United States Department of the Interior

Geological Survey

A C Language Implementation of the SRO (Murdock) Detector/Analyzer

by

James N. Murdock and Scott E. Halbert

Open File Report 87-158

30 April 1991

This report is preliminary and has not been reviewed for conformity with U.S. Geological Survey editorial standards. Any use of tradenames is for descriptive purposes only and does not imply endorsement by the U.S. Geological Survey.

Albuquerque, New Mexico

Abstract

A signal detector and analyzer algorithm was described by Murdock and Hutt in 1983. The algorithm emulates the performance of a human interpreter of seismograms. It estimates the signal onset, the direction of onset (positive or negative), the quality of these determinations, the period and amplitude of the signal, and the background noise at the time of the signal.

The algorithm has been coded in C language for implementation as a "blackbox" for data similar to that of the China Digital Seismic Network. A driver for the algorithm is included, as are suggestions for other drivers. In all of these routines, plus several FIR filters that are included as well, floating point operations are not required. Multichannel operation is supported.

Although the primary use of the code has been for in-house processing of broadband and short period data of the China Digital Seismic Network, provisions have been made to process the long period and very long period data of that system as well. The code for the in-house detector, which runs on a mini-computer, is very similar to that of the field system, which runs on a microprocessor.

The code is documented.

Contents

1 Introduction 1

2 Program Flow 3

3 Detector Routines Overview 5

4 Example Drivers 11

5 Support Routines Overview 17

6 Event Detector Parameters and Onset Printouts 25

6.1 Event Detector Parameters 25

6.2 Parameters That Are Output By The Detector 28

7 Characteristics of the Code 31

8 Remarks 41

8.1 Caveats 41

8.2 Acknowledgements 41

8.3 References 42

A Event Detector C Code 43

A.1 CSSTND.H – Main project definitions 43

A.2 CSCONT.H – Context and interface definitions 45

A.3 DETECT.H – Detector variables definitions 50

A.4 CSCONFIG.H – Detector configuration definitions 51

A.5 E_DETECT.C – Detector main loop and dispatch 52

A.6 EVENT.C – Event determination routine 57

A.7 P_ONE.C – P-T value generator 66
A.8 P_TWO.C – Average background estimator 70
A.9 ONSET.C – Onset picker 72
A.10 ONSETQ.C – Onset parameter determination 80
A.11 IBINGO.C – Event flag and timer setup 86
A.12 WBUFF.C – Event storage buffer setup 88
A.13 XTH.C – Threshold calculator 89
A.14 TIME_F.C – Event pick time determination 91
A.15 CK_T_KONT.C – Pick-time adjustments 93
A.16 PERIOD.C – Event period determination 95
A.17 COUNT_DN.C – Event on/off countdown 96
A.18 CONT_SETUP.C – Prepare event structures 98

B Data Management C Code **101**

B.1 E_BUFFER.C – Allocate data buffers 101
B.2 E_CDSNLOAD.C – Convert CDSN data into integers 102
B.3 E_CREATE.C – Allocate user detector structures 105
B.4 E_FILTER.C – Filter the user data 107
B.5 E_REMOVE.C – Remove and cleanup user structures 109
B.6 FFIRBB20.C – 20 SPS BB FIR filter 109
B.7 FFIRSP10.C – 10 SPS SP FIR filter 111
B.8 FFIRSP20.C – 20 SPS SP FIR filter 113
B.9 FFIRSP40.C – 40 SPS SP FIR filter 114
B.10 FFPAV.C – 4 point running average filter 116
B.11 FNULL.C – Dummy filter 117
B.12 DISP PAR.C – Display event parameters 119

C Example Driver **121**

List of Figures

5.1 Response of `FfirSP10()` filter at 10 SPS 21
5.2 Response of `FfirSP20()` filter at 20 SPS 21
5.3 Response of `FfirSP40()` filter at 40 SPS 22
5.4 Response of `FfirBB20()` filter at 20 SPS 22
5.5 Response of `Ffpav10()` filter at 10 SPS 23
5.6 Response of `Ffpav40()` filter at 40 SPS 23

List of Tables

5.1 FIR filters 20

6.1 Parameters used for event detection. 25

6.2 Encoded threshold factors generated by `Xth()` subroutine. 27

6.3 Sample onset printout from event detector 28

7.1 Sizes of the C source files and compiled object files for event detection system as compiled on the Sun 3/160, C compiler. The n/a is used because the code sections of the include files are already added to the sizes of the routines. . . 34

7.2 Example execution profile of the event detector that processes 20 samples/second data on the DEC 11/70. The n/a is used in columns which are not applicable to the statistic. The filter used has 37 coefficients, each of which is implemented with power-of-2 shifts for speed. 35

7.3 Example execution profile of the event detector that processes 40 samples/second data on the DEC 11/70. The n/a is used in columns which are not applicable to the statistic. The filter used has 17 coefficients, each of which is implemented with power-of-2 shifts for speed. 36

7.4 Example execution profile of the event detector that processes 1 sample/second data on the Sun 3/160. The n/a is used in columns which are not applicable to the statistic. A 6-pole low-pass recursive filter was used. Little effort was made to reduce the execution time for this filter. 37

7.5 Example execution profile of the event detector that processes 20 samples/second data on the Sun 3/160. The n/a is used in columns which are not applicable to the statistic. The filter used is the same as in the CDSN BB test. 38

7.6 Example execution profile of the event detector that processes 100 samples/second data on the Sun 3/160. The n/a is used in columns which are not applicable to the statistic. A 2-pole low-pass recursive filter was used. Little effort was made to reduce the execution time for this filter. 39

Chapter 1

Introduction

The purpose of this report is to present a C language (Kernighan and Ritchie, 1978) implementation of the offline detector that is used to process data of the China Digital Seismic Network (CDSN). It is very similar to the on-line detector of that system which employs a microprocessor.

The algorithm was described by Murdock and Hutt (1983) and is sometimes referred to as the "SRO[1] detector", the "Murdock–Hutt detector", or alternatively, as the "Murdock detector". As described in the report, the detector emulates the performance of a human who interprets seismograms. It estimates: (1) the time of onset of the signal (2) the direction of the onset (positive or negative, c or d) (3) the quality of these determinations (4) the period and maximum amplitude of the first several cycles of the signal, and (5) a measure of the RMS background noise at the time the signal was detected.

The algorithm has been implemented herewith in C language without using any floating point operations. The detector was designed and coded in FORTRAN by one of the authors (Murdock) in early 1977. It has been revised and updated several times since then. This most recent version has been designed:

- To process multiple channels of data simultaneously.
- To operate very quickly.
- To operate as a "blackbox" for data similar to that of the CDSN.
- To be portable, readable and understandable.

Whereas our goal is not to present a general purpose signal detector and analyzer, documentation is provided so that the detector might be modified to process seismic data other than that of the China System. One important restriction of the code is imposed by our effort to avoid floating point operations: The sample rate (which ordinarily is an integer) must divide into one thousand with no remainder. (The circumvention of floating point operations has been to facilitate implementation on elementary microprocessors, as well as to enhance speed of execution thereon.) The code could be easily modified to address this restriction. Of course, any such modifications, or alternate use, should be accompanied by thorough tests to demonstrate that the algorithm performs as expected.

[1] Seismic Research Observatory.

The body of the report describes the flow of the code, gives a brief description of each of the routines, and describes parameters of the code (size, speed, etc.). In addition, descriptions of potential drivers are presented. Appendices give the code of the detector and of a driver.

Chapter 2

Program Flow

The following is the flow of the program presented in pseudo C style:

C include files:

detect.h definitions (#define) and externals.

csstnd.h definitions made for portability.

stdio.h C header for standard input/output

csconfig.h parameters that are configurable by the user.

cscont.h structures needed for multichannel operation, macros for the driver (Appendix C).

signal.h for UNIX utility signal().

Event Detector Routine:

E_detect() the data flow manager for the detector.

P_one() find P-T (peak to trough and trough to peak) values and associated times.

P_two() estimate sample standard deviation of P-T values.

Xth() calculate thresholds for Event().

Event() process P-T values to detect signals.

W_buff() write buffers when an event might be in progress.

Ibingo() set flags and raise thresholds when event is detected.

Onsetq() find reference sample number of onset of detected signal.

Onset() convert the reference sample number to time, calculate period of signal, and calculate parameters of the signal.

Period() calculate period of signal.

Time_f() convert sample number to time.

Ck_t_kont() adjust event time calculation when records are $\geq$ 1 minute long.

Count_dn() set flags for processing, and time the interval of the event.

Chapter 3

Detector Routines Overview

These are the C routines which are the event detector. They require input parameters and a driver to be operated. Suggestions for two simple drivers are shown in the next chapter (Chapter 4), and a complex driver is shown in Appendix C. A description of the input parameters is shown in Chapter 6.

– E_detect() –

Called: By User Driver.

`E_detect()` orchestrates data flow within the event detector algorithm. Although filtering is not required, the data that are input to the detector are typically filtered. The function of `E_detect()` is to call routines that:

1. Find the amplitudes and associated times of the (typically) filtered data, `P_one()`.
2. Process these amplitudes and associated times to search for seismic signals, `Event()`.
3. Cause output of the parameters of the detected signal, `Onsetq()`.
4. Time the interval of the event, `Count_dn()`.

The main (calling program) passes to `E_detect()` the address of a structure. Although this structure contains several other variables and constants, the only ones needed by `E_detect()` are:

1. sample rate x 1000 of the digital data. (This multiplication is done so that VLP data of the China system can be detected, if one so desires.)
2. the number of data points that will be processed.
3. the time (yr, day, hour, minute, second, millisecond) of the first data point of the current record (the data that will be processed).
4. the address of the first data point to be processed. (The address of the array of 32 bit data that will be processed.)
5. the address of the list (a structure) of variables and constants that must be maintained for each detector. The parameters of the detector (*filhi*, *fillo*, *xth1*, etc.) are included in this list and must be loaded into the list by the user.

6. the detector name (for instance, BB_Z1, BB_Z2, SP_Z, SP_N, etc).

E_detect() returns TRUE (the value 1) if any the data of the current record are within the interval of an event: Upon declaring an event, E_detect() will remain TRUE for a minimum of NOR_OUT × *wait_blk* samples, where typically NOR_OUT = 4 and *wait_blk* = 1014 or 507. (Because we envision that data will be written to tape during the interval of the event, *wait_blk* typically is chosen as a function of the input record length.) The return value of E_Detect() will remain true longer than NOR_OUT × *wait_blk* if a retrigger occurs in the coda of the event.

Returns: TRUE - Current record is within the interval of an event.

FALSE - Current record is not within the interval of an event.

Fatal Errors: No fatal exits

– P_one() –

Called: By E_detect() to process each seismic data sample. This routine determines the signed amplitudes (and associated times) of the filtered data. (By amplitudes, we mean the difference in value of the consecutive local maximums and minimums, hereafter referred to as the peak-to-trough amplitudes, or P-T value.) Each peak or trough is determined by comparing slopes between the input samples. When a peak or trough is found, the record and time (ie, sample number) where it occurred is documented.

In addition to calculating the peak-to-trough amplitudes, P_one() gathers information to estimate the background noise. To do this, P_one() compares the absolute value of each of the amplitudes to a threshold *thx*. The maximum of 20 successive values less than *thx* is fed to P_two(). (The threshold is used to inhibit anomalously large values, such as spikes, from contributing to the estimate of normal background.) P_two() uses these maximums to estimate the statistical dispersion of the background noise: *twosd*. (By statistical dispersion, we mean an estimate of the sample standard deviation of the P-T values.)

Returns: TRUE - Peak or trough detected

FALSE - No Peak or trough detected

Fatal Errors: No fatal exits

– P_two() –

Called: By P_one() to estimate the dispersion of the P-T values.

P_two() estimates the statistical dispersion of the background noise and calculates the four thresholds that are used in the detector. Remember

that P_one() finds the maximum of 20 rectified P-T values and sends this maximum to P_two(). P_two() averages *val_avg* (typically 8 or 16) of these maximums. The average thus obtained is *twosd*. For zero-mean normally distributed P-T values, *twosd* would be an estimate of twice the sample standard deviation of the P-T values. (See Murdock and Hutt, 1983, for a comparison between the measured sample standard deviation and the estimate of it hereby.) P_two() calls Xth() to calculate the thresholds from *twosd*.

Returns: Nothing (Modifies global variables only)

Fatal Errors: No fatal exits

– Xth() –

Called: By P_two() to calculate each of the four thresholds when *twosd* is calculated.

This routine forms the thresholds from *twosd* and from the encoded factors xth_i (that were input by the operator). The reason for this routine is to circumvent long multiplication of 32 bit integers.

Returns: The threshold value (*th1*, *th2*, *th3*, or *thx*)

Fatal Errors: No fatal exits

– Event() –

Called: By E_detect() when P_one() returns TRUE (ie, for each P-T value).

Routine Event detects signals using thresholds *th1* and *th2*. Typically, an event may be detected if 4 P-T values are greater than *th2*, or if 3 P-T values are greater than *th2* and one (or more) of the 3 is greater than *th1* (Murdock and Hutt, 1983). However, restrictions apply: All of the P-T values must be in a time window (typically 4 sec), and the P-T values must "look like" they are part of a signal. Here, "look like" means that two P-T values must not occur too close together or too far apart. If they are too close together the second P-T value will be discarded and if they are too far apart the window will be moved.

Upon detection, an event is declared when enough P-T values have been processed to estimate the period of it.

Returns: TRUE when an event is declared

FALSE, otherwise

Fatal Errors: No fatal exits

– Wbuff() –

Called: By `Event()` when an event might be in progress, but not yet declared.

This routine updates buffers that are needed when an event might be in progress but not yet declared.

Returns: Nothing (Modifies global variables only)

Fatal Errors: No fatal exits

– Ibingo() –

Called: By `Event()` when an event has been detected.

It is useful to note that although Ibingo is called when an event is detected, an event is not declared (i.e. routine `Event()` returns true), until enough P-T values have been processed to estimate the period of the signal. The purpose of `Ibingo()` is to set parameters for processing the interval of the event.

Returns: Nothing (Modifies global variables only)

Fatal Errors: No fatal exits

– Onsetq() –

Called: By `E_detect()` when `Event()` returns TRUE.

The buffers for each candidate signal have at least 4 P-T values before the first one that was $\geq$ *th2*. `Onsetq()` compares the last two of these four with yet another threshold, *th3* (*th3* $<$ *th2*). In addition, a test is performed to see whether or not the P-T value looks like it is part of the signal. Here "looks like" is determined by the period of the signal. These tests are to search for a signal onset that is smaller than *th2*. When the first P-T value of the signal is found, it is flagged. In `P_one()`, recall the reference time of each P-T value is given when the P-T value is declared; hence the time is for the "trailing edge" of the P-T value. Therefore the signal onset occurs before the time of the first P-T value of the signal. The algorithm considers two possibilities for the onset: It is either the time of the P-T value that immediately preceeds the signal, or if this P-T value occurs too far ahead of the first P-T value of the signal, a correction is applied to the time of the first P-T value of the signal. Here "too far" is determined by the measured period of the signal. The correction (0 or 500 ms for the SP and BB) and an index to the reference P-T are sent to routine `Onset()` for conversion to Universal Time.

In addition to this index and other parameters of the signal, `Onsetq()` sends `Onset()` the amplitudes of the two P-T values that occur on either side of the first P-T value of the signal. These five P-T values (the two prior to the signal and the first three of the signal) are used by `Onset()` to estimate the quality of the time determination of the beginning of the signal (Murdock and Hutt, 1983).

Returns: Nothing (Modifies global variables only)

Fatal Errors: No fatal exits

– Onset() –

Called: By `Onsetq()` each time `Onsetq()` is called.

Calculates the period of the declared signal, its maximum amplitude, the SNR (signal to noise ratio) series (a quality evaluation, see Murdock and Hutt, 1983), and converts the reference sample number of the signal onset to time of signal onset. The output to a log is made here.

Returns: Nothing (Modifies global variables only)

Fatal Errors: No fatal exits

– Period() –

Called: By `Onset()` each time `Onset()` is called.

The routine `Period()` calculates the period of the detected signal from the number of samples per period. The purpose of this routine is to circumvent floating point operations.

Returns: Nothing (Modifies global variables only)

Fatal Errors: No fatal exits

– Time_f() –

Called: By `Onset()` each time `Onset()` is called.

This routine calculates the onset time of the detected signal from a reference sample number and a time correction. One purpose of this routine is to circumvent floating point operations.

Returns: Integer seconds of start time that will be added to the time in the header of the seismic data record

Fatal Errors: No fatal exits

– Ck_t_kont() –

Called: By `Time_f()` each time `Time_f()` is called, returns immediately if the record length is less than one minute.

`Ck_t_kont()` is implemented to process records that are one minute or longer in length. For such records, it reduces the reference sample number to less than the number of samples per minute, if necessary, and adjusts the time field that was read in the record header accordingly.

Returns: Adjusted reference sample number

Fatal Errors: No fatal exits

– Count_dn() –

Called: By `E_detect()` for each seismic data sample after an event is declared, and while a contained counter is greater than zero.

This routine is a clock that times the interval of the event and sets flags for processing the coda of the event.

Returns: Nothing (Modifies global variables only)

Fatal Errors: No fatal exits

Chapter 4

Example Drivers

In this chapter we give two examples of drivers for the detector. The first one is a simple driver which uses few of the data management support routines that are described in Chapter 5.

This is a very basic idea of how to interface to the event detector. The user is providing the data, which may or may not be filtered. The user must provide the information in the example below, plus the values for the structure *struct r_time* (see cscont.h, App A). The time loaded in *r_time* is the time of the first sample of *thedataarray*. The macro `setallparams` (see cscont.h) loads the parameters of the detector and the routine `Cont_setup()` (Appendix A) initializes the continuity structure struct *con_sto*. (Whereas here we are processing only 1 channel of data, the continuity structure is needed for multi-channel applications.) Although the program can be modified to accept any sample rate, without the modifications use only those that divide into 1000 without any remainder. Our field experience is limited primarily to 1, 20, 40 and 100 samples/sec, though tests have been run at many other sample rates.

```
#include "detect.h"              /* Make sure everything is defined */

main()
{

        struct detect_info thedetector;          /* Data (see App A) */
        struct con_sto  thecontinuity;           /* Detector continuity */

        LONG    thedataarray[1000];              /* Where the data goes */

/*-----------Get the data structures above properly filled out-----------*/

        thedetector.samrte = 10000;      /* Sample rate * 10000 (here 10SPS) */
        thedetector.datapts = 1000;      /* Quantity of data points */
        thedetector.indatar = thedataarray;      /* Where data will be */
        thedetector.detname = "SP";      /* The name for the detector */
        thedetector.incontd = &thecontinuity;    /* Where continuity is */

/*-----------Initialize the continuity structure-------------------------*/
```

```
        Cont_setup(&thecontinuity);      /* Appendix A */

/*----Set the event detection parameters into the continuity structure--*/
/*                See Chapter 6                                          */

        /* Load values into the continuity structure */

        setallparams(&thedetector,              /* A macro (in cscont.h) */
                4,    /* Filhi (decimal) -- filhi << fillo */
                40,   /* Fillo (decimal) -- 0 << fillo < iwin */
                80,   /* Iwin  (decimal) -- don't wrap around E_B */
                4,    /* N_hits (decimal) -- reccommend 4 or 5 */
                020,  /* Xth1 (octal) -- xth1 > xth2, xth1 <= 0377 */
                015,  /* Xth2 (octal) -- xth2 > xth3, rec. 015-017 */
                010,  /* Xth3 (octal) -- reccommend 10 */
                015,  /* Xthx (octal) -- ~013 <= xthx <= 0377, rec. 015-030 */
                500,  /* Def_tc (decimal) -- 1/2 nominal signal period, ms */
                507,  /* Wait_blk (decimal) -- See Chap. 6 parameter 15 */
                8);   /* Val_avg (decimal) -- 1 <= val_avg <= 16 */

/*-----------------Display parameters on the console--------------------*/
/*                    Disp_par is in Appendix C                         */

        Disp_par(&thedetector);

/*---------------User's processing loop-----------------------------*/

        FOREVER {

                if (exit_condition) break;

                Users_data_routine(thedataarray,&thedetector.startt);
                        /* User's routine to fill in the data array
                           and provide the starting time */

                E_detect(&thedetector);

        }

        /* Clean up */

}
```

Here is an example of a slightly more complex detector driver which establishes multiple

detectors and uses many of the data management support routines.

This idea is somewhat more complicated than the one shown above. Here we are processing two, instead of one, channels of information. Furthermore, these data are filtered. Moreover, memory is allocated dynamically by the subroutines that employ malloc. (All of these routines are described in Chapter 5, and are shown in Appendix B). The driver is designed to process an arbitrary format (here it is CDSN). In the process below, the data are input from magnetic tape that contains VLP, LP, BB, and SP data. Conceptually, as denoted by the comments, we search for the data of interest (here BB, and SP). Routine E_cdsnload() returns FALSE if the record that was read was not BB or SP. Otherwise, it decodes header information and converts and loads the 16-bit gain-ranged seismic data sample into a 32-bit integer. When all of the integers of the record have been filtered, they are input to the detector.

