". Within the call to DRTABL(), the third argument is the variable in which values for the parameter will be returned. The fourth argument is a number code which identifies that particular parameter definition. A more complete description of function DRTABL() is provided below.

Once the parameters have been defined, their values may be retrieved from the job listing. Subroutine DRPRMS() retrieves each parameter which is specified after the call to the processor in the job listing. Since the parameters may be specified in any order, DRPRMS() matches the parameter name to those given by DRTABL(). DRPRMS will identify the number code of the parameter as specified by DRTABL() and will return this code number ("return code"), the number of values which follow the parameter in the job listing, and the actual values.

In the example of Figure 14, if the code number returned from DRPRMS() is "1", then according to DRTABL() the specified parameter is "TRACES". The code within section three of the subroutine will retrieve the value specified with the parameter and assign it into variable "numtraces". The next call to DRPRMS() will obtain the next parameter from the job listing. When no parameters are left for this processor, a return code number of ".1" is returned, signifying no more calls to DRPRMS() should be made.

Within section three, the values for all parameters which are retrieved should be checked for validity. Values which are unrealistic, will exceed dimension limits, or will produce incorrect operation of the execution subroutine should be detected here. Error statements can be printed in the .IPR file and the job can be made to abort by calls to subroutine LOCMPEROR().

Documentation print statements can be made to the .IPR file using write statements with an output unit number stored in a SIERRASEIS variable "LO". This variable is defined within the include file "SSCUNI.INC" which must be declared at the beginning of the
Figure 14. Schematic Example of Initialization Subroutine

subroutine EXAMP0

C 1. Variable Declarations: include files, defaults
#include "SSCUNI.INC"
real freq(2)
#include "SSCOM.INC"
integer numtraces,ix

numtraces=48 freq(1)=10. freq(2)=40.
ix=0

C 2. Parameter Definition: user choices
DRTABL('name',format,variable,returncode,#values,repeatable?)
call DRTABL('FREQBEG
', 'FLOT', xtemp, 2, 1, .FALSE.)
call DRTABL('FREQEND
', 'FLOT', xtemp, 3, 1, .FALSE.)

C 3. Parameter retrieval and processing
100 call DRPRMS(iretcode,nvalues)
if(iretcode.EQ. 1)go to 200
endif
if(iretcode.EQ. 2)then
numtraces=xtemp
freq(1)=xtemp
200 continue
endif
if(freq(2) .LT. freq(1))call irisMPEROR('EXAMPLE',1)

C 4. Print to .IPR
write(LO, 300)numtraces
write(LO, 400)freq(1),freq(2)
300 format('EXAMPLE> number of traces is',i5)
400 format('EXAMPLE> start, end frequencies:',2f10.3)

C 5. Parameter communication
call DRSAVE(numtraces,4,1) call DRSAVE(freq,5,2)
call DRSAVE(ix,6,1)
C 6. Work space reservation
call DRGNCL('EXAMPLE',KNSAMP,KNSAMP)
call DRRSRV(dummy,7,KNSAMP)

C Result of steps 5 and 6: execution subroutine is defined as:
C subroutine
EXAMPX(jp,trace,work,numtraces,freq,ix,scratch)
C real trace(1),work(1),scratch(1),freq(1)
C integer numtraces,ix
C NOTE: "ix" is a trace counter used in execution subroutine.
RETURN END
subroutine. A WRITE statement should have a companion FORMAT statement which identifies the processor in some way. In the example in Figure 14, two lines of text will appear in the .IPR file from this processor.

Parts 5 and 6 of the schematic subroutine are used to transfer parameters and to reserve work space within the execution subroutine. Each item saved or reserved using functions DRSAVE(), DRRSRV(), or DRGNCL() will appear as a subroutine argument for the execution subroutine:

<table>
<thead>
<tr>
<th>c</th>
<th>lines in initialization subroutine:</th>
</tr>
</thead>
<tbody>
<tr>
<td>call DRSAVE(numtraces,4,1)</td>
<td></td>
</tr>
<tr>
<td>call DRSAVE(freq,5,2)</td>
<td></td>
</tr>
<tr>
<td>call DRSAVE(ix,6,1)</td>
<td></td>
</tr>
<tr>
<td>call DRGNCL('EXAMPLE',KNSAMP,KNSAMP)</td>
<td></td>
</tr>
<tr>
<td>call DRRSRV(dummy,7,KNSAMP*2)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c</th>
<th>execution subroutine declaration:</th>
</tr>
</thead>
<tbody>
<tr>
<td>subroutine EXAMP(jp,trace,work,numtraces,freq,ix,scratch)</td>
<td></td>
</tr>
<tr>
<td>dimension trace(1),work(1),freq(1),scratch(1)</td>
<td></td>
</tr>
<tr>
<td>integer jp,numtraces,ix</td>
<td></td>
</tr>
</tbody>
</table>

DRSAVE() will transfer an already defined value to the execution subroutine. The second argument of DRSAVE() is the location of the value within the execution subroutine argument list. For example, "numtraces" will be the fourth argument in the execution subroutine’s SUBROUTINE statement argument list. Variable "freq" will be available from the fifth argument. The number of elements to be retrieved is given by the third argument of DRSAVE(); note that there will be two values contained in "freq()" in the execution subroutine.

The function DRGNCL() is used to define the work space occupied by the incoming/outgoing trace and its associated work buffer. By default, DRGNCL() defines the first three arguments of the execution subroutine: a job process number ("jp"), the incoming/outgoing trace buffer ("trace"), and a work buffer ("work"). The size of the trace and work buffers are given by the second and third arguments of DRGNCL().

DRRSRV() is used to dynamically allocate work space when the execution subroutine is invoked. This subroutine defines the size of the work space needed (i.e., KNSAMP*2) and the location in the SUBROUTINE argument list (i.e., sixth argument in SUBROUTINE argument list above). The array "scratch" specified as the sixth argument in the execution subroutine argument list will be the array which contains the dynamically allocated work space.

Notes - 1: Variable declaration - include files and defaults

The use of include files within a subroutine allows the subroutine to access variable values which are used by the SIERRASEIS infrastructure. Access to all global common variables is through the include file "SSCOM.INC". Output to the run-time text files (.IPR and .EPR files) requires the inclusion of file "SSCUNI.INC". Other more specialized functionality may require separate include files; the identification of these files is made through examination of SIERRASEIS subroutines which perform similar behavior. Most application subroutines will probably only need the two files mentioned here.

Notes - 2: Parameter definition - the DRTABL function

The DRTABL() function has six arguments which define the full behavior of parameters and their values. These parameters can be used to obtain single or several values; parameters can be also used as flags with no values or can be repeatably given.

The six arguments for DRTABL() in proper order are "name", "format", "variable", "return code", "# elements", and "repeatable". Examples of the arguments within this function are found in Figure 14.
The "name" argument contains the name of a parameter which can be specified by a user within a job listing. The character string for "name" can be up to eight alphabetic characters long; current limitations within the SIERRASEIS utility routines prevent numerals to be used. As a result, a parameter name of "GATETWO" is valid while "GATE2" is not.

The "format" argument specifies the format of the value(s) which are to be specified following the parameter name. The value of "format" is a four-character text string. Valid text strings are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'DEC'</td>
<td>an integer</td>
</tr>
<tr>
<td>'FLOT'</td>
<td>a floating point</td>
</tr>
<tr>
<td>'ALF'</td>
<td>a character string (character*8)</td>
</tr>
<tr>
<td>'ALFS'</td>
<td>a character string terminated with a single quote</td>
</tr>
<tr>
<td>'ALST'</td>
<td>more than one character string terminated with a single quote</td>
</tr>
<tr>
<td>'DELT'</td>
<td>numbers in delta format (first, last, increment)</td>
</tr>
<tr>
<td>'LGL'</td>
<td>a logical value (.TRUE. or .FALSE.)</td>
</tr>
<tr>
<td>'HEX'</td>
<td>a hexadecimal value</td>
</tr>
<tr>
<td>'OCT'</td>
<td>an octal value</td>
</tr>
</tbody>
</table>

An example of an 'ALF' string is 'VAW'. An example of an 'ALFS' string is '/scr/work/DATA.DIO'. An example of an 'ALST' string is 'KSHOT KTRC KNSAMP'.

The "variable" argument is the name of the variable or array in which values are placed into upon retrieval from the job listing. A call to DRPRMS() retrieves the values and places them into the variable given in the "variable" slot.

The "return code" value given in the fourth slot is returned by DRPRMS() whenever the defined parameter is obtained from the job listing. The return code value generally increases from one call to DRTABL() to the next but can be given in any order. Values up to 9999 are acceptable.

The number of elements (values) that can be retrieved for a given parameter is given as the fifth argument "# elements". If a single value is needed, give the integer value "1". If two values are needed, give the value "2".

The "repeatable" argument is used to indicate whether a parameter can be given only once per processor call or can be given more than once. For example, a processor which builds a two-dimensional horizon model may allow for several horizons. The parameter "HORIZON" can be given several times if the "repeatable" argument is defined as ".TRUE.". If the value is ".FALSE.", then "HORIZON" can be given once and thus only one horizon could be given.

The first call to DRTABL() in Figure 14 is:

```
call DRTABL('TRACES', 'DEC', itemp, 1, 1, .FALSE.)
```

In this example, the parameter is named "TRACES". The value to be given with this parameter is a single integer which will be retrieved in variable "itemp" by DRPRMS(). The return code is "1". The parameter "TRACES" cannot be specified more than once per processor call.

Notes - 3A: Parameter retrieval and processing - DRPRMS function

The DRPRMS() function identifies from the job listing the next parameter specified for the given processor. The function returns three items: the parameter return code, the number of values within the job listing associated with the parameter, and the values stored in the variable as specified within the appropriate call to DRTABL(). The return code is used to identify which portion of subsequent code should be used to process the values which occurred within the job listing. The number of values is used to indicate how many values should be processed. When no more parameters exist to be processed for the processor, DRTABL() returns a return code of "-1".
The return code is used to transfer programming control to appropriate parameter processing code. Two programming structures exist which use the return code - IF-ELSEIF's or a conditional GOTO:

<table>
<thead>
<tr>
<th>IF-ELSEIF's</th>
<th>Conditional GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 call DRPRMS(iretcode,nvalues)</td>
<td>100 call DRPRMS(iretcode,nvalues)</td>
</tr>
<tr>
<td>if(iretcode .EQ. -1)goto200</td>
<td>if(iretcode .EQ. -1)goto200</td>
</tr>
<tr>
<td>if(iretcode.EQ. 1)then [process]</td>
<td>goto(110,120)iretcode</td>
</tr>
<tr>
<td>elseif(iretcode.EQ. 2)then [process]</td>
<td>goto100</td>
</tr>
<tr>
<td>endif</td>
<td></td>
</tr>
<tr>
<td>goto100</td>
<td></td>
</tr>
<tr>
<td>200 continue</td>
<td>200 continue</td>
</tr>
</tbody>
</table>

If more than one value is to be retrieved by DRPRMS() for a parameter, then the variable name given to DRTABL() should be a dimensioned variable. For example:

```fortran
subroutine EXAMP0
  dimension itemp(5)
  call DRTABL('VELS', 'DEC', itemp, 1, 5, .FALSE.)
  100 call DRPRMS(iretcode,nvalues)
    if(iretcode.EQ. -1)goto200
    if(iretcode.EQ. 1)then
      velstart=itemp(1)
      velend=itemp(5)
    endif
    goto100
  200 continue
```

Lines from the job listing which call this processor would look like:

```
/EXAMPLE
VELS 4500, 5000, 5700, 6200, 7800
```

Notes - 3B: Parameter retrieval and processing - Error Statements

If an error is detected during parameter processing, either warning or fatal error messages should be invoked. For non-fatal warnings, messages can be written to the .IPR text file which is generated by localinit. Non-fatal warnings are written using the WRITE/FORMAT statements:

```fortran
WRITE(LO,600)itemp
600 FORMAT('EXAMPLE> WARNING: user choice of',i5,' exceeds limits')
```

For fatal errors, LOCAL-SEIS uses a fatal-error subroutine which is identical to one used by SIERRASEIS. Within SIERRASEIS, the subroutine "MPEROR" is invoked with a processor name and an error statement. This subroutine scans the error text file ERRORS.LST in .../sseis13/run to select the proper error statement. The statement is then printed into the .IPR or .EPR file.
For LOCAL-SEIS, the subroutine LOCMPEROR will obtain error messages from the error text file LOCERR.LST in .../local13/run. If an error is detected, invoke the subroutine as follows:

```fortran
IF(itemp.GT.maxvalue)call LOCMPEROR('EXAMPLE',13)
```

LOCMPEROR will scan the error text file for the errors associated with processor 'EXAMPLE'. Error number 13 in this section will be printed to either the .IPR or .EPR for localinit or localexec, respectively. The executable can be asked to abort gracefully.

If the programmer uses subroutine LOCMPEROR, he or she will need to enter the appropriate error statements into text file LOCERR.LST. This text file is scanned to see the format in which error statements are entered. Within the file, the user can specify whether the error is fatal or not; this will determine whether the job will crash or continue.

**Notes - 4: Printing to .IPR listing**

Information can be written to the .IPR using WRITE statement unit number "LO" provided the "SSCUNL.INC" file is included at the top of the subroutine. The unit number "SLO" will print information into the .EPR file. Sufficient information regarding user-selected parameters should be printed into the .IPR to allow the user to determine whether the parameters were processed correctly.

**Notes - 5: Parameter transfer to the EXEC subroutine - the DRSAVE function**

In certain cases where several values need to be saved in order to transfer them to the execution subroutine, one may find a construction where the transfer is not performed one at a time using several calls to DRSAVE(). Rather, a single array may be used to store the numbers and then transfer them via one call to DRSAVE(). The fewer number of calls to DRSAVE() translates into fewer subroutine arguments for the execution subroutine. This increases, however, statements needed to recover the values from the array. Figure 15 illustrates the differences between these two approaches.

Contents of variables defined by DRSAVE() are saved between the initialization and execution subroutines. Just as important, the contents of these variables are saved between calls (i.e., seismograms) to the execution subroutine. In trace-by-trace processing, the contents are available for each successive seismogram. If the execution subroutine modifies the contents of such a variable, the updated value is made available when the subroutine is called for the next seismogram.

**Notes -6: Work space reservation for EXEC subroutine - DRRSRV, DRGNCL functions**

In order to make memory requirements efficient for localinit and localexec, one can use DRRSRV() and DRGNCL() to dynamically allocate memory for localexec. These routines allow the initialization subroutine to determine the amount of memory needed based on various factors (i.e., gather size based on number of traces and samples per trace). The memory will not be allocated until the execution subroutine is invoked in localexec. This approach is more efficient than to declare an array of the needed size within the initialization subroutine and then saving the space using DRSAVE(). It is often difficult to know just how large to declare an array a priori to any specific job.

A fundamental difference between DRSAVE() and DRRSRV()/DRGNCL() is that the contents of variables saved by DRSAVE are transferred from the initialization to the execution subroutines. For the latter functions, no contents exist during the initialization subroutine.

The contents of the work buffer as defined by DRGNCL() are not saved between calls to the same execution subroutine by successive seismograms. To save the contents of a work buffer, an extra array should be allocated by using DRRSRV(). The contents of an array allocated by DRRSRV() will be saved between calls to the subroutine by successive seismograms.
Figure 15. Examples of efficient array transfer for DRSAVE()

A: Approach ONE

Initialization routine code:

```fortran
    call DRSAVE(numtraces,4,1)
call DRSAVE(freq0,5,1)
call DRSAVE(freq1,6,1)
call DRSAVE(cdp0,7,1)
call DRSAVE(cdp1,8,1)
call DRSAVE(time0,9,1)
call DRSAVE(time1,10,1)

call DRGNCL('EXAMPLE',KNSAMP,KNSAMP)
```

Execution routine code:

```fortran
subroutine EXAMPX(jp,trace,work,numtraces,freq0,freq1,+
cdp0,cdpl,time0,timel)
real trace(1),work(1)
```

B: Approach TWO

Initialization routine code:

```fortran
    xpass(1)=freq0
    xpass(2)=freq1
    xpass(3)=cdp0
    xpass(4)=cdpl
    xpass(5)=time0
    xpass(6)=timel

call DRSAVE(numtraces,4,1)
call DRSAVE(xpass,5,6)

call DRGNCL('EXAMPLE',KNSAMP,KNSAMP)
```

Execution routine code:

```fortran
subroutine EXAMPX(jp,trace,work,numtraces,xpass)
real trace(1),work(1),xpass(1)

freq0=xpass(1)
freq1=xpass(2)
cdp0=xpass(3)
cdpl=xpass(4)
time0=xpass(5)
timel=xpass(6)
```
Assignment of Global Common Variables

Values of global common variables (GCV's) which are set within the initialization subroutine will be passed to subsequent subroutines during the execution of localinit. For the processors selected within a job listing, each initialization subroutine will be invoked in the order listed. Updated GCV values will be passed to the successive subroutines.

For example, the IRIS-SEIS processor /NULENGTH is used to modify a seismogram length by changing the number of samples per seismogram. The GCV to be modified is KNSAMP. The initialization subroutine interprets the new trace length from the job length and resets KNSAMP accordingly. Initialization subroutines which follow the subroutine for /NULENGTH will use the modified trace length.

The act of modifying a GCV within the initialization subroutine does not automatically make the change during the execution subroutines. The programmer must ensure that the new value of a GCV is communicated to the execution subroutine and that the GCV is properly reset for appropriate seismograms within the execution subroutine. An example of this behavior can be seen by examining the initialization and execution subroutines ("NULEN0.F" and "NULENX.F", respectively) within ../iris13/seismic for the processor /NULENGTH.

Accessing Global Common Variables

The contents of GCV's can be set in a subroutine by one of two different ways. In the first method, the GCV's which are defined within SIERRASEIS (i.e., listed in include file "SSCOM.INC") can be explicitly accessed by name. For example, the CDP number can be directly set using a line such as "KCDP = 100".

An alternative programming approach is to access a GCV though the use of an index location within the GCV storage arrays located in the include file "SSCOM.INC". As explained in Section II-A, the four types of GCV's are accessible using arrays SSCMAI(), SSCMAR(), SSCMA8(), SSCMA4() within "SSCOM.INC". Each index position in each array is defined in SSCOM.INC as a valid GCV; reference to a particular location in an array is the same as a reference to a specific GCV by name (i.e., SSCMAI(15) is the same as KCDP). Manipulation of a GCV can be performed provided the array index location is known.

A programming aid to access GCV's is provided within IRIS-SEIS to access both GCV's defined within "SSCOM.INC" and user-defined GCV's. The subroutine irisGETHED() will obtain the array type and index location for any GCV. The GCV can then be accessed by use of the array. For example, a call to irisGETHED() to obtain information on KCDP will indicate that the 15th position of the integer GCV array should be used. The CDP number is thus accessible through array SSCMAI(15) as is illustrated in the following lines of programming code:

```plaintext
character*6 cname
integer index,iformat
cname='KCDP'
call irisGETHED(cname,index,iformat)
if(iformat.eq.1)SSCMAI(index)=1001
```

Given a GCV name as stored in variable "cname", the array format and array index position is returned. The array format is =1 for the integer array SSCMAI(), =2 for the real array SSCMAR(), =3 for the CHAR*8 array SSCMA8(), and =4 for the CHAR*4 array SSCMA4(). The values of variables "index" and "iformat" can be transferred to the execution subroutine in order to access the GCV during execution-phase processing.

Use of the subroutine irisGETHED() allows a processor to be constructed in a generic manner in which any GCV can be accessed if necessary. Use of this subroutine can be found in the /HEADLIST processor subroutine "HDLST0.F" in ../iris13/seismic.
**Definition of New Global Common Variables**

If necessary, a new GCV can be defined within a processor. Utility subroutine `irisDEFHED()` can be used to define a new GCV for use within a subroutine or to be made available within subsequent processor subroutines. The subroutine will place the new GCV's name into the list of available GCV's within the "IRISCOMC.INC" include file and will reserve an index location in the appropriate GCV array in "SSCOM.INC".

The `irisDEFHED()` subroutine can be used in the following manner:

```fortran
character*6 cname
character*4 cformat
integer index, iformat
cname='testhd'
cformat='REAL'
call irisDEFHED(cname, cformat, index, iformat)
```

The variable "cname" contains the name of the new GCV; the characters can be alphanumeric (and can even include punctuation). The variable "cformat" describes which GCV type to use; proper values are the 4-character strings "INT", "REAL", "CHR8", are "CHR4". The variables "index" and "iformat" are returned by the subroutine and describe the index location within the GCV array and the array type (as defined by subroutine `irisGETHED()`).

To define a new header within a processor, both the initialization and execution subroutines should use `irisDEFHED()`. The use of the subroutine in the initialization subroutine will make the new GCV available for subsequent processors in the initialization phase of processing. The execution subroutine will need to use `irisGETHED` in order to define the new GCV for each seismogram which enters the subroutine. For examples, see processor /DEFHEAD subroutines "DEFHD0.F" and "DEFHDX.F" in ../iris13/seismic.

**More information**

For more information, read the Programmer's manual provided for SIERRASEIS. Also, much information can be extracted through the examination of already-existing subroutines located in ../sseis13/seismic and ../iris13/seismic. Most likely, the specifics of programming steps needed may already exist located in various subroutines. It is often easier to modify a subroutine which is similar to what the user needs rather than to rewrite one from scratch.
IV-E. Creation of an Execution Subroutine

The execution subroutine is the subroutine called during the execution phase of a seismic processing job. The subroutine is called from within localexec and performs the actual processing to seismic traces or seismic GCV's. This subroutine retrieves user parameter values as defined within the initialization phase, performs run-time resource allocation, processes seismograms, sets or resets GCV values, and sets or resets trace flow behavior as required by the subroutine algorithm. The name of the subroutine is defined within the PROCLOC.LIS file as described in section IV-C.