Note that these examples are not strictly runnable C code as the user must supply some code for loading the seismic data which must be passed to the decoder.

```
#include "detect.h"                /* Define everything important */

#define MAX_LOOKBACK               40 /* For the FIR filters */
#define MAX_DATA_POINTS            1014 /* Size of CDSN data packet */

#define CDSN_INPUT_RECORD_SIZE  2048     /* Size of CDSN record in bytes */
                                         /* 20 bytes header, 2028 data */
#define NUM_DETECTORS              2

BYTE    rawcdsn[CDSN_INPUT_RECORD_SIZE];        /* Storage buffer */

main()
{

        LONG    *Common_data_buffer;    /* Array of longs for detector */
        struct detect_info detector[NUM_DETECTORS];     /* Detector info */
                                        /* (See Appendix A) */
        BOOL    FfirBB20(),             /* We will use this broadband filter
*/
                FfirSP40(),             /* And this short period filter */
                useflag;                /* Flag for detection below */
        WORD    i,j;                    /* Indexes */

/*----A Common data buffer will be allocated.  One could do this manually----*/

        Common_data_buffer = E_buffer(MAX_DATA_POINTS,MAX_LOOKBACK);

/*-----------------Create the short period detector-----------------------*/
/*  ,                          calls Cont_setup                           */
```

```
        if (!(E_create(&detector[0],     /* Detector 0 is SP */
                18,                      /* Number filter lookback points */
                1,                       /* SP record id # */
                "SPZ",                   /* Give short period a name */
                Common_data_buffer,      /* Where data is stored */
                FfirSP40()) {            /* Name of filter */

                printf("Unable to E_create() the SPZ detector\n");
                exit();

        }

/*-----------------The SP parameters must be set-------------------------*/

        setallparams(&detector[0],       /* Set up SP detector */
                8,      /* Filhi (decimal) */
                80,     /* Fillo (decimal) */
                160,    /* Iwin  (decimal) */
                4,      /* N_hits (decimal) */
                020,    /* Ith1 (octal) */
                015,    /* Ith2 (octal) */
                010,    /* Ith3 (octal) */
                015,    /* Ithx (octal) */
                500,    /* Def_tc (decimal) */
                507,    /* Wait_blk (decimal) */
                8);     /* Val_avg (decimal) */

/*-----------------Create the broad band detector------------------------*/
/*                          calls Cont_setup                             */

        if (!(E_create(&detector[1],     /* Detector 1 is BB */
                38,                      /* Number filter lookback points */
                2,                       /* BB record id # */
                "BBZ",                   /* Give broad band a name */
                Common_data_buffer,      /* Where data is stored */
                FfirBB20()) {            /* Name of filter */

                printf("Unable to E_create() the BBZ detector\n");
                exit();

        }

/*-------------------Set params for the BB detector----------------------*/
```

```
        setallparams(&detector[1],4,40,200,5,077,017,010,030,500,1014,16);

/*--------------------Display the parameters on the STDOUT-----------------*/

        for (i=0; i<NUM_DETECTORS; i++)
                Disp_par(&detector[i]);

/*--------------------Begin the data I/O loop-----------------------------*/

        FOREVER {

                if (end_of_file_in_all_input_data) break;

                --- Data is loaded into place from raw input ---
                        ------Data is placed in rawcdsn------

/*-------Search for the desired data, convert, and event detect----------*/

                for (i=0; i<NUM_DETECTORS; i++) {

                        useflag = E_cdsnload(&detector[i],rawcdsn,0);
                        /* Search for the ith channel, and when found,
                                convert component 0 (Z) */

                        if (useflag) {  /* useflag = TRUE for BB and SP only
*/

                                E_filter(&detector[i]);
                                E_detect(&detector[i]);

                        }
                }
        }

        Close your files here

}
```

For an example of a more complex event detector driver, please see the source to Detmain.c in Appendix C. This is a functioning piece of code, and is used at the ASL for testing and development of the event detector code systems.

Drivers can range from very simple to very complex. It is intended however that the event detector itself be viewed and used as a complete blackbox which only need be provided with appropriate data. It is hoped that the user would not need to modify the detector inside; however, again, we have tried to supply sufficient documentation to aid in such an

effort should one so desire.

Chapter 5

Support Routines Overview

The following is a description of the subroutines provided to take care of details involving data I/O management and other housekeeping support. As noted by example 1 in chapter 4, these routines are not required for use of the event detector, but are available to take care of details for which the user might otherwise need to provide similar code.

– Subroutine: E_buffer() –

Called: `LONG *E_buffer(maxdata, maxlookback)` – called to allocate the buffers for the seismic data samples. Lookback is required because some of the routines use FIR filters. Allocates an array (size *maxdata* + *maxlookback*) of longs and returns a pointer to the new array.

maxdata The maximum number of data points expected. This must be as large as or larger than the maximum number of data points which the decoder (see `E_cdsnload()`) will need.

maxlookback The maximum number of lookback points which will be required by any of the filters. May be 0 (zero) if no filtering is to be done, or if the user will be managing the filtering.

Returns: TRUE – All was sucessful.

FALSE – Some allocation failed. Probably due to insufficient memory.

Fatal Errors: No fatal exits

– Subroutine: E_create() –

Called: `BOOL E_create(detector, looksize, recordtype, detname, dataarray, filter)` – called by the main user program to initialize the user's detector structures. The user creates an array of detector structures (*struct detect_info*), and calls `E_create()`, once for each detector to be established. `E_create()` will allocate memory internally for the lookback array, and the private copy of the continuity structure each detector requires. It then sets the other values in the *detect_info* structure to initial values.

detector The pointer to the *detect_info* structure for this detector. Note that the user must allocate the storage for this. Done by simply specifying "`struct detect_info mycopyofdetect`", and passing "`&mycopyofdetect`" to `E_create()`. There should be one of these structures for each detector to be created.

looksize This is the maximum number of lookbacks which will be required for the current detector's data filter. Note that *looksize* may be less than, but must not be greater than, the maximum lookback parameter sent to `E_buffer()`. If the input for the current detector will not be filtered, use a 0 (zero) here. An array of *looksize* size will be allocated, and a pointer to it will be placed in the user's detector structure.

recordtype This is an arbitrary component type identifier. It is used to uniquely identify the different sample rates and/or individual instruments in the input data. *Recordtype* is also used by the decoder to determine which data is to be processed. (Our tape format has SP, BB, LP, and VLP data, typically we process only the SP and BB data.) See `E_cdsnload()` for an example of how *recordtype* is used to select the components to be processed.

detnam This a pointer to a string which contains the name of the detector. Used on various printouts to identify the component and channel.

dataarray This is the pointer to the array of longs which will be used to store the data and the lookbacks during conversion and event detection. Ordinarily, *dataarray* is the pointer returned by the `E_buffer()` routine. Alternatively, the user may allocate *dataarray* by hand and send the pointer to `E_create()`. The array must be as large as the maximum lookback anticipated plus the maximum number of data points that the decoder being used will need.

filter This is the pointer to the filter function that will be used to process the decoded data when `E_filter()` is called. The user may supply a pointer to a filter provided in the library, or may use one of these filters as a template to code a custom filter. If no filter is required, NULL should be specified here. Note that if no filtering is desired, but a lookback greater than zero is specified, the `Fnull()` filter should be specified here to get the data array moved so it is positioned properly for the detector. The following is a situation in which a non-zero lookback, along with a null filter, would be employed: The user has developed custom filters which require lookback space but will not use the `E_filter()` facility.

Returns: TRUE - All allocations were sucessful.

FALSE - Some allocation failed. There was probably not enough memory to satisfy the request.

Fatal Errors: No fatal exits

– Subroutine: E_cdsnload() –

Called: `BOOL E_cdsnload(detector, indata, offset)` – Routine used to convert from a raw (station tape) format to the LONG (long) array format.

Routine first checks to verify that the data in *indata* matches the selection criteria in *decoder* (specified with recordtype to `E_create()`). If they do not match, it stops decoding, and returns FALSE.

If the record type of the input data (determined from the header) matches the recordtype in the detector structure (specified when `E_create()` was called), this data is to be event detected. The routine goes on to convert the raw input data (here the China Network gain-ranged format) into the elements of our 32-bit data array. The value of the *looksize* parameter which was specified to the `E_create()` routine will be used as the initial position for storing data in this array.

This routine might be used as a template for the users to write their own decoders.

detector The *struct detect_info* detector structure. Contains the information needed to operate the current detector.

indata Pointer to the raw CDSN data record to be processed.

offset The multiplexed channel offset. Here 0=Z, 1=N–S, and 2=E–W.

Returns: TRUE - Data matched, and was decoded. It can now be filtered, and sent to the event detector.

FALSE - This data did not match the current detector. The driver program should still attempt to run this data on all of the other detectors.

Fatal Errors: No fatal exits

– Subroutine: E_filter() –

Called: `BOOL E_filter(detector)` - Filter driver. Performs housekeeping involved with the lookback data arrays, and invokes the user's filter to actually filter the input data.

Before this routine is called, the data array should be filled with data by the decoder. Upon exiting `E_filter()`, the filtered data will begin at the top of the data buffer and will be ready for event detection.

Returns: TRUE - Success.

FALSE - Some processing error - not expected to happen.

FIR filters for `E_filter()`			
Name	Nominal Rate	General Type	Figure
`FfirSP10()`	10 SPS	Band Pass	See Figure 5.1
`FfirSP20()`	20 SPS	Band Pass	See Figure 5.2
`FfirSP40()`	40 SPS	Band Pass	See Figure 5.3
`FfirBB20()`	20 SPS	High Pass	See Figure 5.4
`Ffpav()`	Any	Low Pass	See Figs 5.5 & 5.6

Table 5.1: FIR filters

Fatal Errors: No fatal exits

– Subroutine: E_remove() –

Called: `E_remove(detector)` – Destroy a detector. Puts the detector structure back to the state it was in before `E_create()` was called. It deallocates the continuity structure and the lookback storage areas. It also sets variables so that the decoder will never select this detector.

Probably, this routine is not useful in normal situations unless all malloc'd areas must be free'd on your operating system before exiting the program. (One such operating system is used on the AMIGA). This routine might also be useful when the detector is implemented to process incoming telemetered data, and there is a need to dynamically reconfigure the event detectors. Once `E_remove()` is called, `E_create()` can be called to initialize a new detector using the old detector slot.

Returns: TRUE - success.

FALSE - Unanticipated problem.

Fatal Errors: No fatal exits

There are also 5 FIR filters provided to reduce noise that is anticipated to be outside of the signal band. The code is of general form, and might be adapted by the user to construct new custom filters. See Table 5.1 for a list of these filters.

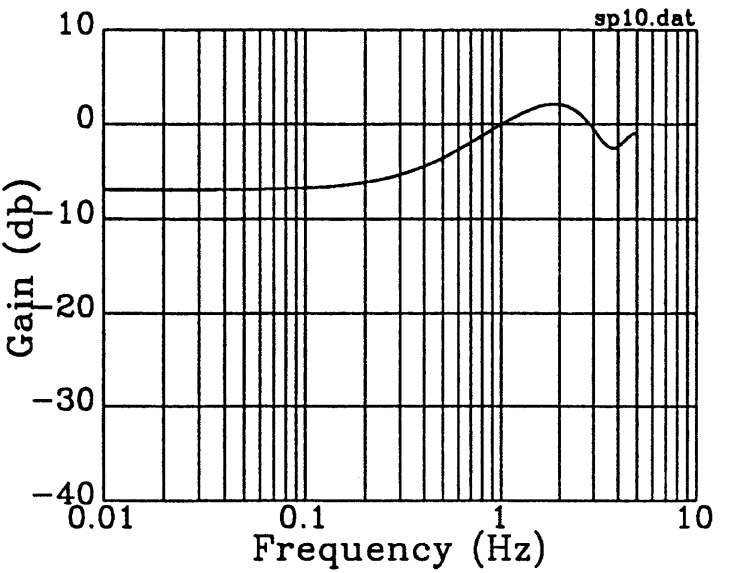

Figure 5.1: Response of `FfirSP10()` filter at 10 SPS

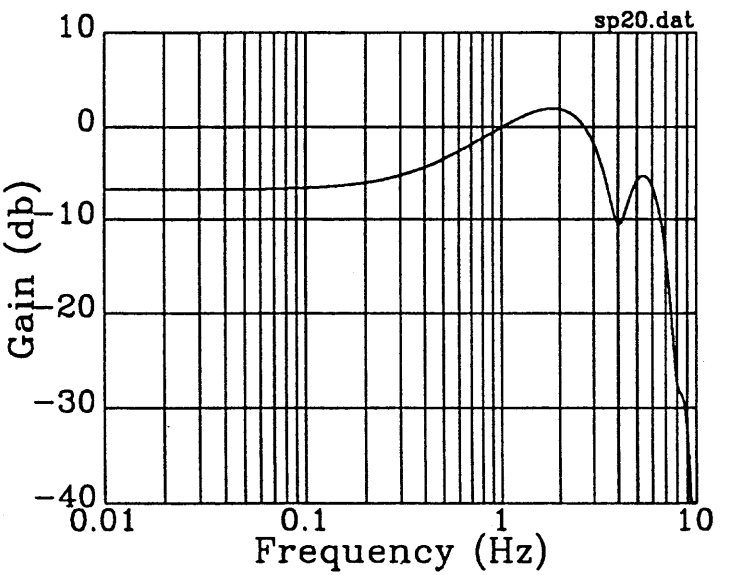

Figure 5.2: Response of `FfirSP20()` filter at 20 SPS

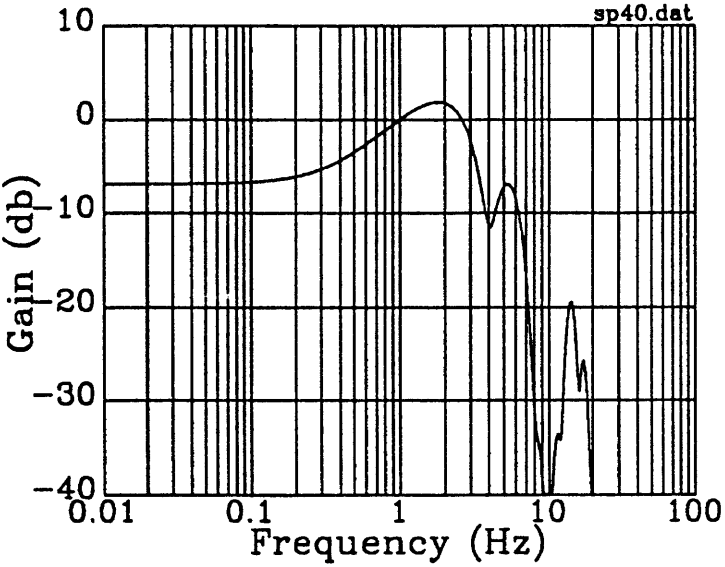

Figure 5.3: Response of `FfirSP40()` filter at 40 SPS

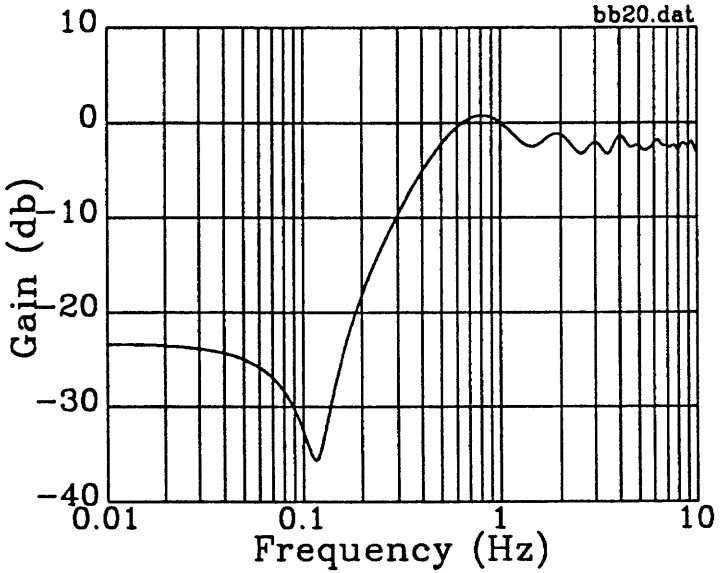

Figure 5.4: Response of `FfirBB20()` filter at 20 SPS

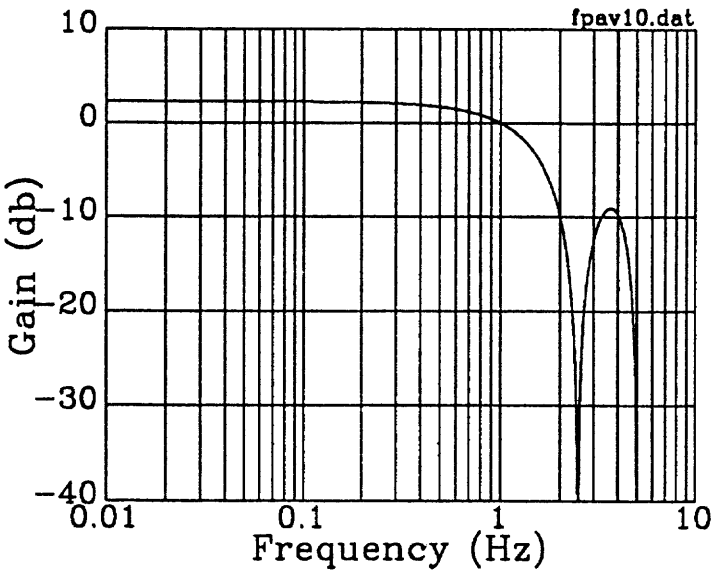

Figure 5.5: Response of `Ffpav10()` filter at 10 SPS

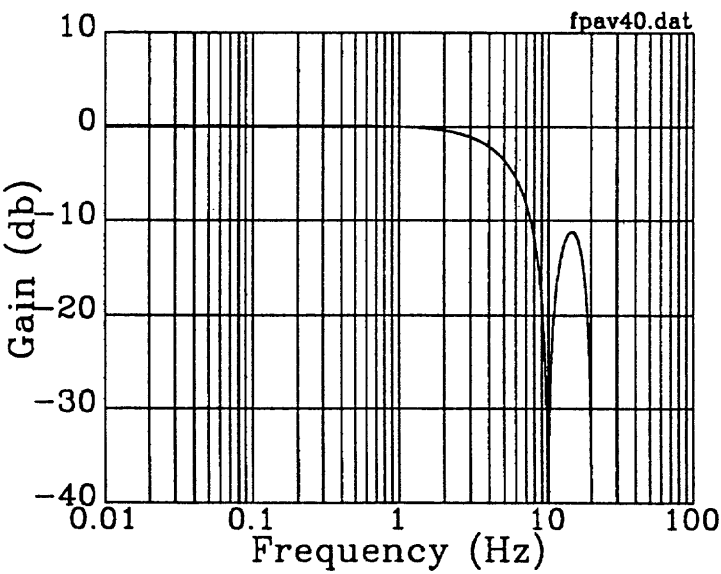

Figure 5.6: Response of `Ffpav40()` filter at 40 SPS

Chapter 6

Event Detector Parameters and Onset Printouts

6.1 Event Detector Parameters

An example of parameters that are input to the detector is given below. In this particular example, the detector is operating on the vertical component of the BB, SP, and LP data channels of the CDSN system.

The format of the parameter file, parameters pertaining to device names, and blocking factors are pertinent only to the detmain driver which is listed in Appendix C. The rest of the parameters, however, are used to tune the detector itself, or to select data that will be processed.

The explanation of the input parameters (see Table 6.1) is:

1. The name of the file (device) that will be read by the UNIX System.
2. The name of the component that will be processed.
3. The filter type that will be used (see Detmain source in Appendix C for table which maps these codes to filter routines).
4. The offset into the multiplexed data of each channel. For instance the SP have three

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
/dev/nrmtb	BBZ	2	0	16	4	40	199	77	17	10	30	5	500	1014	16
/dev/nrmtb	SPZ	1	0	16	8	80	160	20	15	10	15	4	500	0507	8
/dev/nrmtb	LPZ	3	0	16	8	20	50	40	17	10	30	4	12000	0507	16

Table 6.1: Parameters used for event detection.

Read in by the `Detmain` driver (see Appendix C).

components of data multiplexed in one record. 0 = Z, 1 = N, 2 = E.

5. Blocking factor for reading the tape (in this instance, the tape controller will cause 16 records to be read in one block).

6. **Filhi**, samples (see Murdock and Hutt, 1983, pg. 4). This parameter is used to control winnowing. In particular, it can be used to reject spiking. Should spiking be a problem, set *filhi* equal to the number of samples between (and including) the first maximum (or minimum, if the spike is negative going) and the subsequent minimum (or maximum). Set this way, *filhi* will cause zero weight to be assigned to the second P-T of the spike. However, it must be remembered that setting *filhi* to reject such spikes might cause the detector to be less sensitive to signals of short duration. The value of *filhi* should be much less than that of *fillo*. See `Event()`.

7. **Fillo**, samples (see Murdock and Hutt, 1983, pg. 4). This parameter controls how the window (*iwin*, below) moves. (As a rule of thumb, set *fillo* equal to twice the expected period of the signal, or $\frac{1}{2}$ the period of the dominant long-period noise, whichever is smaller.) *Fillo* causes the window to be moved if only one weighted P-T value is $\geq$ *th2* in the subwindow defined by *fillo*. See `Event()`.

8. **Iwin**, the window length, samples. This value should be larger than *fillo*. Assuming the time criteria of *filhi* and *fillo* are met, a detection will occur if, in the window, *nhits* P-T values are $\geq$ *th2*, or, if 3 P-T values are $\geq$ th2 with at least one of the 3 being $\geq$ *th1*. Typically, we set the window length equal to twice *fillo*. *Fillo* controls the number of detections (and false alarms) more strongly than *iwin*. See `Event()`.

9. **Xth1**, dimensionless. The encoded factor (octal, described more fully below) of *twosd* that forms *th1*. We usually start tuning the detector with *xth1* set to 20_8 and raise it to reduce false alarms. The value of this parameter must be no larger than 377_8, and *xth1* > *xth2*. See `Xth()`.

10. **Xth2**, dimensionless. The encoded factor (octal, described more fully below) of *twosd* that forms *th2*. We usually start tuning the detector with *xth2* set to 15_8 and raise it to reduce false alarms. The value of this parameter must be no larger than octal 377_8, and *xth2* < *xth1*. Normally, it is 15_8, 16_8, or 17_8. See `Xth()`.

11. **Xth3**, dimensionless. The encoded factor (octal, described more fully below) of *twosd* that forms *th3*. We use 10_8 for *xth3*. It is used to time the onset of signals. The value 10_8 will produce *th3* equal to approximately two standard deviations of the background. Hence, the SNR series (the quality evaluation) is easily related to the statistics of the background. The value of this parameter must be no larger than 377_8. See `Xth()`.

12. **Xthx**, dimensionless. The encoded factor (octal, described more fully below) of *twosd* that forms *thx*. *Thx* is used to inhibit anomalously large values from raising the estimate of normal background. Typically, we start tuning with *xthx* of 15_8. If we believe that *thx* is not tracking the background properly, we raise *xthx* to allow larger values to contribute to the estimate of normal background. *Xthx* should be set no smaller than about 13_8 and must be set no larger than 377_8. See `Xth()`.

xthi (octal)	Value Factor (decimal)	xthi (octal)	Value Factor (decimal)
8 10	1.000	24 30	3.000
9 11	1.125	32 40	4.000
10 12	1.250	40 50	5.000
11 13	1.375	48 60	6.000
12 14	1.500	56 70	7.000
13 15	1.625	64 100	8.000
14 16	1.750	192 300	24.000
15 17	1.875	255377	31.875
16 20	2.000		

Table 6.2: Encoded threshold factors generated by `Xth()` subroutine.

13. **N_hits** ("m" in the report of Murdock and Hutt, 1983). The number of weighted P-T values in the window that is $\geq$ *th2*, but $<$ *th1*, that will cause a detection. Normally, use 4_{10}. Values less than 4_{10} will disable the effect of *th1* and hence are not recommended. Increasing *n_hits* will both decrease the sensitivity of the detector to small emergent events and decrease the number of false alarms, with the other input parameters held constant. See `Event()`.

14. **Def_tc**, milliseconds. If the detector can not find a suitable onset for the signal, it will use the time of the first P-T value of the signal minus *def_tc*. Hence *def_tc* should be one-half of the expected period of the signal. In later revisions, *def_tc* might be calculated by the algorithm. See `Onsetq`.

15. **Wait_blk**, samples. After an event has been declared, the detector will be disabled for (NOR_OUT − TIM_OFF) × *wait_blk* samples. The detector will return TRUE for NOR_OUT × *wait_blk* samples. Thus TIM_OFF is used to suppress spurious detections in the coda of the event and NOR_OUT is used to determine the interval that might be recorded on magnetic tape, if such recording is indeed desired. See `Count_dn()`.

16. **Val_avg**, values per average. This is the number of sets that is used to form *twosd*. The maximum permitted number is hardcoded to 16_{10} (see Cscont.h). Values larger than the dimensioned array likely will cause aberrant behavior of the detector. Smaller values produce a quicker response in the estimate of *twosd*, but can also produce larger short-term variations in the estimate. See Cscont.h, `P_two()`.

Table 6.2 defines the octal codes that are used for *xth1*, *xth2*, *xth3*, *xthx*. In the way we operate, decimal values are input for all of the parameters except xthi, and octal values are input for xth_i. Octal values were selected for consistency with the on-line SRO detector. In addition, at compile time, one may set the compiler flag LOCAL_EV (see `Onset()`) if

1	2	3	4	5	6	7	8	9	10	11
BBZ	d	0	01343	1986	168	20:14:14.760	127	0.450	32	B
SPZ	d	0	00356	1986	168	20:14:14.655	2066	0.425	341	A
BBZ	d	0	10300	1986	168	20:25:28.510	98	0.200	32	B
SPZ	d	0	01344	1986	168	20:41:27.105	1365	0.500	330	A
SPZ	c	0	11221	1986	168	21:04:16.405	805	0.375	417	A
SPZ	c	2	11122	1986	168	21:05:17.755	1971	0.550	1001	B
SPZ	c	0	00300	1986	168	21:16:24.680	731	0.125	257	A
SPZ	d	1	00121	1986	168	21:39:33.055	829	0.250	365	A
SPZ	c	1	10158	1986	168	21:50:50.730	3199	0.525	379	A

Table 6.3: Sample onset printout from event detector

local and regional events are the primary interest. Setting this flag aids in the detection of secondary phases of regional events.

6.2 Parameters That Are Output By The Detector

This is the explanation of the detector output (see Table 6.3).

1. The component name.
2. Estimate of the direction of the first break (polarity of the initial onset of the signal). The two possibilities are **c** or **d** (compression or dilatation).
3. The number of P-T values that Onsetq() looked back to find the onset of the signal (0, 1, or 2).
4. The quality evaluation of the estimate of the onset (see Murdock and Hutt, 1983 pg. 16). Dimensionless.
5. The year in which the signal occurred.
6. The day of the year.
7. The hour, minute, second of the estimated onset.
8. The maximum amplitude (digital counts) of the first 4 cycles of the signal.
9. The average period (seconds) of the first 4 cycles of the signal.
10. The value of the background (digital counts) at the time the signal was detected (it is an estimate of twice the sample standard deviation of the P-T values; *twosd*).
11. The algorithm that produced the detection:

A - 1 P-T $\geq$ *th1*, 3 P-T $\geq$ *th2*.

B - *n_hits* P-T $\geq$ *th2*. (see code of `Event()` for more information)

```
BOOL E_cdsnload(detector, indata, offset)
struct detect_info *detector;
UBYTE * indata;
WORD offset;
{

        WORD i, rect, numcomp, leap;
        LONG samrat, sampr, nmsec, fmsec, fsec, fmin, fhour;
        BOOL nak;

        WORD lp, j, gr, ct;
        LONG l_data, *oarray;
        UBYTE * bytarr;

        samrat = HINIB(16) * 10000L +
            LONIB(16) * 1000L + HINIB(17) * 100L;

        i = samrat / 10;
        switch (i) {
            case 4000:          /* 40 samples per second */
                rect = 1;
                break;
            case 2000:          /* 20 samples per second */
                rect = 2;
                break;
            case 100:           /*  1 sample per second */
                rect = 3;
                break;
            case 10:            /* 0.1 sample per second */
                rect = 4;
                break;
            default:            /* Unknown, therefore illegal sample rate */
                rect = 0;
                break;
        }

        if (detector->drectyp != rect)
            return(FALSE);                      /* Not this one */

        detector->samrte = samrat;              /* Detector needs this */

        numcomp = LONIB(8);     /* Number of components in this record */

/*-----------------Compute a complete date----------------------------*/
```

Chapter 7

Characteristics of the Code

This chapter is intended to give the user an overview of how fast the code might execute on the user's machine. In viewing the data, one must consider that the input data sizes are records of 1024 or 2048 bytes — in a real-time interrupt-driven system, the record sizes will impact on the speed of operation. Another factor that will impact on the speed of execution is the dominant frequency of the input data. The frequency affects the speed because the first operation of the detector is to find the local maximums and minimums of the data, and these are the data (together with their associated times) that are processed by the remainder of the detector.

For a conservative estimate of the speed, we have implemented the C code as described herein, rather than integrating the code so that execution could be performed without calling functions: Placing some of the heavily used code inline likely will result in a savings of 20-30 percent in the time of execution.

The tests herein were performed on the following two machines and software:

- Machine: DEC PDP 11-70
  - Operating System: Berkeley UNIX 2.9
  - Compiler: Standard BSD 2.9 PDP-11 C
- Machine: Sun Model 3/160 Work Station. (Motorola 68020 microprocessor, 16.67 Mhz, *mathematics coprocessor disabled.*)
  - Operation System: Sun OS V3.5 (BSD UNIX 4.2 equiv)
  - Compiler: Standard Sun C Compiler

To show the likely sizes of the execution modules, Table 7.1 gives them for the Sun Model 3/160.

To demonstrate the times of execution for the detector and the support routines, data of 20 and 40 samples/second have been processed by the PDP-11/70, and data of 1, 20, and 100 samples/second have been processed by the Sun 3/160. Because the runs were made several years apart, the input data are not the same for the two different machines, hence an exact comparison between the two machines cannot be made. However, it was deemed useful to have execution times on a modern microprocessor as well as on a familiar minicomputer, therefore we have given the execution times for both the Sun 3/160 and the DEC 11/70.

Table 7.2 shows the results for the 20 sample/second data of the 11/70. Routine `P_one()` is called once per sample and routine `Event()` is called once per P-T value. Hence, the average number of samples/P-T value is easily calculated to be 2.2, and the number of the average frequency of the input data can be determined to be 4.5 *Hz* (remember that there are 2 P-T values/cycle). The number of P-T values processed per second is calculated by dividing the total calls to `Event()` (778,216) by the total time required to execute the detector (412 sec, including overhead); which yields a speed of 1886 P-T values/second. Similarly, one can calculate the speed of execution, relative to real time, using data for the detector itself as 209 × real time, or for the detector plus support routines as 92 × real time, both including the unix overhead.

Table 7.3 demonstrates execution times for the 40 sample/second data on the 11/70. As for the above, the execution times may be calculated:

- Samples/P-T Value = 4.5, which translates to 4.4 *Hz*.
- P-T values processed per second = 1398.
- Detector speed, relative to real time = 157 × real time (includes all of system overhead).
- Detector plus support = 75 × real time.

It is interesting to note that the execution times do not increase in direct proportion to the sample rate: The 40 sample/second data is not processed one-half as fast as the 20 sample/second data. This is due primarily to the compression of the data by `P_one()`, as discussed above.

Table 7.4 demonstrates the execution of times for the 1 sample/second data on the Sun 3/160. As for above, the execution times may be calculated.

- Samples/P-T Value = 13.4, which translates to .037 *Hz* (26.8 *sec*).
- P-T values processed per second = 293.
- Detector speed, relative to real time = 3936 × real time (includes all of system overhead).
- Detector plus support = 408 × real time (Note the large time required for floating point operations for the 6 pole low-pass recursive filter).

Table 7.5 demonstrates the execution of times for the 20 sample/second data on the Sun 3/160. As for above, the execution times may be calculated.

- Samples/P-T Value = 2.5, which translates to 4.0 *Hz*.
- P-T values processed per second = 3664.

- Detector speed, relative to real time = 460 × real time (includes all of system overhead).
- Detector plus support = 262 × real time.

Table 7.6 demonstrates the execution of times for the 100 sample/second data on the Sun 3/160. As for above, the execution times may be calculated.

- Samples/P-T Value = 4.39, which translates to 11.4 *Hz.*
- P-T values processed per second = 1621.
- Detector speed, relative to real time = 71 × real time (includes all of system overhead).
- Detector plus support = 11 × real time (Note large time required for floating point operations for the 2 pole low-pass recursive filter).

To calculate the likely execution times for other data that may be of interest, use the information that routine `P_One()` is called once per sample, routine `Event()` is called once per P-T value (twice per cycle), routine `P_two()` is called approximately once every 20 P-T values, and routine `Xth()` is called four times per call to `P_two()`.

It is important to note that the execution speeds calculated (above) inlcude all of the unix overhead. If we exclude the unix overhead plus the execution times for the support routines, the execution for the detector subroutine itself may be calculated. These data may be useful in comparing the detector described herein with others. The speed of execution for the detector routine are as follows:

- On the 11-70, 20 samples/second data is 289 × real time.
- On the 11-70, 40 samples/second data is 236 × real time.
- On the 3/170, 20 samples/second data is 541 × real time.
- On the 3/170, 100 samples/second data is 172 × real time.

Again, these speeds probably can be increased 20-30% by merely placing some of the more frequently called routines inline, rather than calling them as functions.

Comparitive sizes of code (bytes)					
Section	Routine_Name	Code	Data	Total	Source
Event Detector	Ck_t_kont()	436	4	440	3313
	Count_dn()	128	4	132	2479
	E_detect()	580	132	712	9010
	Event()	2112	4	2116	18833
	Ibingo()	88	4	92	3004
	Onset()	1620	80	1700	12572
	Onsetq()	1440	4	1444	10004
	P_one()	484	4	488	5052
	P_two()	436	4	440	3773
	Period()	148	4	152	1869
	Time_f()	320	4	324	3028
	Wbuff()	224	4	228	3229
	Xth()	296	4	300	2744
Data Support	FfirBB20()	456	4	460	3224
	FfirSP10()	132	4	136	2178
	FfirSP20()	168	4	172	2721
	FfirSP40()	272	4	276	2408
	Ffpav()	136	4	140	2193
	Fnull()	100	4	104	2174
	Cont_setup()	320	4	324	3042
	Detmain()	4672	628	5300	18107
	Disp_par()	196	152	348	1311
	E_buffer()	88	4	92	1260
	E_cdsnload()	1312	4	1316	4421
	E_create()	264	4	278	2694
	E_filter()	240	4	244	1563
	E_remove()	128	4	132	1238
Include Files	csconfig.h	n/a	n/a	n/a	1426
	cscont.h	n/a	n/a	n/a	7671
	csstnd.h	n/a	n/a	n/a	3437
	detect.h	n/a	n/a	n/a	1611

Table 7.1: Sizes of the C source files and compiled object files for event detection system as compiled on the Sun 3/160, C compiler. The n/a is used because the code sections of the include files are already added to the sizes of the routines.

CDSN BB Test					
Routine	Time (secs)	Calls	Section%	All%	ms/call
Data Support Section					
FfirBB20()	432.47	5106	83.5%	46.5%	84.70
E_cdsnload()	85.48	15600	16.5%	9.2%	5.48
E_buffer()	0.00	1	0.0%	0.0%	4.17
Cont_setup()	0.00	1	0.0%	0.0%	16.67
Event Detector Section					
P_one()	167.65	1725828	56.2%	18.0%	0.07
Event()	91.51	778216	30.7%	9.8%	0.12
P_two()	16.92	38909	5.7%	1.8%	0.30
Xth()	14.36	151384	4.8%	1.5%	0.08
Wbuff()	5.10	75125	1.7%	0.5%	0.06
Count_dn()	2.43	1109574	0.8%	0.3%	0.02
Ck_t_kont()	0.08	28	0.0%	0.0%	0.21
Onset()	0.03	28	0.0%	0.0%	1.28
Period()	0.00	28	0.0%	0.0%	0.43
Ibingo()	0.00	28	0.0%	0.0%	0.21
System Overhead Section					
unix overhead	114.43	n/a	100.0%	12.3%	n/a

Table 7.2: Example execution profile of the event detector that processes 20 samples/second data on the DEC 11/70. The n/a is used in columns which are not applicable to the statistic. The filter used has 37 coefficients, each of which is implemented with power-of-2 shifts for speed.

CDSN SP Test					
Routine	Time (secs)	Calls	Section%	All%	ms/call
Data Support Section					
FfirSP40()	427.54	10212	72.1%	37.5%	41.87
E_cdsnload()	165.40	15600	27.9%	14.5%	10.60
E_buffer()	0.02	1	0.0%	0.0%	16.67
Cont_setup()	0.00	1	0.0%	0.0%	8.33
Event Detector Section					
P_one()	245.46	3451656	67.1%	21.5%	0.07
Event()	92.39	766196	25.2%	8.1%	0.12
Xth()	12.00	150204	3.3%	1.1%	0.08
P_two()	11.64	38285	3.2%	1.0%	0.30
Count_dn()	2.92	154634	0.8%	0.3%	0.02
Wbuff()	1.45	23479	0.4%	0.1%	0.06
Onset()	0.10	78	0.0%	0.0%	1.28
Period()	0.04	78	0.0%	0.0%	0.43
Ck_t_kont()	0.02	78	0.0%	0.0%	0.21
Ibingo()	0.01	78	0.0%	0.0%	0.21
System Overhead Section					
unix overhead	182.29	n/a	100.0%	16.0%	n/a

Table 7.3: Example execution profile of the event detector that processes 40 samples/second data on the DEC 11/70. The n/a is used in columns which are not applicable to the statistic. The filter used has 17 coefficients, each of which is implemented with power-of-2 shifts for speed.

Steim LP Test					
Routine	Time (secs)	Calls	Section%	All%	ms/call
Data Support Section					
Cont_setup()	0.04	1	0.0%	0.0%	40.00
CtlWordType()	4.33	330240	0.3%	0.3%	0.01
E_detect()	7.78	1376	0.5%	0.5%	5.65
E_filter()	0.23	1376	0.0%	0.0%	0.05
E_steimload()	11.35	80000	0.8%	0.7%	0.14
Frecgen()	163.59	1376	11.6%	10.4%	118.89
FP overhead	1224.64	n/a	86.5%	77.5%	n/a
main()	3.16	1	0.2%	0.2%	3159.86
Event Detector Section					
Ck_t_kont()	0.00	14	0.0%	0.0%	0.00
Count_dn()	0.88	49189	4.4%	0.1%	0.02
Event()	3.07	48204	15.3%	0.2%	0.06
Ibingo()	0.00	14	0.0%	0.0%	0.00
Onset()	0.04	14	0.2%	0.0%	2.86
Onsetq()	0.38	14	1.9%	0.0%	27.14
P_one()	15.00	645730	74.9%	0.9%	0.02
P_two()	0.24	2119	1.2%	0.0%	0.11
Period()	0.00	14	0.0%	0.0%	0.00
Time_f()	0.00	14	0.0%	0.0%	0.00
Wbuff()	0.20	7796	1.0%	0.0%	0.03
Xth()	0.22	8432	1.1%	0.0%	0.03
System Overhead Section					
Unix overhead	144.02	n/a	100.0%	9.1%	n/a

Table 7.4: Example execution profile of the event detector that processes 1 sample/second data on the Sun 3/160. The n/a is used in columns which are not applicable to the statistic. A 6-pole low-pass recursive filter was used. Little effort was made to reduce the execution time for this filter.

Steim BB Test					
Routine	Time (secs)	Calls	Section%	All%	ms/call
Data Support Section					
Cont_setup()	3.03	1	0.3%	0.1%	3029.86
CtlWordType()	45.48	3335760	4.3%	1.8%	0.01
E_filter()	9.39	13899	0.9%	0.4%	0.68
E_steimload()	157.78	80000	14.9%	6.4%	1.97
FfirBB20()	842.01	13899	79.3%	34.2%	60.58
Frecgen()	0.48	0	0.1%	0.0%	0.00
main()	3.62	1	0.3%	0.1%	3619.84
Event Detector Section					
Ck_t_kont()	0.02	3115	0.0%	0.0%	0.01
Count_dn()	144.41	9888951	12.1%	5.9%	0.02
E_detect()	185.18	13899	15.5%	7.5%	13.32
Event()	321.05	5139293	26.9%	13.0%	0.06
Ibingo()	0.00	3115	0.0%	0.0%	0.00
Onset()	1.58	3115	0.1%	0.1%	0.51
Onsetq()	7.01	3115	0.6%	0.3%	2.25
P_one()	480.57	12931076	40.2%	19.5%	0.04
P_two()	15.70	251510	1.3%	0.6%	0.06
Period()	0.12	3115	0.0%	0.0%	0.04
Time_f()	0.41	3115	0.0%	0.0%	0.13
Wbuff()	25.36	795413	2.1%	1.0%	0.03
Xth()	13.49	541544	1.1%	0.5%	0.03
System Overhead Section					
Unix overhead	207.30	n/a	100.0%	8.4%	n/a

Table 7.5: Example execution profile of the event detector that processes 20 samples/second data on the Sun 3/160. The n/a is used in columns which are not applicable to the statistic. The filter used is the same as in the CDSN BB test.

Steim VSP Test					
Routine	Time (secs)	Calls	Section%	All%	ms/call
Data Support Section					
Cont_setup()	0.21	1	0.0%	0.0%	209.99
CtlWordType()	93.93	6906720	0.5%	0.4%	0.01
E_filter()	3.46	28778	0.0%	0.0%	0.12
E_steimload()	323.69	28800	1.6%	1.3%	11.24
Frecgen()	2639.76	28778	12.7%	10.8%	91.73
FP overhead	17682.66	n/a	85.2%	72.2%	n/a
main()	1.18	1	0.0%	0.0%	1179.95
Event Detector Section					
Ck_t_kont()	0.00	583	0.0%	0.0%	0.00
Count_dn()	17.43	1213081	1.1%	0.1%	0.01
E_detect()	336.69	28778	21.7%	1.4%	11.70
Event()	327.34	6095004	21.1%	1.3%	0.05
Ibingo()	0.00	583	0.0%	0.0%	0.00
Onset()	0.36	583	0.0%	0.0%	0.62
Onsetq()	13.55	583	0.9%	0.1%	23.24
P_one()	793.82	26773880	51.2%	3.2%	0.03
P_two()	33.70	303032	2.2%	0.1%	0.11
Period()	0.02	583	0.0%	0.0%	0.03
Time_f()	0.07	583	0.0%	0.0%	0.12
Wbuff()	4.21	130123	0.3%	0.0%	0.03
Xth()	22.95	1183716	1.5%	0.1%	0.02
System Overhead Section					
Unix overhead	2209.35	n/a	100.0%	9.0%	n/a

Table 7.6: Example execution profile of the event detector that processes 100 samples/second data on the Sun 3/160. The n/a is used in columns which are not applicable to the statistic. A 2-pole low-pass recursive filter was used. Little effort was made to reduce the execution time for this filter.

Chapter 8

Remarks

8.1 Caveats

Our experience with the detector has been primarily with the BB and SP CDSN and the SP SRO data. Sample rates for these are 20 and 40 samples per second and the record sizes are about 1000 16-bit gain-ranged words. In addition, we have a limited amount of experience with 1 sps data (the LP of the CDSN system), and 100 sps data (the VSP of the IRIS-GSN system). Hence, whereas provision is made for processing other data, care should be exercised when doing so. In particular, *E_B*, which defines the size of the large buffers in `Event()`, should be great enough so that values in the last part of the window (*i_win*) do not wrap around to write over values in the first part of the window. As a precaution, one might choose *E_B* as large as, or larger than, the largest of all of the values in the set

$$e\_b_i = 2 \times (window\_length_i\ (\ sec) \times Nyquist\_frequency_i\ (Hz)) + 4$$

Furthermore, we reemphasize, to circumvent floating-point operations, we assume that the integer (except for VLP) sample rate will divide into one thousand with no remainder (this limitation is easily addressed by modifying the code that depends primarily on the variable *ms_sam*). In addition, one must remember that the clocks of the detector are based on 16-bit integers: One might have severe problems processing individual records that are 32768 data bytes long, for instance. Finally, although we do not forsee major problems in processing other seismic data, if one operates outside of our experience, thorough tests should be conducted to demonstrate that the detector performs as expected.