The execution subroutine is constructed to have seven important parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Note:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. subroutine argument list</td>
<td>as defined within init</td>
</tr>
<tr>
<td>2. variable declarations</td>
<td>include files and declarations</td>
</tr>
<tr>
<td>3. initial allocations</td>
<td>first time ever in subroutine</td>
</tr>
<tr>
<td>4. processing algorithm</td>
<td>actual processing work</td>
</tr>
<tr>
<td>5. trace flow</td>
<td>single or multiple traces?</td>
</tr>
<tr>
<td>6. printing to .EPR listing</td>
<td>WRITE(SLO,*))</td>
</tr>
<tr>
<td>7. end of job cleanup</td>
<td>release resources</td>
</tr>
</tbody>
</table>

Similar to the initialization subroutine, there are rules and regulations which govern each part. Prior to constructing a subroutine, the programmer should examine subroutines within ../sseis13/seismic and ../iris13/seismic for examples. In particular, subroutines for IRIS-SEIS processors /EASYADD, /EXTRACTR, and /EXMULTRC should be examined. The SIERRASEIS Programmer's Manual should also be consulted, particularly for detailed information on SIERRASEIS function-type subroutines.

The execution subroutine is used to process seismic data. An algorithm is emplaced into the subroutine which works on single seismograms (trace-by-trace operations) or with multiple sets of seismograms. The execution subroutine should be programmed to conduct initial resource allocation, monitor trace flow, send print statements to the .EPR text file if necessary, and clean up resources upon passage of the last seismogram. Figure 16 illustrates a schematic execution subroutine which is consistent with the initialization subroutine in Figure 14.

1. Subroutine argument list: consistency with initialization routine

The declaration of the subroutine ("SUBROUTINE" line of code) should have an argument list which is consistent with the information which was saved in the initialization subroutine by the subroutines DRSAVE, DRGNCL, and DRRSRV (see previous section). The variables within the argument list are used either to retrieve information which is to be conveyed from the initialization subroutine or to dynamically allocate space during use of the execution subroutine.

The first three arguments of a subroutine is by SIERRASEIS convention always defined to be an integer variable used by the localexec driver, the trace array, and a work array. The three arguments are defined by a call to DRGNCL within the initialization subroutine; the size of the trace and work arrays are specified there.

Any subsequent variables must have been defined by DRSAVE, DRGNCL, or DRRSRV. Any of these values within the argument list will be saved between calls to the execution subroutine; this is one way to communicate values between successive seismograms.

2. Variable declarations: declarations and include files

Any variable which appears in the argument list should be declared at the top of the execution subroutine. Dimensioned arrays can be specified without the full dimension as if to indicate the address of the array (i.e., "trace(1)" is the address of element one but represents
Figure 16. Schematic Example of Execution Subroutine

C 1. Subroutine Argument list: as defined in init routine. C This subroutine is compatible with Figure 14.

C subroutine EXAMPX(jp,trace,work,numtraces,freq,ix,scratch)

C 2. Variable Declarations: include files, declarations
#include "SSCOM.INC"
#include "SSCUNI.INC"

real trace(1),work(1),scratch(1),freq(1)
integer numtraces,ix

C (Part of 5. Trace flow goes at the top)
C if(KSTATE.eq.3)goto500

C 3. Initial allocations
i=1,KNSAMP
endif
ix=ix+1

C 4. Processing
algorithm using arrays trace(), work(), freq(), scratch(),
C and variable numtraces
C

C 5. Trace flow (GCV's KSTATE, KCMBCK)
KSTATE=2
KCMBCK=0

C 6. Print to .EPR listing
C Be careful about a write() statement for every trace
C if the expected data sets will be large.
write(SLO,IOO)ix
100 format('EXAMPLE> Trace ',i5, 'processed.')
goto1000

C 7. End-of-Data (KSTATE=3): Cleanup or release resources.
write(SLO,200) 200 format('EXAMPLE> Only reached when End-of-
Data occurs')
1000 RETURN END
the entire array). Dynamically-allocated double-subscripted arrays can be used by passing the array and dimension sizes via the argument list. This information can thus be used in the declaration as follows:

```
subroutine EXAMPX(jp,trace,work,array,isize,jsize)
dimension trace(1), work(1), array(isize,jsize)
```

Do not place a GCV name such as KNSAMP in the argument list. Remember also that the initialization subroutine must have specified the proper array size within its calls to DRGNCL(), DRRSRV(), and DRSAVE():

```
subroutine EXAMP0
  call DRGNCL("EXAMPLE",KNSAMP,KNSAMP)
  call DRRSRV(temp,4,isize*jsize)
  call DRSAVE(isize,5,1)
  call DRSAVE(jsize,6,1)
```

Include files should be declared at the start of an execution subroutine if variables defined within the include files are to be used. If any GCV's are used, the include file "SSCOM.INC" should be used. If any output to the .EPR file is to be written, the file "SSCUNI.INC" should be included.

Other variables which are used within the subroutine will be cleared upon each successive entry (each new seismogram). Prior values will not be available for re-use. The exception is any variable which exists within the subroutine argument list.

3. Initial allocations: first time setups

Certain processing algorithms require initial setup upon first time entry into a processor subroutine. For example, prior to processing the first seismogram in a data set, algorithm variables or flags may require initial values or clearing (see Figure 16). Devices can be allocated or disk files can be opened or created as needed. The subroutine code must bookkeep the condition of "first-time entry" in order to set initial values, then bookkeep that the initial values were set. A flag or counter should be used to keep track of initial subroutine entry.

A flag counter can bookkeep the number of traces encountered by a processor. This counter is incremented for each trace which enters the subroutine. Provided that the counter is first set to zero within the initialization subroutine, the counter can indicate when the first-ever trace is encountered:

```
Initialization subroutine:
  subroutine EXAMP0
    integer counter
    counter=0
    call DRSAVE(counter,4,1)
    call DRGNCL("EXAMP0",KNSAMP,KNSAMP)
  return
end
```

```
Execution subroutine
  subroutine EXAMPX(jp,trace,work,counter)
    integer counter
    counter=counter+1
    if(counter.EQ.1)then
      [set initial values]
    endif
    return
end
```
In this example, "counter" is passed to the execution subroutine with an initial value of "0". For each seismogram entered into the subroutine, the counter is incremented. Upon the first entry into the subroutine (counter=0), initial values can be set. Subsequent entries will bypass this portion of the code.

Rather than use a seismogram counter, a logical variable can be defined. A logical variable which questions whether the entry has been previously entered can be set to "/FALSE." in the initialization subroutine. If the logical variable value is "/FALSE." in the execution subroutine, initial values can be set after which the logical variable can be set to "/TRUE.". Subsequent entries into the subroutine will bypass the local code.

4. Processing algorithm

The processing algorithm can be any process which works on the seismic data or GCV header values. The processing algorithm can operate on single seismograms or can be programmed to require more than one seismogram. In addition, there is nothing to prevent a programmer from accessing other peripherals or graphical devices within the processor.

The primary programming consideration is that the algorithm must ultimately be compatible with single seismogram re-entry and release ("one-trace-at-a-time" flow). Any programming calls to other subroutines or peripherals must be compatible with the SIERRASEIS programming environment.

The best guide as to how processing algorithms are incorporated into subroutines can be found in the actual processor subroutines. These can be found in ../sseis13/seismic or ../iris13/seismic.

5. Trace flow: single and multiple traces

Each execution-phase subroutine must explicitly state the trace flow behavior for the subroutine. The trace flow can be trace-by-trace or multiple-trace algorithms; additionally, the processor can be made to operate when no data is left (end of the job). The accumulation or release of multiple traces must also be properly accounted for within the subroutine.

The behavior of trace flow is defined by a subroutine by the values of GCV's "KSTATE" and "KCMBCK". The execution-phase driver localexec always assumes that a single trace enters the subroutine. The trace flow behavior upon leaving the subroutine is determined by the driver based on the values of these GCV's. According to the value of KSTATE, the driver will either pass the seismogram to the next processor in trace-by-trace flow or return to a previous processor to obtain another seismogram in multiple accumulation flow. The value of KCMBCK indicates whether the subroutine has a number of seismograms ready for multiple release. The values of these GCV's must be set prior to leaving the subroutine.

The status of KSTATE should also be checked upon entry to the subroutine. When no data is left to be processed, KSTATE will have the value of "3". An IF conditional can be emplaced to skip the entire subroutine if the end-of-data status is reached. Other processors may wish to perform end-of-data operations such as cleanup, resource release, or final printouts to the .EPR file.

A more full explanation of KSTATE and KCMBCK is available in section II-A. Programming examples can be found in the seismic directories for SIERRASEIS or IRIS-SEIS. For examples of trace-by-trace processor subroutines, examine those processors which operate in trace-by-trace mode (/AGC, for example). Similarly, examine multiple trace processors for examples of how to manipulate the proper GCV's. Examples are also provided in /EASYADD, /EXTACTRC, and /EXMULTRC within the

A final word regarding KSTATE and KCMBCK values which has implications as to bookkeeping of the values of these GCV's. The values of KSTATE and KCMBCK are passed from one processor to the next. Since each processor resets the values of these GCV's, the values when re-entering a specific processor subroutine may not be the same as
when it was exited one seismogram previously. For example, if one processor sets KSTATE/KCMBCK to imply multiple-release flow and the next processor operates in trace-by-trace flow, the values of KSTATE/KCMBCK when re-entering the first processor's subroutine will indicate trace-by-trace flow. This subroutine cannot be written with the assumption that the KSTATE/KCMBCK values are "remembered"; rather, it is best to internally maintain these values within storage variables so that the GCV's can be retrieved and thus explicitly reset to the proper flow values. An example of this can be found for the processor EXMULTRC (subroutines "EXMULO.F" and "EXMULX.F" in ../iris13/seismic).

6. Printing to .EPR listing

Run-time printing from an execution subroutine is performed in a manner similar to the printing to an .IPR file. The FORTRAN unit number for a WRITE statement is defined within the SIERRASEIS variable "SLO" which is defined in the include file "SSCUNI.INC". A write statement which will appear in the .EPR file is thus:

```fortran
subroutine EXAMPX(jp,trace, work)
    # include "SSCUNI.INC"
    write (SLO,100)
    100   format('EXAMPLE> Entered subroutine')
    return
end
```

The results of a write statement will appear in the .EPR file once for each time the subroutine is re-entered unless some type of selectivity is installed. An internal trace counter can be used to identify some desired re-entry interval.

7. End of job cleanup

The value of GCV KSTATE will become "3" when no more input data is available to the job stream. At this point, a processor should be programmed to either simply exit its subroutine or to conduct end-of-job shutdown. The end-of-job shutdown can include final printouts to the .EPR file, resource release, diskfile closings, or any other operations prior to complete termination of the processing job.

The value of KSTATE should be checked upon entry into the execution subroutine so that the end-of-job status can be immediately detected. If found, flow within the subroutine should not reach other portions of code which is designed for KSTATE values of 1, 2, or 4.
V - USER DEVELOPMENT WORK AREA

LOCAL-SEIS is provided within IRIS-SEIS to be a platform where development of processors can take place. This third tier of seismic routines allows the user to mix locally-developed processors with those which exist within SIERRASEIS and IRIS-SEIS. The separation of local routines from all others allows for a seismic processing package tailored to the applications at an individual locality.

For a single user site, the data processor/programmer can add routines directly to LOCAL-SEIS. In a multi-user environment, however, there are advantages to having a version which contains "final" versions of new routines. In this form, development by two or more persons can occur in separate areas which do not affect each other nor the "final" version.

For these multi-user sites, the development areas exist similar to LOCAL-SEIS. In fact, the relationship between the development areas and LOCAL-SEIS is similar to various versions of LOCAL-SEIS and IRIS-SEIS: when a processor is finalized locally, it can be installed into the library of "approved" processors (LOCAL-SEIS for an individual site and IRIS-SEIS, hopefully, as the repository of processors developed among the SIERRASEIS community).

Within the development areas, a programmer can install a processor into a library which gets linked with all other SIERRASEIS, IRIS-SEIS, and LOCAL-SEIS processors. The resulting executables are then available for testing of the processor by the programmer or user. The development version can be modified for debugging purposes without affecting the performance of LOCAL-SEIS.

The structure of this work area is described in section II-D. The developer works within a directory structure similar to LOCAL-SEIS but creates executables known as userinit and userexec rather than localinit and localexec. The installation of a routine into the user version is essentially identical to the procedures used to install a routine into LOCAL-SEIS.

A user development area template directory is provided within LOCAL-SEIS and is named ../local13/user1. For each person who wishes to install a processor, a copy of this directory should be made as described below. This somewhat custom directory separates his or her developmental code from the code of other developers.

V-A. Installation of template directory "local13/user1"

This section presents a step-by-step installation procedure to create the template development directory user1. This installation needs to be performed only once (preferably when LOCAL-SEIS is installed). It is not necessary to use the "SIERRA" account used for installation of user1, however, write-permission is required within the LOCAL-SEIS directory ../local13. The files in this directory should be kept as master copies and that a copy of the directory should be made where modifications can be made.

The template development directory is named user1 and exists at the same level as other directories which define LOCAL-SEIS (install, lib, main, run, and seismic, for example). The installation of this directory is made easier since LOCAL-SEIS is already installed prior to installation of this directory.

Changes to pathname locations in make files during the installation of LOCAL-SEIS, should be repeated for user1. If symbolic links were used, this step can be bypassed.

The installation procedures below discuss a quick and a full approach to installation of user1. The quick approach can be used because needed object code and executables are already provided in compiled/linked form within the distribution tar tape. These files can be used provided the host CPU is compatible with a SUN 4 (SPARC) with OS 4.1. If your host
CPU is not compatible, then you should use the full installation procedures.

The following procedures are described with the assumption that LOCAL-SEIS has been already installed (see section IV-A).

Quick Installation of template user development directory

To install the template user development directory using the quick steps, you will need to perform the following steps:

1. locate the user1 directory in ../../local13.
2. either:
   (A) if symbolic link /usr/sierra or sierra within /usr was created to point to the overall SIERRA directory, skip, or
   (B) make full pathname change to the following file (#lines to change):
       -local13/user1/install/makefile: one line
3. run shell local13/user1/install/liball,
4. use the makefile to create executables userinit and userexec,
5. test executables to verify installation, then remove executables to save space if desired

(1) Locate the user1 directory in ../../local13.

The initial installation procedure for IRIS-SEIS should have extracted the user1 directory within the local13 directory from the distribution tape. If these files are not present, then they can be extracted from the distribution tape. Change the working directory to the overall SIERRASEIS directory (e.g., ../../sierra).

Use the tar command to extract the files, e.g.;

(%) tar xvf /dev/rmtl6 ./local13/user1

Use the proper tape drive device name.

(2) Directory pathnames: symbolic links or explicit definition

If IRIS-SEIS was installed using either the explicit path /usr/sierra or a symbolic link pointing to this path, then no action should be taken at this step. Either of these options which were used for IRIS-SEIS and LOCAL-SEIS will work for the user development area.

If the installation of IRIS-SEIS did not use the symbolic link or if the location of the overall sierra directory is not at /usr/sierra, then the user should explicitly give some directory paths. Make the change to the following file:

<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
</tr>
</thead>
<tbody>
<tr>
<td>local13/user1/install/makefile</td>
<td>ROOT_DIR = /usr/sierra/local13/user1</td>
</tr>
</tbody>
</table>

The reference to /usr/sierra should be modified to the true location of the sierra/ directory. When this change is made, the new pathname should point to the true location of the development directory.

No modification to file ../../local13/user1/install/compone is required. The include flag used during the compilation of a LOCAL-SEIS subroutine is the same as that used for the compilation of an IRIS-SEIS subroutine within ../../iris13/install/compone.

(3) Remake the user development archive library

In order to remake ("archive" and "ranlib") the archive library of user development object code, use the liball shell. Change directory to the install directory:

(%) cd ../../local13/user1/install
(%) liball

Running the liball shell will create or update the archive library userseis.a.
(4) Create development executables

Create the development executables `userinit` and `userexec` using the makefile located in `../local13/user1/install`:

```bash
(%) make userinit
```

and

```bash
(%) make userexec.
```

The executables will be placed into `../local13/user1/run`.

(5) Test executables to verify installation

By running a LOCAL-SEIS job the user can test the executables. If this developmental version of LOCAL-SEIS is to be used, the user must either place into `../sierra/bin13/fR` symbolic links which point to the executables or place the `../local13/user1/run` directory within his or her path variable. Alternatively, the user can move the executables to the directory in which the test job will run.

The symbolic links for the development versions can be created as follows:

```bash
(%) cd ../sierra/bin13
(%) ln -s /usr/sierra/local13/user1/run/userinit userinit
```

and

```bash
(%) ln -s /usr/sierra/local13/user1/run/userexec userexec
```

Use the correct path to point to `../local13/user1/run` (if not `/usr/sierra`).

After running the test job successfully, the executables can be removed in order to save space.

Full Installation of user development area

A full installation of the user development area includes only a few more steps beyond the quick installation:

1. locate the `user1` directory in `../local13`.
2. either:
   A) if symbolic link `/usr/sierra` or `sierra` within `/usr` was created to point to the overall SIERRA directory, skip, or
   B) make full pathname change to the following file (#lines to change):
      ```bash
      -local13/install/makefile: one line
      ```
3. compile all source code
4. create archive libraries using shell `local13/user1/install/liball`
5. create `usergntabl` by using the makefile within `local13/user1/install`
6. run shell `makelk in local13/user1/install`
7. use the makefile to create executables `userinit` and `userexec`
8. Test executables to verify installation

(1) Locate the `user1` directory in `../local13`.

The initial installation procedure for IRIS-SEIS should have extracted the `user1` directory within the `local13` directory from the distribution tape. If these files are not present, then they must be extracted from the distribution tape. Change the working directory to the overall SIERRASEIS directory (e.g., `../sierra`).

Use the `tar` command to extract the files, e.g.;

```bash
(%) tar xvf /dev/rmt16 /local13/user1
```

Use the proper tape drive device name.
(2) Directory pathnames: symbolic links or explicit definition

If IRIS-SEIS was installed using either the explicit path /usr/sierra or a symbolic link pointing to this path, no action should be taken at this step. Either of these options which were used for IRIS-SEIS and LOCAL-SEIS will work for the user development area.

If the installation of IRIS-SEIS did not use the symbolic link or if the location of the overall sierra directory is not at /usr/sierra, then some directory paths must be given explicitly. Make the change to the following file:

<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
</tr>
</thead>
<tbody>
<tr>
<td>local13/user1/install/makefile</td>
<td>ROOT_DIR = /usr/sierra/local13/user1</td>
</tr>
</tbody>
</table>

The reference to /usr/sierra should be modified to the true location of the user’s sierra/ directory. When this change is made, the new pathname should point to the true location of the development directory.

No modification to the file ../local13/user1/install/compone is required. The include flag used during the compilation of a LOCAL-SEIS subroutine is the same as that used for the compilation of an IRIS-SEIS subroutine within ../iris13/install/compone.

(3) Recompile all source code

Steps to recompile all source code within the user1 directory:

```bash
(cd ../local13/user1/install
(compall
```

The compall shell will recompile all appropriate source code. There are few subroutines which exist within the essentially empty user1 directory, so the recompilation should be quick.

(4) Remake the user development archive library

In order to remake ("archive" and "ranlib") the archive library of user development object code, use the liball shell. Change directory to the install directory:

```bash
(cd ../local13/user1/install
(liball
```

Running the liball shell will create or update the archive library userseis.a.

(5) Create "usergntabl" by using the install makefile

At this point, nearly all the necessary object code will exist. The missing subroutines are the lookup subroutines used by the userinit and userexec drivers. These subroutines are created by running the stand-alone program usergntabl (as discussed in section II A-C). Before the creation of the lookup tables, the user must verify that the stand-alone executable exists.

Within the local13/install directory, run:

```bash
(make usergntabl
```

(6) Create lookup subroutines: userlkinit.F, userlkexec.F

Now that the stand-alone executable exists from step (5), the lookup subroutines can be created. Within the local13/user1/install directory, run the shell file makelk:

```bash
(makelk
```

This shell will run the stand-alone program usergntabl, using the text file ../local13/run/PROCUSER.LIS as input. The SIERRASEIS banner should flash on the screen as the executable is run.
From the execution of these programs, the output will be the lookup subroutines. These subroutines will also be compiled to produce object code. The source and object code of these routines will be placed into the \texttt{./local13/user1/run} directory.

7) Create development executables "userinit" and "userexec"

Create the user development executables \texttt{userinit} and \texttt{userexec} using the makefile located in \texttt{./local13/user1/install}:

\begin{verbatim}
(\%) make userinit
\end{verbatim}

and

\begin{verbatim}
(\%) make userexec.
\end{verbatim}

The executables will be placed into \texttt{./local13/user1/run}.

8) Test executables to verify installation

The user may run a \texttt{LOCAL-SEIS} job in order to test the executables. If this developmental version of \texttt{LOCAL-SEIS} is to be used, the user must either place into \texttt{./sierra/bin13/fR} symbolic links which point to the executables or place the \texttt{./local13/user1/run} directory within his or her path variable. Alternatively, the user can move the executables to the directory in which he or she wishes to run the test job.

The symbolic links for the development versions can be created as follows:

\begin{verbatim}
(\%) cd \texttt{./sierra/bin13}
(\%) ln -s /usr/sierra/local13/user1/run/userinit userinit
\end{verbatim}

and

\begin{verbatim}
(\%) ln -s /usr/sierra/local13/user1/run/userexec userexec
\end{verbatim}

Use the correct path to point to \texttt{./local13/user1/run} (if not \texttt{/usr/sierra}).