8.2 Acknowledgements

The coefficients of the FIR filters that are implemented were calculated by C. R. Hutt who is preparing a report on the design of the filters. The routine for finding the peaks and troughs of the input time series was adapted from a 1977 program by L. G. Holcomb. The routine `Ith()` was adapted from one in 1984 by R. R. Reynolds and C. R. Hutt. John J. McDermott, Jr. cooperated in the early part of the project to convert from FORTRAN to C language. Stuart Flicker contributed suggestions on portability of the code. Also, thanks to Dave Barnett from Lawrence Livermore Laboratories who contributed code for enhanced local event processing.

During the implementation of the algorithm in PASCAL for the IRIS-1 and IRIS-2 GSN Data Aquisition Systems, Dr. Joseph Steim repaired some bugs that kept the detector from operating with sample rates less than 1 sample per second. These changes were then incorporated into the C detector. In 1989, Sean Keane performed extensive off-line simulations to test for proper operation of the detector for odd sample rates and through end-of-year boundaries and with leap years.

8.3 References

Kernighan, B. W. and D. M. Ritchie (1978). *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 228 pp.

Murdock, J. N. and C. R. Hutt (1983). "A new event detector designed for the Seismic Research Observatories", USGS Open File Report 83-785, 42pp.

Appendix A

Event Detector C Code

Here is the actual C code for the event detector itself. It is extensively documented and should be easily ported from one machine type to another.

A.1 CSSTND.H – Main project definitions

```
/*--------------------------------------------------------------------*
 *   G L O B A L   D I G I T A L   S E I S M I C   N E T W O R K      *
 *                                                                    *
 *    gdsnstd.h - Standards and constants definition for project      *
 *--------------------------------------------------------------------*/

/*--------------------------------------------------------------------*
 * Albuquerque Seismological Laboratory - USGS - US Dept of Interior  *
 *--------------------------------------------------------------------*/

/*--------------------------------------------------------------------*
 *    Modification and history                                        *
 *                                                                    *
 *---Edit---Date----Who-------Description of changes-----------------*
 *    001 24-Apr-86 SH        Set up standards file                   *
 *    002  9-Jul-86 SH        Add mfree to free for UNIX29            *
 *    003 22-Dec-86 SH        Tailor VMS definitions                  *
 *--------------------------------------------------------------------*/

/*--------------------Determine System Dependancies-----------------*/

#ifdef VMS
#define QUAD_UWORDSIZE          /* Ints and pointers are 32 bit */
#define UNSGND unsigned         /* Non-Ints can be specified as unsigned */
#define VOIDER void             /* The compiler really has a void type */
#define STDFLT double           /* Standard floating */
#endif
#ifdef UNIX29
```

```
#define DUAL_UWORDSIZE          /* Ints and pointers are 16 bit */
#define UNSGND                  /* You cannot have unsigned non-ints */
#define VOIDER int              /* The compiler does not have real void */
#define STDFLT float            /* Standard single precision floating */
#define mfree(a) free(a)        /* no mfree in unix 2.9 */
#endif

/*-----------------------Define Data types--------------------------*/

#define VOID VOIDER             /* For functions which return nothing */
#define BOOL UNSGND char        /* Flag quantities */
#define TEXT UNSGND char        /* For character strings */

        /*-------------------------------------------*
         *                   8-Bit   16-Bit  32-Bit  *
         *    Numbers:       BYTE    WORD    LONG    *
         *    Unsigned:      UBYTE   UWORD   ULONG   *
         *-------------------------------------------*/

/*      8-Bit quantities        */

#define UBYTE UNSGND char       /* An 8-Bit definition */
#define BYTE char               /* Numeric 8-bit definition */

/*      16-Bit quantities       */

#ifdef QUAD_UWORDSIZE
#define UWORD UNSGND short      /* 16 bit unsigned */
#define WORD short              /* 16 bit numeric quantity */
#endif
#ifdef DUAL_UWORDSIZE
#define UWORD unsigned int      /* Assume an int is a 16 bit number */
#define WORD int                /* 16 bit numeric quantity */
#endif

/*      32-Bit quantities       */

#ifdef QUAD_UWORDSIZE
#define ULONG unsigned int      /* Definition of a 32 bit mask */
#define LONG int                /* A 32 bit number */
#endif
#ifdef DUAL_UWORDSIZE
#define ULONG UNSGND long       /* 32 bit mask */
#define LONG long               /* 32 bit number */
#endif
```

```
#define DSKREC ULONG            /* Pointer to a disk record */

/*------------Define some macros and contants we will use-----------------*/

#define FOREVER for(;;)         /* Infinite loop        */

#include <stdio.h>              /* Get from standard include library */

#ifndef NULL
#define NULL    (0)             /* Impossible pointer   */
#endif
#define TRUE    1               /* if (TRUE)            */
#define FALSE   0               /* if (!TRUE)           */
#define EOS     '\0'            /* End of string        */

#define min(a,b) ((a)<(b)?(a):(b))
#define max(a,b) ((a)>(b)?(a):(b))

#ifdef UNIX29
#include <macros.h>              /* Get from UNIX include library */
#endif

#define IUBYTE(x) (((WORD) x)&0xFF)
```

A.2 CSCONT.H – Context and interface definitions

```
/*-------------------------------------------------------------------*
 * C h i n a  D i g i t a l  S e i s m o g r a p h  N e t w o r k  *
 *                                                                   *
 *      CSCONT.H - Main structure definitions                        *
 *                                                                   *
 *      Contains information used to analyze seismic events.  There  *
 *      is one of these buffers for each discrete component of seismic *
 *      data that is being event detected.                           *
 *-------------------------------------------------------------------*/

#undef min

/*---------------------Continuity Structure----------------------*/

struct con_sto {
/*------------Variables used in e_detect-----------------------------*/
```

```
        TEXT *ch_name;          /* Name of detector stored in e_detect  */
        BYTE cur_rec;           /* Index of the current record */
        WORD sam_tab[CUR_MAX];/* Number of samples per record per channel
                                    in current and prev records */

/*--------Variable Used in event subroutine-------------------------*/

        WORD buf_flg[E_B]; /* flags: 1 if >= th2; 2 if >= th1 */
        WORD buf_sc[E_B];  /* summed delta sample counts */
        LONG buf_amp[E_B]; /* P-T amplitudes values */
        WORD buf_tim[E_B]; /* time coordinate of P-T values */
        BYTE buf_rec[E_B]; /*  record number array */

        WORD abuf_sc[4];   /* last 4 delta sample counts */
        LONG abuf_amp[4];  /* last 4 P-T amplitudes */
        WORD abuf_tim[4];  /* time cord. of last 4 P-T values */
        BYTE abuf_rec[4];  /* companion to buf_rec */

        LONG last_amp;     /* last P-T value       */

        BOOL epf;          /* event possible flag */
        BOOL evon;         /* event detected flag */
        BOOL icheck;       /* flag, ensures period estimate */

        WORD filhi;        /* see OF Report 83-785 */
        WORD fillo;        /* see OF Report 83-785 */
        WORD iwin;         /* window length (samples) */
        WORD n_hits;       /* # P-T >= th2 for detection  */

        WORD fst_flg;      /* index to first flagged P-T value  */
        WORD indx;         /* saves fst_flg */
        WORD lst_flg;      /* index to last flagged P-T value */
        WORD lst_flg2;     /* lst_flg corrected for overflow */
        WORD lst_pt;       /* index to last P-T value processed */
        WORD lst_pt2;      /* lst_pt corrected for overflow */
        WORD index2;       /* counter,checked for overflow */

        BOOL iset;         /* flag, = 1 when P-T >= th1 */

        WORD jj;           /* index for abrev buffers abuf */
        WORD sumdsc;       /* sum delta sample count */
        WORD sumflg;       /* number of P-T values > th2 */
        WORD last_sc;      /* loaded to abuf_sc[] */
        WORD last_tim;     /* loaded to abuf_tim[] */
```

```
        BYTE last_rec;      /*  record of last_tim */

        LONG th1;           /* largest detection threshold */
        LONG th2;           /* smallest detection threshold */

/*----------------Variables used in p_one----------------------------*/

        LONG last_y;        /* amplitude coordinate of previous sample */
        WORD last_x;        /* time coordinate of previous sample */
        BYTE rec_last_x;    /*  Record of last time coordinate */
        WORD sum_s_c;       /* samples from last P or T to current sample */
        WORD s_sum_sc;      /* samples between last two P-T values */
        WORD index;         /* counter, calls ptwo */
        LONG max_y;         /* amplitude coordinate of P or T value */
        WORD tim_of_max;    /* time coordinate of P or T value */
        BYTE rec_of_max;    /*  record of tim_of_max */
        BOOL prev_slope;    /* sign of last difference  */
        LONG maxamp;        /* abs max of 20 consec. P-T values < thx */
        LONG thx;           /* upper bound for noise est. */
        LONG s_amp;         /* signed amplitude of P-T value */

/*----------------Variables used in p_two----------------------------*/

        LONG tsstak[16];    /* contains set of maxamp values */
        LONG twosd;         /* statistical dispersion of P-T values */
        WORD kk;            /* index for tsstak[] */
        WORD val_avg;       /* the number of values in tsstak[] */

/*----------------Variables used in onsetq---------------------------*/

        LONG th3;           /* threshold for estimating onset */
        LONG def_tc;        /* time correction for onset (default) */
        BOOL ponset;        /* Print onsets? */
        VOID (*onsproc)(); /* Call subroutine upon onset */

/*----------------Variables used in count_dn-------------------------*/

        UWORD wait_blk;     /* controls re-activation of detector and *
                             *  recording time in event coda        */
        WORD itc;           /* counter for interval of the event */
        WORD nn;            /* counter for itc */

/*----------------Variables used in xth------------------------------*/

        UWORD xth1;         /* coded threshold factor -- xth*/
```

```
        UWORD xth2;         /*          "                   */
        UWORD xth3;         /*          "                   */
        UWORD xthx;         /*          "                   */
                            /* th1>th2>thx>th3 or th1>thx>th2>th3   */

/*-----------------Variables to set up the detector----------------------*/

        WORD kont_per;      /* counts per period */
        LONG ms_sam;        /* milliseconds per sample */
        WORD haf_per;       /* samples per one-half period */
        WORD sam_sec;       /* sample rate of the digital data */
        LONG s_r_x_1000;    /* Sample rate * 1000 */

};

/*-------------------Time Structure--------------------------------*/

struct  r_time  {
        LONG day_yr;
        LONG prv_yr;
        LONG yr;
        LONG day;
        LONG hr;
        LONG min;
        LONG sec;
        LONG msec;
};

/*--------------------Call interface-----------------------------------*/

struct detect_info {
        LONG    samrte;     /* Sample rate * 1000 this detector */
        WORD    drectyp;    /* Record type designator */
        WORD    datapts;    /* Number of data points last decode */
        struct r_time startt;/* Starting time of first data point */
        LONG    *indatar;   /* ptr to data array */
        WORD    lbksize;    /* Space reserved for lookback */
        LONG    *lbkarr;    /* ptr to lookback storage array */
        struct con_sto *incontd; /* This detector's continuity structure */
        TEXT    *detname;   /* Name of this event detector */
        BOOL    (*filterc)();   /* Pointer to the filter function (or NULL) */
#ifdef FFILTER
        double  ivstack[LBSIZE][MAXSECTION]; /* Input value stack */
        double  fbstack[LBSIZE][MAXSECTION]; /* Feedback stack */
        double  coeff[2][LBSIZE][MAXSECTION]; /* Input Coefficients */
```

```
                        /* [NUMER/DENOM][LOOKBACK][SECTION] */
        int     numsec;     /* Number of sections in use */
        int     flstaki;    /* Stack index variable (ring buffer) */
#endif
};

/*-----------------Auxillarly onset information----------------------*/

struct s_onset {
        BOOL    new_onset;  /* Flag TRUE if new onset here */
        TEXT    o_polar;    /* A 'd' or 'c' for polarity of first break */
        TEXT    o_dalgo;    /* Detection algorithm 'A' or 'B' */
        UBYTE   o_snr[5];   /* SNR onset quality estimate values */
        UWORD   o_year;     /* Year of onset */
        UWORD   o_days;
        UWORD   o_hours;
        UWORD   o_mins;
        UWORD   o_secs;
        UWORD   o_msecs;
        ULONG   o_amps;     /* Amplitude of signal */
        UWORD   o_pl;       /* Integer part of period */
        ULONG   o_pr;       /* Next two decimals of period */
        ULONG   o_large;    /* twosd, background noise estimate */
};

/*-----------Macros for convenience--------------------------*/

#define filhi_set(detector,value) (detector)->incontd->filhi = (value)
#define fillo_set(detector,value) (detector)->incontd->fillo = (value)
#define iwin_set(detector,value) (detector)->incontd->iwin = (value)
#define nhits_set(detector,value) (detector)->incontd->n_hits = (value)

#define xth1_set(detector,value) (detector)->incontd->xth1 = (value)
#define xth2_set(detector,value) (detector)->incontd->xth2 = (value)
#define xth3_set(detector,value) (detector)->incontd->xth3 = (value)
#define xthx_set(detector,value) (detector)->incontd->xthx = (value)

#define deftc_set(detector,value) (detector)->incontd->def_tc = (value)
#define wait_set(detector,value) (detector)->incontd->wait_blk = (value)
#define avg_set(detector,value) (detector)->incontd->val_avg = (value)

#define setallparams(cs,fhi,flo,iw,nht,x1,x2,x3,xx,tc,wa,av) \
 { filhi_set(cs,fhi); fillo_set(cs,flo); iwin_set(cs,iw); nhits_set(cs,nht);
\
   xth1_set(cs,x1); xth2_set(cs,x2); xth3_set(cs,x3); xthx_set(cs,xx); \
```

```
   deftc_set(cs,tc); wait_set(cs,wa); avg_set(cs,av); }
```

A.3 DETECT.H – Detector variables definitions

```
/*-------------Primary declarations for event detector package---------*/


#include <csstnd.h>
#ifdef UNIX
#include <signal.h>
#endif
#include <csconfig.h>
#include <cscont.h>

#define absval(inval) (((inval)>=0)? (inval):-(inval))

#ifdef MAINDEF /* See e_detect() */
   /* Initializations added to comply with objections of Whitesmith linker */

struct r_time etime = {0};      /* Time of beginning of record */
struct r_time htime = {0};      /* Backup of beginning time */

WORD sam_no = 0;        /*the number of the current seismic sample*/
WORD th_wt = 0; /*weight, = 1 if >= th2, = 2 if >= th1 -- event*/
LONG fil_out = 0L;      /*output of the seismic data filter -- filter*/
WORD lastmin = 0;       /* fix the TIMESTAMP function */

WORD p_rval = 0;      /* fraction of period expressed in millsec--period*/
LONG p_lval = 0;      /* integer value of period--period*/
LONG t_rval = 0;      /* fraction of onset sec  expressed as decimal--time_f*/

struct con_sto *con_ptr;        /* Global continuity structure pointer */
#else

extern  struct r_time etime,htime;
extern  WORD sam_no,th_wt,lastmin,p_rval,p_lval,t_rval;
extern  LONG fil_out;
extern  struct con_sto *con_ptr;

#endif

#define B_M1    (E_B - 1)       /*buffer index for routine Event*/
#define B_M2    (E_B - 2)
```

```
#define B_M3    (E_B - 3)
#define B_M4    (E_B - 4)       /* "    "       "       "       */

/*------------Be friendly to lint-----------------------------*/

WORD    Ck_t_kont();
VOID    Count_dn();
BOOL    E_detect();
BOOL    Event();
VOID    Ibingo();
VOID    Period();
LONG    Time_f();
VOID    Onset();
VOID    Onsetq();
BOOL    P_one();
VOID    P_two();
VOID    Period();
VOID    Wbuff();
LONG    Xth();

UBYTE   *malloc();

#define mfree(x) free(x)
```

A.4 CSCONFIG.H – Detector configuration definitions

```
/*---------Detector configurable parameters and flags-------------------*/
/* Within limits, the user may adjust these to tune the detector        */

#define EV_OFF 2  /* following an event, disable the detector
                     for NOR_OUT - EV_OFF wait_blk's */
                  /* NB EV_OFF must <= than NOR_OUT. See count_dn(). */
#define NOR_OUT 4  /* the number of wait_blk's that controls the *
                      interval of the event. See count_dn(). (WORD) */
#define CUR_MAX 20 /* Define modulus for the record sequences (max, 127) */
#define E_B     700/* size of buffers in event(). (WORD)   */

#undef RAMPUP      /* Automatic quick backround estimate */

#define PONSET

#undef FFILTER     /* Generic floating point filter routines
                             Don't define if there's no FP */
```

```
#define MAXSECTION 4 /* Number of stages deep */
#define LBSIZE 3     /* Current lookback size - probably always 3 */

#undef RESYNCRONIZE  /* Cause detector to reset pt and loop index values
                        after events so that the outputs of online and
                        offline detectors will be identical -- otherwise
                        small timing differences will cause the answers to
                        vary slightly */

#define ITC_PER_CNTRL/* Duration of recording made a function of the
                        period of the signal for high frequency events */

#define VSP_SPS 80   /* Affect all sample rates this or higher */
#define PER_TRIG 100 /* Set new ITC on periods this or longer */
#define ITC_UP 8     /* Set the ITC to this */
```

A.5 E_DETECT.C – Detector main loop and dispatch

```
#define MAINDEF              /* Make "detect" allocate storage for all global
                                variables in this module - see detect.h */
#include <detect.h>

/*-------------------------------------------------------------------*
 *    Function: BOOL E_detect(detector) - Event detect               *
 *                    a data record                                  *
 *-------------------------------------------------------------------*
 *    Arguments:      struct detect_info *detector - information for *
 *                                    running the detector           *
 *-------------------------------------------------------------------*
 *    Returns:        TRUE - Current record is within the interval   *
 *                            of an event.                           *
 *                    FALSE - Current record is not within the       *
 *                             interval of an event.                 *
 *                                                                   *
 *                         ---The interval of the event---           *
 *                    Upon declaring an event, E_detect will remain  *
 *                     TRUE for a minium of NOR_OUT x wait_blk       *
 *                     samples, where NOR_OUT = 4 and                *
 *                     wait_blk = 1014 or 507.  It will remain true  *
 *                     longer than NOR_OUT x wait_blk if a retrigger *
 *                     occurs in the coda of the event.              *
 *-------------------------------------------------------------------*
 *    Fatal Errors:   This routine does not have fatal exits         *
 *-------------------------------------------------------------------*
```

```
*    E_detect is the main driver for the event detect algorithm.     *
*     The function of E_detect is to call routines that              *
*           (1)  Find the amplitudes and associated times            *
*                of the filtered data, P_one.                        *
*           (2)  Process these amplitudes and associated             *
*                times to search for seismic signals, Event.         *
*           (3)  Cause output of the parameters of the               *
*                detected signal, Onsetq.                            *
*                                                                    *
*    The main (calling program) passes to E_detect the address       *
*     of a structure.  Although this structure (detector) con-       *
*     tains several other variables and constants, the only          *
*     ones needed by E_detect are:                                   *
*           (1)  sample rate x 1000 of the digital data. (This       *
*                multiplication is done so that VLP data of          *
*                the China system can be detected, if one so         *
*                desires.)                                           *
*           (2)  the number of data points that will be processed.   *
*           (3)  the time (yr, day, hour, minute, second, milli-     *
*                second) of the first data point of the current      *
*                record (the data that will be processed).           *
*           (4)  the address of the first data point to be pro-      *
*                cessed.  (The address of the array of 32 bit        *
*                data that will be processed.)                       *
*           (5)  the address of the list ( a structure) of           *
*                variables and constants that must be maintained     *
*                for each detector.  The parameters of the detec-    *
*                tor (filhi, fillo, xth1,etc.) are included in       *
*                this list and must be loaded into the list by       *
*                the user.                                           *
*           (6)  the detector name (for instance, BB_Z1, BB_Z2,      *
*                SP_Z, SP_N, etc).                                   *
*                                                                    *
*    E_detect returns TRUE (the value 1) if any the data of the      *
*     current record are within the interval of the event--          *
*     "the interval of the event" is a programmable feature,         *
*     as explained above.  Otherwise E_detect returns False (the     *
*     value 0).                                                      *
*--------------------------------------------------------------------*/

BOOL E_detect(detector)
struct detect_info *detector;
{

        LONG *ldp,tdat;
```

```
        BOOL tapewrite;

        con_ptr = detector->incontd;   /* User's continuity structure */

        tapewrite = FALSE;

        ldp = detector->indatar;
                                                          /* Note 1 */
/*----------Associate the data with a record number-----------------*/

        con_ptr->cur_rec++;
        if (con_ptr->cur_rec >= CUR_MAX)
            con_ptr->cur_rec = 0;

/*------Provide for changing the sample rate during processing------*/

        if (detector->samrte != con_ptr->s_r_x_1000) {

            con_ptr->s_r_x_1000 = detector->samrte;
            con_ptr->sam_sec = detector->samrte / 1000L;
            /* con_ptr->ms_sam = 1000 / con_ptr->sam_sec; */
            con_ptr->ms_sam=1000000L/((LONG) con_ptr->s_r_x_1000);

                printf("s_r_x_1000 = %d\n",con_ptr->s_r_x_1000);
                printf("sam_sec = %d\n",con_ptr->sam_sec);
                printf("ms_sam = %d\n",con_ptr->ms_sam);
        }

        con_ptr->sam_tab[con_ptr->cur_rec] = detector->datapts;
                              /* # of data points to be processed */

        etime.day_yr = detector->startt.day_yr;
                                      /* load time of first data point */
        etime.prv_yr = detector->startt.prv_yr;
        etime.yr = detector->startt.yr;
        etime.day = detector->startt.day;
        etime.hr = detector->startt.hr;
        etime.min = detector->startt.min;
        etime.sec = detector->startt.sec;
        etime.msec = detector->startt.msec;

/*---etime will be changed when events are detected - backup----*/

        htime.day_yr = etime.day_yr;
        htime.prv_yr = etime.prv_yr;
```

```
        htime.yr = etime.yr;
        htime.day = etime.day;
        htime.hr = etime.hr;
        htime.min = etime.min;
        htime.sec = etime.sec;
        htime.msec = etime.msec;

/*-----------------Process the input data-------------------------*/

        for (sam_no = 0; sam_no < con_ptr->sam_tab[con_ptr->cur_rec];
                sam_no++) {

                                                        /* Note 2 */
/*---P-T detected, p_one=TRUE; event declared, Event=TRUE-----------*/
/*              Send input data to P_one                           */

            tdat = *ldp;
            if (P_one(tdat) && Event())
                Onsetq();

            ldp++;

            if (con_ptr->itc) { /* is data within interval of event? */
                Count_dn();    /* decrements itc */
                tapewrite = TRUE;
            }
        }

/*      printf("tsd==%d t1=%d t2=%d t3=%d tx=%d\n",con_ptr->twosd,
                con_ptr->th1,
                con_ptr->th2,
                con_ptr->th3,
                con_ptr->thx);*/
                                                        /* Note 3 */
        return(tapewrite);
}
#ifdef JNMCOMMENT
```

NOTES

1. In P_one, a peak or trough is not declared until one sample past the maximum or minimum. Thus, should the last sample of a record be a maximum or minimum, neither will be detected until the next record has been read. We need to keep track of in which record the data occur to cope with this situation.

2. The input time series is sent to routine P_one which calculates the signed amplitudes (peak-to-trough differences) and their corresponding times. P_one calls P_two which estimates the statistical dispersion of the amplitudes and sets the thresholds. When a peak or trough has been found by P_one, P_one returns TRUE and Event is called to determine whether or not the current amplitude might be part of a signal. When a signal is declared by Event, it returns TRUE, and Onsetq is called for further processing of the signal. Onsetq calls Onset which outputs information about the signal.

3. The counter itc is greater than zero if the current sample is within the interval of the event. Routine Count_dn decrements itc.

EXPLANATION OF THE VARIABLES

con_ptr – global pointer which allows all routines of the event detector to access the continuity structure. See cscont.h, detect.h.

CUR_MAX – the upper bound of the arbitrary record numbers. The record numbers are reset here.

cur_rec – an index that keeps track of the record in which the amplitudes of the input time series occur.

itc – a counter that is greater than zero if the current data sample is within the interval of the event. Initialized when an event is detected to NOR_OUT. Decremented in Count_dn.

incontd – the slot in the detect_info structure containing this detectors continuity structure. (It is a pointer to the continuity structure for this detector, see cscont.h)

ldp – a pointer that is initialized to the address of the first seismic data sample of the current record.

new_onset – a flag set=TRUE if a signal was declared in the current record. Set=FALSE otherwise.

sam_ch – samples per channel array for this and CUR_MAX-1 previous records

sam_no – sample number of seismic data of the current demultiplexed record.

s_r_x_1000 – adjusted sample rate to permit processing VLP data that is less than 1 sample/sec. (sam_sec × 1000).

samrte – adjusted sample rate to permit processing VLP data that is less than 1 sample/sec. (sam_sec × 1000).

sam_sec – sample rate of the current seismic data record (samples per second).

tapewrite – a flag set=TRUE if any data of the current record is within the interval of the event. Set=FALSE otherwise. E_detect returns the value of tapewrite. The concept here is that the calling routine might wish to write data to tape during, and prior to, the interval of the event.

```
#endif
```

A.6 EVENT.C – Event determination routine

```
#include <detect.h>

/*-------------------------------------------------------------*
 *  Function: BOOL Event() - Detect events                     *
 *-------------------------------------------------------------*
 *  Arguments:      No arguments                               *
 *-------------------------------------------------------------*
 *  Returns:        TRUE - if event declared                   *
 *                 FALSE - no event declared                   *
 *-------------------------------------------------------------*
 *  Fatal Errors:   This routine does not have fatal exits     *
 *-------------------------------------------------------------*
 *  Detects signals using thresholds th1 and th2. For n_hits = 4  *
 * an event may be detected if 4 values are greater than th2, or  *
 * if 3 values are greater than th2, and one (or more) of the 3   *
 * is greater than th1 (see 0-F report 83-785).                   *
 *-------------------------------------------------------------*/

BOOL Event() {

        WORD j, m;
        WORD tfst_flg;
        LONG ab_amp;

        ab_amp = absval(con_ptr->s_amp);

        if (con_ptr->jj > 3)
            con_ptr->jj = 0;
                                        /* Note 1 */
        con_ptr->abuf_rec[con_ptr->jj] = con_ptr->last_rec;
        con_ptr->abuf_sc[con_ptr->jj] = con_ptr->last_sc;
        con_ptr->abuf_amp[con_ptr->jj] = con_ptr->last_amp;
        con_ptr->abuf_tim[con_ptr->jj++] = con_ptr->last_tim;
                                       /* NB jj incremented   */
        con_ptr->last_sc = con_ptr->s_sum_sc;
```

```
        con_ptr->last_amp = con_ptr->s_amp;
        con_ptr->last_tim = con_ptr->tim_of_max;
        con_ptr->last_rec = con_ptr->rec_of_max;

                                        /* Note 2 */
        if (con_ptr->evon) {           /* evon set=TRUE in Ibingo, =FALSE in
                                          Count_dn */
            if (con_ptr->fst_flg > (con_ptr->lst_pt2 = con_ptr->lst_pt))
                con_ptr->lst_pt2 += E_B;

            if ((con_ptr->lst_pt2 - con_ptr->fst_flg) == 7) {
                con_ptr->sumdsc += con_ptr->s_sum_sc;
                Wbuff();
                con_ptr->icheck = TRUE;  /* Inhibit writing to buffers */
                return(TRUE);          /* Event Declared! */
            }

            if ((con_ptr->lst_pt2 - con_ptr->fst_flg) < 8) {
                con_ptr->sumdsc += con_ptr->s_sum_sc;
                Wbuff();
                return(FALSE);
            }

                                        /* Note 3 */
            if (con_ptr->icheck)
                return(FALSE);

            con_ptr->icheck = TRUE;
            return(TRUE);

        }

/*--------Was event possible set in previous pass(es)?-----------------*/
                                        /* Note 4 */

        if (!con_ptr->epf) {

            if (ab_amp < con_ptr->th2)
                return(FALSE);
            if (ab_amp < con_ptr->th1)
                th_wt = 1;
            else
                th_wt = 2;

            con_ptr->epf = TRUE;
```

```
            con_ptr->fst_flg = 4;
            con_ptr->lst_flg = 4;
            con_ptr->lst_pt = 3;

            m = con_ptr->jj;

            if (m > 3)
                m = 0;

/*-A P-T value is >= th2, hence a candidate signal just began---------*/
/*-- --------------Load large buffers from abbrev buffers----------------*/
/*-These two different buffer types are a holdover from FORTRAN -----*/

            con_ptr->sumdsc = -con_ptr->abuf_sc[m];

            for (j = 0; j < 4; j++) {
                if (m > 3)
                m = 0;

                con_ptr->sumdsc += con_ptr->abuf_sc[m];
                con_ptr->buf_rec[j] = con_ptr->abuf_rec[m];
                con_ptr->buf_sc[j] = con_ptr->sumdsc;
                con_ptr->buf_amp[j] = con_ptr->abuf_amp[m];
                con_ptr->buf_tim[j] = con_ptr->abuf_tim[m++];
                                        /* m incremented */
            }
            con_ptr->sumdsc += con_ptr->s_sum_sc;

            Wbuff();
            return(FALSE);
        }

        con_ptr->sumdsc += con_ptr->s_sum_sc;

/*--------Event possible flag set; however, is current----------------*/
/*---------------P-T value >= smallest threshold?--------------------*/

                                        /* Note 5 */
        if (ab_amp >= con_ptr->th2) {
            th_wt = 1;          /* Assign weight=1 to current P-T value*/

/*------------------------  >= largest threshold?-------------------*/

            if (ab_amp >= con_ptr->th1)
```

```
                th_wt = 2;                          /* Weight=2 */

/*----------Is interval between P-T values too small?-----------------*/
/*----------If so, assign weight=0 to current P-T value---------------*/

            if ((con_ptr->sumdsc - con_ptr->buf_sc[con_ptr->lst_flg])
                <= con_ptr->filhi) {
                th_wt = 0;
                Wbuff();
                return(FALSE);
            }

/*----------Is interval between P-T values too large?-----------------*/
/*----------If so, move beginning of window to current P-T value------*/

            if ((con_ptr->sumdsc - con_ptr->buf_sc[con_ptr->lst_flg])
                >= con_ptr->fillo) {
                Wbuff();
                con_ptr->fst_flg = con_ptr->lst_pt;
                con_ptr->lst_flg = con_ptr->lst_pt;
                return(FALSE);
            }
                                                /* Note 6 */
/*----------Is this the second P-T value of the candidate signal, ------*/
/*--------------with no intervening ones? (a special case) --------------*/
/*----------If so, write buffer and set lst_flg = lst_pt---------------*/

            if (con_ptr->fst_flg == con_ptr->lst_flg) {
                Wbuff();
                con_ptr->lst_flg = con_ptr->lst_pt;
                return(FALSE);
            }

                                                /* Note 7 */
/*------------Current P-T value >= smallest threshold,----------------*/
/*-nevertheless slide window, if necessary, to satisfy time criterion---*/
/*----------------------Set lst_flg = lst_pt -----------------------*/

            while ((con_ptr->sumdsc - con_ptr->buf_sc[con_ptr->fst_flg])
                > con_ptr->iwin) {
                con_ptr->indx = con_ptr->fst_flg;
                con_ptr->lst_flg2 = con_ptr->lst_flg;
                if ((con_ptr->indx + 1) > con_ptr->lst_flg)
                con_ptr->lst_flg2 = con_ptr->lst_flg + E_B;
                for (j = (con_ptr->indx + 1); j <= con_ptr->lst_flg2; j++) {
```

```
                    /* Is fst_flg == largest index of buffer ? */
                if (con_ptr->fst_flg == B_M1)
                    con_ptr->fst_flg = -1;
                    /* changed from ==, 8 jul 86, jnm */
                if (con_ptr->buf_flg[++con_ptr->fst_flg] != 0)
                    break;              /* fst_flg incremented */
                }
            }
            Wbuff();

            con_ptr->lst_flg = con_ptr->lst_pt;
            con_ptr->iset = FALSE;
            con_ptr->sumflg = 0;
            con_ptr->lst_flg2 = con_ptr->lst_flg;

            if (con_ptr->lst_flg < con_ptr->fst_flg)
                con_ptr->lst_flg2 = con_ptr->lst_flg + E_B;

/*---Check for event by evaluating values in flag buffer--------------*/

            for (j = con_ptr->fst_flg; j <= con_ptr->lst_flg2; j++) {
                con_ptr->index2 = j;
                if (con_ptr->index2 > B_M1)
                    con_ptr->index2 -= E_B;
                if (con_ptr->buf_flg[con_ptr->index2] != 0)
                    con_ptr->sumflg++;
                if (con_ptr->buf_flg[con_ptr->index2] == 2)
                    con_ptr->iset = TRUE;       /* P-T >= th1 */
            }
            if (con_ptr->sumflg < 3)
                return(FALSE);
            if (con_ptr->iset) {
                Ibingo();                       /* Event detected */
                return(FALSE);                  /* But not yet declared */
            }
            if (con_ptr->sumflg >= con_ptr->n_hits)
                Ibingo();
            return(FALSE);              /* Event detected but not declared */
        }

/*--Event possible flag is set, however current P-T value is less-------*/
/*--than smallest threshold.  Do not set lst_flg = lst_pt---------------*/

        th_wt = 0;
        Wbuff();
```

```
                                                /* Note 8 */
/*--------------Slide window if necessary so that--------------------*/
/*----first value in window is a flagged value, if possible.----------*/
/*-----------Otherwise, reset event possible flag.-------------------*/

        while ((con_ptr->buf_sc[con_ptr->lst_pt] -
                con_ptr->buf_sc[con_ptr->fst_flg]) > con_ptr->iwin) {
            con_ptr->lst_flg2 = con_ptr->lst_flg;
            if (con_ptr->fst_flg > con_ptr->lst_flg)
                con_ptr->lst_flg2 = con_ptr->lst_flg + E_B;
            if (con_ptr->fst_flg >= con_ptr->lst_flg2) {

/*------fst_flg has been rotated to, or (fail safe) past lst_flg, ------*/
/*--------------and lst_flg is outside of the window.------------------*/
/*-------Therefore, there are no flagged P-T values in the window---*/
/*--------------ie, a candidate signal is not in progress ----------*/

                con_ptr->epf = FALSE;
                return(FALSE);
            }

/*--------------Rotate fst_flg thru the circular buffer---------------*/
            for (j = (con_ptr->fst_flg + 1); j <= con_ptr->lst_flg2; j++) {
                tfst_flg = j;
                if (tfst_flg > B_M1)
                tfst_flg -= E_B;
                if (con_ptr->buf_flg[tfst_flg] > 0)
                break;
            }
            con_ptr->fst_flg = tfst_flg;
        }

/*-------epf remains TRUE, ie, a candidate signal may be in progress--*/
/*-------However, the candidate signal has not yet been verified--*/

        return(FALSE);
}

#ifdef JNMCOMMENT
```

NOTES

1. The abbreviated buffers (abuf) save the last 4 P-T values and their associated time fields. Four P-T values are needed because, if the current P-T value is the first P-T

value of a signal that is $\geq$ th2, the detector looks back 2 P-T values to test for the onset. If the earliest of these two P-T values is deemed the onset, then the detector needs the two P-T values before this onset to compute the signal-to-noise series, hence the 4 values.

2. The event on (evon) flag is set=TRUE when routine Ibingo is called. After evon=TRUE, we want to ensure that enough P-T values have been processed to permit an estimate of the period of the signal. When enough values have been processed, Event returns TRUE. It is important to note that Event returns TRUE only once in the interval of the event, unless a new detection in the coda is made. Evon set to False in routine Count_dn when the counter itc $\leq$ EV_OFF (ie, evon set to false when (NOR_OUT - EV_OFF) $\times$ wait_blk samples have been processed). In the China System, EV_OFF = 2, NOR_OUT = 4, and wait_blk = 1014 for the BB and 507 for the SP. Because we envision that data will be written to tape during the interval of the event, wait_blk typically is chosen as a function of the input record length.

3. Again, we want to return event=TRUE only once per detection. The flag icheck was set to FALSE in routine Ibingo. Therefore, if there are 8 or more P-T values currently in the window, icheck will be set to TRUE and event returns TRUE. On subsequent calls of routine Event, while evon = TRUE, Event returns FALSE, see [2] above for how evon is controlled.

4. This is where processing of data for detection of an event begins. The event possible flag (epf) is set = TRUE when a P-T value in the window is $\geq$ th2. During the time epf = TRUE, data of the large buffers are processed and data are written into them by Wbuff. The concept of large and abbreviated buffers is a holdover from the FORTRAN version.

5. How the detector works has been described by Murdock and Hutt in the USGS Open File Report 83-785. In brief, an event will be detected if, in the window, 3 P-T values are $\geq$ th2 and one (or more) of them is $\geq$ th1; alternatively, an event will be detected if n_hits P-T values are $\geq$ th2, where n_hits is $>$ 3. However, these n_hits P-T values must not occur too close together (no closer than filhi +1 samples), or too far apart (no farther apart than fillo$-$1 samples).

6. This tests a special case and was implemented to enhance speed. The index fst_flg was initialized to lst_flg, and lst_pt was initialized to fst_flg-1. lst_pt is incremented in wbuff. Therefore, this if statement will return Event = FALSE for the second consecutive P-T value of a candidate signal, assuming the large buffers were loaded from the abbreviated buffers when the first of these two P-T values was processed. Incidently, the concept of abbreviated buffers and large buffers is a holdover from the FORTRAN version of the code.

7. If the number of samples from the beginning of the window to the current P-T value is greater than the window length (samples), move the beginning of the window to the next flagged (th_wt $>$ 0) P-T value. Continue with this process until the difference between the first P-T value in the window and the last P-T value in the window (both of which are flagged) is $\leq$ the window length.

8. The current P-T value is less than th2, nevertheless, we may have a signal in progress but not yet detected. This is certainly true if the distance between the fst_flg and the lst_flg is less than iwin. Alternatively, if this distance is greater than iwin, we move the end of the window to the (index of the) current P-T value and then check for flagged P-T values in the window. If there are no flagged values (ie lst_flg is outside of the window) epf is set=FALSE. On the other hand, if there is a flagged P-T value in the window (be it lst_flg or an earlier one), the beginning of the window is moved to this point (ie fst_flg is set to the index of the first flagged P-T value in the window).

DESCRIPTION OF VARIABLES

ab_amp – absolute amplitude of the current P-T. Counts.

abuf_amp – an abbreviated buffer that contains the signed P-T amplitudes.

abuf_rec – an abbreviated buffer that contains (arbitrary) record numbers in which each P-T value occurred.

abuf_tim – an abbreviated buffer that contains time information of the P-T values. This information is represented by the number of sample counts from a reference time. Check for numerical overflow is not made.

abuf_sc – an abbreviated buffer that contains the number of sample counts between the P-T values. Here it is important to note that check for numerical overflow is not made.

B_M1 – E_B - 1. The largest index of the large buffers.

buf_amp – a large buffer that contains signed P-T amplitudes.

buf_flg – a large buffer that contains the values of th_wt for each P-T value of the time window. The parameter th_wt is a weighting value.

buf_rec – a large buffer that contains record numbers in which the P-T values of the window occurred.

buf_sc – a large buffer that contains the sum of the number of samples. This clock is initialized when the abbreviated buffers are downloaded into the large buffers. It is important to note that values of this buffer are not checked for numerical overflow. For very high sample rates (ie thousands of samples per second), check for impending overflow, with appropriate subsequent action, should be implemented.

buf_tim – a large buffer that contains time information of the P-T values of the window. This information is represented by the number of sample counts from a reference time. The reference time is obtained from the header of a record. Check for numerical overflow is not made. In subsequent versions of the detector, one might be able to combine buf_tim and buf_sc.

E_B – size of the large buffers. The large buffers should be large enough to inhibit wrap-around in the window. For example, if there is a maximum of 10 P-T values per second (Nyquist frequency = 10), and the window is 4 sec long, E_B should be ≥ 44 (4 P-T values that are loaded from the abbreviated buffers plus the maximum of 40 of the window).

epf – event possible flag. It is set=TRUE when a P-T value in the window is $\geq$ th2. Set=FALSE when an event is detected or when no values in the window are $\geq$ th2.

evon – event on flag. It is set=TRUE in Ibingo. Set=FALSE in Count_dn in the interval of the coda of the event.

fst_flg – an index to the first flagged P-T value of the window. It is an index to the first P-T value $\geq$ th2.

index2 – an index that is within the range of the large buffers: If index2 exceeds the size of the buffer, index2 is set=0.

indx – saves the value of fst_flg when fst_flg may be changed.

iset – a flag that is set=TRUE when a P-T value of the window $\geq$ th1.

icheck – a flag that is set=FALSE in Ibingo, set=TRUE in Event when enough P-T values have been processed (8) to estimate the period of the signal.

iwin – the length of the window (samples).

jj – an index for the abbreviated buffers.

last_amp – signed amplitude of the previous (as opposed to current) P-T.

last_rec – arbitrary record number in which the previous P-T occurred.

last_sc – the number of samples (time axis) between the two previous P-T values.

last_tim – the sample number of the last P-T value (the sample number is initialized at the beginning of each record). This sample number is used to calculate the time of the P-T value by using the information in the record header.

lst_flg – an index to the last P-T value that is $\geq$ th2.

lst_flg2 – circular buffers are employed, lst_flg2 is a bound that is corrected for buffer length.

lst_pt – an index to the last P-T value that was processed, it is incremented in routine Wbuff.

lst_pt2 – circular buffers are employed, lst_pt2 is a bound that is corrected for buffer length.

m – an index for the abbreviated buffers.

n_hits – the number of P-T values in the window that is required for a detection if they are all $\geq$ th2 but $<$ th1.

rec_of_max – arbitrary record number in which the current P-T occurred.

s_amp – signed amplitude of the current P-T. Counts.

s_sum_sc – the number of samples (time axis) between the current P-T value and the previous one.

sumdsc – the sum of the s_sum_sc that is initialized when the abbreviated buffers are downloaded. Not checked for numerical overflow.

sumflg – the number of flagged P-T values (ie, those with an associated th_wt > 0) in the window.

tfst_flg – temporary fst_flg.

th1 – the largest threshold for detection of an event. Typically th1 = 2.0 $\times$ twosd, where twosd is an approximation of twice the sample standard deviation of the P-T values. Counts.

th2 – the smallest threshold for detection of an event. Typically th2 = 1.5 $\times$ twosd, where twosd is an approximation of twice the sample standard deviation of the P-T values. Counts.

th_wt – threshold weight. Equals 2 if current P-T value is $\geq$ th1. Equals 1 if current P-T value is $\geq$ th2, but $<$ th1. Otherwise it is zero.

tim_of_max – the sample number of the current P-T value (the sample number is initialized at the beginning of each record). This sample number is used to calculate the time of the P-T value by using the information in the record header.

```
#endif
```