After running the test job successfully, the executables can be removed in order to save space.

**Verification of Installation**

The best test of a successful installation is to run a \texttt{LOCAL-SEIS} job. The executables can be tested in two parts. First, by using a job which calls only \texttt{SIERRASEIS} processors (a plot job, for example) and is known to run correctly within \texttt{SIERRASEIS}. The user should get the same results by using \texttt{userinit/userexec}. Second, by inserting an \texttt{IRIS-SEIS} processor into this job stream, then running with \texttt{userinit/userexec}. If the processor is called correctly, then no run-time errors should occur.
V-B. Creation of user work area

A user work area can be created by the duplication of the development directory template user1. The copy of this directory can be customized to allow an individual programmer to develop and test algorithms without impacting the local version of LOCAL-SEIS. Once a processor has been successfully developed within a user work area, it can be "officially" installed into the local version of LOCAL-SEIS.

For a multi-programmer environment, separate custom directories should be created. The separate sets of work space will allow for processor development which will be safe from accidental programming mistakes from other efforts. While each programmer will have a separate directory, the programmers should try to benefit from each other’s experiences.

A custom work area is created by copying the directory contents of the user1 directory template. Minor installation procedures are needed. In addition, customization of executable names can be performed.

It is not necessary to use the "SIERRA" account used for installation of user1, however, write-permission is required within the LOCAL-SEIS directory ./locall3. It must be ensured that the new subdirectories of the work area have sufficient file/directory permissions to allow the user read/write access to the development files.

Since the creation of a user work area is performed by copying the user1 directory template, there is no need for quick/full installation procedures. Here, the assumption is that the template directory has been properly installed so that only one set of procedures is required to set up the work area.

Installation of a custom development directory

To install a user’s custom development directory, the following steps should be performed: Assume the user’s development directory name will be represented by "XXXX"

1. copy user1 into a user’s directory ("XXXX")
2. either:
   (A) if symbolic link /usr/sierra or sierra within /usr was created to point to the overall SIERRA directory, skip, or
   (B) make full pathname change to the following file (#lines to change):
       -locall3/XXXX/install/makefile: one line,
3. run shell locall3/XXXX/install/makeall,
4. use the makefile to create executables userinit and userexec.

(1) copy user1 into a user’s directory ("XXXX")

The contents of the user1 directory should be copied into a directory for the new work area:

(%) cd .../local13
(%) cp -R user1 XXXX

Fix any file permissions which may be incorrect by running a shell file in ...
/local13/XXXX/install:

(%) cd .../local13/XXXX/install
(%) sh.chmod

(2) Directory pathnames: symbolic links or explicit definition

If IRIS-SEIS was installed using either the explicit path /usr/sierra or a symbolic link pointing to this path, then it is not necessary to do anything at this step. Either of these options which were used for IRIS-SEIS and LOCAL-SEIS will work for the user development area.

If the installation of IRIS-SEIS did not use the symbolic link or if the location of the overall sierra directory is not at /usr/sierra, then the programmer must give explicitly some
directory paths. Make the change to the following file:

<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
</tr>
</thead>
<tbody>
<tr>
<td>local13/XXXX/install/...</td>
<td>ROOT_DIR = /usr/sierra/local13/XXXX</td>
</tr>
</tbody>
</table>

The reference to /usr/sierra should be modified to the true location of your sierra/ directory. When this change is made, the new pathname should point to the true location of the development directory "XXXX".

Not modification to the file .../local13/user1/install/compone is required. The include flag used during the compilation of a LOCAL-SEIS subroutine is the same as that used for the compilation of an IRIS-SEIS subroutine within .../iris13/install/compone.

(3) Remake the user development archive library

In order to remake ("archive" and "ranlib") the archive library of user development object code, use the liball shell. Change directory to the install directory:

```
(%) cd ../local13/XXXX/install
(%) liball
```

Running the liball shell will create or update the archive library userseis.a.

(4) Create development executables

Create the development executables userinit and userexec using the makefile located in ../local13/XXXX/install:

```
(%) make userinit
(%) make userexec.
```

The executables will be placed into ../local13/XXXX/run.

User customization of executable names

An individual user may wish to replace the names of userinit and userexec with more personal names. These names can be used to distinguish between executables created by different developers. There are two methods to change the executable names; both methods affect the manner in which the executables are made.

The first method requires a simple modification to the "makefile" in the developer's local13/XXXX/install directory. Within this file, there are two references to the compile flag "-o" which specify the names of the executables userinit and userexec. The developer can customize his or her executable names by replacing within these lines the new names for userinit and userexec. For example, the line

```bash
-o ../(RUN_DIR)/userinit
```

will become

```bash
-o ../(RUN_DIR)/ekinit
```

if the executable userinit is to be renamed ekinit. The basic drawback of this simple approach is that the "make" command will not recognize the new name but will require the old name:

```
(%) make userinit
```

rather than

```
(%) make ekinit.
```

The second approach is to fix the names of userinit and userexec so that the "make" command will recognize the new names. These changes are more pervasive through the user work area:

(1) rename the files userinit.F and userexec.F and associated files:

```
(%) cd ../local13/XXXX/main
(%) mv userinit.F ekinit.F (or whatever new name is to be used)
```
(%) mv userinit.o ekinit.o
(%) mv userexec.F ekexec.F
(%) mv userexec.o ekexec.o

(2) update "makefile" to reflect new names
(%) cd ...
    local13/XXXX/install

Edit file "makefile". Change all references of userinit to the new name, i.e., "ekinit". Similarly, change all references of userexec to the new name (such as "ekexec").

(3) remake executables:
    (%) make ekinit

and
    (%) make ekexec

for example.

User access to the executables

For testing purposes, a user may wish to access the executables from some directory other than the directory where the executables are stored. This access can be made by (1) placing the development run directory into the PATH environment variable list, (2) placing the executables into an accessible directory, or (3) placing symbolic links pointing to the executables into an accessible directory. Any of these methods is acceptable.

Reducing the size of the executables

The executables userinit and userexec contain all processors provided by SIERRASEIS, IRIS-SEIS, and LOCAL-SEIS. The two executables will reside on about 5 Mbytes of disk. For developmental purposes, the user may find that only a few processors are needed (input, new processor, and output/display). It is possible to link only the subroutines for these processors with the SIERRASEIS libraries in order to generate executables tailored only for the processors needed. The size of these tailored executables will be smaller than the fully functional executables.
V-C. Developing a New Processor in the Work Area

The development of a new processor has two phases: initial addition to the local work version of LOCAL-SEIS and the update of existing processor subroutines. Both phases are similar to those needed to add a processor to the "official" version of LOCAL-SEIS and are described below.

Adding a new processor

The addition of a new processor to a development version of LOCAL-SEIS requires seven steps. These steps involve construction of the processor initialization and execution subroutines along with slight modifications to the structure of the work version of LOCAL-SEIS. Shell files are provided within ..\local13/XXXX/install to help implement the modifications.

The reference to file names and executables below will use the names userinit and userexec. However, customized names if used should be inserted in their places.

Briefly, the steps needed to add a new processor are:
1. Select new, unused names for the processor and the associated initialization and execution subroutines.
2. Add these names to the PROCUSER.LIS file.
4. Create the source code for the initialization and execution subroutines.
5. Compile the initialization and execution subroutines.
6. Remake the userseis.a archive library.
7. Remake the executables userinit and userexec.

1. Select new names for processor and associated subroutines.

The naming convention for a new processor is identical to the convention for LOCAL-SEIS. See section IV-C for a full discussion regarding the names for a new processor and its associated initialization and execution subroutines.

2. Add new names to PROCUSER.LIS file

The new names of the processor and the initialization and execution subroutines should be entered into the PROCUSER.LIS file within ..\local13/XXXX/run. The initial contents of PROCLOC.LIS is:

```
processor: init name: exec name:
(LOCAL FILES)
NULLMODU NULUM0 NULUMX
```

The null processor has empty subroutines which exist within ../local13/XXXX/seismic.

Suppose a new processor is /FFTPLOT. The initialization and execution subroutines are named "fftpl0" and "fftplx", respectively. The PROCUSER.LIS file will look like:

```
processor: init name: exec name:
(LOCAL FILES)
NULLMOD NULLM0 NULLMX
FFTPLOT FFTPL0 FFTPLX
```


The newly modified PROCUSER.LIS file is used to create the lookup subroutines "userlkinit.F" and "userlkexec.F". The executable usergntabl is used to create these subroutines. The shell file makelk located in ../local13/XXXX/install can be used to invoke usergntabl:

```
(%) cd ../local13/XXXX/install.
```
The shell file will place the lookup subroutines into \texttt{./local13/XXXX/run} and will also compile the subroutines in order to create object code.

4. Create the source code for initialization and execution subroutines.

The initialization and execution subroutines should be written according to rules as discussed in sections IV-D and IV-E. The format and structure of these subroutines should try to follow the structure of SIERRASEIS subroutines as much as possible; this ensures compatibility. Remember that at least two subroutines are required for each new processor.

Once completed, the subroutines should be placed into directory \texttt{./local13/XXXX/seismic}.

5. Compile the initialization and execution subroutines.

The initialization and execution subroutines can be compiled using the shell compone located in \texttt{./local13/XXXX/install}. For example, to compile subroutines "examp0.F" and "exampx.F":

\begin{verbatim}
(\%) cd ./local13/XXXX/seismic
(\%) ./install/compone examp0.F
(\%) ./install/compone exampx.F
\end{verbatim}

Note the relative directory definition of file "compone" compared to the location of the source code files. Two new object files, examp0.o and exampx.o, will appear.

6. Remake the userseis.a archive library.

Once the new object code exists, the archive library userseis.a needs to be updated:

\begin{verbatim}
(\%) cd ./local13/XXXX/install.
(\%) liball
\end{verbatim}

The liball shell will re-archive the object code within \texttt{./local13/XXXX/seismic} and "ranlib" the archive library.

7. Remake the development LOCAL-SEIS executables.

The development executables, userinit and userexec can be recompiled using the make shell:

\begin{verbatim}
(\%) make userinit
\end{verbatim}

and

\begin{verbatim}
(\%) make userexec.
\end{verbatim}

If the executable names were changed, the new names should be used. The executables should be properly linked and constructed; they should appear in the \texttt{./local13/XXXX/run} directory.

If an object code file is missing, return to the \texttt{./local13/XXXX/seismic} directory to see if it was compiled correctly.

Revising a Processor

If an error occurs in one of the subroutines related to a processor, the programmer may have to fix the code within the subroutine. Upon debugging and revision, the updated subroutine needs to be reinstalled into the proper executable.

To revise an executable, the following steps are needed:

1. Edit the (initialization or execution) subroutine.
2. Compile the subroutine
3. Remake the userseis.a archive library
4. Remake the appropriate executable (userinit or userexec).
1. Edit the (initialization or execution) subroutine.
   
   If a mistake exists within the initialization or execution subroutine, the code should be debugged and revised. The updated subroutine should be placed within .. /local13/XXXX/seismic.

2. Compile the subroutine.
   
   The revised subroutine should be recompiled:
   
   (%) cd .. /local13/XXXX/seismic.
   
   (%) .. /install/compone subroutine.F
   
   Note the relative location of file "compone" relative to the location of the source code subroutine.

3. Remake the userseis.a archive library.
   
   Once the revised object code exists, the archive library userseis.a needs to be updated:
   
   (%) cd .. /local13/XXXX/install.
   
   (%) liball
   
   The liball shell will re-archive the object code within .. /local13/XXXX/seismic and "ranlib" the archive library.

4. Remake the appropriate executable (userinit or userexec).
   
   If the revised subroutine was an initialization subroutine, then remake the initialization executable:
   
   (%) cd .. /local13/XXXX/install.
   
   (%) make userinit
   
   If the revised subroutine was an execution subroutine, then remake the execution executable:
   
   (%) cd .. /local13/XXXX/install.
   
   (%) make userexec
   
   If custom names were installed, use the new names instead. Either of the two new executables will be placed within .. /local13/XXXX/run.

   **Testing a new processor**
   
   A simple processing job should be constructed to test the new processor. The test should examine the validity of both the processor function and the trace flow. Small amounts of data or synthetic seismograms with known GCV values can be used in conjunction with plot displays and header printing.

   WRITE statements within the initialization and execution phase subroutines are useful at the development stage. WRITE statements in the initialization subroutine can be used to verify that input parameters are obtained and translated properly. Care must be exhibited in the execution subroutine regarding WRITE statements; too many WRITE statements for large test data sets will create very large .EPR text files.

   If a new processor works properly, the processor can then be installed into LOCAL-SEIS. If it does not function correctly, make the necessary programming fixes and update the object code, the archive library of processors, and the executables.
## APPENDIX I - IRIS-SEIS Manual Pages

<table>
<thead>
<tr>
<th>Processor</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIRWVMUT</td>
<td>75</td>
</tr>
<tr>
<td>AMPDUMP</td>
<td>77</td>
</tr>
<tr>
<td>AMPMATH</td>
<td>79</td>
</tr>
<tr>
<td>BULKSHFT</td>
<td>81</td>
</tr>
<tr>
<td>CLIPIT</td>
<td>83</td>
</tr>
<tr>
<td>COMVARPR</td>
<td>85</td>
</tr>
<tr>
<td>DEFHEAD</td>
<td>87</td>
</tr>
<tr>
<td>DIPFIL</td>
<td>89</td>
</tr>
<tr>
<td>EASYADD</td>
<td>91</td>
</tr>
<tr>
<td>EXCORTPR</td>
<td>92</td>
</tr>
<tr>
<td>EXMULTRC</td>
<td>94</td>
</tr>
<tr>
<td>EXTRCTRC</td>
<td>96</td>
</tr>
<tr>
<td>FILTER</td>
<td>98</td>
</tr>
<tr>
<td>GATHCNTR</td>
<td>100</td>
</tr>
<tr>
<td>HDMINMAX</td>
<td>102</td>
</tr>
<tr>
<td>HEADEQL</td>
<td>104</td>
</tr>
<tr>
<td>HEADLIST</td>
<td>107</td>
</tr>
<tr>
<td>INTRLEAV</td>
<td>109</td>
</tr>
<tr>
<td>MONITOR</td>
<td>112</td>
</tr>
<tr>
<td>NULENGTH</td>
<td>114</td>
</tr>
<tr>
<td>NUMTRFLO</td>
<td>116</td>
</tr>
<tr>
<td>OKAGC</td>
<td>118</td>
</tr>
<tr>
<td>OMIT</td>
<td>120</td>
</tr>
<tr>
<td>OMITTRAC</td>
<td>122</td>
</tr>
<tr>
<td>PEAKVAL</td>
<td>124</td>
</tr>
<tr>
<td>RMSAMP</td>
<td>126</td>
</tr>
<tr>
<td>ROT</td>
<td>128</td>
</tr>
<tr>
<td>TRCOUNTR</td>
<td>130</td>
</tr>
<tr>
<td>TRTAPER</td>
<td>132</td>
</tr>
<tr>
<td>UNIXIO</td>
<td>134</td>
</tr>
</tbody>
</table>
FUNCTION: Automated Surgical Mute of Air Wave

AIRWVMUT performs automated surgical muting of an air wave. Using the source-receive offset (GCV 'RANGE') and the speed of sound in air, the air wave arrival time is computed by AIRWVMUT. Amplitudes are muted in a window centered about this time. An additional taper zone can be specified on each side of the muted window.

The width of the surgical mute zone can widen with offset to account for the changing shape of the air wave with offset distance. Mute zone widths are specified in RANGE-width pairs (width in milliseconds). For traces whose offset falls between RANGE-width pairs, the mute zone width is interpolated between the nearest given RANGE-width pairs. Traces which fall before the first given RANGE value are not muted. Traces which fall beyond the last given RANGE value has a mute zone width equal to the last specified width value.

The surgical mute zone has an amplitude which is dropped to a relative dB level. By default, amplitudes in the mute zone are 60 dB lower than their original level.

The side tapers are based on a cosine function and range in scaling from 100% beyond the taper zone to the surgical mute zone dB level when the taper reaches the surgical mute zone.

PARAMETERS

Synopsis:

```
/AIRWVMUT
AIRVEL velocity
MUTWIDTH range1 width1 range2 width2...
DBLEVEL dbdrop
TPRWIDTH taperwidth
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIRVEL</td>
<td>velocity</td>
<td>real</td>
<td>Opt 1090</td>
<td>Velocity of sound in air. Default in ft/sec.</td>
</tr>
<tr>
<td>MUTWIDTH</td>
<td>range width</td>
<td>real</td>
<td>Opt --&gt;</td>
<td>defines surgical muting zone by using RANGE-width pairs. Width in msec, RANGE is GCV value. Default is two pairs, (100,250) and (5280, 500). Specify up to 50 pairs. Amplitudes in mute zone are dropped from original amplitude by this amount. Units are in dB. Width of taper zones on either size of surgical mute. Measured in millisec.</td>
</tr>
<tr>
<td>DBLEVEL</td>
<td>dbdrop</td>
<td>real</td>
<td>Opt 60</td>
<td></td>
</tr>
<tr>
<td>TPRWIDTH</td>
<td>taperwidth</td>
<td>real</td>
<td>Opt 100</td>
<td></td>
</tr>
</tbody>
</table>
GLOBAL COMMON VARIABLES
RANGE must be defined prior to using this routine. KSORT type does not matter. GCV’s are not changed.

EXAMPLE or OTHER NOTES
Example of usage:
/DIN FILENAME 'SHOTS'
/AIRWVMUT

Using all defaults, these shot gathers will have air waves muted based on a speed of sound in air of 1090 ft/sec. Offsets less than 100 ft will not be muted. At 100 ft offset, the mute zone is 250 msec; at 5280 ft, the mute zone is 500 msec. Offsets between these distances have mute zones which are interpolated between these mute times. Offsets greater than 5280 have mute zones of 500 msec. Amplitudes in the surgical mute zone are suppressed by 60 dB. Side tapers are 100 msec in width.

/DIN FILENAME 'SHOTS'
/AIRWVMUT
    MUTWIDTH 440. 100. 1000. 300. 8000. 750.
    TPRWIDTH 250.

Traces less than 440 ft offsets are not tapered. Traces between 440 and 1000 ft have mute zones between 100 and 300 msec. Traces between 1000 and 8000 ft have mute zones between 300 and 750 msec. The taper width is 250 msec.

SUBROUTINE NAMES
INIT subroutine: AIRWVO.F
EXEC subroutine: AIRWVX.F
DIRECTORY: .../iris13/seismic

REVISION HISTORY
26jan90  D. Okaya  Initial installment.
**IRIS** **IRIS-SEIS**

<table>
<thead>
<tr>
<th>PROCESSOR: AMPDUMP</th>
<th>User notes__ New XX Changes__</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: January 27, 1990</td>
</tr>
</tbody>
</table>

**FUNCTION:** Print trace amplitudes

AMPDUMP prints the amplitudes of the first NSAMPLE samples of each seismic trace. The amplitudes can be printed in either integer, real, or exponential format, depending on amplitude magnitudes. Integer format uses FORTRAN '115' format. Real numbers are printed using 'F15.5', and exponential numbers use 'E15.7'. The range of amplitudes to be printed must be known, otherwise a printout overflow may occur.

The amplitudes for each trace are identified by a 'gather' and 'trace' number. These values can be any two GCV's, however a pair of GCV's which uniquely identify each trace should be selected (i.e., KSHOT and KTRC or KCDP and KTRACE).

**PARAMETERS**

Synopsis:

```shell
/AMPDUMP

GATHER GCVgath        Opt
TRACE GCVtrace         Opt
NSAMPLES samptoprint  Req
FLOAT none             Opt
INT none               Opt
EXP none               Opt
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>GATHER</td>
<td>GCVgath</td>
<td>chr*6</td>
<td>Opt</td>
<td>KSHOT</td>
<td>Gather type for trace identification.</td>
</tr>
<tr>
<td>TRACE</td>
<td>GCVtrace</td>
<td>chr*6</td>
<td>Opt</td>
<td>KTRC</td>
<td>Trace type for trace identification.</td>
</tr>
<tr>
<td>NSAMPLES</td>
<td>samptoprint</td>
<td>int</td>
<td>Req</td>
<td>none</td>
<td>Number of samples to print.</td>
</tr>
<tr>
<td>FLOAT</td>
<td>none</td>
<td>none</td>
<td>Opt</td>
<td>yes</td>
<td>Flag requesting real format (This is selected by default).</td>
</tr>
<tr>
<td>INT</td>
<td>none</td>
<td>none</td>
<td>Opt</td>
<td>none</td>
<td>Flag requesting integer format.</td>
</tr>
<tr>
<td>EXP</td>
<td>none</td>
<td>none</td>
<td>Opt</td>
<td>none</td>
<td>Flag requesting exponential format.</td>
</tr>
</tbody>
</table>

**GLOBAL COMMON VARIABLES**

GATHER and TRACE GCV's must be defined. None are changed.

**EXAMPLE or OTHER NOTES**

Example of usage:
```
/DIN FILENAME 'DATA'
/AMPDUMP
NSAMPLES 100
EXP
```
TRACE 'RANGE'

First 100 samples of each trace is dumped in exponential format. Each trace is identified by KSHOT and RANGE values.

SUBROUTINE NAMES

INIT subroutine: AMPDM0.F
EXEC subroutine: AMPDMX.F
DIRECTORY: ../iris13/seismic

REVISION HISTORY

11Jul89  D. Okaya  Initial installment.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: AMPMATH</th>
<th>User notes__ New__ Changes XX</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: January 27, 1990</td>
</tr>
</tbody>
</table>

**FUNCTION:** Perform scalar math on amplitudes of seismic traces.

AMPMATH performs scalar math on trace amplitudes. All amplitudes are affected in the same manner. Math operations are as follows:

\[ OUT(t) = (MULTA \times IN(t) + ADDA) \times MULTB + ADDB \]

The input trace \( IN(t) \) is first multiplied by MULTA, then added to ADDA. The net result is then multiplied by MULTB and subsequently added to ADDB.

The use of DIVA and DIVB is similar to MULTA and MULB:

\[ OUT(t) = (IN(t) / DIVA + ADDA) / DIVB + ADDB \]

The MUL and DIV parameters can be used in combination (e.g., MULTA and DIVB).

By default, amplitudes of outgoing traces are the same as those coming in. Analogous scalar math can be formed to trace headers (GCV’s) using HEADEQL.

**PARAMETERS**

Synopsis:

```
/AMPMATH
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTA</td>
<td>float_number</td>
<td>float</td>
<td>Opt</td>
<td>1.0</td>
<td>inner multiplier.</td>
</tr>
<tr>
<td>ADDA</td>
<td>float_number</td>
<td>float</td>
<td>Opt</td>
<td>0.0</td>
<td>inner additive.</td>
</tr>
<tr>
<td>MULTB</td>
<td>float_number</td>
<td>float</td>
<td>Opt</td>
<td>1.0</td>
<td>outer multiplier.</td>
</tr>
<tr>
<td>ADDB</td>
<td>float_number</td>
<td>float</td>
<td>Opt</td>
<td>0.0</td>
<td>outer additive.</td>
</tr>
<tr>
<td>DIVA</td>
<td>float_number</td>
<td>float</td>
<td>Opt</td>
<td>1.0</td>
<td>inner divisor.</td>
</tr>
<tr>
<td>DIVB</td>
<td>float_number</td>
<td>float</td>
<td>Opt</td>
<td>1.0</td>
<td>outer divisor.</td>
</tr>
</tbody>
</table>

**GLOBAL COMMON VARIABLES**

None are affected or used.
EXAMPLE or OTHER NOTES
Example of usage:
/DIN
    FILENAME 'DATA'
/AMPMATH
    MULTA 2.
    ADDA 5000.
    DIVB .25

Incoming trace amplitudes from disk file DATA.DIO are first doubled, then DC-shifted by +5000, after which they are divided by 0.25.

SUBROUTINE NAMES
  INIT subroutine: AMATH0.F
  EXEC subroutine: AMATHX.F
  DIRECTORY: ../iris13/seismic

HISTORY
  6sep88  D. Okaya Initial installment.
  23nov88 D. Okaya Add DIVA/DIVB parameters.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: BULKSHFT</th>
<th>User notes</th>
<th>New XX Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: February 6, 1990</td>
<td></td>
</tr>
</tbody>
</table>

FUNCTION: Whole sample trace shifting (statics)

BULKSHFT performs whole sample statics by shifting trace amplitudes up or down. This processor is faster than STATAPLY since whole (integral) sample shifts are involved; no amplitude interpolations are applied. The specified time shift, SHIFTMS, is rounded to the nearest sample.

A positive time shift pushes the amplitudes later into the trace; a negative time shift pulls the amplitudes up towards \( t=0 \). Samples which are vacated by the shifting have zero amplitude.

PARAMETERS

Synopsis:

```
/BULKSHFT  SHIFTMS time  Req
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/ Opt</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHIFTMS</td>
<td>time</td>
<td>real</td>
<td>Req</td>
<td>0.0</td>
<td>amount of msec to shift traces.</td>
</tr>
</tbody>
</table>

GLOBAL COMMON VARIABLES

GCV's are not affected.

EXAMPLE or OTHER NOTES

```
/IN
  DATA 1 12000 4 96 0
/BULKSHFT
   SHIFTMS 2000.
```

Traces are delayed in time by 2000 msec (500 samples).

REVISION HISTORY

06feb90  dao  Initial installment.
08jun90  dao  modified for SUN/UNIX IRIS-SEIS installation.

SUBROUTINE NAMES

INIT subroutine: BULKS0.F
EXEC subroutine: BULKSX.F
DIRECTORY: ../iris13/seismic
FUNCTION:  Clip excessive amplitudes to a user-specified level

CLIPIT is similar to a despiking routine in that it removes amplitudes above a user-specified acceptable level. The replaced values are set to the user-specified level. Run-time action can be listed in the .EPR file.

PARAMETERS

Synopsis:

```
/CLIPIT
  CLIPAMP level
  DOC none
  NODOC none
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIPAMP</td>
<td>level</td>
<td>real</td>
<td>Opt</td>
<td>100000000.</td>
<td>amplitudes above this level are reset to this level.</td>
</tr>
<tr>
<td>DOC</td>
<td>none</td>
<td>-</td>
<td>Opt</td>
<td>--</td>
<td>print action in .EPR file.</td>
</tr>
<tr>
<td>NODOC</td>
<td>none</td>
<td>-</td>
<td>Opt</td>
<td>--</td>
<td>do not print action in .EPR file (default).</td>
</tr>
</tbody>
</table>

GLOBAL COMMON VARIABLES

No GCV's are affected.

EXAMPLE or OTHER NOTES

```
/IN
   /CLIPIT
     CLIPAMP 1000.
     DOC
```

All trace amplitudes above 1000. are reset to 1000. Amplitudes below -1000. are reset to -1000. Any substitutions are documented in the .EPR file.

SUBROUTINE NAMES

- INIT subroutine: CLIPT0.F
- EXEC subroutine: CLIPTA.F
- DIRECTORY: ../iris13/seismic

REVISION HISTORY

- 00oct88  E. Karageorgi  Initial installment.
- 25nov89  D. Okaya       Changed run-time print statements to user options. Fixed sign preservation.
FUNCTION: Print number of Global Common Variables for this job.

COMVARPR prints the number of GCV’s defined for the job. The job will contain Sierra-defined GCV’s and possibly user-defined GCV’s. COMVARPR will print these numbers once in the .IPR listing. If desired, the numbers can be printed in the .EPR listing, once for each trace that enters the routine. Output will look like:

COMVARPR> NNAMI, NNAMR, NNAM8, NNAM4: 137 71 10 1

NNAMI, NNAMR, NNAM8, and NNAM4 are the number of integer, real, CHAR*8, and CHAR*4 GCV’s, respectively.

This routine performs a different, valuable service. While quite an innocuous routine, it provides a routine which is defined but does not take up any computer resources. There are cases where, for reasons unknown, other processors may need to have something following them. This is particularly true when a processor has string or character arguments. SierraSeis for some reason wants SOMETHING to follow the string/character arguments; COMVARPR provides a valid processor in the .DAT or .IN file yet does not perform operations which will slow the job down.

PARAMETERS
Synopsis:

<table>
<thead>
<tr>
<th>/COMVARPR</th>
<th>EXEC</th>
<th>Opt</th>
</tr>
</thead>
</table>

Parameter  Argument  Fmt  Req/ Opt  Default  Comment
EXEC none --- Opt not used Specify this parameter to get listings in the .EPR file. If not used, then COMVARPR will operate just once in the .IPR file.

GLOBAL COMMON VARIABLES
None used.

EXAMPLE or OTHER NOTES
Example of usage:
/DIN
FILENAME 'DATA'
/HEADLIST
   HEADERS 'KCDP KTRACE KSHOT KTRC'
/COMVARPR

SUBROUTINE NAMES
  INIT subroutine: CVPRI0.F
  EXEC subroutine: CVPRIX.F
  DIRECTORY: ../iris13/seismic

HISTORY
  29aug88  D.Okaya  Initial installment.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: DEFHEAD</th>
<th>User notes__ New XX Changes__</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: November 26, 1988</td>
</tr>
</tbody>
</table>

**FUNCTION:** Define new trace header / Global Common Variable

DEFHEAD allows a new trace header or GCV to be defined for the job. DEFHEAD places the new name into the GCV common blocks (SSCOM.INC) and obtains an index position within the common blocks. The new GCV name is then available for subsequent definition and manipulation (by reference using the index position).

SierraSeis makes a distinction between integer, real, CHAR*8, and CHAR*4 variables. Each new name must be no more than 6 characters long. Alphanumeric characters and symbols are permissible. A fatal error will occur if a name has been previously defined (by Sierra or by the user).

**PARAMETERS**

**Synopsis:**

```
/DEFHEAD
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/ Opt</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>GCV-name</td>
<td>char*6</td>
<td>Opt</td>
<td>none</td>
<td>defines an INT header.</td>
</tr>
<tr>
<td>REAL</td>
<td>GCV-name</td>
<td>char*6</td>
<td>Opt</td>
<td>none</td>
<td>defines a REAL header.</td>
</tr>
<tr>
<td>CHREIGHT</td>
<td>GCV-name</td>
<td>char*6</td>
<td>Opt</td>
<td>none</td>
<td>defines a CHR*8 header.</td>
</tr>
<tr>
<td>CHRFour</td>
<td>GCV-name</td>
<td>char*6</td>
<td>Opt</td>
<td>none</td>
<td>defines a CHR*4 header.</td>
</tr>
</tbody>
</table>

**GLOBAL COMMON VARIABLES**

The new name is entered into the appropriate common block. The appropriate hidden GCV variable NNAMI, NNAMR, NNAM8, or NNAM4 is incremented properly (see COMVARPR for printing these values out).

**EXAMPLE or OTHER NOTES**

Example of usage:

```
/DIN
   FILENAME 'DATA'
/DEFHEAD
   REAL 'DEPTH'
/DEFHEAD
   INTEGER 'P-WAVE'
```
A new REAL header named 'DEPTH' and a new INTEGER header named 'P-WAVE' are defined.

SUBROUTINE NAMES
INIT subroutine: DEFHDO.F
EXEC subroutine: DEFHDX.F
DIRECTORY: ../iris13/seismic

HISTORY
10jun88   D. Okaya       Initial installment.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: DIPFIL</th>
<th>User notes</th>
<th>New XX Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: June 24, 1990</td>
<td></td>
</tr>
</tbody>
</table>

**FUNCTION:** Dip-filtering in the time domain

DIPFIL applies dip-filtering in the time domain. The processor is based on a Butterworth dip-filter algorithm by Hale and Claerbout (GEOPHYSICS, 48, pp. 1033-1038, 1985).

Parameters required are the dip-filter slope in msec/trace, the dominant frequency of the data in Hz, and the selection of low-dip pass or reject.

**PARAMETERS**

Synopsis:

```
/DIPFIL
    DIP dip
    FREQ dom_freq
    MODE passornot
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIP</td>
<td>dip</td>
<td>real</td>
<td>Req</td>
<td>none</td>
<td>Dip of filter edge in msec/trace. Filtering will be between +dip and -dip.</td>
</tr>
<tr>
<td>FREQ</td>
<td>dom_freq</td>
<td>real</td>
<td>Req</td>
<td>none</td>
<td>Dominant frequency of data. passornot=0 is low-pass (reject dipping events outside dip range). passornot=1 is high-pass (reject flat dips).</td>
</tr>
<tr>
<td>MODE</td>
<td>passornot</td>
<td>int</td>
<td>Req</td>
<td>none</td>
<td></td>
</tr>
</tbody>
</table>

**GLOBAL COMMON VARIABLES**

No change to GCV's.

**EXAMPLE or OTHER NOTES**

```
/IN
   DATA 1 6000 4 48 0
/DIPFIL
   DIP 4. FREQ 25. MODE 0
```

The 48-channel shot gathers will be dipfiltered at 4 msec/trace. The input data has a dominant frequency of 25 Hz. Low dips will be passed.
SUBROUTINE NAMES
 INIT subroutine: DIPIN.F
 EXEC subroutine: DIPEX.F
 DIRECTORY: ../../iris13/seismic

REVISION HISTORY
14apr85  D. Okaya  Installed into DISCO from Stanford SEP program.
25apr90  T. Daley  Converted from DISCO routine into SierraSeis routine.
28jul90  D. Okaya  Revised run-time memory allocation code.
FUNCTION: Shell routine for trace in-trace out processing.

EASYADD is a shell routine to allow a user to easily add a processing algorithm. The processor must be a trace in-trace out algorithm and cannot require any run-time parameters; i.e., the algorithm is hard-wired into the subroutines.

For trace-to-trace processing with user parameters, see EXTRCTRC. For multiple trace processing, see EXMULTRC.

PARAMETERS
Synopsis:

/EASYADD
[no parameters]

Parameter | Argument | Fmt | Req/ Opt | Default | Comment
--- | --- | --- | --- | --- | ---
none

GLOBAL COMMON VARIABLES
None modified.

EXAMPLE or OTHER NOTES
This routine can be copied and subsequently modified to construct a new processor which processes traces on an individual manner.

SUBROUTINE NAMES
INIT subroutine: EASYIN.F
EXEC subroutine: EASYEX.F
DIRECTORY: ..../iris3/seismic

REVISION HISTORY
25Jun90 dao Initial installment.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: EXCORTPR</th>
<th>User notes ___ New XX Changes ___</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: February 5, 1990</td>
</tr>
</tbody>
</table>

FUNCTION: Prepare uncorrelated Vibroseis data for extended correlation

EXCORTPR prepares uncorrelated Vibroseis shot gathers for extended correlation. SierraSeis's /VCORR processor will not allow for extended correlation. /EXCORTPR will allow for self-truncating extended correlation by tapering the uncorrelated seismograms, then zero-padding to a sufficient length to allow /VCORR to perform Vibroseis correlation.

The taper at the ends of the uncorrelated seismograms is used to remove a cross-correlation edge effect. A cosine taper is used; the taper length is a run-time parameter.

The length of zero-padding is determined in one of two ways; either explicitly or by computing the needed length based on sweep length and desired extended-correlation length. The relationship between padded uncorrelated length, sweep length, and resulting extended correlation length is

\[ t_{\text{ext-corr}} = t_{\text{uncorr-pad}} - t_{\text{sweep}} \]

The amount of padding which is necessary is the difference between the original uncorrelated trace length and the amount \( t_{\text{uncorr-pad}} \). The amount of padding is also equal to the difference between the original correlated length and the desired extended correlation length. The new uncorrelated trace length \( t_{\text{uncorr-pad}} \) can be explicitly given with the PADTO parameter. Alternatively, one may specify the sweep length and the amount of desired extended correlation time (using then SWEEPLEN and CORRLEN parameters). Given the incoming seismogram trace length, the processor will determine the amount of padding required to produce the desired correlation time.

The actual Vibroseis correlation is conducted using the /VCORR processor.

The only GCV affected by this processor is KNSAMP; if either the PADTO or SWEEPLEN/CORRLEN parameters are given, the uncorrelated seismograms are zero-padded to a longer amount of travel time (i.e., KNSAMP is increased). If none of these parameters are given, the output length is the same as the input; no padding is performed.

PARAMETERS

Synopsis:

/EXCORTPR TPRWIDTH tapermsec Opt
PADTO padmsec Opt
SWEEPLEN sweepmsec Opt
CORRLEN corrmsec Opt
### Global Common Variables

KNSAMP is changed if either PADTO or SWEEPLEN/CORRLEN is specified.

### Example or Other Notes

**/DMX**

**/EXCORTPR**

- TPRWIDTH 400.
- PADTO 30000.

Suppose /DMX outputs uncorrelated data of 24000 msec length with 18000 msec sweeps. Conventional correlation will produce 6000 msec traces. **EXCORTPR** will use a 400 msec cosine taper at 23000-24000 msec, then zero-pad the traces to 30000 msec. Subsequent conventional correlation (/VCORR) will then produce 12000 msec data (30000 msec uncorrelated length - 18000 msec sweep length).

**/DMX**

**/EXCORTPR**

- TPRWIDTH 400.
- SWEEPLEN 18000.
- CORRLEN 12000.

Since SWEEPLEN and CORRLEN are specified, **EXCORTPR** will determine that 30000 msec uncorrelated traces are required for subsequent conventional correlation to produce the requested 12000 msec extend-correlated traces (sweep length plus desired correlation length). The amount of zero-padding which will be performed is the difference between the incoming uncorrelated trace length and the determined 30000 msec. Traces are cosine tapered, then zero-padded.

### Subroutine Names

- INIT subroutine: EXTPR0.F
- EXEC subroutine: EXTPRX.F
- DIRECTORY: ../iris13/seismic

### Revision History

02feb90 dao Initial installment.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: EXMULTRC</th>
<th>User notes__ New XX Changes__</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: June 24, 1990</td>
</tr>
</tbody>
</table>

FUNCTION: Example of Multiple-trace processing

EXMULTRC is an example routine of multiple trace processing. This routine will accumulate a gather of traces, allow for gather processing, then release the traces. The trace flow conditions will allow for partial gathers to be received based on indexing using a GCV given by the GATHNAME parameter.

Other parameters may be defined in the INIT phase subroutine for use in the EXEC subroutine.

By itself, EXMULTRC will accumulate then release a gather of traces. No gather processing will be conducted.

PARAMETERS

Synopsis:

```
/EXMULTRC
GATHNAME gather_GCV
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>GATHNAME</td>
<td>gather_GCV</td>
<td>char*6</td>
<td>Req</td>
<td>none</td>
<td>GCV to key gathering.</td>
</tr>
</tbody>
</table>

GLOBAL COMMON VARIABLES

A GCV must be used to indicate gather changes within the trace flow.

EXAMPLE or OTHER NOTES

/IN

/EXMULTRC

GATHNAME 'KSHOT'

In this example, shot gathers will be accumulated for shot gather processing.

SUBROUTINE NAMES

INIT subroutine: EXMULO.F
EXEC subroutine: EXMULX.F
DIRECTORY: ../../iris13/seismic

REVISION HISTORY

25jan90  dao  Initial installment.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: EXTRCTRC</th>
<th>User notes New XX Changes__</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: June 24, 1990</td>
</tr>
</tbody>
</table>

FUNCTION: Example of trace-to-trace processing

EXTRCTRC is a shell routine which illustrates trace-to-trace processing. Each trace which enters the processor exists prior to the arrival of the next trace.

Examples of single and multiple integer and real parameters are defined.

This routine can be copied and modified in order to install a trace-to-trace routine into LOCAL-SEIS.

PARAMETERS

Synopsis:

```
/EXTRCTRC
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONEINT</td>
<td>int single</td>
<td>int</td>
<td>Opt</td>
<td>none</td>
<td>parameter to get single integer value.</td>
</tr>
<tr>
<td>TWOINT</td>
<td>int1 int2</td>
<td>int</td>
<td>Opt</td>
<td>none</td>
<td>parameter to get two integer values.</td>
</tr>
<tr>
<td>ONEFLOAT</td>
<td>flt single</td>
<td>real</td>
<td>Opt</td>
<td>none</td>
<td>parameter to get single floating point value.</td>
</tr>
<tr>
<td>TWOINT</td>
<td>fl f2 f3 f4 f5</td>
<td>real</td>
<td>Opt</td>
<td>none</td>
<td>parameter to get five floating point values.</td>
</tr>
</tbody>
</table>

GLOBAL COMMON VARIABLES

No GCV’s are modified in this shell routine.

EXAMPLE or OTHER NOTES

None.

SUBROUTINE NAMES

INIT subroutine: EXTRC0.F
EXEC subroutine: EXTRCX.F
DIRECTORY: ../iris13/seismic

REVISION HISTORY

25jan90 dao Initial installment.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: FILTER</th>
<th>User notes</th>
<th>New Changes XX</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: January 20, 1990</td>
<td></td>
</tr>
</tbody>
</table>

FUNCTION: Simple bandpass filter using 4 corner points in frequency domain

FILTER is a simple bandpass filter routine which applies a 4-corner filter in the frequency domain. The four corner points define a boxcar which is cosine tapered on the edges. The filter is applied to the entire trace and is the same for all traces.

The bandpass filter is applied in the frequency domain and uses an FFT routine from Claerbout FGDP (e.g., does not use AP FFT routine).

As an option, the amplitude level of the filtered region beyond the edges of the boxcar taper can be specified. Default value is 60 dB from full amplitude of the boxcar.

PARAMETERS

Synopsis:

```
/FILTER

CORNERS c1 c2 c3 c4
DBLEVEL base_level
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/ Opt</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORNERS</td>
<td>c1 c2 c3 c4</td>
<td>real</td>
<td>Req</td>
<td>none</td>
<td>c1-c4 are four corner points: c1=0% left corner, c2=100% left corner, c3=100% right corner, and c4=0% right corner, with frequencies increasing from left to right.</td>
</tr>
<tr>
<td>DBLEVEL</td>
<td>base_level</td>
<td>real</td>
<td>Opt</td>
<td>60 dB</td>
<td>specifies amplitude drop surrounding boxcar. Boxcar tapers rise from this level to full amplitude of boxcar.</td>
</tr>
</tbody>
</table>

GLOBAL COMMON VARIABLES

No GCV's are modified.

EXAMPLE or OTHER NOTES

Example of usage:

```
/DIN
FILENAME 'DATA'
/FILTER
   CORNERS 10. 15. 50. 55. DBLEVEL 75.
```

In this job, the input data are filtered using a 10-15-50-55 Hz boxcar. The edges are cosine-tapered (10-15 and 50-55 Hz). Beyond the filter boxcar, frequencies are suppressed by 75 dB.
SUBROUTINE NAMES

INIT subroutine: FILTR0.F
EXEC subroutine: FILTRX.F and FILTRX FFT.F
DIRECTORY: ../iris13/seismic

HISTORY

07Jul89  D. Okaya  Initial installment.
10Oct89  D. Okaya  Taper switched from cosine-squared to cosine.
20Jan90  D. Okaya  Added DBLEVEL option.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: GATHCNTR</th>
<th>User notes</th>
<th>New XX Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: June 24, 1990</td>
<td></td>
</tr>
</tbody>
</table>

**FUNCTION:** Adjust KSHOT header keyed on KFLDFN value

GATHCNTR resets KSHOT values which are improperly set by /IN. Due to a self-counting feature in IN, uniform KSHOT/KTRC numbers are created. When irregular gathers are read (i.e., traces are missing), the KSHOT/KTRC numbers do not properly increment but are counted so that each KSHOT gather has KNTR traces/gather. As soon as one irregular gather is reached in the processing stream, all subsequent traces are misidentified.