A.7 P_ONE.C – P-T value generator

```
#include <detect.h>
#define NEG 1
#define POS 0

/*-------------------------------------------------------------------*
 *    Function: BOOL P_one(in_data) - Form the P-T series            *
 *              The algorithm for finding the peaks and troughs was  *
 *                adapted from one by L.G. Holcomb.                  *
 *-------------------------------------------------------------------*
 *    Arguments:      LONG in_data, the seismic data sample          *
```

```
*-------------------------------------------------------------------*
*    Returns:         TRUE - Peak or trough detected                *
*                     FALSE - No Peak or trough detected            *
*-------------------------------------------------------------------*
*    Fatal Errors:    This routine does not have fatal exits        *
*-------------------------------------------------------------------*
*    This routine determines the signed amplitudes (and associated  *
*     times) of the filtered data. (By amplitudes, we mean the      *
*     difference in value of the consecutive local maximums and     *
*     minimums, hereafter referred to as the peak-to-trough ampli-  *
*     tudes, or P-T value.)  Each peak or trough is determined by   *
*     comparing slopes between  the input samples.  When a peak     *
*     or trough is found, the record and time (ie, sample number)   *
*     where it occurred is documented.  In P_one, "y" denotes an    *
*     amplitude coordinate and "x" denotes a time coordinate.       *
*                                                                   *
*    In addition to calculating the peak-to-trough amplitudes,      *
*     P_one also compares each of these amplitudes to a threshold   *
*     thx.  The maximum of 20 successive values less than thx is    *
*     fed to P_two.  P_two uses these maximums to calculate the     *
*     statistical dispersion of the background noise.  The thresh-  *
*     hold is used to inhibit anomalously large values (such as     *
*     spikes) from contributing to the estimate of the normal       *
*     background.                                                   *
*-------------------------------------------------------------------*/

BOOL P_one(in_data)
register    LONG in_data;
{
        register    BOOL cur_slope;
        LONG del_amp, ab_amp;

        del_amp = in_data - con_ptr->last_y;

        con_ptr->sum_s_c++;

        if (del_amp == 0) cur_slope = con_ptr->prev_slope;
                /* Ensure that no zero values pass through */
        else
                if (del_amp < 0)
                    cur_slope = NEG;
                else
                    cur_slope = POS;

/*--------Was the last sample a peak or trough?-----------------*/
```

```
                                                    /* Note 1 */
        if (con_ptr->prev_slope != cur_slope) {
                        /* a maximum or minimum has been found */
            con_ptr->s_amp = con_ptr->max_y - con_ptr->last_y;
            con_ptr->tim_of_max = con_ptr->last_x;
            con_ptr->rec_of_max = con_ptr->rec_last_x;

            con_ptr->prev_slope = cur_slope;
                                        /* slope between samples */
            con_ptr->max_y = con_ptr->last_y;
                                    /* coordinate of max or min */
            con_ptr->last_y = in_data;
                                /* current amplitude coordinate */
            con_ptr->last_x = sam_no;
                                        /* current time coordinate */
            con_ptr->rec_last_x = con_ptr->cur_rec;

            con_ptr->s_sum_sc = con_ptr->sum_s_c;/* save sum */
            con_ptr->sum_s_c = 0;
                                /* initialize sum of sample counts */

/*--------Get max of 20 P-T values less than thx,-----------*/

            if ((ab_amp = absval (con_ptr->s_amp)) > con_ptr->thx)
                return(TRUE);
                        /* P-T > thx (chg frm >= 31-Aug-90/SH) */

            if (ab_amp > con_ptr->maxamp)
                con_ptr->maxamp = ab_amp;

            if (++con_ptr->index == 20) {   /* increment index */

/*---------------and send this maximum to P_two.-----------------*/

                if (con_ptr->maxamp>0) P_two(con_ptr->maxamp);

            /* reset for the next 20 */
                con_ptr->maxamp = 0L;
                con_ptr->index = 0;
            }

            return(TRUE);                       /* P-T < thx */

        }
```

```
        con_ptr->last_y = in_data;
                                /* current amplitude coordinate */
        con_ptr->last_x = sam_no;      /* current time coordinate */
                               /* Note 2 */
        con_ptr->rec_last_x = con_ptr->cur_rec;

        return(FALSE);                 /* peak or trough not found */

}

#ifdef JNMCOMMENT
```

NOTES

1. If a change in slope occurs, the last sample was a peak or trough. Calculate the peak-to-trough amplitude and document the time at which this amplitude occurred.
2. The peak or trough might occur in the last sample of the previous record. Keep track of in which record the peak or trough occurrs.

EXPLANATION OF VARIABLES

ab_amp – the absolute value of the peak-to-trough difference.

cur_slope – slope (positive or negative) between the last data sample (amplitude coordinate) and the current one.

del_amp – the current amplitude coordinate minus the previous one.

in_data – the current data sample.

index – a counter that is used to control calls to P_two.

last_y – amplitude coordinate immediately preceeding the current one.

maxamp – the absolute maximum of 20 successive P-T values < thx.

max_y – maximum (or minimum) amplitude coordinate of the last one-half cycle. (It is the amplitude coordinate of the last peak or trough.)

prev_slope – slope (positive or negative) between the two samples (amplitude coordinates) immediately prior to the current one.

rec_last_x – record in which the previous time coordinate occurred.

rec_of_max – the record in which the peak or trough occurred.

s_amp – signed amplitude (signed peak-to-trough difference).

s_sum_sc – the number of samples between the last two P-T values. (It is the number of samples between the last peak and trough.)

sam_no – the current sample number. Set in E_detect.

sum_s_c – sum of sample counts in the interval from the last peak or trough to the current sample.

tim_of_max – time coordinate of the last one-half cycle. (It is the time coordinate of the last peak or trough, ie of the last P-T value.)

```
#endif
```

A.8 P_TWO.C – Average background estimator

```
#include <detect.h>

/*-------------------------------------------------------------------*
 *    Function: VOID P_two(maxamp) - Estimate average P-T            *
 *-------------------------------------------------------------------*
 *    Arguments:      LONG maxamp - maximum (absolute) of 20         *
 *                          successive P-T values < thx              *
 *-------------------------------------------------------------------*
 *    Returns:        Nothing (VOID)                                 *
 *-------------------------------------------------------------------*
 *    Fatal Errors:   This routine does not have fatal exits         *
 *-------------------------------------------------------------------*
 *    P_two estimates the statistical dispersion of the background   *
 *     noise and calculates the four thresholds that are used in the *
 *     detector.  Remember that P_one finds the maximum of 20 rec-   *
 *     tified P-T values and normally sends this maximum to P_two.   *
 *     P_two averages val_avg (8 or 16) of these maximums. The       *
 *     average thus obtained is twosd.  For zero-mean normally       *
 *     distributed P-T values, twosd would be an estimate of twice   *
 *     the sample standard deviation of the P-T values. (See OF 83-785*
 *     for a comparison between the measured sample standard devia-  *
 *     tion and the estimate of it hereby.)  P_two calls xth to cal- *
 *     culate the thresholds from twosd.                             *
 *-------------------------------------------------------------------*/

VOID P_two(maxamp)
LONG maxamp;
{
        register WORD i,c;
        LONG sum,Xth();
```

```
        if(!con_ptr->evon) { /* twosd not calculated while evon=TRUE */

/*--------------------------Load buffer---------------------------------*/

                con_ptr->tsstak[con_ptr->kk] = maxamp;
                if(++con_ptr->kk == con_ptr->val_avg) con_ptr->kk = 0;

/*-------------------------Average buffer-------------------------------*/

                sum = 0;        /* prepare to average */
#ifndef RAMPUP
                for(i= 0;i < con_ptr->val_avg; i++) sum += con_ptr->tsstak[i];
                c = con_ptr->val_avg;
#else
                c=0;
                for (i=0; i<con_ptr->val_avg; i++) {
                        if (con_ptr->tsstak[i]>0) {
                                sum += con_ptr->tsstak[i];
                                c++;
                        }
                }
#endif

                switch (c) { /* perform average */
                case 16:
                        con_ptr->twosd = sum >> 4;
                        break;
                case 8:
                        con_ptr->twosd = sum >> 3;
                        break;
                default:
/*              we try to avoid 32-bit division                 */
                        if (c<=0) con_ptr->twosd=0;
                        else    con_ptr->twosd = sum / c;
                        break;
                }

                        /* Special condition -- if twosd ever
                                got to be zero, it would stay there */
                if(con_ptr->twosd <= 0)
                        con_ptr->twosd = 1000000L;

/*-----------------------Calculate thresholds-------------------------*/
```

```
                con_ptr->th1 = Ith(con_ptr->xth1);
                con_ptr->th2 = Ith(con_ptr->xth2);
                con_ptr->th3 = Ith(con_ptr->xth3);
                con_ptr->thx = Ith(con_ptr->xthx);

        }
        return;
}

#ifdef JNMCOMMENT
```

DEFINITION OF VARIABLES

evon – event on flag. Set = TRUE in Ibingo (event detected) Set = FALSE in routine Count_dn when itc is decremented to EV_OFF. EV_OFF typically is 2 (evon set=FALSE when itc - EV_OFF × wait_blk samples have been processed, where itc typically is 4 and wait_blk typically is 1014).

kk – index for the tsstak.

maxamp – the absolute maximum of 20 successive P-T values that are less than thx.

sum – the sum of the maximum amplitudes that are stored in the buffer tsstak.

tsstak – a buffer that contains the history of the maximum amplitudes that were input by P_one.

val_avg – the number of maximum values (input from P_one) that is used to estimate each twosd.

```
#endif
```

A.9 ONSET.C – Onset picker

```
#include <detect.h>

/*-------------------------------------------------------------------*
 *    Function: VOID Onset(ibak,sper,tm_indx,                        *
 *                    tc,amp_indx,pt0,pt1,pt2,pt3,pt4)               *
 *                                    -Calculate onset parameters    *
 *-------------------------------------------------------------------*
 *    Arguments:                                                     *
 *                    WORD ibak    - # of P-T values looked back     *
 *                    WORD sper    - period of signal (samples)      *
 *                    WORD tm_indx - index to reference time of      *
```

```
 *                                    first P-T value of signal      *
 *                    WORD tc      - time correction                  *
 *                    WORD amp_indx- index to first P-T value of      *
 *                                      the signal                    *
 *                    LONG pt0     - value 1 for SNR calc             *
 *                    LONG pt1     - value 2 for SNR calc             *
 *                    LONG pt2     - value 3 for SNR calc             *
 *                    LONG pt3     - value 4 for SNR calc             *
 *                    LONG pt4     - value 5 for SNR calc             *
 *-----------------------------------------------------------------*
 *    Returns:        Nothing (VOID)                                 *
 *-----------------------------------------------------------------*
 *    Fatal Errors:   This routine does not have fatal exits         *
 *-----------------------------------------------------------------*
 *    Calculates period, amplitude,SNR series (O-F 83-785), and      *
 *    converts sample number to time. Outputs the estimates.         *
 *-----------------------------------------------------------------*/

VOID Onset(ibak, sper, tm_indx, tc, amp_indx, pt0,
        pt1, pt2, pt3, pt4)
WORD ibak, sper, tm_indx, amp_indx;
LONG pt0, pt1, pt2, pt3, pt4, tc;
{

        WORD i, indx2, j, isnr[5], walkback, idx;
        LONG mx_amp, base, pt[5], temp, onset1;
        WORD t_samp, d_samp;
        TEXT ipol;

        struct s_onset  onsetdata;

/*------------get polarity of first break of the signal--------------*/

        if (pt2 > 0)
            ipol = 'd';              /* pt2 is first P-T of signal */
        else
            ipol = 'c';

/*----------------load array for SNR calculation------------------*/
/*------------------see OF Report 83-785---------------------------*/
        pt[0] = pt0;              /* 2nd P-T before signal */
        pt[1] = pt1;              /* P-T immediately before signal */
        pt[2] = pt2;              /* 1st P-T of signal */
        pt[3] = pt3;              /* 2nd P-T of signal */
        pt[4] = pt4;              /* 3rd P-T of signal */
```

```
/*----------do the SNR calculation without using long division----*/

        for (j = 0; j < 5; j++) {
            base = con_ptr->twosd - (con_ptr->twosd >> 1);
            for (i = 0; i < 10; i++) {
                isnr[j] = i;
                if (absval(pt[j]) < base)
                break;
                base += con_ptr->twosd;
            }
        }

/*------------get the maximum amplitude of the first 4 cycles-----*/

        indx2 = amp_indx;
        mx_amp = 0;
        for (j = 0; j < 8; j++) {
            temp = absval(con_ptr->buf_amp[indx2]);
            if (temp > mx_amp)
                mx_amp = temp;
            if (indx2++ == B_M1)
                indx2 = 0;                  /* indx2 incremented */
        }

/*-----calculate the Period(in seconds) of the first 4 cycles----*/

        Period(sper);

/*-Compute which record the data came from so that the time will-*/
/*-be calculated correctly.  ----------------------------------*/

        t_samp = con_ptr->buf_tim[tm_indx];
        d_samp = con_ptr->cur_rec - con_ptr->buf_rec[tm_indx];
        if (d_samp < 0)
            d_samp += CUR_MAX;

        for (walkback = 0; walkback < d_samp; walkback++) {
                idx = con_ptr->cur_rec - walkback - 1;
                if (idx < 0)
                    idx += CUR_MAX;

                t_samp -= con_ptr->sam_tab[idx];
        }
```

```
/*----------set up the onset time for output----------------------*/

        onset1 = Time_f(t_samp, tc);

        while (onset1<0)     {  onset1+=60;      etime.min--;    }
        while (onset1>59)    {  onset1-=60;      etime.min++;    }

        while (etime.min<0)  {  etime.min+=60;   etime.hr--;     }
        while (etime.min>59) {  etime.min-=60;   etime.hr++;     }
        while (etime.hr<0)   {  etime.hr+=24;    etime.day--;    }
        while (etime.hr>23)  {  etime.hr-=24;    etime.day++;    }

        while (etime.day>etime.day_yr) {
                etime.day -= etime.day_yr-1;
                etime.yr++;
        }

        while (etime.day<1) {
                etime.yr--;
                etime.day += etime.prv_yr;      /* Fix 3-Feb-89/SH */
        }

        if (con_ptr->ponset) {
            printf(" %s", con_ptr->ch_name);
            printf(" %c %d ", ipol, ibak);
            for (i = 0; i < 5; i++)
                printf("%d", isnr[i]);
            printf(" %d ", etime.yr);
            printf("%3d ", etime.day);
            printf("%02d", etime.hr);
            printf(":%02d:", etime.min);
            printf("%02d.%03ld %7ld %2ld.%03d %7ld",
             onset1, t_rval, mx_amp, p_lval, p_rval, con_ptr->twosd);

/*              Was th1 exceeded in the detection? A=yes, B=no. */

            printf(" %c ", con_ptr->iset ? 'A' : 'B');
            printf("\n");
        }

/*--This code is for writing data into headers
             of China System records --*/

        onsetdata.new_onset = TRUE;             /* New onset has occured */
        onsetdata.o_polar = ipol;
```

```
        onsetdata.o_dalgo = con_ptr->iset ? 'A' : 'B';
        for (i = 0; i < 5; i++)
            onsetdata.o_snr[i] = isnr[i];
        onsetdata.o_year = etime.yr;
        onsetdata.o_days = etime.day;
        onsetdata.o_hours = etime.hr;
        onsetdata.o_mins = etime.min;
        onsetdata.o_secs = onset1;
        onsetdata.o_msecs = t_rval;
        onsetdata.o_amps = mx_amp;
        onsetdata.o_pl = p_lval;
        onsetdata.o_pr = p_rval;
        onsetdata.o_large = con_ptr->twosd;

        /* Optional processing routine for onsets */
        if (con_ptr->onsproc != NULL)
            (*(con_ptr->onsproc))(&onsetdata);
                    /* send onsproc the address of
                                        the output list */

/*-Duration of recording made a function of the period of the---*/
/* signal for high frequency events                              */

#ifdef ITC_PER_CNTRL

        if (con_ptr->sam_sec >= VSP_SPS)
                if (p_rval >= PER_TRIG) con_ptr->itc = ITC_UP;

#endif

/*-etime will be changed when events are detected - backup---*/

        etime.day_yr = htime.day_yr;
        etime.prv_yr = htime.prv_yr;
        etime.yr = htime.yr;
        etime.day = htime.day;
        etime.hr = htime.hr;
        etime.min = htime.min;
        etime.sec = htime.sec;
        etime.msec = htime.msec;

/* The thresholds are raised to permit processing the coda of events,*/
/* also, the indices kk and index are set so that Ptwo is called  */
/* expeditiously when processing of the signal coda begins.  */
```

```
/* There is a provision for tuning to process local events.  Here */
/* the maximum P-T (first four cycles) of the signal is loaded into */
/* the thresholds and into the tsstak of P_two.                    */

                                                        /* Note 1 */
#ifdef LOCAL_EV
        con_ptr->twosd = mx_amp;
        for (i = 0; i < con_ptr->val_avg; i++)
            con_ptr->tsstak[i] = mx_amp;
#else
#define bump(x) ((x) + ((x)>>4))
                 /*bump value by a factor of 1.0625 (1-1/16) */
        for(i = 0; i < 16; i++)
                con_ptr->tsstak[i] = bump(con_ptr->tsstak[i]);
        con_ptr->twosd = bump(con_ptr->twosd);
#endif

/*------Raise the thresholds here by calling xth------------*/
/*        after raising twosd (above)                       */

        con_ptr->th1 = Ith(con_ptr->xth1);
        con_ptr->th2 = Ith(con_ptr->xth2);
        con_ptr->th3 = Ith(con_ptr->xth3);
        con_ptr->thx = Ith(con_ptr->xthx);

/*-------To synchronize on-line and off-line detectors--------*/
/*                 See P_one for definitions                   */
                                                /* Note 2 */

#ifdef RESYNCRONIZE
        con_ptr->maxamp = 0;
        con_ptr->s_sum_sc = 0;
        con_ptr->sum_s_c = 0;
        con_ptr->prev_slope = 0;
        con_ptr->max_y = con_ptr->last_y = 0;
        con_ptr->tim_of_max = con_ptr->last_x = 0;

        con_ptr->index = 19;   /*  ensure call to Ptwo expeditiously */

        con_ptr->kk = con_ptr->val_avg - 1;
                                /* ensure new average expeditiously */
#endif

        return;
}
```

```
#ifdef JNMCOMMENT
```

NOTES

1. The thresholds should be raised so that the coda of events can be processed. Indeed, if the thresholds were not raised, and if the noise field changed dramatically (as happens at some stations), the detector could not track the background and would continuously produce detections. Provision is made for two different modes of processing the event coda. If one is primarily interested in local events, one might use the code included when LOCAL_EV is defined. Here one might input a small number for val_avg (see P_two) so that the detector could track changes in the coda more efficiently. Alternatively, if one is processing data of teleseisms, and one wishes to record much of the coda, the default option, with large values (8 - 16) for val_avg, should be employed. In this instance, for large events, one can reasonably expect several retriggers in the coda, some of which might be caused merely by the elevated level of overall activity, rather than by distinct phases.

2. In implementing the code to run on a new system, one often wishes to compare the output on the system being developed with a benchmark that is run on a different system. The variables are set to facilitate such comparisons.

EXPLANATION OF VARIABLES

amp_indx – an index to the first P-T value of the signal. Set in routine Onsetq.

base – a reference value used to calculate the quality numbers of the onset of the signal.

B_M1 – the maximum index of the circular buffers of size E_B.

buf_amp – a buffer of size E_B (maximum index B_M1) that contains the signed P-T values. Loaded in routine Event.

buf_tim – a buffer that contains the number of samples from a reference time for each P-T value in buf_amp. Same size as buf_amp.

buf_rec – contains the (arbitrary) record number of each P-T value. Same size as buf_amp.