GATHCNTR resets KSHOT values based on values of KFLDFN. When a change in the KFLDFN value is detected to signify a new field gather, the KSHOT value is updated. All subsequent traces have this new KSHOT value until KFLDFN changes. The KSHOT value can begin at a user-specified value (FIRST) and be updated by a user-specified increment (INCREM).

While GATHCNTR will fix KSHOT values to be compatible with KFLDFN values, TRCOUNTR can be used to fix KTRC values.

For this routine to work properly, KFLDFN must be read correctly by the /IN processor.

**PARAMETERS**

Synopsis:

```
/GATHCNTR
    FIRST first_KSHOT
    INCREM del_KSHOT
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST</td>
<td>first_KSHOT</td>
<td>int</td>
<td>Opt</td>
<td>1</td>
<td>value of KSHOT for first gather in processing stream.</td>
</tr>
<tr>
<td>INCREM</td>
<td>del_KSHOT</td>
<td>int</td>
<td>Opt</td>
<td>1</td>
<td>increment between successive gathers.</td>
</tr>
</tbody>
</table>

**GLOBAL COMMON VARIABLES**

KSHOT is updated based on values of KFLDFN. KTRC values can be changed by using /HEADEQL or /TRCOUNTR.

**EXAMPLE or OTHER NOTES**

/IN
/GATHCNTR
/TRCOUNTR
KSHOT
In this example, irregular gathers incorrectly numbered by /IN are reset by GATHCNTR. New KSHOT numbering starts at 1 and increments by 1. Improper KTRC numbers are subsequently renumbered using the KSHOT option in GRCOUNTR.

/IN
/GATHCNTR
/HEADEQL
  HEADIN 'KFLDTN' HEADOUT 'KTRC'

If the KFLDTN trace numbering is correct and matches the desired KTRC numbering, one may set KTRC equal to KFLDTN without performing the resetting conducted by TRCOUNTR.

SUBROUTINE NAMES
  INIT subroutine: GACNT0.F
  EXEC subroutine: GACNTX.F
  DIRECTORY: ../iris13/seismic

REVISION HISTORY
18dec89 dao Initial installment.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: HDMINMAX</th>
<th>User notes</th>
<th>New XX Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: January 26, 1990</td>
<td></td>
</tr>
</tbody>
</table>

**FUNCTION:** Prints minimum and maximum values of a given GCV/header.

**HDMINMAX** will print the minimum and maximum value of a given GCV or trace header. Any GCV may be searched. All input traces are checked before the extrema are printed into the .EPR file.

On occasion, GCV values for some traces will not be defined and will thus be set to zero. In order to ignore zero values and search for minima above zero, the NONZERO option can be selected to search only for non-zero minima/maxima.

**PARAMETERS**

**Synopsis:**

```
/HDMINMAX
   HEADIN GCVin
   NONZERO none
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEADIN</td>
<td>GCVin</td>
<td>chr*6</td>
<td>Req/Opt</td>
<td>none</td>
<td>name of GCV to check.</td>
</tr>
<tr>
<td>NONZERO</td>
<td>none</td>
<td>none</td>
<td>Opt/Opt</td>
<td>none</td>
<td>Ignore zero values during search.</td>
</tr>
</tbody>
</table>

**GLOBAL COMMON VARIABLES**

The specified GCV is scanned for extrema values. No GCV values are changed.

**EXAMPLE** or **OTHER NOTES**

Example of usage:

```
/DIN FILENAME 'DATA'
/HDMINMAX
   HEADIN RANGE
```

The offset values of all traces are scanned with the minimum and maximum offsets printed in the .EPR file.

**SUBROUTINE NAMES**

- INIT subroutine: HDMIN0.F
- EXEC subroutine: HDMINX.F
- DIRECTORY: ../iris13/seismic

**REVISION HISTORY**

26nov88  D. Okaya  Initial installment.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: HEADEQL</th>
<th>User notes</th>
<th>New XX Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: November 26, 1988</td>
<td></td>
</tr>
</tbody>
</table>

FUNCTION: Set GCV values using another GCV and scalar math.

HEADEQL allows for the assignment of a GCV value using another GCV and scalar math. This module is similar to /PRMODCOM except that more flexible scalar math is possible. The GCV value is assigned using the following math:

\[
GCVout = (MULTA \times GCVin + ADDA) \times MULTB + ADDB
\]

Math is conducted using real operations; if the outgoing GCV is an integer GCV, its value is converted to an integral number at the last possible moment.

A GCV can be operated upon itself.

For scalar or constant definition of a GCV, set MULTA equal to 0.0.

A self-incrementing trace counter exists and can be used to increment the GCV. The resulting trace counter sum/product is then added to the modified incoming GCV value. The trace counter can have its own math performed onto it:

\[
COUNTout = (CMULTA \times COUNTER + CADDA) \times CMULTB + CADDB
\]

where

\[
COUNTER = CSTART + DELC \times (\text{ith trace} - 1)
\]

The resulting output GCV value is then

\[
GCVout = (MULTA \times GCVin + ADDA) \times MULTB + ADDB + COUNTout
\]

To use the counter, specify any of the counter parameters. To get a simple trace counter (1 to ntrace...) requiring no math, use the "COUNTER" parameter.

* To use header math without the counter, do not specify any counter parameter.
* To use just counter math, set the MULTA and MULTB parameters to zero.
* To use both header and counter math, set the appropriate parameters.
* The resulting GCV value is the sum of both header and counter math. To not use either one, the header or counter sum/product can be set to zero.

Due to the commutative property of multiplication, the product of a multiplier with the sum of a GCV value and the trace counter is the same as the sum of the products of the multiplier with both the CGV value and trace counter:

\[
SCALAR \times (GCVin + COUNTER) = SCALAR \times GCVin + SCALAR \times COUNTER.
\]
PARAMETERS

Synopsis:

```
/HEADEQL

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEADIN</td>
<td>GCVin</td>
<td>chr*6</td>
<td>Opt</td>
<td>none</td>
<td>name of GCV to retrieve value from.</td>
</tr>
<tr>
<td>HEADOUT</td>
<td>GCVout</td>
<td>chr*6</td>
<td>Opt</td>
<td>none</td>
<td>name of GCV to transfer value to</td>
</tr>
<tr>
<td>MULTA</td>
<td>multiplier</td>
<td>float</td>
<td>Opt</td>
<td>1.0</td>
<td>inner multiplier</td>
</tr>
<tr>
<td>ADDA</td>
<td>additive</td>
<td>float</td>
<td>Opt</td>
<td>0.0</td>
<td>inner additive</td>
</tr>
<tr>
<td>MULTB</td>
<td>multiplier</td>
<td>float</td>
<td>Opt</td>
<td>1.0</td>
<td>outer multiplier</td>
</tr>
<tr>
<td>ADDB</td>
<td>additive</td>
<td>float</td>
<td>Opt</td>
<td>0.0</td>
<td>outer additive</td>
</tr>
<tr>
<td>COUNTER</td>
<td>none</td>
<td>none</td>
<td>Opt</td>
<td>none</td>
<td>specifies a simple incrementing trace counter is to be added to the header (GCV) math. This counter starts at 1 and increments by one for each trace. initial value of trace counter for first trace. increment value of trace counter per trace inner counter multiplier inner counter additive outer counter multiplier outer counter additive</td>
</tr>
<tr>
<td>CSTART</td>
<td>c_initial</td>
<td>float</td>
<td>Opt</td>
<td>1.0</td>
<td>initial value of trace counter for first trace.</td>
</tr>
<tr>
<td>DELC</td>
<td>c_increm</td>
<td>float</td>
<td>Opt</td>
<td>1.0</td>
<td>increment value of trace counter per trace</td>
</tr>
<tr>
<td>CMULTA</td>
<td>multiplier</td>
<td>float</td>
<td>Opt</td>
<td>1.0</td>
<td>inner counter multiplier</td>
</tr>
<tr>
<td>CADDAA</td>
<td>additive</td>
<td>float</td>
<td>Opt</td>
<td>0.0</td>
<td>inner counter additive</td>
</tr>
<tr>
<td>CMULTB</td>
<td>multiplier</td>
<td>float</td>
<td>Opt</td>
<td>1.0</td>
<td>outer counter multiplier</td>
</tr>
<tr>
<td>CADDAB</td>
<td>additive</td>
<td>float</td>
<td>Opt</td>
<td>0.0</td>
<td>outer counter additive</td>
</tr>
</tbody>
</table>

Note: If no counter parameter (COUNTER, CSTART...) is used, then by default all counter parameters are reset to zero; that is, CSTART, CDEL, CMULTA, CADDAA, CMULTB, and CADDAB equal zero. 

GLOBAL COMMON VARIABLES

Output GCV value is modified from the incoming value. If the output GCV does not already exist, it should be defined by DEHEAD prior to calling this routine.

EXAMPLE or OTHER NOTES

Example of usage:

```
/DIN
   FILENAME 'DATA'
/DEFHEAD
```

FLOAT OFFSET /HEADEQL
HEADIN RANGE
HEADOUT OFFSET
ADDA -5280.

In this job, a new GCV named "OFFSET" is created. Its value is taken from RANGE and is modified by -5280.

/DIN
FILENAME 'DATA'
/DEFHEAD
FLOAT KM
/HEADEQL
HEADIN KCDP
HEADOUT KM
MULTA .05
CSTART 101.
DELC 2.
CMULTA .05

In this job, a new GCV header named "KM" is created. The value of KM is dependent on the KCDP number and an incrementing counter:

\[ KM = 0.05 \times KCDP + 0.05 (101 + 2 \times COUNTER). \]

SUBROUTINE NAMES
INIT subroutine: HDEQL0.F
EXEC subroutine: HDEQLX.F
DIRECTORY: ../iris13/seismic

HISTORY
10jun88 D. Okaya Initial installment
29nov88 D. Okaya Install COUNTER options.
** IRIS ** IRIS-SEIS **

PROCESSOR: HEADLIST  User notes  New XX Changes

By: David Okaya  Manual date: November 27, 1988

FUNCTION: List up to six GCV values per seismic trace

HEADLIST prints the values in the .EPR file of up to six GCV's for each seismic trace. Its function is similar to /PRMODCOM except that the listing is more concise and that user-defined GCV's are accessible.

There is a quirk in the way SierraSeis accepts character string input and this processor encounters it often. If an INIT-phase error is obtained describing an illegal parameter, follow the call to HEADLIST with a call to COMVARPR or with "$EOJ". The quirk in SierraSeis wants to see some additional text in the job description file. COMVARPR is a processor with very little impact on the net seismic job.

PARAMETERS
Synopsis:

```plaintext
/HEADLIST
    HEADERS GCVname1... GCVname6  Req
```

Parameter  Argument  Fmt  Req/ Opt  Default  Comment
HEADERS   GCVname   chr*6  Opt  none  up to six GCV names, entire set should be enclosed in single quotes.

GLOBAL COMMON VARIABLES
The specified GCV's are printed in the .EPR file, one listing per trace.

EXAMPLE or OTHER NOTES
Example of usage:

```plaintext
/DIN
    FILENAME 'DATA'
/HEADLIST
    HEADERS 'KCDP KTRACE CDPX CDPY'
$EOJ
```

SUBROUTINE NAMES
INIT subroutine: HDLST0.F
EXEC subroutine: HDLSTX.F
DIRECTORY: ../irisl3/scismic

HISTORY
9jun88  D. Okaya  Initial installment.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: INTRLEAV</th>
<th>User notes</th>
<th>New XX Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: June 24, 1990</td>
<td></td>
</tr>
</tbody>
</table>

**FUNCTION:** Interleave seismic traces or pad irregular gathers

**INTRLEAV** is a routine which creates regular gathers from gathers of irregular trace numbers. Gathers which are short from an expected number of traces are back-padded with zero traces. In addition, traces can be interleaved if stored in consecutive separate half-gathers.

The interleaving is performed by storing traces in memory - much faster than using temporary disk space. Up to 700 traces per gather can be accommodated. The incoming gathers should be divided into two equally-sized halves.

Examples for padding or interleaving are given below.

**PARAMETERS**

**Synopsis:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/ Opt</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITKIP</td>
<td>skip0</td>
<td>int</td>
<td>Opt</td>
<td>0</td>
<td>number of traces in incoming gather to not use.</td>
</tr>
<tr>
<td>FIRSTNX</td>
<td>nx1</td>
<td>int</td>
<td>Opt</td>
<td>KNTR/2</td>
<td>Number of traces in first half of output gather.</td>
</tr>
<tr>
<td>FIRSTTRC</td>
<td>first1</td>
<td>int</td>
<td>Opt</td>
<td>1</td>
<td>Position of first output trace within input gather.</td>
</tr>
<tr>
<td>FIRSTINC</td>
<td>inc1</td>
<td>int</td>
<td>Opt</td>
<td>1</td>
<td>Increment within input gather to obtain next output trace.</td>
</tr>
<tr>
<td>SECNDNX</td>
<td>nx2</td>
<td>int</td>
<td>Opt</td>
<td>KNTR/2</td>
<td>Number of traces in second half of output gather.</td>
</tr>
<tr>
<td>SECNDTRC</td>
<td>first2</td>
<td>int</td>
<td>Opt</td>
<td>1</td>
<td>Position of first trace in second half of output gather within input gather.</td>
</tr>
<tr>
<td>SECNDINC</td>
<td>inc2</td>
<td>int</td>
<td>Opt</td>
<td>1</td>
<td>Increment within input gather to obtain next output trace for second half.</td>
</tr>
<tr>
<td>OUTNX</td>
<td>outkntr</td>
<td>int</td>
<td>Opt</td>
<td>none</td>
<td>number of desired output traces if not equal to input KNTR.</td>
</tr>
</tbody>
</table>
GLOBAL COMMON VARIABLES

GCV's are affected in following manner:

KFLDFN: preserved as is.
KFLDTN: preserved as is.
KSHOT: preserved as is.
KTRC: renumbered so that output traces are consecutive starting at 1.
KNTR: same as input KNTR unless OUTNX parameter is given.
KRECS: set to output KNTR value.
KTRECS: set to output KNTR * KNSHOT.
KMTRC: set to new KTRECS.

All other GCV's are preserved.

EXAMPLE or OTHER NOTES

To pad irregularly sized gathers:

/IN
  DATA 1 6000 4 96 0
  /GATHCNTR
  /TRCOUNTR KSHOT
  /INTRLEAV
    FIRSTNX 48 FIRSTTRC 1 FIRSTINC 1
    SECDNX 48 SECDTRC 49 SECDINC 1

/IN will create uniform KSHOT/KTRC gathers using 96 traces per gather, even if the field gathers do not necessarily have 96 traces/gather. /GATHCNTR will first properly set KSHOT to change when KFLDFN changes. /TRCOUNTR will reset KTRC to begin recounting at one whenever KSHOT changes. The sequence of routines will result in proper (irregularly-sized) KSHOT-KTRC numbering.

The processor /INTRLEAV will detect and pad irregularly-sized gathers. The parameters describe that each output gather is composed of two halves of 48 traces each. The first output half is obtained from the input gather by choosing the first 48 consecutive traces beginning at trace 1. The second output half is obtained by choosing the next 48 consecutive input traces starting at trace 49. The KTRC headers are renumbered starting at 1.

To pad irregularly sized gathers which are nominally two traces too large:

/IN
  DATA 1 6000 4 98 0
  /GATHCNTR
  /TRCOUNTR KSHOT
  /INTRLEAV
    INITSKIP 2
    FIRSTNX 48 FIRSTTRC 3 FIRSTINC 1
    SECDNX 48 SECDTRC 51 SECDINC 1

In this case, due to magnetic tape irregularities (i.e., SEG-D format rather than SEG-Y format), two file header records are read by /IN to be data traces (resulting in 98 traces/gather instead of 96 traces/gather). The INITSKIP parameter allows us to skip these first two "traces", outputting the desired 96 traces in proper order.

To interleave traces:

/IN
  DATA 1 6000 4 96 0
  /INTRLEAV
    FIRSTNX 48 FIRSTTRC 1 FIRSTINC 2
    SECDNX 48 SECDTRC 2 SECDINC 2

The first input trace is output trace 1. The second input trace is output trace 3. The 48th input
trace is output trace 95. The 49th input trace (first trace in second set of 48) is output trace 2. The 50th input trace (second trace in second set of 48) is output trace 4.

SUBROUTINE NAMES
INIT subroutine: INRLV0.F
EXEC subroutine: INRLVX.F
DIRECTORY: ../iris13/seismic

REVISION HISTORY
19oct89 D. Okaya Initial installation.
20nov89 D. Okaya fix for EOD condition (partial gather release).
23nov89 D. Okaya redo situation flagging.
20jan90 D. Okaya modify to allow partial gather input with full gather output (input halves can be less than desired output).
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: MONITOR</th>
<th>User notes__ New XX Changes__</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: November 27, 1988</td>
</tr>
</tbody>
</table>

**FUNCTION:** Monitor the progress of a seismic job

MONITOR allows the user to externally monitor the status of a seismic job. The emplacement of MONITOR in a job allows the user to identify the gather and trace that is currently being processed by the seismic job.

MONITOR stores the gather and trace number of the current seismic trace into a temporary disk file which is either named "MONITOR.TMP" or is named by the user. This file will contain the GCV names of the gather and trace and then the gather and trace numbers.

While a seismic job is running, the user can monitor the job by running an external program named "MONITOR" or 'monitor'. This job will read and display the contents of MONITOR.TMP or whatever file the user has named.

**PARAMETERS**

Synopsis:

```
/MONITOR
FILENAME filename
GATHER GCVname
TRACE GCVname
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILENAME</td>
<td>filename</td>
<td>chr*40</td>
<td>Opt</td>
<td>'MONITOR.TMP'</td>
<td>diskfile to store gather and trace information. Enclose in single quotation marks.</td>
</tr>
<tr>
<td>GATHER</td>
<td>GCVname</td>
<td>chr*6</td>
<td>Opt</td>
<td>'KSHOT'</td>
<td>GCV name of gather grouping of traces.</td>
</tr>
<tr>
<td>TRACE</td>
<td>GCVname</td>
<td>chr*6</td>
<td>Opt</td>
<td>'KTRC'</td>
<td>GCV name of trace grouping of traces.</td>
</tr>
</tbody>
</table>

**GLOBAL COMMON VARIABLES**

Gather and trace GCV's are stored in diskfile read by external program called 'MONITOR'.

**EXAMPLE or OTHER NOTES**

Example of usage:

```
/DIN
FILENAME 'DATA'
/MONITOR
FILENAME '/scr/okaya/MONITOR.TMP'
```
GATHER 'KSHOT'
TRACE 'RANGE'

While each trace is processed during this seismic job, the GCV values stored in KSHOT and RANGE are stored in the diskfile MONITOR.TMP. The external program MONITOR will read the disk file, printing the KSHOT and RANGE values. A screen update is printed each time the KSHOT or RANGE value is changed.

SUBROUTINE NAMES
INIT subroutine: MONITO.F
EXEC subroutine: MONITX.F
DIRECTORY: ../iris13/seismic

HISTORY
25nov88  D. Okaya  Initial installment.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: NULENGTH</th>
<th>User notes</th>
<th>New XX Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: August 24, 1988</td>
<td></td>
</tr>
</tbody>
</table>

**FUNCTION:** New Trace Length

NULENGTH changes trace lengths. Number of samples per trace (KNSAMP) changes accordingly. Subsequent processors in both INIT and EXEC run-streams reflect the new sample rate.

If the outgoing trace length is less than the incoming trace length, the trace is simply truncated. If the outgoing trace is longer than the incoming trace, the padded portion is zero'd.

**PARAMETERS**

/NULENGTH
   NEWTIME time

where time is the new trace length in millisec.

**GLOBAL COMMON VARIABLES**

KNSAMP is changed to the new number of samples/trace.

**EXAMPLE or OTHER NOTES**

/IN
   DATA 1 6000. 4 96 0
   /NULENGTH
   NEWTIME 8000.

In this example, 6 sec traces within 96-trace shot gathers are lengthened to 8 sec.

A similar way to apply this processor is to use /PRMDCOM twice, one each for the INIT and EXEC run streams.

/IN

/PRMDCOM
   INITPR
   MODIFY KNSAMP SET nsample

/PRMDCOM
   MODIFY KNSAMP SET nsample

**SUBROUTINE NAMES**

INIT subroutine: NULEN0.F
EXEC subroutine: NULENX.F

**HISTORY**

24aug88 D. Okaya Initial installment.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: NUMTRFLO</th>
<th>User notes_ New XX Changes__</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: June 24, 1990</td>
</tr>
</tbody>
</table>

FUNCTION: Count number of traces in trace flow

NUMTRFLO determines the number of traces which pass through a specific portion of the job's trace flow. The total number of traces which pass through NUMTRFLO is printed into the .EPR file at the completion of the entire job. NUMTRFLO can be used to determine the expected size of output tape or disk files or to monitor the correctness of trace flow modified by /AUX or trace omitting.

NUMTRFLO can be effectively used by simulating a large job using the /GENCSR or /GENCDP routines. For example, to determine the number of traces flowing out of a CDP-sort job, one can define very short dummy traces using /GENCSR, apply geometry, sort using /GATHER, then run the traces through NUMTRFLO. The total number of output traces can be then used for the real CDP-sort job (i.e., /DOUT NTRACES parameter).

PARAMETERS
Synopsis:

```
/NUMTRFLO [none]
```

Parameter | Argument | Req/ Opt | Default | Comment
---|---|---|---|---
None | | | | 
T] | | | | 

GLOBAL COMMON VARIABLES
None are used.

EXAMPLE or OTHER NOTES
/GENCSR BI 1 EI 100 KNTR 48 KNSAMP 10 SR 4
/GEOGRAPHY USE
/GATHER
/NUMTRFLO
$EOJ

Given the 100 48-channel shot gathers, NUMTRFLO will indicate the number of traces which will be contained in the CDP gathers produced by /GATHER.

SUBROUTINE NAMES
INIT subroutine: NUMTR0.F
EXEC subroutine: NUMTRX.F
DIRECTORY: ../iris13/seismic

REVISION HISTORY
02may90 D. Okaya Initial installment.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: OKAGC</th>
<th>User notes__ New XX Changes__</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual Date: November 27, 1988</td>
</tr>
</tbody>
</table>

FUNCTION: Automatic Gain Control

OKAGC applies an AGC function to seismic traces. The Sierra-provided AGC processor can leave traces not balanced in the time direction. OKAGC applies an AGC function which is more severe than /AGC but yields traces which are better balanced in the time direction.

PARAMETERS

Synopsis:

```
/OKAGC
    WINDOW length
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/ Opt</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>WINDOW</td>
<td>length</td>
<td>float</td>
<td>Req</td>
<td>none</td>
<td>Window length in milliseconds for AGC moving gate.</td>
</tr>
</tbody>
</table>

GLOBAL COMMON VARIABLES

No GCV's are used or modified

EXAMPLE or OTHER NOTES

Example of usage:

```
/DIN
    FILENAME 'DATA'

/OKAGC
    WINDOW 1000.

/DISPLAY
```

SUBROUTINE NAMES

INIT subroutine: OKAGC0.F
EXEC subroutine: OKAGCX.F
DIRECTORY: ../iris13/seismic

HISTORY

11oct88  D. Okaya  Initial installment.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: OMIT</th>
<th>User notes__ New XX Changes__</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: January 26, 1990</td>
</tr>
</tbody>
</table>

**FUNCTION:** Physically omit blocks of traces from the processing stream

OMIT removes traces from the processing stream. Traces which are omitted do not pass through subsequent processors. Traces to omit are identified by a GCV value.

Trace selection for omitting is performed in one of two ways: either by specifying a range of GCV values or by listing the GCV's of the traces to omit. One GCV type and one type of trace selection can be specified per call to /OMIT.

If a range of header values is specified, the range can be used with either inclusive or exclusive logic. To omit traces with CDP number between 400 and 500, inclusive logic will omit traces between and equal to 400-500, while exclusive logic will omit traces between 400-500 but will keep CDP's at these boundary values.

OMIT is best used to drop blocks of traces (i.e., entire gather or a consecutive range of a particular GCV). For selective trace omitting, OMITTRAC may be a better routine to use.

**PARAMETERS**

Synopsis:

```
/OMIT
   HEADER GCVname    Req
   RANGE minlimit maxlimit  Opt
   LOGIC logictype  Opt
   LIST value1 value2...  Opt
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEADER</td>
<td>GCVname</td>
<td>chr*6</td>
<td>Opt</td>
<td>none</td>
<td>GCV name to key trace selection.</td>
</tr>
<tr>
<td>RANGE</td>
<td>minlimit</td>
<td>real</td>
<td>Opt</td>
<td>0.999999999</td>
<td>Range of header values to omit.</td>
</tr>
<tr>
<td></td>
<td>maxlimit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOGIC</td>
<td>logictype</td>
<td>chr*8</td>
<td>Opt</td>
<td>INCLUSIV</td>
<td>INCLUSIV or EXCLUSIV logic for RANGE option.</td>
</tr>
<tr>
<td>LIST</td>
<td>value1 value2</td>
<td>real</td>
<td>Opt</td>
<td>none</td>
<td>Up to 100 GCV values for trace selection.</td>
</tr>
</tbody>
</table>

**GLOBAL COMMON VARIABLES**

A GCV is used to identify traces to omit. These traces are dropped from the processing stream. While the number of traces in post-OMIT processing is decreased, the hidden GCV values such as KRECS and NTRECS are not modified (as total number is not know until after all traces are processed).
EXAMPLE or OTHER NOTES

Example of usage:
/DIN FILENAME 'DATA'
/OMIT
  HEADER KFLDFN
  RANGE 900 999
  LOGIC INCLUSIV

All traces with KFLDFN between 900-999 (including 900 and 999) are omitted. A true situation, where 900-999 represent daily test field records before production collection.

SUBROUTINE NAMES

  INIT subroutine: OMITIN.F
  EXEC subroutine: OMITEX.F
  DIRECTORY: ../iris13/seismic

REVISION HISTORY

18dec89  D. Okaya  Initial installment.
21jan90  D. Okaya  Add LIST option.
** IRIS ** ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: OMITTRAC</th>
<th>User notes</th>
<th>New XX Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: June 24, 1990</td>
<td></td>
</tr>
</tbody>
</table>

FUNCTION: Physically omit individual traces from a processing stream.

OMITTRAC, like OMIT, will drop traces from a processing stream. OMITTRAC will allow for selection of particular traces while OMIT is intended to be used to drop blocks of traces.

Trace omission is accomplished by selecting two GCV's as gather and trace keys, then identifying each trace to omit by providing the "gather" and "trace" numbers. Within a given gather, a range of traces can be omitted by providing the beginning and ending traces. One may also select (inclusive or exclusive) trace logic to deal with the boundaries. Up to 100 sets of gather/trace edits may be given per OMITTRAC call.

PARAMETERS

Synopsis:

```
/OMITTRAC
  GATHNAME GCV_gath    Req
  TRACNAME GCV_trace   Req
  LOGIC logic         Opt
  GATHER num_gath     Req
  TRACE trace1 trace2  Req
```

Parameter | Argument | Fmt | Req/ | Default | Comment
-----------|----------|-----|-------|---------|-------------------
GATHNAME   | GCV_gath | char*6 | Opt  | none    | Name of GCV for gather keying.
TRACNAME   | GCV_trace| char*6 | Opt  | none    | Name of GCV for trace keying.
LOGIC      | logic    | char*8 | Opt  | INCLUSIV| INCLUSIV or EXCLUSIV logic for trace boundaries.
GATHER     | num_gath | float  | Opt  | none    | Value of "gather" to drop traces.
TRACE      | trace1   | float  | Opt  | none    | Range of "traces" to drop. For single trace, specify only trace1. For a range of consecutive traces, specify both trace1 and trace2.

GLOBAL COMMON VARIABLES

Two (integer or real) GCV's are to be used to identify traces. While the total number of traces in the job flow will be reduced, the GCV's KTRECS, KMTRC are not reduced. These may need to be externally reduced using /PRMODCOM or /HEADEQL.
For the fifty 48-channel shot gathers, the following traces will be dropped: shot 102, traces 15-28; shot 120, traces 1-48; shot 138, traces 2 and 13. Trace logic by default is inclusive.

In this case, trace indexing is by KSHOT and offset distance (RANGE). The traces with inner offset distances of 0-330 will be dropped.

SUBROUTINE NAMES
- INIT subroutine: OMITRO.F
- EXEC subroutine: OMITRX.F
- DIRECTORY: .. ./iris13/seismic

REVISION HISTORY
01may90  D. Okaya  Modified from /OMIT to allow for gather/trace selection.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: PEAKVAL</th>
<th>User notes__ New XX Changes__</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: June 24, 1990</td>
</tr>
</tbody>
</table>

FUNCTION: Print Minimum/Maximum Amplitude Per Trace

PEAKVAL will print the minimum and maximum amplitude encountered for each trace. Traces are referenced according to the type of gather (KSORT) the data is sorted. Output amplitudes are in exponential format.

By default, all samples of each trace is examined. To examine a window of data, use the WINDOW parameter.

PARAMETERS
Synopsis:

```
/PEAKVAL WINDOW time1 time2 Opt
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/</th>
<th>Opt</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>WINDOW</td>
<td>time1 time2</td>
<td>real</td>
<td>Opt</td>
<td>none</td>
<td></td>
<td>start and end times of window (in msec).</td>
</tr>
</tbody>
</table>

GLOBAL COMMON VARIABLES

GCV's are used to reference traces according to the data sort type (KSORT). No modification is performed.

EXAMPLE or OTHER NOTES

/IN
/PEAKVAL

For each trace passed by /IN, the minimum and maximum amplitude will be listed in the .EPR file.

SUBROUTINE NAMES

INIT subroutine: PEAKV0.F
EXEC subroutine: PEAKVX.F
DIRECTORY: ../iris13/seismic

REVISION HISTORY

01jun88  D. Okaya  Modified from SASTT0 to print just peak amplitudes.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: RMEAN</th>
<th>User notes</th>
<th>New XX Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: Jim Fowler</td>
<td>Manual date: October 2, 1990</td>
<td></td>
</tr>
</tbody>
</table>

FUNCTION: Remove mean from REFTEK seismograms.

RMEAN removes mean from REFTEK data.

PARAMETERS

Synopsis:

```
/RMEAN
```

```
Parameter    Argument  Fmt  Req/Opt  Default  Comment
none
```

GLOBAL COMMON VARIABLES

No changes.

EXAMPLE or OTHER NOTES

Example of usage:

```
/DIN FILENAME 'SHOTS'
/RMEAN
```

SUBROUTINE NAMES

- INIT subroutine: RMEAN0.F
- EXEC subroutine: RMEANX.F
- DIRECTORY: ../iris13/seismic

REVISION HISTORY

21sep89  J. Fowler Initial creation.
26jan90  D. Okaya QC and installment into IRIS-SEIS.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: RMSAMP</th>
<th>User notes</th>
<th>New</th>
<th>Changes XX</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: November 26, 1989</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FUNCTION: Compute Absolute or RMS-Amplitude for entire trace, set of traces

RMSAMP computes the RMS-amplitude for each seismic trace that enters the processor. Also computed is a running RMS-amplitude for the set of traces up to the current trace. The running RMS-amplitude computed for the last trace will represent the RMS-amplitude for the entire set of traces (gather or section) passed into RMSAMP.

RMSAMP will list for each trace the trace and running RMS amplitudes in the .EPR file. The identification of each trace is provided by gather and trace numbers; the user can specify which two GCV’s to use for trace identification (default is KSHOT and KTRC).

The user can also specify that only positive amplitudes be used for the RMS-amplitude; negative numbers are internally set to zero before the RMS-amplitude computation. Normalization still counts these samples; that is, each trace's mean-square amplitude is normalized by KNSAMP before the square root it taken. Traces released by RMSAMP do not have zero amplitudes but contain the original incoming amplitudes.

As an alternative choice, the absolute value of the amplitudes can be examined instead of the RMS-amplitude.

PARAMETERS

Synopsis:

```
/RMSAMP
  NONEG
  GATHER GCVname
  TRACE GCVname
  ABSAMP
```

Parameter | Argument | Fmt | Req/ | Default | Comment
--- | --- | --- | --- | --- | ---
NONEG | none | -- | Opt | not used | Specifies negative amplitudes are zero’d before computing the RMS amplitudes.
GATHER | GCVname | C*6 | Opt | 'KSHOT' | name of GCV to use for gather indexing in .EPR file.
TRACE | GCVname | C*6 | Opt | 'KTRC' | name of GCV to use for trace indexing in .EPR file.
ABSAMP | none | -- | Opt | not used | Use Absolute amplitude rather than RMS-amplitude.

GLOBAL COMMON VARIABLES

Two GCV’s are used for .EPR trace identification; no GCV values are modified.
EXAMPLE or OTHER NOTES
Example of usage:
/DIN
    FILENAME 'DATA'
/RMSAMP
    GATHER 'KCDP'
    TRACE 'RANGE'
    NONEG

This example will read the disk file DATA.DIO and compute the RMS-amplitude for each trace. A running RMS-amplitude for all traces is also computed. The .EPR file will reference each trace by printing its KCDP and RANGE values. No negative numbers are used in the RMS-amplitude computation.

SUBROUTINE NAMES
    INIT subroutine: RMSAM0.F
    EXEC subroutine: RMSAMX.F
    DIRECTORY: ../iris13/seismic

HISTORY
    25nov88    D. Okaya    Initial installment.
    17dec88    D. Okaya    Install examination of regular amplitudes.
**IRIS** **IRIS-SEIS**

<table>
<thead>
<tr>
<th>PROCESSOR: ROT</th>
<th>User notes__ New XX Changes__</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: Tom Daley</td>
<td>Manual date: July, 1988</td>
</tr>
</tbody>
</table>

**FUNCTION:** Rotation of three component data

ROT rotates three component data into a new coordinate system defined by two rotation angles. PHI is measured clockwise from component one and THETA is measured counterclockwise from component three.

In normal use, each group of three traces input are assumed to be vertical, horizontal, and horizontal. They will be output as radial, SV, and SH, respectively.

**PARAMETERS**

Synopsis:

```
/ROT
   LEVEL level_num
   PHI phi_value
   THETA theta_value
```

Parameter  | Argument  | Fmt | Req/ Opt | Default | Comment |
-----------|-----------|-----|----------|---------|---------|
LEVEL      | level_num | real | Req       | -       | each group of three traces has a level number. Maximum is 200. |
PHI        | phi_value | real | Req       | -       | angle from vertical to radial (in degrees). |
THETA      | theta_value | real | Req       | -       | angle from horizontal to SH (in degrees). |

**GLOBAL COMMON VARIABLES**

No common variables are changed.

**EXAMPLE or OTHER NOTES**

/JOB ACCT 'TEST TEST TRCROT CCS'
/DIN
   FILENAME 'VSPSHOT' /ROT
   LEVEL 1 PHI 80.561 THETA 110.134
   LEVEL 2 PHI 82.399 THETA 113.403
   LEVEL 3 PHI 84.047 THETA 111.484
   LEVEL 4 PHI 86.123 THETA 109.921

**SUBROUTINE NAMES**

INIT subroutine: ROTAIN.F
EXEC subroutine: ROTAEX.F
DIRECTORY: ../iris13/seismic

**REVISION HISTORY**

00Jun88 T. Daley - Initial installment
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: TRCOUNTR</th>
<th>User notes</th>
<th>New XX Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: November 26, 1989</td>
<td></td>
</tr>
</tbody>
</table>

FUNCTION: Trace Counter and Adjustment

TRCOUNTR examines the trace index GCV for continuity within the gather GCV. When the gather GCV value changes, signifying a new gather, the trace index GCV is reset to start at 1. The need for this ability comes from /IN's self-counting of traces from 1 to KNTR regardless of a change in gather number. TRCOUNTR fixes the trace index GCV to smoothly increment from 1 whenever a new gather is detected.

Three gather GCV's can be used for checking. The corresponding trace index GCV is used: KFLDFN-KFLDTN, KSHOT-KTRC, KCDP-KTRACE.

Use /INTRLEAV to fill (pad) incomplete gathers after running /TRCOUNTR.

PARAMETERS

Synopsis:

```
/ TRCOUNTR
  KFLDFN
  KSHOT
  KCDP
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/ Opt</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>KFLDFN</td>
<td>none</td>
<td>---</td>
<td>Opt</td>
<td>yes</td>
<td>Use KFLDFN to key changes in KFLDTN.</td>
</tr>
<tr>
<td>KSHOT</td>
<td>none</td>
<td>---</td>
<td>Opt</td>
<td>no</td>
<td>Use KSHOT to key changes in KTRC.</td>
</tr>
<tr>
<td>KCDP</td>
<td>none</td>
<td>---</td>
<td>Opt</td>
<td>no</td>
<td>Use KCDP to key changes in KTRACE.</td>
</tr>
</tbody>
</table>

GLOBAL COMMON VARIABLES

KFLDTN, KTRC, or KTRACE is changed depending on which gather GCV is specified. Use /HEADEQL to set other GCV's accordingly.

EXAMPLE or OTHER NOTES

/IN

/TRCOUNTR KFLDFN

This job will check KFLDTN numbers with respect to KFLDFN, adjusting trace numbers so that whenever a new KFLDFN number is reached, the KFLDTN numbers begin at 1.
This job adjusts KFLDTN to be consistent with changes in KFLDFN. The value of KTRC is set equal to KFLDTN; KSHOT is set equal to KFLDFN.

SUBROUTINE NAMES

INIT subroutine: TRCNT0.F
EXEC subroutine: TRCNTX.F
DIRECTORY: ../iris13/seismic

HISTORY

25nov89  D. Okaya  Initial installment.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: TRTAPER</th>
<th>User notes__ New XX Changes__</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: June 24, 1990</td>
</tr>
</tbody>
</table>

**FUNCTION:** Taper data dropouts

Data dropouts in uncorrelated seismic traces create correlation artifacts which can be removed by tapering the dropouts. **TRTAPER** applies a cosine-squared taper to the edges of the dropout.

User parameters are the minimum length of a dropout in samples to taper (ZEROSMPL) and the taper width as a percent of the overall trace length (TAPERPCT).

Run-time documentation on any tapering can be obtained by using the DOC parameter.

**PARAMETERS**

Synopsis:

```
/RTTAPER
  ZEROSMPL num_samp Opt
  TAPERPCT pct Opt
  NODOC [none] Opt
  DOC [none] Opt
```

Parameter | Argument       | Fmt | Req/Opt | Default | Comment |
----------|----------------|-----|---------|---------|---------|
ZEROSMPL  | num_samp       | int | Opt     | 10      | number of samples over which a dropout is tapered. |
TAPERPCT  | pct            | real| Opt     | 10%     | taper length as a percent of overall trace length (KNSAMP). |
NODOC     | none           | --- | Opt     | none    | Do not print run-time documentation into the .EPR file. |
DOC       | none           | --- | Opt     | none    | Print run-time documentation into the .EPR file. |

**GLOBAL COMMON VARIABLES**

GCV's are not affected.

**EXAMPLE or OTHER NOTES**

```
/IN
/RTTAPER
  ZEROSMPL 50
  TAPERPCT 15.
```

Dropouts which are 50 samples or greater in width are sampled. The taper width on either end is 15% of the overall trace length.
SUBROUTINE NAMES

INIT subroutine: TRTAP0.F
EXEC subroutine: TRTAPX.F
DIRECTORY: ../iris13/seismic

REVISION HISTORY

13jan89  E. Karageorgi  Initial installation.
25nov89  D. Okaya   installed DOC/NODOC option.
** IRIS ** IRIS-SEIS **

<table>
<thead>
<tr>
<th>PROCESSOR: UNIXIO</th>
<th>User notes</th>
<th>New Changes XX</th>
</tr>
</thead>
<tbody>
<tr>
<td>By: David Okaya</td>
<td>Manual date: July 28, 1990</td>
<td></td>
</tr>
</tbody>
</table>

FUNCTION: Read or write seismic data using UNIX binary internal format

UNIXIO reads or writes seismic data to or from a UNIX binary file. A UNIX binary file contains seismic traces in contiguous manner in either 32-bit floating point or 16-bit integer format. If no trace headers (GCV's) are stored, the seismic data is stored essentially as a two-dimensional block of data (samples vs. arrays). This format is compatible with C language I/O routines.

The user can decide to save SierraSeis trace headers (GCV's) or to store the data with no trace headers. If GCV's are stored, traces are contained in the disk file in contiguous header-trace, header-trace format.

File I/O is handled using the CREAD/CWRITE functions which is compatible with C language file I/O. Using this type of file I/O allows for faster I/O to be performed.

UNIXIO allows for the following I/O combinations:

<table>
<thead>
<tr>
<th>I/O operation</th>
<th>GCV Presence</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A) READ</td>
<td>NO GCV's</td>
<td>REAL*4</td>
</tr>
<tr>
<td>(B) READ</td>
<td>NO GCV's</td>
<td>INT*2</td>
</tr>
<tr>
<td>(C) READ</td>
<td>GCV's</td>
<td>REAL*4</td>
</tr>
<tr>
<td>(D) READ</td>
<td>GCV's</td>
<td>INT*2</td>
</tr>
<tr>
<td>(E) WRITE</td>
<td>NO GCV's</td>
<td>REAL*4</td>
</tr>
<tr>
<td>(F) WRITE</td>
<td>NO GCV's</td>
<td>INT*2</td>
</tr>
<tr>
<td>(G) WRITE</td>
<td>GCV's</td>
<td>REAL*4</td>
</tr>
<tr>
<td>(H) WRITE</td>
<td>GCV's</td>
<td>INT*2</td>
</tr>
</tbody>
</table>

- To read data from disk, use the READ parameter.
- To write data to disk, use the WRITE parameter.
- To read or write GCV's along with data, use the USEGCV parameter.
- To not read or write GCV's, use the NOGCV parameter.
- To read or write data traces in 32-bit floating point, use FORMAT 32.
- To read or write data traces in 16-bit integers, use FORMAT 16.

A more complete description of the UNIXIO file format (with or without GCV's) is given below.

The FILENAME parameter must be given. A filename suffix is not automatically attached (unlike the '.DIO' from /DOUT). A suggested suffix to use is '.UIO'.

Writing data to disk

For the WRITE option, all traces which enter UNIXIO are written to the requested disk file. The only parameters needed to write a diskfile are: FILENAME, WRITE, USEGCV or NOGCV, and FORMAT. By default, FORMAT assumes 4-byte (32-bit) floating point numbers.

A diskfile written with the USEGCV parameter will contain a diskfile header and trace headers. This format is described below. To decode the GCV header information from a diskfile, use the stand-alone utility program 'UIOLength'. This external program will dump the number of GCV's, list any user-defined GCV names, and then indicate the trace sample length and number of traces stored in the file.
A diskfile written with the NOGCV parameter will have neither a diskfile header nor trace headers. The seismic traces will be written in sequential order so that the first sample of one trace directly follows the last sample of the previous trace. This format allows one to apply external processing to the data and feed it back in. Reading data with the NOGCV option also allows one to create uniform traces from continuously recorded data.