CUR_MAX – MAX value for record identification. Value allows a relative age of record to be determined. When the age reaches CUR_MAX, the record number is reset.

cur_rec – the (arbitrary) current record number. Incremented in E_detect, saved in P_one, loaded into buf_rec in Event.

d_samp – Delta samples - age of data in reference to the current record. Used to compute times accurately.

etime.xxx – the time that was initially read in the headers of the records. These times may be changed in the routine Ck_t_kont.

etime.prv_yr – the number of days in the previous year.

htime.xxx – Since the time calculation routines modify etime for the onset time, we must backup etime and restore it so that the next event in the same record can be correctly calculated.

ibak – the number of P-T values that Onsetq looked back to find the first P-T value of the signal. (It is the number of P-T values before fst_flg.)

indx2 – an index that references values in the circular buffers that were set up in routine Event.

ipol – the polarity of the onset of the signal (c or d).

iset – TRUE if P-T $\geq$ th1, FALSE otherwise. See Event.c.

isnr[ ] – the quality numbers (signal-to-noise ratios) that that are integer values each of which is in the interval $0 \leq$ isnr ≤ 9. Dimensionless.

mx_amp – maximum amplitude of the first four cycles (first 8 P-T values) of the signal. Counts.

new_onset – a flag set=TRUE if a signal was declared in the current record. Set=FALSE otherwise.

onset1 – the integer number of seconds from the reference header time. The remainder of the seconds is in t_rval.

p_lval – the integer part of the signal period. Set in routine Period. Seconds.

p_rval – the remainder of the signal period. Set in routine Period. Milliseconds.

pt[ ] – this array contains the values of the first P-T amplitude of the signal (pt[2]), plus the 2 P-T amplitudes on either side of it.

sam_tab – an array containing the number of samples per record for this and the previous CUR_MAX-1 record.

sper – the period of the signal in samples. Set in routine Onsetq.

tc – the time correction that is applied to the reference time. Set in routine Onsetq. Milliseconds.

tm_indx – index to the reference time of the signal. Set in routine Onsetq.

t_rval – the fraction part of onset. This value is set in routine Time_f. Milliseconds.

t_samp – the reference time (in samples) for the onset of the signal. By an intermediate process, the value is initialized at the beginning of each record in E_detect.

twosd – an estimate of twice the sample standard deviation of the P-T values. Set in routine P_two. Counts.

```
#endif
```

A.10 ONSETQ.C – Onset parameter determination

```
#include <detect.h>
#define RO 2 /* round off used for period estimated from 8 P-T values */

/*-------------------------------------------------------------------*
 *    Function: VOID Onsetq() - Estimate onset                       *
 *-------------------------------------------------------------------*
 *    Arguments:      None                                           *
 *-------------------------------------------------------------------*
 *    Returns:        Nothing (VOID)                                 *
 *-------------------------------------------------------------------*
 *    Fatal Errors:   This routine does not have fatal exits         *
 *-------------------------------------------------------------------*
 *    As one might remember,the buffers for each candidate signal    *
 *     have at least 4 P-T values before the first one that was      *
 *     >= th2.  Onsetq compares the last two of these four with yet  *
 *     another threshold, th3 (th3 < th2).  In addition, a test is   *
 *     performed to see whether or not the P-T value looks like      *
 *     it is part of the signal.  Here "looks like" is determined    *
 *     by the period of the signal.  These tests are to search       *
 *     for a signal onset that is smaller than th2.  When the first  *
 *     P-T value of the signal is found, it is flagged.  In P_one,   *
 *     recall the reference time of each P-T value is given when the *
 *     P-T value is declared; hence the time is for the "trailing    *
 *     edge" of the P-T value.  Therefore the signal onset occurs    *
 *     before the time of the first P-T value of the signal.         *
 *     The algorithm considers two possibilities for the onset:      *
 *     It is either the time of the P-T value that immediately pre-  *
 *     ceeds the signal, or if this P-T value occurs too far ahead   *
 *     of the first P-T value of the signal, a correction is applied *
 *     to the time of this P-T value. Here "too far" is deter-       *
 *     mined by the measured period of the signal.  The correction   *
 *     (0 or 500 ms for the SP and BB) and an index to the reference *
 *     P-T are sent to routine Onset for conversion to Universal Time.*
 *                                                                   *
 *    In addition to this index and other useful information, Onsetq *
 *     sends Onset the amplitudes of the two P-T values that occur   *
 *     on either side of the first P-T value of the signal.  These   *
```

```
 *     five P-T values (the two prior to the signal and the first     *
 *     three of the signal) are used by Onset to estimate the         *
 *     quality of the time determination of the beginning of the      *
 *     signal (see O-F Report 83-785).                                *
 *---------------------------------------------------------------------*/

VOID Onsetq() {

        WORD flg_p1, flg_p2, flg_m1, flg_m2, flg_m3, flg_m4, lb,
            i1, i2, i3, i4, i5, i6, per_bnd, per_sc, kase, per_sav;
        LONG tc;

        kase = 0;

/*------Setting up indices that are corrected for a circular buffer --*/

        flg_m4 = con_ptr->fst_flg - 4;   /* 4 P-T values before the first
                                            flagged one */
        if (flg_m4 < 0)
            flg_m4 = con_ptr->fst_flg + B_M4;   /* etc */

        flg_m3 = con_ptr->fst_flg - 3;
        if (flg_m3 < 0)
            flg_m3 = con_ptr->fst_flg + B_M3;

        flg_m2 = con_ptr->fst_flg - 2;
        if (flg_m2 < 0)
            flg_m2 = con_ptr->fst_flg + B_M2;

        flg_m1 = con_ptr->fst_flg - 1;
        if (flg_m1 < 0)
            flg_m1 = con_ptr->fst_flg + B_M1;

        flg_p1 = con_ptr->fst_flg + 1;   /* 1 P-T after the first flagged one
*/
        if (flg_p1 >= E_B)
            flg_p1 = con_ptr->fst_flg - B_M1;

        flg_p2 = con_ptr->fst_flg + 2;   /* etc */
        if (flg_p2 >= E_B)
            flg_p2 = con_ptr->fst_flg - B_M2;

        per_bnd = con_ptr->fst_flg + 8;  /* index of the last P-T used to
                                            estimate the period */
        if (per_bnd > B_M1)
```

```
            per_bnd -= E_B;

        per_sc = con_ptr->buf_sc[per_bnd] -
                con_ptr->buf_sc[con_ptr->fst_flg] + R0;
        per_sc >>= 2;
        per_sav = per_sc;          /* per_sc may be changed below, in the
                                      process that estimates the onset of
                                      the signal */

        if (per_sc < con_ptr->sam_sec)
            per_sc = con_ptr->sam_sec;   /* 1 sec */
        con_ptr->haf_per = per_sc >> 1;

/*--------------Code for looking back 2 P-T values---------------------*/
                                      /* Note 1 */

        if ((con_ptr->buf_sc[flg_m1] - con_ptr->buf_sc[flg_m2]) <= per_sc)
            if (absval (con_ptr->buf_amp[flg_m2]) >= con_ptr->th3)
                kase = ((con_ptr->buf_sc[flg_m2] -
                        con_ptr->buf_sc[flg_m3])
                    > con_ptr->haf_per) ? 1 : 2;

/*--------------Code for looking back 1 P-T value----------------------*/
                                      /* Note 2 */

        if ((kase == 0) && (con_ptr->buf_sc[con_ptr->fst_flg] -
                con_ptr->buf_sc[flg_m1]) <= per_sc)
            if (absval(con_ptr->buf_amp[flg_m1]) >= con_ptr->th3)
                kase = ((con_ptr->buf_sc[flg_m1] -
                        con_ptr->buf_sc[flg_m2])
                    > con_ptr->haf_per) ? 3 : 4;

/*--------------Code for looking back 0 P-T values---------------------*/

        if ((kase == 0) && (con_ptr->buf_sc[con_ptr->fst_flg] -
                con_ptr->buf_sc[flg_m1]) <= con_ptr->haf_per)
            kase = 5;

        switch (kase) {

            case 1:
                lb = 2;
                tc = con_ptr->def_tc;

                i1 = flg_m2;           /* index to reference time */
```

```
    i2 = flg_m4;            /* index to amplitude  of P-T */
    i3 = flg_m3;
    i4 = flg_m2;
    i5 = flg_m1;
    i6 = con_ptr->fst_flg;   /*  "  "   "  "  " */

    break;
case 2:
    lb = 2;
    tc = 0;

    i1 = flg_m3;            /* index to reference time */
    i2 = flg_m4;            /* index to amplitude  of P-T */
    i3 = flg_m3;
    i4 = flg_m2;
    i5 = flg_m1;
    i6 = con_ptr->fst_flg;  /* "     "   "   " */

    break;
case 3:
    lb = 1;
    tc = con_ptr->def_tc;

    i1 = flg_m1;            /* index to reference time */
    i2 = flg_m3;            /* index to amplitude  of P-T */
    i3 = flg_m2;
    i4 = flg_m1;
    i5 = con_ptr->fst_flg;
    i6 = flg_p1;            /*  "  "   "  "  " */

    break;
case 4:
    lb = 1;
    tc = 0;

    i1 = flg_m2;            /* index to reference time */
    i2 = flg_m3;            /* index to amplitude  of P-T */
    i3 = flg_m2;
    i4 = flg_m1;
    i5 = con_ptr->fst_flg;
    i6 = flg_p1;            /*  "  "   "  "  " */

    break;
case 5:
    lb = 0;
```

```
            tc = 0;

            i1 = flg_m1;            /* index to reference time */
            i2 = flg_m2;            /* index to amplitude  of P-T */
            i3 = flg_m1;
            i4 = con_ptr->fst_flg;
            i5 = flg_p1;
            i6 = flg_p2;            /*  "  "   "  "  " */

            break;
        default:
            lb = 0;
            tc = con_ptr->def_tc;

            i1 = con_ptr->fst_flg;   /* index to reference time */
            i2 = flg_m2;            /* index to amplitude  of P-T */
            i3 = flg_m1;
            i4 = con_ptr->fst_flg;
            i5 = flg_p1;
            i6 = flg_p2;            /*  "  "   "  "  " */

            break;

    }

    /* printf("Case=%d, tc=%d, per_sc=%d,haf_per=%d\n",kase,tc,per_sc,
            con_ptr->haf_per); */

    /* printf("m1=%d,m2=%d,m3=%d,m4=%d\n",
            con_ptr->buf_sc[flg_m1],
            con_ptr->buf_sc[flg_m2],
            con_ptr->buf_sc[flg_m3],
            con_ptr->buf_sc[flg_m4]); */

    Onset(lb, per_sav, i1, tc, i4, con_ptr->buf_amp[i2],
            con_ptr->buf_amp[i3], con_ptr->buf_amp[i4],
            con_ptr->buf_amp[i5], con_ptr->buf_amp[i6]);

    return;
}

#ifdef JNMCOMMENT
```

NOTES

1. Algorithm for looking back two P-T values. First we check to see that the difference between the two P-T values (fst_flg-1 and fst_flg-2) is not too large (1 sec or the period, whichever is greater). If the P-T values pass this test, we then check whether or not fst_flg-2 $\geq$ th3. If either of these two tests fail, we proceed to the algorithm for looking back one P-T value. However, if both tests succeed, we then test to see what time we will use for the onset; if the preceeding (fst_flg-3) P-T value occurs too far ahead of fst_flg-2, we use the time of fst_flg-2 - 0.5 sec (an input value), otherwise we use the time of fst_flg-3. Here "too far" is per_sc/2. (This contrasts with the FORTRAN progam which used a constant of 10 (sam_sec/2) instead of per_sc/2: For general purpose application, especially for the Long Period System, and perhaps for the Broad Band as well, the 0.5 sec (10 samples) appears too small.)

2. Algorithm for looking back one P-T value. The logic parallels that explained in [1] above. Note that kase $\neq$ 0 if both time and amplitude criteria were met in [1] above.

EXPLANATION OF VARIABLES

B_M1

B_M2

B_M3

B_M4 – the size (E_B) of the large P-T buffers minus 1,...,minus 4

buf_amp – a buffer that contains the signed P-T amplitudes.

buf_rec – a buffer that contains the record numbers in which each of the P-T values of buf_amp occurred.

buf_sc – a buffer that indicates time by using the sum of the samples. This clock is initialized when the abbreviated buffers are downloaded in Event (when an event is judged possibly to be in progress).

def_tc – an input value that is loaded into tc (for SP and BB of the China System it is 500 (ms)).

fst_flg – an index to the first P-T value of the current signal that is $\geq$ th2.

flg_m1 – fst_flg minus 1

flg_m2 – fst_flg minus 2

flg_m3 – fst_flg minus 3

flg_m4 – fst_flg minus 4

flg_p1 – fst_flg plus 1

flg_p2 – fst_flg plus 2

i1

i2

i3

i4

i5

i6 – indices that are used to reference values in the large buffers of Event (buf_amp, buf_rec, buf_tim). The index i1 references time, the others, amplitude.

kase – an index that is used in the lookback proceedure.

lb – the number of P-T values that Onsetq looked back to determine the first P-T of the signal (0, 1, or 2).

per_bnd – a bound that is used to estimate the period of the first four cycles of the signal.

per_sav – saved value of per_sc, used for outputting the period of the signal. Samples.

per_sc – the period of the signal (samples).

RO – round off for estimating the period of the signal. Samples.

sam_sec – digitizing rate of the current seismic data record. Samples per second.

haf_per – samples per one-half period.

tc – a time correction that is applied (see discussion in the header of this routine). Milliseconds.

```
#endif
```

A.11 IBINGO.C – Event flag and timer setup

```
#include <detect.h>

/*-----------------------------------------------------------------------*
 *    Function: VOID Ibingo() - Sets parameters                          *
 *-----------------------------------------------------------------------*
 *    Arguments:      No arguments                                       *
 *-----------------------------------------------------------------------*
 *    Returns:        Nothing (VOID)                                     *
 *-----------------------------------------------------------------------*
 *    Fatal Errors:   This routine does not have fatal exits             *
 *-----------------------------------------------------------------------*
```

```
 *    Ibingo is called when an event has been detected.  (Here it is  *
 *     useful to note that although Ibingo is called when an event    *
 *     is detected, an event is not declared, ie routine Event re-    *
 *     turns true, until enough P-T values have been processed to     *
 *     estimate the period of the signal.)  The purpose of Ibingo     *
 *     is to set parameters for processing the interval of the event. *
 *     The interval of the event is interval in which E_detect re-    *
 *     turns TRUE.  Typically the interval of the event is            *
 *     itc x wait_blk samples, where usually itc is 4 and wait_blk    *
 *     is 1014 (samples).                                             *
 *-----------------------------------------------------------------*/

VOID Ibingo()
{

        con_ptr->epf = FALSE;
        con_ptr->evon = TRUE;
        con_ptr->icheck = FALSE;
        con_ptr->itc = NOR_OUT;
        con_ptr->nn = 0;        /*  Initialize for Count_dn */

        return;
}
#ifdef JNMCOMMENT
```

EXPLANATION OF VARIABLES

epf – event possible flag, set to TRUE by Event when a signal is deemed possibly to be in progress (but not yet detected), set to FALSE upon detection.

evon – event on flag, set to TRUE when an event is detected, set to FALSE when NOR_OUT-EV_OFF × wait_blk samples have been processed by Count_dn. Typically EV_OFF = 2, wait_block = 1014.

icheck – used in routine Event to inhibit declaring an event until enough P-T values have been processed to estimate the period of the signal. Set to TRUE in Event when these values have been processed.

itc – counter for the interval of the event. Decremented to zero when the last sample of the interval of the event has been processed. The interval of the event typically is itc × wait_blk samples. (The interval will be longer if a retrigger occurs.) Typically, itc = 4, wait_blk = 1014. See Count_dn.

nn – an index used in routine Count_dn.

NOR_OUT – typically 4. It is a factor of wait_blk through itc.

th1 – threshold for detection of event. See Xth, Event. (Counts.)

th2 – threshold for detection of event. See Xth, Event. (Counts.)

th3 – threshold for estimating onset of event. See Xth, Onsetq. (Counts.)

thx – upper bound for estimate of twosd. See P_one, Xth. (Counts.)

```
#endif
```

A.12 WBUFF.C – Event storage buffer setup

```
#include <detect.h>

/*-------------------------------------------------------------------*
 *    Function: VOID Wbuff() - Write buffers                         *
 *-------------------------------------------------------------------*
 *    Arguments:      No arguments                                   *
 *-------------------------------------------------------------------*
 *    Returns:        Nothing (VOID)                                 *
 *-------------------------------------------------------------------*
 *    Fatal Errors:   This routine does not have fatal exits         *
 *-------------------------------------------------------------------*
 *    Wbuff updates buffers when an event might be in progress, but  *
 *     not yet detected, ie when the event possible flag (epf) is    *
 *     set = TRUE plus during the interval that is required for      *
 *     enough to data to estimate the period of the event.           *
 *-------------------------------------------------------------------*/

VOID Wbuff()
{

        if(con_ptr->lst_pt++ == B_M1) con_ptr->lst_pt = 0;
                                                /* increment lst_pt */

        con_ptr->buf_flg[con_ptr->lst_pt] = th_wt;
        con_ptr->buf_sc[con_ptr->lst_pt] = con_ptr->sumdsc;
        con_ptr->buf_amp[con_ptr->lst_pt] = con_ptr->s_amp;
        con_ptr->buf_tim[con_ptr->lst_pt] = con_ptr->tim_of_max;
        con_ptr->buf_rec[con_ptr->lst_pt] = con_ptr->rec_of_max;

        return;
}

#ifdef JNMCOMMENT
```

EXPLANATION OF VARIABLES

buf_amp – the circular buffer of size E_B that contains the signed P-T values. See Event.

buf_flg – the circular buffer of size E_B that contains the weights (0,1,or 2) for each P-T value of buf_amp. See Event.

buf_rec – the circular buffer of size E_B that contains the (arbitrary) record number for each P-T value of buf_amp. See E_detect, P_one, Event.

buf_sc – the circular buffer of size E_B that contains the sum of the number of samples (time axis) from initilization to each P-T of buf_amp. See Event.

buf_tim – the circular buffer of size E_B that contains a sample number for each P-T value of buf_amp. The sample number is initialized at the beginning of each record and is incremented for each seismic data sample. See E_detect, P_one, Event.

lst_pt – the index (within the circular buffers) of the last P-T value. The circular buffers are of size E_B, maximum index of B_M1.

rec_of_max – the (arbitrary) record number for each tim_of_max. See P_one, Event.

s_amp – the signed P-T amplitude. See P_one.

sumdsc – sum of the sample counts that was initialized when the epf flag was set = TRUE, ie, when the abbreviated buffers were downloaded in Event. See Event.

th_wt – a weight that is assigned by Event to each P-T value of a possible signal. Its values are 0, 1, or 2.

tim_of_max – the sample number of each P-T value of buf_amp. The sample number is initialized at the beginning of each record and is incremented for each seismic data sample thereafter. tim_of_max is used to time each P-T value. See E_detect, P_one, Event.

```
#endif
```

A.13 XTH.C – Threshold calculator

```
#include <detect.h>

/*-------------------------------------------------------------------*
 *    Function: LONG Xth(xthi) - Calculate threshold                 *
 *              adapted from a routine by R.R. Reynolds              *
 *-------------------------------------------------------------------*
 *    Arguments:      UWORD xthi - factor                            *
 *-------------------------------------------------------------------*
```

```
 *    Returns:        Return the threshold value                        *
 *----------------------------------------------------------------------*
 *    Fatal Errors:   This routine does not have fatal exits            *
 *----------------------------------------------------------------------*
 *    The routine Xth forms the thresholds from twosd and from the      *
 *     encoded factors xthi.  Shifts and encoded factors are em-        *
 *     ployed to circumvent the need for 32 bit multiplication.         *
 *                                                                      *
 *    As coded here, the maximum permitted value of xthi is 0377        *
 *----------------------------------------------------------------------*/

LONG Xth(xthi)
register UWORD xthi;
{

        LONG th = 0;
        register UWORD x_left;

        x_left = xthi >>3;      /* remove first three bits */

/*--------Calculate high order part of threshold-------------------*/

        if(x_left) {
                if(x_left & 020) th  = con_ptr->twosd<<4;
                if(x_left & 010) th += con_ptr->twosd<<3;
                if(x_left & 004) th += con_ptr->twosd<<2;
                if(x_left & 002) th += con_ptr->twosd<<1;
                if(x_left & 001) th += con_ptr->twosd;
        }

/*---------Calculate low order part of threshold------------------*/

        if(xthi & 07) {         /* use the first 3 bits */
                if(xthi & 01) th += (con_ptr->twosd >>3);
                if(xthi & 02) th += (con_ptr->twosd >>2);
                if(xthi & 04) th += (con_ptr->twosd >>1);
        }
        return(th);
}



#ifdef JNMCOMMENT
```

EXPLANATION OF VARIABLES

twosd - an estimate of twice the sample standard deviation of the P-T values (counts). See P_one, P_two.

x_left - an unsigned 16 bit word that contains xthi, with the first 3 bits of xthi removed. After shifting to remove these bits, the remaining that are set in x_left are used to control the amount twosd is multiplied (shifted) to obtain the high order part of the threshold, th.

xthi - the input encoded factor of twosd. Octal values. Octal values were selected for continuity with the SRO on-line detector.

```
#endif
```

A.14 TIME_F.C - Event pick time determination

```
#include <detect.h>

/*-------------------------------------------------------------------*
 *    Function: WORD Time_f(t_kont,tc) - "floating point"            *
 *                                          for output               *
 *-------------------------------------------------------------------*
 *    Arguments:                                                     *
 *                    WORD t_kont - onset sample number              *
 *                    WORD tc - time correction                      *
 *-------------------------------------------------------------------*
 *    Returns:        LONG - onset time                              *
 *-------------------------------------------------------------------*
 *    Fatal Errors:   This routine does not have fatal exits         *
 *-------------------------------------------------------------------*
 *    This routine calculates the onset time of the detected signal  *
 *     from t_kont and tc.  One purpose of this routine is to cir-   *
 *     cumvent floating point operations.                            *
 *-------------------------------------------------------------------*/

LONG Time_f(t_kont,tc)
WORD t_kont;
LONG tc;
{

        LONG reftr,t_lval;

/*-----------------More than one minute per record?-----------------*/
/*                   If so, adjust values read in headers.           */
```

```
        t_kont = Ck_t_kont(t_kont);

        if (con_ptr->sam_sec>0) {

                t_lval = t_kont/con_ptr->sam_sec;       /* Integer Seconds */
                reftr = t_kont - t_lval * con_ptr->sam_sec; /* Rem in samples
*/
                t_rval = reftr * con_ptr->ms_sam + etime.msec - tc; /* Rem ms
*/

        } else {

                t_lval = t_kont * (con_ptr->ms_sam / 1000);
                reftr = t_kont - ((t_lval*1000) / con_ptr->ms_sam);
                t_rval = etime.msec - tc;

                /* printf("time_f t_kont=%d, t_lval=%d, t_rval=%d, reftr=%d\n",
                        t_kont,t_lval,t_rval,reftr); */

        }

/*-------------------Obtain remainder < 1000 ms ---------------------*/

        while(t_rval >= 1000) {
                t_lval++;
                t_rval -= 1000; /* milliseconds remainder */
        }

/*-------The time correction could have caused remainder < 0 -------*/

        while(t_rval <= -1000) {
                t_lval--;
                t_rval += 1000; /* milliseconds remainder */
        }

        if(t_rval < 0) {
                t_lval--;
                t_rval += 1000; /* milliseconds remainder */
        }

/*------------Integer seconds relative to header time--------------*/

        t_lval += etime.sec;

        return(t_lval);
```

```
}

#ifdef JNMCOMMENT
```

EXPLANATION OF VARIABLES

etime.sec – the reference time (seconds) of a record header.

ms_sam – sample rate expressed as milliseconds per sample.

sam_sec – digitizing rate of the current seismic data record. Samples per second.

reftr – remainder of integer division expressed in samples.

tc – the time correction (milliseconds) that is applied to the reference onset. For BB and SP data of the China System tc = 500 (ms) or 0.

t_kont – the number of samples from a reference time of a record header to the reference onset of a signal.

t_lval – the integer part of the onset time (seconds).

t_rval – the remainder of the onset time expressed in milliseconds.

```
#endif
```