**Reading data from disk**

For the READ option, one must specify the FILENAME, READ, USEGCV or NOGCV, and FORMAT parameters. By default, the data format is 4-byte floating point numbers.

Important parameters which describe the physical size of the data must be given. When a file which does not contain GCV’s is to be read, key values such as data type, trace length, sample rate, and traces per gather must be given. Thus for the combination READ/NOGCV, a DATA parameter identical to that used by /IN must be provided. From this parameter, GCV’s KSORT, KNSAMP, SR, KNTR, and KNAUX are determined. The data traces are read from disk using these dimensions.

When a file which contains GCV’s is to be read (USEGCV parameter), the key values are read from the first data trace in the disk file. Thus for the combination READ/USEGCV, no DATA card is needed.

One additional piece of information which is needed to properly read data from disk is the amount of data to read. This information is provided by a combination of the BI, EI, NGATHERS, and NTRACES parameters. The interplay between these parameters is described in the below table.

If data is read from disk using the USEGCV values, the values of BI and EI act as data limiters; that is, only traces within the BI-EI range will be read by UNIXIO. When not specified, BI is assumed to be the KSHOT or KCDP value of the first trace (depending on the KSORT data type). When EI is not given, it is calculated from BI and either NGATHERS or NTRACES (with the help of KNTR).

A file without headers (i.e., the NOGCV parameter) will have GCV’s calculated for the data. The BI (specified or defaulted to 1) and EI (either specified or calculated) will serve as the range of GCV’s. All traces asked for will be read; BI and EI do not act as data limiters in this case. The total number of traces as computed from BI, EI, and NGATHERS or as specified by NTRACES determines the number of traces read by UNIXIO.

The behavior of the BI, EI, NGATHERS, and NTRACES parameters is as follows. "total" is the total number of traces which UNIXIO will read. "nx" is the sum of the KNTR and KNAUX values (as specified in DATA card or read from first trace).

<table>
<thead>
<tr>
<th>Specified:</th>
<th>Resulting behavior:</th>
</tr>
</thead>
<tbody>
<tr>
<td>BI</td>
<td>total=(EI-BI+1)*nx</td>
</tr>
<tr>
<td>EI</td>
<td></td>
</tr>
<tr>
<td>BI NGATHERS</td>
<td>total=NGATHERS*nx; EI=BI+NGATHERS-1</td>
</tr>
<tr>
<td>BI NTRACES</td>
<td>total=NTRACES</td>
</tr>
<tr>
<td></td>
<td>EI=BI+(NTRACES/nx)-1</td>
</tr>
<tr>
<td>NGATHERS</td>
<td>total=NGATHERS*nx; BI=1 (NOGCV); EI=BI+NGATHERS-1;</td>
</tr>
<tr>
<td>NTRACES</td>
<td>total=NTRACES</td>
</tr>
<tr>
<td></td>
<td>BI=1 (NOGCV); EI=BI+(NTRACES/nx)-1</td>
</tr>
<tr>
<td>EI NGATHERS</td>
<td>not supported</td>
</tr>
<tr>
<td>EI NTRACES</td>
<td>not supported</td>
</tr>
</tbody>
</table>

For the READ/NOGCV combination, one may skip an initial set of traces prior to reading data using the SKIPTRCS parameter. The first data trace which is read will have the BI value as determined above. This feature is not available with READ/USEGCV; rather, the OMIT or OMITRAC processor should be used.
PARAMETERS

Synopsis:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
<th>Fmt</th>
<th>Req/</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILENAME</td>
<td>filename</td>
<td>CHR*40</td>
<td>Req</td>
<td>none</td>
<td>UNIX binary internal format filename.</td>
</tr>
<tr>
<td>READ</td>
<td>none</td>
<td></td>
<td>Opt</td>
<td>none</td>
<td>Specify to read data from disk file.</td>
</tr>
<tr>
<td>WRITE</td>
<td>none</td>
<td></td>
<td>Opt</td>
<td>none</td>
<td>Specify to write data to disk file.</td>
</tr>
<tr>
<td>USEGCV</td>
<td>none</td>
<td></td>
<td>Opt</td>
<td>none</td>
<td>Specify to read or write trace headers to/from the disk file.</td>
</tr>
<tr>
<td>NOGCV</td>
<td>none</td>
<td></td>
<td>Opt</td>
<td>none</td>
<td>Specify to read or write trace headers with no GCV's involved.</td>
</tr>
<tr>
<td>FORMAT</td>
<td>bit_code</td>
<td>int</td>
<td>Opt</td>
<td>32</td>
<td>Data format (32= REAL<em>4, 16= INT</em>2). Required with READ/USEGCV option to specify trace format. Similar to DATA card used with /IN processor.</td>
</tr>
<tr>
<td>DATA</td>
<td>sort_type</td>
<td>float</td>
<td>req</td>
<td>none</td>
<td>sort type (KSORT).</td>
</tr>
<tr>
<td>trace_length</td>
<td>float</td>
<td>req</td>
<td>none</td>
<td>none</td>
<td>number of millisec per trace.</td>
</tr>
<tr>
<td>samp_rate</td>
<td>float</td>
<td>req</td>
<td>none</td>
<td>none</td>
<td>sample rate in millisec.</td>
</tr>
<tr>
<td>trace_per_gather</td>
<td>float</td>
<td>req</td>
<td>none</td>
<td>none</td>
<td>number of traces per gather (KNTR).</td>
</tr>
<tr>
<td>BI</td>
<td>first_gather</td>
<td>int</td>
<td>opt</td>
<td>none</td>
<td>number of the first gather (KSHOT or KCDP).</td>
</tr>
<tr>
<td>EI</td>
<td>last_gather</td>
<td>int</td>
<td>opt</td>
<td>none</td>
<td>number of the last gather (KSHOT or KCDP).</td>
</tr>
<tr>
<td>NGATHERS</td>
<td>num_gathers</td>
<td>int</td>
<td>opt</td>
<td>none</td>
<td>number of gathers to read.</td>
</tr>
<tr>
<td>NTRACES</td>
<td>tot_traces</td>
<td>int</td>
<td>opt</td>
<td>none</td>
<td>total number of traces to read.</td>
</tr>
</tbody>
</table>

GLOBAL COMMON VARIABLES

If USEGCV option is used, then GCV's are read from or written to disk.

If NOGCV is selected with the READ option, then GCV values are computed to be consistent with the DATA card values. KSHOT or KCDP numbers are computed using the BI-EI or NGATHERS/NTRACES parameters. If NOGCV is selected with the write option,
GCV's are written to disk (GCV information is lost).

EXAMPLE or OTHER NOTES
Example of usage:

To write a file with no GCV's:
/DIN
    FILENAME 'DATA'
/UNIXIO
    FILENAME 'DATA.UIO'
    WRITE
    NOGCV

To write a file with GCV's:
/DIN
    FILENAME 'DATA'
/UNIXIO
    FILENAME 'DATA.UIO'
    WRITE
    USEGCV

To write a file with GCV's and 16-bit (2-byte) integer traces:
/DIN
    FILENAME 'DATA'
/UNIXIO
    FILENAME 'DATA.UIO'
    WRITE
    USEGCV
    FORMAT 16

To read a file without GCV's: 1000 CDP stacked traces (3001 samples per trace):
/UNIXIO
    FILENAME 'DATA.UIO'
    READ
    NOGCV
    DATA 4 6000 2 1
    BI 101 EI 1100

To read file with GCV's: 9600 traces in shot gathers (3001 samples/sec) (DATA values are read from first trace in file):
/UNIXIO
    FILENAME 'DATA.UIO'
    READ
    USEGCV
    BI 1 EI 100

To read file with GCV's: 55 traces in 10-trace shot gathers (3001 samples/sec):
/UNIXIO
    FILENAME 'DATA.UIO'
    READ
    USEGCV
    NTRACES 55
UNIX DISKFILE FORMAT WHEN GCV's ARE STORED

The UNIX binary internal diskfile has a file header and trace headers. The file header indicates the number of trace headers and the names of any user-defined headers. The GCV headers stored for each trace are mapped directly from the SSCOM.INC common block.

This SSCOM.INC common block consists of four trace header arrays - one each for integer, real, character*8, and character*4 header values. These four arrays have the following characteristics:

<table>
<thead>
<tr>
<th>type</th>
<th>defined</th>
<th>used by SierraSeis</th>
<th>array name</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>200</td>
<td>134</td>
<td>SSCMA1</td>
</tr>
<tr>
<td>REAL</td>
<td>100</td>
<td>71</td>
<td>SSCMAR</td>
</tr>
<tr>
<td>CHAR*8</td>
<td>25</td>
<td>10</td>
<td>SSCMA8</td>
</tr>
<tr>
<td>CHAR*4</td>
<td>10</td>
<td>1</td>
<td>SSCMA4</td>
</tr>
</tbody>
</table>

The undefined slots in these arrays are available to be filled with user-defined headers.

The UNIX internal diskfile header keeps track of the number of GCV’s defined for each trace and also the names of user-defined GCV’s. The first four four-byte words of the diskfile header contain the total number of integer, real, CHAR*8, and CHAR*4 GCV’s which exist for the job. If the number of any header type is larger than that needed by SierraSeis, the difference is the number of user-defined headers. Following the four 4-byte numbers will be the names of any user-defined GCV’s. These names are stored in CHAR*6 format in the order that they are indexed in the GCV common block.

Immediately following the diskfile header will be the header values for the first trace followed by the first data trace. The size of the trace header is dependent on the values provided in the first four words of the file header. The order of the trace headers is integer, real, character*8, and character*4.

<table>
<thead>
<tr>
<th>File block</th>
<th>Bytes</th>
<th>Format</th>
<th>Format of UNIXIO file Contents</th>
<th>Example</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>#headers_1</td>
<td>1-4</td>
<td>I*4</td>
<td># int GCV’s</td>
<td>136</td>
<td>134 Sierra + 2 user</td>
</tr>
<tr>
<td>#headers_2</td>
<td>5-8</td>
<td>I*4</td>
<td># real GCV’s</td>
<td>72</td>
<td>71 Sierra + 1 user</td>
</tr>
<tr>
<td>#headers_3</td>
<td>9-12</td>
<td>I*4</td>
<td># char*8 GCV’s</td>
<td>10</td>
<td>10 Sierra</td>
</tr>
<tr>
<td>#headers_4</td>
<td>11-16</td>
<td>I*4</td>
<td># char*4 GCV’s</td>
<td>1</td>
<td>1 Sierra</td>
</tr>
<tr>
<td>user GCV</td>
<td>17-22</td>
<td>chr*6</td>
<td>1st user int GCV</td>
<td>'DEPTH '</td>
<td></td>
</tr>
<tr>
<td>user GCV</td>
<td>23-28</td>
<td>chr*6</td>
<td>2nd user int GCV</td>
<td>'COMPON'</td>
<td></td>
</tr>
<tr>
<td>user GCV</td>
<td>29-34</td>
<td>chr*6</td>
<td>1st user real GCV</td>
<td>'OFFSET'</td>
<td></td>
</tr>
<tr>
<td>1st trace</td>
<td>35-578</td>
<td>I*4</td>
<td>136 GCV values</td>
<td>101</td>
<td>136 integer GCV’s</td>
</tr>
<tr>
<td>header</td>
<td>579-866</td>
<td>real*4</td>
<td>72 GCV values</td>
<td>.004</td>
<td>72 real GCV’s</td>
</tr>
<tr>
<td>&quot;</td>
<td>867-946</td>
<td>chr*8</td>
<td>10 GCV values</td>
<td>'WM-3'</td>
<td>10 char*8 (i.e., KLINE)</td>
</tr>
<tr>
<td>&quot;</td>
<td>947-950</td>
<td>chr*4</td>
<td>1 GCV value</td>
<td>'xxxx'</td>
<td>1 char*4 (i.e., KSEQ)</td>
</tr>
<tr>
<td>1st trace</td>
<td>951-xxx</td>
<td>real*4</td>
<td>1st data trace</td>
<td></td>
<td>KNSAMP*4 byte words</td>
</tr>
</tbody>
</table>

See also utility programs UIOlength, UIOminmax, UIOdump.

SUBROUTINE NAMES

INIT subroutines: UNIXIN.F, UNIXI2.F
EXEC subroutines: UNIXEX.F, UNIXE2.F
Utility subroutine: UIOLIB.F
DIRECTORY: ../iris13/seismic
Initial modification from /CONVEXIN to allow input and output.
modified to allow read and write of GCV's.
fixed KSEQ behavior.
installed internal trace flow to allow the routine to be the first (trace
driver) within a job.
(a) READ/USEGCV does not need DATA parameter;
(b) use BI/EI as data limiters for READ/USEGCV;
(c) allow initial input trace skipping for READ/NOGCV;
(d) allow data storage in REAL*4 or INT*2 format;
(e) removed extraneous parameters, redefined run-time behavior of
those kept.
Appendix II: FTOCIO Library

C-Language I/O for FORTRAN Programs:
FORTRAN Unformatted I/O Access of Seismic Data
using C-equivalent Routines

A useful library of C-derived I/O statements exists which makes the access of seismic data files much easier. These C-equivalent statements replace FORTRAN statements such as OPEN, WRITE, READ, and CLOSE. Values are stored in unformatted, internal binary format. Their implementation and use can be far simpler than the FORTRAN statements.

The benefit of the ftocio (FORTRAN to C I/O) library is that the byte-address access of disk files found in the C language is made available to FORTRAN. This powerful I/O ability is ideal for large size seismic data files when individual seismic traces are requested.

The use of C-compatible I/O statements in FORTRAN programs makes data files available to both C and FORTRAN programs. One can intermix the application of C and FORTRAN programs to the same data files; the output from one program can be the input of another without having to worry about disk file format.

Structure of FORTRAN and C unformatted records

A major form of FORTRAN unformatted READ/WRITE statements require that a stored record in a disk file have a 4-byte integer record header which specifies the length of the record. The statements:

```
OPEN(UNIT=11,FILE=NAME,STATUS='OLD',FORM='UNFORMATTED')
READ(11)(X(I),I=1,N)
```

require that a 4-byte integer precede the 4*N bytes which comprise the stored array X(i). While the programmer only keeps track of the stored array, FORTRAN will acknowledge the record header. In other words, the programmer or user may think that only 4*N bytes are in the disk file when in reality 4*N + 4 bytes are stored.

If two successive FORTRAN records are stored, each record will have a 4 byte record header. If the first record is read using less than the total record length, the next READ statement will skip to the start of the next record; the trailing data in the first record is ignored.

An array written in C will only contain the number of bytes needed to store the array values. The statements

```
in = open('name',2);
notin= write(in,x,4*N);
```

will write N samples of the array X(i) to disk. The disk file will contain only 4*N bytes; the programmer must keep track of how many samples (bytes) to read back.

Two successive read statements using the C "open" function will read continuous data - the second read statement will read the first value available after the last value from the first read statement.

An additional advantage to C I/O calls is the availability of the seek command. This functionality allows the user to skip a desired number of bytes into a disk file in order to perform an operation. A loop of READ statements is not needed to advance into the disk file. While FORTRAN can provide a similar feature by specifying a desired record number, this feature only works for files which are written with fixed record lengths. C allows for variable record lengths and immediate seek capability. In this form, the programmer can directly access any particular data sample by knowing its true sample or byte position in the disk file.
FORTRAN I/O functions in FTOCIO: C-equivalent I/O statements

The C-equivalent I/O statements allow the user to create, open, and close a data file and to read, write, and seek data within the file. The FORTRAN functions are:

<table>
<thead>
<tr>
<th>ftocio</th>
<th>functionality</th>
<th>C-equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>ccreat</td>
<td>create a file with a given ASCII name</td>
<td>create</td>
</tr>
<tr>
<td>copen</td>
<td>open a file which has been previously created</td>
<td>open</td>
</tr>
<tr>
<td>cclose</td>
<td>close a file which has been created or opened</td>
<td>close</td>
</tr>
<tr>
<td>cread</td>
<td>read (an array of) data</td>
<td>read</td>
</tr>
<tr>
<td>cwrite</td>
<td>write (an array of) data</td>
<td>write</td>
</tr>
<tr>
<td>cseek</td>
<td>seek within a file specified by byte location</td>
<td>lseek</td>
</tr>
</tbody>
</table>

If used, each of these functions must be declared as integer functions in the declaration level of a main or subroutine program:

```
INTEGER copen, ccreat, copen, ccclose, cwrite, cread, cseek.
```

These functions are used as (integer) functions of the form

```
istatus = command (arg1, arg2, ...)
```

where `istatus` is an integer variable, `command` is the I/O statement to be performed, and the `arg`'s are function arguments which define things such as the file name, dimensioned array to read or write, and number of bytes.

The variable `istatus` serves two useful purposes. When the command has been performed successfully, the value of `istatus` is a positive integer. If the command was not successful, then `istatus` = -1. By checking the value of `istatus` one can tell whether to proceed in the program or stop to check why the command did not work.

For commands `ccreat` and `copen`, the positive integer value of `istatus` are significant. For each file that is created or opened, a different number is returned. The `istatus` values serve as file I.D. numbers (i.e., logical device numbers) which are used to identify the files within the program which are acted upon by the other I/O commands.

Creating, opening, and closing files

An aside on File Permission Codes (mode of a file)

Upon executing the UNIX command "ls -l", a list of files within the current directory will appear. The column of file attributes preceding the file names will consist of rows of r's, w's, and x's such as

```
-rw-rw-r-
```

or

```
-rwxr-xr-x.
```

An 'r' stands for READ, 'w' stands for WRITE, and 'x' stands for executable; this shows what can be done with the file and by whom. The first 'rwx' means the person whose login is listed as the file owner can read and write (change) the file and can run the program if it is an executable or shell file. If any of the three letters are missing, that command cannot be done to the file. The second set of 'rwx' shows what other people in the same user-group can do; the last set of 'rwx' shows what all other users can do to the file.

The sequence of 'rwx' can be represented by a number code. Each of the three letters can represent binary digits; the three letters together can represent both a three-digit binary number (000 through 111) and a single-digit octal number (0-7). Since the file permission code (or 'mode') is represented by three sets of 'rwx' s, a numerical representation exists in the form of three sets of three-digit binary numbers or, more easily, three octal digits:
The leading zero is set to one if the file name represents a directory; otherwise the named item is some type of file.

A file's 'mode' can be reset using the UNIX command chmod. As will be seen below, one can define a mode setting when creating a file.

Creating a file (CCREAT)

Creating a new file takes the following form:

\[ \text{ID} = \text{ccreat ('NEWNAME',IPERM)} \]

where ID is the internal file I.D. number (logical device number), 'NEWNAME' is the external (actual) name of the file, and IPERM is the file's permission code (mode).

Using the octal representation for a file permission code, one can create a data file using the following FORTRAN statements:

```
INTEGER ID,IPERM,ccreat
IPERM = 6*64 + 6*8 + 4  \quad \text{(representing -rw-rw-r--)}
ID = ccreat('DATAFILE',IPERM).
```

A data file named 'DATAFILE' is created with the owner and group users having read/write permission with all other owners having read-only permission. The internal file identification number (logical unit number) is referenced by using ID; this value is returned from ccreat upon valid execution of the function.

The data file name can be a character variable:

```
INTEGER ID,IPERM,ccreat
CHARACTER*20 CNAME
CNAME = 'DATAFILE'
IPERM = 6*64 + 6*8 + 4
ID = ccreat(CNAME,IPERM).
```

Closing a file (CCLOSE)

Now that the datafile 'DATAFILE' has been created, it will eventually have to be closed. The FORTRAN statement

\[ \text{ICHK} = \text{cclose(ID)} \]

will close the file. ICHK is an integer variable. ID is the internal file identification number which came from the ccreat (or copen) command. This statement simply closes the file after all accessing has been done; it is good to close all files before the program ends.

ICHK will equal -1 if the closing of the file was not possible.

Opening a pre-existing file (COPEN)

To open a pre-existing file, use

\[ \text{ID} = \text{copen('OLDFILE',ACCESS)} \]
where ID is the internal file identification number, 'OLDFILE' is the name of a file which already exists, and ACCESS is an integer which decides whether one can only read from or write to the file or do both:

- ID = copen('OLDFILE',0)  read only
- ID = copen('OLDFILE',1)  write only
- ID = copen('OLDFILE',2)  read and write

If you never want your data file to change, use ACCESS=0. Now that the file 'OLDFILE' is opened, read and/or write operations are possible. Remember to close the file when done.

An example of opening and closing a file is as follows:

```fortran
INTEGER ID,ICHK,copen, cclose
CHARACTER*20 CNAME

CNAME = 'DATAFILE'
ID = copen(CNAME,0)
IF (ID .EQ. -1) THEN [FATAL ERROR]
C perform read-only operations to data file, then
ICHK = cclose(ID)
END
```

### Reading, writing, and skipping data arrays

When reading, writing, or skipping data values or arrays, one needs to think in terms of the bytes used to store the data. Floating point or integer arrays (INTEGER*4) use four bytes per number (32-bit format); INTEGER*2 integers use two bytes per number; and character arrays use single bytes per character.

### Reading an array of seismic data (CREAD)

Suppose we wish to read an array of seismic data which contains a thousand time samples (number of samples = NT = 1000). Then, provided the data file 'DATAFILE' already exists, the seismic data can be read into a dimensioned array using the following:

```fortran
INTEGER NT, ID, NREAD, copen, cread
DIMENSION TRACE(1000)
ID = copen('DATAFILE',0)
NREAD = cread(ID,TRACE,4*NT)
```

The variable NREAD is used only to check if the read operation was successful; if equal to -1, the cread operation failed; otherwise it will equal the number of bytes actually read. The cread operation takes the first 4*NT bytes stored in 'DATAFILE' and stores them into the single-subscripted floating point array TRACE. The values of TRACE are thus available for subsequent manipulation.

If the statement below was used instead of the one above,

```fortran
NREAD = cread(ID,TRACE,4*50)
```

then only the first 50 samples of the data would be stored in the array TRACE.

Based on these examples, we can see that the cread statement looks like:

```fortran
NREAD = cread(ID,ARRAY,NBYTES)
```

where ID is the internal file identification number, ARRAY is the declared array (or single-
variable) in which to store the read bytes, and NBYTES is the number of bytes to place into ARRAY. NREAD is the number of bytes successfully read (or -1 if the operation failed).

The cread command will only map the specified number of bytes into the variable ARRAY; the manner in which ARRAY is declared determines whether the bytes become integers, real, or character data. In the following examples, the declaration statements define the type of data which is read:

```
INTEGER ARRAY(25)
NREAD = cread(ID,ARRAY,100)

REAL ARRAY(25)
NREAD = cread(ID,ARRAY,100)

CHARACTER*1 ARRAY(100)
NREAD = cread(ID,ARRAY,100)
```

In the first example, ARRAY will contain 25 integers, in the second, ARRAY will contain 25 real values, in the last, ARRAY will contain 100 ASCII characters.

**Writing an array of seismic data (CWRITE)**

Writing data to a disk file is structured quite similar to the read statement:

```
NWRITE = cwrite(ID,ARRAY,NBYTES)
```

where ID is the file identification number, ARRAY is the array or variable to output, and NBYTES is the number of bytes to write. NWRITE will be returned from cwrite to contain the actual number of bytes which were written (or equal -1 if the write failed). Examples of the use of cwrite are given below.

Two consecutive calls to cwrite will place the written data in continuous byte order; the bytes of the last value of the first write will be followed by the first set of bytes from the first value of the second write. No record header bytes will exist between the two sets of data.

**Seeking to specific byte locations (CSEEK)**

We can skip over traces before reading or writing by using a seek command. Again, we think in terms of bytes. The seek command allows us to actually skip through whole or partial data arrays if we want to obtain the value(s) of a specific data sample(s). Since the seek command skips in units of single bytes, we need to make sure the seek command places us at the start of a set of bytes, not, for example, in the middle of a (4-byte) number.

The seek command is:

```
NSEEK = cseek(ID, NBYTES, FROM WHERE).
```

The file in which to skip is indicated by ID; the total number of bytes to skip is given by NBYTES. FROM WHERE indicates the reference point from where to skip from within the file:

```
nseek = cseek(id, nbytes, 0)    skip # bytes from the beginning of the file
nseek = cseek(id, nbytes, 1)    skip # bytes from the current location in the file
nseek = cseek(id, nbytes, 2)    skip # bytes backwards from the end of the file
```

NSEEK is an integer variable which behaves like NREAD or NWRITE in that it will equal -1 if the seek fails or will return the number of bytes which were skipped.
Suppose a data file exists which contains several seismograms of length $NT=1000$ samples per seismogram. Then

$$\text{NSEEK} = \text{cseek}(\text{ID}, 3\times 4\times NT, 0)$$

will skip three seismograms worth of bytes from the beginning of the data file. Remember that there are 4 bytes per sample and thus $4\times NT$ bytes per seismogram. A subsequent

$$\text{NSEEK} = \text{cseek}(\text{ID}, 1000, 1)$$

will skip the next 1000 bytes (250 samples) from the end of the third seismogram (current position after the first seek).

The notion of a pointer should be mentioned. When a file is created or opened, a pointer is positioned at the beginning of the file. A read or write command will start from the pointer, initially at the beginning of the file. Once an array (seismogram) is read, the pointer gets moved to the end of the array and actually points to the start of the next array. So two consecutive read commands will read two consecutive arrays. The write command behaves similarly. The seek command will move the pointer around. The sequence

```c
ID = \text{copen}('OLDNAME', 2)
NREAD = \text{cread}(\text{ID}, \text{TRACE}, 4\times NT)
NSEEK = \text{cseek}(\text{ID}, 3\times 4\times NT, 1)
NWRITE = \text{cwrite}(\text{ID}, \text{TRACE}, 4\times NT)
ICHK = \text{cclose}(\text{ID})
```

opens the file OLDNAME for reading and writing (ACCESS=2), reads the first array (seismogram), skips over the next three seismograms, then writes the array as the overall fifth array in the file. The file is then closed.

**Example 1: Copy a file containing several seismic traces**

Suppose we need to copy a file of seismic traces (and we don't want to use the UNIX copy command 'cp'). If we know that there are $NT=1000$ samples per trace, $NX=20$ seismic traces in the file, and the data are stored in the file 'OLDNAME', we can proceed:

```
C COPYFILE: a program to copy one data file to another
dimension TRACE(1000)
integer copen,ccreat,cclose,cread,cwrite,cseek

NT=1000
NX=20
iperm= 6\times 64 + 6\times 8 + 4

C Open input, create output, seek to beginning of both
in = \text{copen}('OLDNAME', 2)
iout = \text{ccreat}('NEWNAME', iperm)
nseek = \text{cseek}(\text{in}, 0, 0)
nseek = \text{cseek}(\text{iout}, 0, 0)

C Now loop over traces
do 100 i=1,NX
nread = \text{cread}(\text{in}, \text{TRACE}, 4\times NT)
nwrite = \text{cwrite}(\text{iout}, \text{TRACE}, 4\times NT)
100 continue
```
C now shut down
ichk=cclose(in)
ichk=cclose(iout)
END

This program copies file 'OLDNAME' into a new file 'NEWNAME'. No error checking was conducted, but for each flocio call, the return status could have been checked.

Additional Functionality: Command Line Arguments

The C language, unlike FORTRAN, has a provision to allow for the access of command line arguments for executable programs. These command line arguments can be option flags or run-time parameters which help influence how the program will function. The use of this capability allows for a program to be written generically rather than for a specific application. Unfortunately, FORTRAN does not provide for this capability.

The flocio library contains functions which introduces this flexibility into FORTRAN. These functions allow the programmer (and user) to obtain command line arguments for subsequent use within the programming code.

The command line arguments can be either integer, floating point, or character string arguments. The following functions provide this capability:

<table>
<thead>
<tr>
<th>function or subroutine</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iarl</td>
<td>&quot;integer argument, long&quot; (get INT*4 value)</td>
</tr>
<tr>
<td>iars</td>
<td>&quot;integer argument, short&quot; (get INT*2 value)</td>
</tr>
<tr>
<td>far</td>
<td>&quot;floating point argument&quot; (get REAL value)</td>
</tr>
<tr>
<td>getname</td>
<td>subroutine to get character string</td>
</tr>
<tr>
<td>openr</td>
<td>subroutine to open file given in command line</td>
</tr>
<tr>
<td>creatr</td>
<td>subroutine to create file given in command line</td>
</tr>
</tbody>
</table>

Command line arguments are given for an executable program "main" in the following form:

```
main arg1 arg2 arg3 ...
```

The functions and subroutines named above are programmed to obtain a value from a specific location in the command line. The locations are specified by the programmer within the source code. The user must follow this order.

Command line functions: IARL, IARS, FAR

To obtain a 4-byte integer from the command line, the line

```
IVARIABLE = iarl (LOCATION)
```

is used. LOCATION is the spot in the command line where the integer value is to be given by the user when the program is run. IVARIABLE is the integer variable to receive the value. An example is given below.

To obtain a 2-byte integer, the line

```
IVARSHORT = iars (LOCATION)
```

is used. IVARSHORT must be declared as an INTEGER*2 variable. Again, LOCATION is the spot in the command line where the value is to be given.

A floating point value is obtained in a similar manner:
The floating point value in the LOCATIONth spot is placed into variable XVALUE.

Example of command line functions

A program to compute the source-receiver distance for an evenly spaced refraction (or reflection) receiver array is given below. First, an interactive program is given, followed by a command line version.

Interactive version:

```
C Compute receiver array source-receiver offsets: INTERACTIVE
integer nrec
real drec,d0

C Get # receivers, receiver spacing, 1st receiver distance
write(*,10)
10 format('Enter # receivers, receiver spacing, 1st receiver:')
read(*,20)nrec,drec,d0
20 format(i5,2f10.3)

C Now compute offsets
do i = 1,nrec
   offset=d0 + drec * float(i-1)
   write(*,30)i,offset
30 format('receiver',i5,' is offset by',f10.3)
enddo
END
```

Command line version:

```
C Compute receiver array source-receiver offsets: COMMAND LINE
integer nrec
real drec,d0

C Get # receivers, receiver spacing, 1st receiver distance;
C User runs program as: (prompt) main nrec drec d0
nrec=iarl(1)
drec=far(2)
d0=far(3)

C Now compute offsets
do i = 1,nrec
   offset=d0 + drec * float(i-1)
   write(*,30)i,offset
30 format('receiver',i5,' is offset by',f10.3)
enddo
END
```

An example of the execution of the latter program would be:

```
main 48 100. 150.
```

where the number of receivers is 48, the receiver spacing is 100 m, and the first receiver is 150 m from the source.
The latter program can be run from shell or batch files while the former program is designed for user input (or with redirected input files).

Command line subroutines: GETNAME, OPENR, CREATR

The command line functions obtain numerical values from the command line. These values are obtained by first extracting the character string for the argument and then converting the string to the appropriate numerical value.

In order to obtain just the character string, the subroutine getname can be used. The subroutine is called in the following manner:

```
CALL getname(LOCATION,STRING)
```

where LOCATION is the position in the command line arguments and STRING is the returned character string. A typical use for this subroutine is to obtain the name of a file to open:

```fortran
character*20 FILENAME
integer open, in

call getname(1, FILENAME)
in = open(FILENAME, 0)
```

In this example, the first command line argument after the executable program name is the input file name. This name is obtained by getname; the file is subsequently opened by open.

Since much of the use of character string command line arguments is for file names which are to be opened or created, subroutines are provided to obtain the character string name and open or create the file. For example, the above four lines of FORTRAN code can be replaced by a one line call to the subroutine openr:

```
call openr(1, 0, in).
```

This subroutine will obtain the first command line argument as a character string (file name), invoke the open function to open the file with read-only access (ACCESS=0), and return the file identification number into the variable "in"). The range of ACCESS values (0-2) can be used here. If the subroutine fails, IN will equal -1.

In a similar manner, one can create a filename given a command line file name using the creatr subroutine:

```
call creatr(LOCATION, MODE, ACCESS, IOUT).
```

LOCATION is the location in the command line of the new file name. MODE is the permission code (e.g., 6*64 + 6*8 + 4 = 0664 octal = rw-rw-r-- file permission) in which to create the file. ACCESS is the resulting file access to be given the file for this program (a la open). The resulting file identification number is returned into variable IOUT. If the subroutine cannot create the file with the given permission and access, IOUT will equal -1.

Example 2: A versatile data file copy program

Example 1 above described a program to copy a data file 'OLDNAME' into a new file 'NEWNAME'. The loop to copy twenty 1000-sample traces was hardwired into the program. The same program can be constructed in a very generic manner using the command line functions and subroutines to create a program that will copy any number of traces of a user-specified length.
COPYFILE: a generic program to copy one data file to another.

To run: (prompt) copyfile oldfile newfile nt nx

where
oldfile = file name to copy
newfile = file name to create
nt = number of samples/trace
nx = number of traces to copy

dimension TRACE(1000)
integer cclose,cread,cwrite

iperm= 6*64 + 6*8 + 4

Open input, create output, seek to beginning of both

call openr(1,0,in)
call creatr(2,iperm,1,iout)

NT=iarl(3)
NX=iarl(4)
if(NT .GT. 1000) then [fatal error]

Now loop over traces

do 100 i=1,NX
nread = cread(in, TRACE, 4*NT)
nwrite = cwrite(iout, TRACE, 4*NT)
100 continue

Now shut down
ichk=cclose(in)
ichk=cclose(iout)

As an example, this program could be run as:

copyfile OLDNAME NEWNAME 1000 50

Subroutine openr will open file 'OLDNAME' with read-only access. Subroutine creatr will create file 'NEWNAME' with 'rw-rw-r--' file permission and will open the file with write-only access (ACCESS=1). Both subroutines will essentially place file pointers at the beginning of the files (i.e., execute cseek(file,0,0) commands).

The dimensions of the data (number of traces, number of samples per trace) are not hardwired, but are obtained from the third and fourth command line arguments. For this example, fifty 1000-sample traces will be copied. The program, as written, will copy any file containing any number of traces into another file provided the internal trace array dimension (i.e., 1000) is not exceeded.

Compiling FORTRAN Source Code When Using FTOCIO LIBRARY

The functions and subroutines are not part of the standard FORTRAN compiling library. The archive library ftocio.a is found in the IRIS-SEIS directory ../irisl3/lib. This library must be listed in the link list when compiling the original source code of a program:

f77 main.f ../irisl3/lib/ftocio.a [other flags, libraries] -o main.

IRIS-SEIS and LOCAL-SEIS shell and make files include this library in their link lists.
Appendix IIB: FTOCIO Library

C-Language I/O for FORTRAN Programs:
FORTRAN Unformatted I/O Access of Seismic Data
using C-equivalent Routines

Summary Table

NOTE: Always declare the I/O functions as integers:

INTEGER*4 ccreat, copen, cclose, cread, cwrite, cseek

**CCREAT**

<table>
<thead>
<tr>
<th>ICREAT = ccreat('FNAME',IPERM)</th>
<th>ccreat creates a file named FNAME. FNAME is a string or a character variable. IPERM is the file’s RWX mode in octal. ICREAT is the file identification number, equal to -1 if failed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ccreat = ccreat('output',iperm)</td>
<td>creates file named 'output'</td>
</tr>
<tr>
<td>character*8 cname</td>
<td></td>
</tr>
<tr>
<td>name = 'output'</td>
<td></td>
</tr>
<tr>
<td>ccreat = ccreat(cname,iperm)</td>
<td>creates file named 'output'</td>
</tr>
<tr>
<td>iperm = 6<em>64 + 6</em>8 + 4</td>
<td>octal 554 = binary 0 110 110 100 creates 'data' with -rw-rw-r-- file mode</td>
</tr>
<tr>
<td>icreat = ccreat('data',iperm)</td>
<td></td>
</tr>
<tr>
<td>iperm = 7<em>64 + 5</em>8 + 7</td>
<td>octal 757 = binary 0 111 101 111 creates 'data' with -rwxr-xrwx file mode</td>
</tr>
</tbody>
</table>

**COPEN**

<table>
<thead>
<tr>
<th>IOPEN = copen('FILENAME',ACCESS)</th>
<th>open opens an existing file. FILENAME is a string or a character variable. ACCESS is a temporary file access (read/write); file permission (IPERM in ccreat) must be compatible. IOPEN is the file identification number, equal to -1 if failed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in = copen('input',0)</td>
<td>opens file 'input' with READ-ONLY status</td>
</tr>
<tr>
<td>in = copen('input',1)</td>
<td>opens file 'input' with WRITE-ONLY status</td>
</tr>
<tr>
<td>in = copen('input',2)</td>
<td>opens file 'input' with READ/WRITE status</td>
</tr>
<tr>
<td>character*8 cinput, coutput</td>
<td></td>
</tr>
<tr>
<td>cinput = 'data'</td>
<td></td>
</tr>
<tr>
<td>coutput = 'results'</td>
<td></td>
</tr>
<tr>
<td>in = copen(cinput,0)</td>
<td>opens input file 'data' (read-only)</td>
</tr>
<tr>
<td>iout = copen(coutput,2)</td>
<td>opens existing output file 'results' (read/write)</td>
</tr>
</tbody>
</table>
### CCLOSE

<table>
<thead>
<tr>
<th>ICHK = cclose(ID)</th>
<th>cclose closes a file previously created or opened.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID is a file identification number defined by ccreat or copen.</td>
</tr>
<tr>
<td></td>
<td>ICHK = status flag = -1 if cclose failed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ichk = cclose(creat)</th>
<th>closes file created by ccreat.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ioout = open('input',0)</td>
<td>opens file input.</td>
</tr>
<tr>
<td>ichk = cclose(ioout)</td>
<td>closes file opened by open.</td>
</tr>
</tbody>
</table>

### CREAD

<table>
<thead>
<tr>
<th>NREAD = cread(ID,ARRAY,NBYTES)</th>
<th>cread reads an array of bytes, translating bytes into proper format based on declaration of ARRAY.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID is file identification number defined by copen or ccreat.</td>
</tr>
<tr>
<td></td>
<td>ARRAY can be real, integer, or character arrays.</td>
</tr>
<tr>
<td></td>
<td>NBYTES is total number of bytes to read.</td>
</tr>
<tr>
<td></td>
<td>NREAD is status flag, = -1 if read failed.</td>
</tr>
</tbody>
</table>

| dimension trace(100) | reads 400 bytes, translates into 100 floating point values stored in array 'trace'. |
| nsample = 100 | (1) reads 4 bytes, translates into integer number stored in 'nsample'. |
| in = open('input',0) | (2) reads 4*nsample bytes, translates into nsample floating point numbers stored in array 'trace'. |
| nread = cread(in,nsample,4*1) | reads 10 traces, each 100 samples long |
| nread = cread(in,trace,4*nsample) | reads 10 traces, each 100 samples long |

### CWRITE

<table>
<thead>
<tr>
<th>NWRITE = cwrite(ID,ARRAY,NBYTES)</th>
<th>cwrite writes an array of bytes, translating from ARRAY.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID is file identification number defined by copen or ccreat.</td>
</tr>
<tr>
<td></td>
<td>ARRAY can be floating point, integer, or character array.</td>
</tr>
<tr>
<td></td>
<td>NBYTES is total number of bytes to write.</td>
</tr>
<tr>
<td></td>
<td>NWRITE is status flag, = -1 if write failed (check read/write permissions).</td>
</tr>
</tbody>
</table>

| dimension trace(100) | writes 400 bytes, translating from array 'trace'. |
| nsample=100 | writes 400 bytes, translating from array 'trace'. |
| iout=open('output',2) | writes 400 bytes, translating from array 'trace'. |
| nwrite=cwrite(iout,trace,4*nsample) | writes 400 bytes, translating from array 'trace'. |
CSEEK

\[
\text{NSEEK} = \text{cseek} (\text{ID, NBYTES, FROM\_WHERE})
\]

\[
\text{cseek moves location pointer to a new byte position. ID is file identification number defined by copen or ccreat. NBYTES is number of bytes to move to. FROM\_WHERE is a position reference location from which to move NBYTES bytes. NOTE: each cread or cwrite moves the location pointer to the end of the bytes read or written.}
\]

\[
\text{in} = \text{copen('input',0)}
\]
\[
\text{nsamples} = 1000
\]
\[
\text{nbytes} = 4 * \text{nsamples}
\]
\[
\text{nseek = cseek(in,nbytes,0)}
\]
\[
\text{nseek = cseek(in,nbytes,1)}
\]
\[
\text{nseek = cseek(in,nbytes,2)}
\]

\[
\text{skip nbytes bytes from beginning of file. skip nbytes from current pointer location. skip nbytes back from end of file.}
\]

Command line functions IARL, IARS, FAR

\[
\text{IVAR = iarl(LOC)}
\]
\[
\text{ialrl obtains INT*4 value from command line argument list. LOC is position in command line argument list. IVAR is variable receiving INT*4 value.}
\]

\[
\text{IVAR = iars(LOC)}
\]
\[
\text{iars obtains INT*2 value from command line argument list. LOC is position in command line argument list. IVAR is variable receiving INT*2 value.}
\]

\[
\text{XVAR = far(LOC)}
\]
\[
\text{far obtains REAL value from command line argument list. LOC is position in command line argument list. XVAR is variable receiving REAL value.}
\]
**Command line subroutines GETNAME, OPENR, CREATR**

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>call getname(LOC,CSTRING)</code></td>
<td><code>getname</code> returns a character string located in the command line. LOC is the position in the command line argument list. CSTRING is a character string containing the returned command line string.</td>
</tr>
<tr>
<td><code>call openr(LOC,ACCESS,ID)</code></td>
<td><code>openr</code> opens a file whose name is given in the command line argument list. LOC is the position of the file name in the command line argument list. ACCESS is the <code>open</code> access code (0-2) in which to open the file. ID is the file identification number returned by <code>openr</code>.</td>
</tr>
<tr>
<td><code>call openr(1,0,ID)</code></td>
<td>File named at argument 1 is opened READ-only.</td>
</tr>
<tr>
<td><code>call openr(1,1,ID)</code></td>
<td>File named at argument 1 is opened WRITE-only.</td>
</tr>
<tr>
<td><code>call openr(1,2,ID)</code></td>
<td>File named at argument 1 is opened READ/WRITE.</td>
</tr>
<tr>
<td><code>call creatr(LOC,IPERM,ACCESS,ID)</code></td>
<td><code>creatr</code> creates a file whose name is given in the command line argument list. LOC is the position of the file name in the command line argument list. IPERM is the file permission (mode) to assign to the new file. ACCESS is the <code>open</code> access code (0-2) in which to open the file. ID is the file identification number returned by <code>creatr</code>.</td>
</tr>
<tr>
<td><code>IPERM = 6*64 + 6*8 + 4</code></td>
<td>File named at argument 2 is created with 664 file mode. File is opened with WRITE-only status and is identified by ID.</td>
</tr>
<tr>
<td><code>call creatr(2,IPERM,1,ID)</code></td>
<td></td>
</tr>
<tr>
<td><code>IPERM = 6*64 + 6*8 + 4</code></td>
<td>File named at argument 2 is created with 664 file mode. File is opened with READ/WRITE status and is identified by ID.</td>
</tr>
<tr>
<td><code>call creatr(2,IPERM,2,ID)</code></td>
<td></td>
</tr>
</tbody>
</table>