A.15 CK_T_KONT.C – Pick-time adjustments

```
#include <detect.h>

/*-------------------------------------------------------------------*
 *    Function: WORD Ck_t_kont(t_kont) -- reduces t_kont             *
 *-------------------------------------------------------------------*
 *    Arguments:      WORD t_kont - number of counts from beginning  *
 *                    or end of record to reference time of event    *
 *-------------------------------------------------------------------*
 *    Returns:        t_kont                                         *
 *-------------------------------------------------------------------*
 *    Fatal Errors:   This routine does not have fatal exits         *
 *-------------------------------------------------------------------*
 *    Ck_t_kont is implemented to process records that are one minute *
 *     or longer in length.  For such records, it reduces t_kont to  *
 *     less than the number of counts per minute, and it adjusts the *
 *     time field that was read in the record header accordingly.    *
 *-------------------------------------------------------------------*/
```

```
WORD Ck_t_kont(t_kont)
WORD t_kont;
{
        LONG min_cor;

        if (con_ptr->sam_sec>0) { /* sam_sec is a WORD */

                min_cor = t_kont / (con_ptr->sam_sec * 60);

                if (min_cor == 0)
                    return(t_kont);

                t_kont -= 60L * con_ptr->sam_sec * min_cor;

        } else {

                /* printf("org t_kont=%d ",t_kont); */

                min_cor = (((LONG) t_kont) * (con_ptr->ms_sam / 1000L));
                t_kont = (min_cor % 60L) / (con_ptr->ms_sam / 1000L);
                min_cor = min_cor / 60L; /* Calculate minutes */

                /* printf("fix t_kont=%d min_cor=%d\n",t_kont,min_cor); */

                if (min_cor == 0)
                    return(t_kont);
        }

/*---------If the correction != 0, header time must be adjusted -----*/

        etime.min += min_cor;

        while (etime.min<0)  {  etime.min+=60;          etime.hr--;     }
        while (etime.min>59) {  etime.min-=60;          etime.hr++;     }
        while (etime.hr<0)   {  etime.hr+=24;           etime.day--;    }
        while (etime.hr>23)  {  etime.hr-=24;           etime.day++;    }

        while (etime.day>etime.day_yr) {
                etime.day -= etime.day_yr-1;
                etime.yr++;
        }

        while (etime.day<1) {
                etime.yr--;
```

```
                etime.day += etime.prv_yr;      /* Fix 3-Feb-89/SH */
        }

        return(t_kont);
}

#ifdef JNMCOMMENT
```

EXPLANATION OF VARIABLES

etime.day – the time (day) that was read from the header of the seismic data record.

etime.hr – the time (hour) that was read from the header of the seismic data record.

etime.min – the time (min) that was read from the header of the seismic data record.

etime.prv_yr – the number of days in the previous year.

etime.yr – the time (year) that was read from the header of the seismic data record.

ms_sam – Milliseconds per sample - used for sample rates less than one

min_cor – the correction (minute) that is applied to the time in the header of the seismic data record.

prev_yr – the number of days in the previous year.

sam_sec – digitizing rate of the current seismic data record. Samples per second.

t_kont – the number of samples from a reference minute (of the record header) to the reference point of the signal. See Onset (t_samp).

```
#endif
```

A.16 PERIOD.C – Event period determination

```
#include <detect.h>

/*-------------------------------------------------------------------*
 *    Function: VOID Period(p_kont) - Find "floating point"          *
 *                                          for output               *
 *-------------------------------------------------------------------*
 *    Arguments:                                                     *
 *                    WORD p_kont - samples in the period            *
 *-------------------------------------------------------------------*
 *    Returns:        Nothing (VOID)                                 *
```

```
 *-------------------------------------------------------------------*
 *    Fatal Errors:   This routine does not have fatal exits         *
 *-------------------------------------------------------------------*
 *    The routine Period calculates the period of the detected signal *
 *     from the value P_kont.  The purpose of this routine is to      *
 *     circumvent floating point operations.                          *
 *-------------------------------------------------------------------*/

VOID Period(p_kont)
WORD p_kont;
{
        if (con_ptr->sam_sec>0) {
                p_lval = p_kont/con_ptr->sam_sec;
                p_rval = (p_kont % con_ptr->sam_sec) * con_ptr->ms_sam;
        } else {
                p_lval = p_kont * (con_ptr->ms_sam / 1000);
                p_rval = 0;
                /* printf("p_kont=%d,ms_sam=%d,p_lval=%d\n",
                        p_kont,con_ptr->ms_sam,p_lval); */
        }

        return;
}

#ifdef JNMCOMMENT
```

EXPLANATION OF VARIABLES

ms_sam – milliseconds between successive samples of the digital data.

p_kont – the average number of samples per cycle in the first four cycles of the declared signal. See Onsetq.

p_lval – the integer part of the period. Seconds.

p_rval – the remainder (decimal fraction) of the period expressed in milliseconds.

sam_sec – samples per second of the digital data. See E_detect.

```
#endif
```

A.17 COUNT_DN.C – Event on/off countdown

```
#include <detect.h>
```

```
/*-------------------------------------------------------------------*
 *    Function: VOID Count_dn() - Count Down Interval of Event       *
 *-------------------------------------------------------------------*
 *    Arguments:      No arguments                                   *
 *-------------------------------------------------------------------*
 *    Returns:        Nothing (VOID)                                 *
 *-------------------------------------------------------------------*
 *    Fatal Errors:   This routine does not have fatal exits         *
 *-------------------------------------------------------------------*
 *    As implemented here, to inhibit multiple printouts immediately *
 *     after a detection, the detector is disabled for (NOR_OUT -    *
 *     EV_OFF) x wait_blk samples and then reenabled.  The interval  *
 *     of the event is NOR_OUT x wait_blk samples.                   *
 *-------------------------------------------------------------------*/

VOID Count_dn()
{

        if(con_ptr->itc > 0) {  /* itc was set to NOR_OUT in Ibingo */
                if(++con_ptr->nn >= con_ptr->wait_blk) {  /* increment nn */
                        con_ptr->nn = 0;
                        con_ptr->itc--; /* decrement wait-block counter */
                }
        }

                        /* enable detecting events */
        if(con_ptr->itc <= EV_OFF) con_ptr->evon = FALSE;

        return;
}


#ifdef JNMCOMMENT
```

EXPLANATION OF VARIABLES

EV_OFF – controls the interval in which events are detected in the P coda. Typical value = 2. To enable detections throughout the P coda, set EV_OFF = NOR_OUT. To completely disable detections as well as recomputation of thresholds in the P coda, set EV_OFF = 0. Note: $0 \leq$ EV_OFF $\leq$ NOR_OUT.

evon – the event on flag. While evon = TRUE, events are not detected and P_two is not called (thresholds are not computed). We want to inhibit detections in the P_coda to suppress spurious printouts by Onset.

itc – counter for the interval of the event. Set = NOR_OUT in Ibingo. When itc = 0, the interval of the event is terminated.

nn – counter for itc.

wait_blk – wait block. The variable itc is a factor of wait_blk. The interval of the event is NOR_OUT × wait_blk. As stated previously, because we envision that data will be written to tape during the interval of the event, wait_blk typically is chosen as a function of the input record length.

```
#endif
```

A.18 CONT_SETUP.C – Prepare event structures

```
#include <detect.h>

/*-----------------------------------------------------------------*
 *    Function: VOID Cont_setup(conptr) - Initialize continuity    *
 *-----------------------------------------------------------------*
 *    Arguments:      struct con_sto *contin - pointer to the      *
 *                     continuity info (data from the previous run)*
 *-----------------------------------------------------------------*
 *    Returns:        Nothing (VOID)                               *
 *-----------------------------------------------------------------*
 *    Fatal Errors:   This routine does not have fatal exits       *
 *-----------------------------------------------------------------*
 *    This routine initializes parameters on the first pass.       *
 *    In particular, the routine sets itc to a value greater than  *
 *    zero so that E_detect returns true during the initilization  *
 *    process.  If one expects to change the sample rate during the*
 *    processing, Main should call this routine to reinitialize the*
 *    parameters.                                                  *
 *-----------------------------------------------------------------*/

VOID Cont_setup(contin)
register struct con_sto *contin;
{

        WORD i;
        contin->last_x = 0;             /* initialize for P_one */
        contin->rec_last_x = 0;         /* initialize for P_one */
        contin->sum_s_c = 0;            /* initialize for P_one */
        contin->s_sum_sc = 0;           /* initialize for P_one */
        contin->evon = FALSE;           /* initialize for Event */
```

```
        contin->epf = FALSE;            /* initialize for Event */
        contin->fst_flg = 0;            /* initialize for Event */
        contin->index = 0;              /* initialize for P_one */
        contin->kk = 0;                 /* initialize for P_two */
        contin->jj = 0;                 /* initialize for Event */

        contin->prev_slope = 2;         /* initialize for P_one */
        contin->s_amp = 600000L;        /* initialize for P_one */
        contin->last_y = 0L;            /* initialize for P_one */
        contin->maxamp = 0L;            /* initialize for P_one */
        contin->max_y = 0L;             /* initialize for P_one */

/*---------Set the thresholds high enough to suppress retriggers------*/
/*                      in the initialization process                  */

        contin->th1 = 500000L;
        contin->th2 = 500000L;
        contin->th3 = 500000L;

/*--------------Ensure that the thresholds will remain high ---------*/
/*                 during the initialization process. See P_two, Xth. */

        contin->twosd = 300000L;
        contin->thx = contin->twosd <<1;
#ifndef RAMPUP
        for(i = 0;i < 16;i++) contin->tsstak[i] = 1000000L;
#else
        for(i = 0;i < 16;i++) contin->tsstak[i] = 0;
#endif

/*-------------Initialize the buffers for Event----------------------*/

        for(i = 0;i < 4; i++) {
                contin->abuf_sc[i] = contin->abuf_tim[i] = 0;
                contin->abuf_amp[i] = 0L;
        }
        for(i = 0;i < E_B; i++) {
                contin->buf_flg[i] = contin->buf_sc[i] = 0;
                contin->buf_tim[i] = 0;
                contin->buf_amp[i] = 0L;
        }

        contin->itc = NOR_OUT;

/*--------------------Defaults for onset processing-----------------*/
```

```
#ifdef NOPONSET
        contin->ponset=FALSE;    /* No output on standard out */
#else
        contin->ponset=TRUE;     /* Output on standard out */
#endif

        contin->onsproc=NULL;    /* No auxillarly output */

        return;
}
```

Appendix B

Data Management C Code

Here is the C code for the optional subsidiary routines which can assist in assembling the desired customized event detection system.

B.1 E_BUFFER.C – Allocate data buffers

```
#include <detect.h>

/*-------------------------------------------------------------------*
 *    Function: LONG *E_buffer(maxdata,maxlookback) - allocate and   *
 *                        initialize data area                       *
 *-------------------------------------------------------------------*
 *    Arguments:      WORD maxdata - Maximum number of expected      *
 *                           data points for data array              *
 *                    WORD maxlookback - Maximum lookback area       *
 *                           expected (if any).  Leave 0 if          *
 *                           no lookback is to be used.              *
 *-------------------------------------------------------------------*
 *    Returns:        Address of allocated buffer, or NULL if        *
 *                    there was not enough memory or total size<=0   *
 *-------------------------------------------------------------------*
 *    Allocates memory for the user's data.  Provides memory for the *
 *    "lookback" that is required by the FIR filters.                *
 *-------------------------------------------------------------------*/

LONG *E_buffer(maxdata,maxlook)
WORD maxdata,maxlook;
{

        UWORD total;
        LONG *newptr;

        /* printf("E_buffer(%d,%d)\n",maxdata,maxlook); */
```

```
        total=maxdata+maxlook;
        total*=sizeof(LONG);            /* Calculate size */

        if (total==0) return(NULL);     /* No action */

        newptr=(LONG *) malloc(total);          /* Get storage */

        return(newptr);                 /* Pass or fail */

}
```

B.2 E_CDSNLOAD.C – Convert CDSN data into integers

```
#include <detect.h>

/*-------------------------------------------------------------------*
 *    Function: BOOL E_cdsnload(detector,indata,offset) - load cdsn  *
 *-------------------------------------------------------------------*
 *    Arguments:      struct detect_info *detector - calling info    *
 *                    UBYTE *indata - pointer to raw CDSN data record*
 *                    WORD offset - what multiplexed channel         *
 *-------------------------------------------------------------------*
 *    Returns:        TRUE - Data valid and converted                *
 *                    FALSE - the recordtype did not match           *
 *-------------------------------------------------------------------*
 *    Fatal Errors:   This routine does not have fatal exits         *
 *-------------------------------------------------------------------*
 *    Examine a CDSN record in its internal format.  Determine a     *
 *    record type key from the sample rate and compare this key      *
 *    with the selection criteria in the structure "detector".  If   *
 *    the record types are the same, decode the rest of the data     *
 *    and return TRUE. Otherwise, do no more, and return FALSE       *
 *    immediately.                                                   *
 *-------------------------------------------------------------------*/

#define SAM_RC 1014             /* Size of the data in words */
#define CDSN_EPOCH 83           /* No data before 1983 */
#define CDSN_CENT 1900          /* This is the 20th century */

#define IUBYTE(x)       ((x)&0xFF)                      /* Low order byte */
#define HINIB(x)        ((IUBYTE(indata[x])>>4)&0xF)    /* High order nibble
*/
#define LONIB(x)        (IUBYTE(indata[x])&0xF)         /* Low order nibble */
```

```
BOOL E_cdsnload(detector, indata, offset)
struct detect_info *detector;
UBYTE * indata;
WORD offset;
{

        WORD i, rect, numcomp, leap;
        LONG samrat, sampr, nmsec, fmsec, fsec, fmin, fhour;
        BOOL nak;

        WORD lp, j, gr, ct;
        LONG l_data, *oarray;
        UBYTE * bytarr;

        samrat = HINIB(16) * 10000L +
            LONIB(16) * 1000L + HINIB(17) * 100L;

        i = samrat / 10;
        switch (i) {
            case 4000:          /* 40 samples per second */
                rect = 1;
                break;
            case 2000:          /* 20 samples per second */
                rect = 2;
                break;
            case 100:           /*  1 sample per second */
                rect = 3;
                break;
            case 10:            /* 0.1 sample per second */
                rect = 4;
                break;
            default:            /* Unknown, therefore illegal sample rate */
                rect = 0;
                break;
        }

        if (detector->drectyp != rect)
            return(FALSE);                      /* Not this one */

        detector->samrte = samrat;              /* Detector needs this */

        numcomp = LONIB(8);     /* Number of components in this record */

/*-----------------Compute a complete date----------------------------*/
```

```
        i = LONIB(1) * 10 + HINIB(2);
        if (i < CDSN_EPOCH)
            i += 100;
        i += CDSN_CENT;

        if (i != detector->startt.yr) {
            leap = 0;
            detector->startt.yr = i;
            if (!(i % 4))
                leap = 1;
            if (!(i % 100))
                leap = 0;
            if (!(i % 400))
                leap = 1;
            detector->startt.day_yr = 365 + leap;
            i = i - 1;
            if (!(i % 4))
                leap = 1;
            if (!(i % 100))
                leap = 0;
            if (!(i % 400))
                leap = 1;
            detector->startt.prv_yr = 365 + leap;
        }

        detector->startt.day = LONIB(2) * 100 + HINIB(3) * 10 + LONIB(3);
        detector->startt.hr = HINIB(4) * 10 + LONIB(4);
        detector->startt.min = HINIB(5) * 10 + LONIB(5);
        detector->startt.sec = HINIB(6) * 10 + LONIB(6);
        detector->startt.msec = HINIB(7) * 100 + LONIB(7) * 10;

        bytarr = &indata[20];
        oarray = &detector->indatar[detector->lbksize];

        ct = 0;

        if (offset > numcomp)
            offset = numcomp;           /* Offset specified is illegal */
        if (offset < 0)
            offset = 0;                 /* Offset illegal */

/*----------------------Convert data from gain-ranged--------------------*/

        for (lp = offset * 2; lp < (SAM_RC * 2); lp +=
```

```
            numcomp * 2) {

        j = (bytarr[lp] & 0x3F) << 8;        /* Strip off exponent */
        j |= (bytarr[lp + 1] & 0xFF);

        j -= 8191;                           /* Form signed value */

        l_data = (LONG) j;

        gr = (bytarr[lp] & 0xC0) >> 6;       /* Get the exponent */

        switch (gr) {       /* Apply exponent to form 32 bit integer */
            case 1:
                l_data <<= 2;
                break;

            case 2:
                l_data <<= 4;
                break;

            case 3:
                l_data <<= 7;
                break;

        }

        *oarray++ = l_data;          /* Load data into output array */
        ct++;                        /* increment count */
    }
    detector->datapts = ct;          /* Save count */

    return(TRUE);                    /* Sucess */

}
```

B.3 E_CREATE.C – Allocate user detector structures

```
#include <detect.h>

/*-------------------------------------------------------------------*
 *   Function: BOOL E_create(detector,looksize,rectyp,detnam,dataarr, *
 *         filter) - create an event detector with given parameters   *
 *-------------------------------------------------------------------*
 *    Arguments:      struct detect_info *detector - calling detector *
```

```
 *                          configuration information                *
 *                  WORD looksize - lookback size for detector       *
 *                          (0 if no lookback)                       *
 *                  WORD rectyp - record type to detect - see the    *
 *                          decoder (such as E_cdsnload()) you       *
 *                          are using to find what                   *
 *                          numbers to use here                      *
 *                  TEXT *detnam - name of detector.                 *
 *                  LONG *dataarr - Pointer to the beginning of      *
 *                          data array.  The size of the array must  *
 *                          be at least as large as the maximum      *
 *                          number of data points plus the maximum   *
 *                          value of lookback.                       *
 *                  BOOL (*infilter)() - function to be used as a    *
 *                          filter.  Fnull() is used if NULL is      *
 *                          specified here.                          *
 *-----------------------------------------------------------------*
 *    Returns:      Returns TRUE if job completed sucessfully        *
 *-----------------------------------------------------------------*
 *    Creates an event detector.  Allocates memory for the lookback  *
 *    array and the continuity structure.  It then calls the         *
 *    Cont_setup() routine to initialize the continuity structure.   *
 *-----------------------------------------------------------------*/

BOOL E_create(detector,looksize,rectyp,detnam,dataarray,infilter)
struct detect_info *detector;
WORD looksize,rectyp;
TEXT *detnam;
LONG *dataarray;
BOOL (*infilter)();
{

        UWORD msize;
        BOOL Fnull();

/*----------------------First allocate the continuity structure--------*/

        detector->incontd= (struct con_sto *) malloc((UWORD)
                        sizeof(struct con_sto));
        if (detector->incontd==NULL) return(FALSE);     /* Nope */

/*-------------Allocate the lookback storage array--------------------*/

        detector->lbkarr=NULL;          /* Initialize to no lookback */
```

```
        if (looksize>0) {
                msize=looksize;
                msize*=sizeof(LONG);

                detector->lbkarr=(LONG *) malloc(msize);

                if (detector->lbkarr==NULL) {
                        mfree((UBYTE *) detector->incontd);
                                                /* Don't leave a mess */
                        return(FALSE);
                }
        }

/*-------------Do all of the rest of the setup-------------------------*/

        detector->samrte=0;
        detector->drectyp=rectyp;
        detector->datapts=0;
        detector->indatar=dataarray;
        detector->lbksize=looksize;
        detector->detname=detnam;
        detector->filterc=infilter;
        if (infilter==NULL) detector->filterc=Fnull;

/*-------------Call the continuity setup-------------------------------*/

        Cont_setup(detector->incontd);
        detector->incontd->ch_name=detnam;

        return(TRUE);

}
```

B.4 E_FILTER.C – Filter the user data

```
#include <detect.h>

/*-------------------------------------------------------------------*
 *    Function: BOOL E_filter(detector) - Implements filtering       *
 *-------------------------------------------------------------------*
 *    Arguments:      struct detect_info detector - detector         *
```

```
 *                        configuration parameters                        *
 *------------------------------------------------------------------------*
 *    Returns:        Returns TRUE if job completed sucessfully           *
 *------------------------------------------------------------------------*
 *    Gets the old lookback from the temporary buffer in detector,        *
 *    and places it in the data array.  E_filter() then saves the         *
 *    lookback data which will be used the next time the current          *
 *    detector is called. Routine then calls the filtering                *
 *    subroutine specified in E_create() and placed as a pointer          *
 *    to a function in detector structure.                                *
 *------------------------------------------------------------------------*/

BOOL E_filter(detector)
struct detect_info *detector;
{

        WORD i;

/*-------------Retrieve the old lookback---------------------------*/

        if (detector->lbksize>0) {

                for (i=0; i<detector->lbksize; i++) detector->indatar[i]=
                        detector->lbkarr[i];

        }

/*--------------Put new lookback away------------------------------*/

        if (detector->lbksize>0) {

                for (i=0; i<detector->lbksize; i++) detector->lbkarr[i]=
                        detector->indatar[i+detector->datapts];

        }

/*--------------Do the required filtering--------------------------*/

        (*(detector->filterc))
                (detector,detector->datapts,detector->lbksize,
                                detector->indatar);

        return(TRUE);

}
```

B.5 E_REMOVE.C – Remove and cleanup user structures

```
#include <detect.h>

/*-----------------------------------------------------------------*
 *    Function: BOOL E_remove(detector) - Remove an event detector  *
 *-----------------------------------------------------------------*
 *    Arguments:      struct detect_info *detector - calling info   *
 *-----------------------------------------------------------------*
 *    Returns:        Returns TRUE if job completed sucessfully     *
 *-----------------------------------------------------------------*
 *    This routine is called when one of the detectors is no longer *
 *    needed or is to be changed by calling E_create.  It frees the *
 *    memory the detector was using and ensures that the detector   *
 *    won't be called again unless it is created anew.              *
 *-----------------------------------------------------------------*/

BOOL E_remove(detector)
struct detect_info *detector;
{

        /* free input continuity area */
        if (detector->incontd!=NULL) mfree((BYTE *) detector->incontd);
        /* free lookback area */
        if (detector->lbkarr!=NULL) mfree((BYTE *) detector->lbkarr);

        detector->samrte=0;
        detector->drectyp=9999;         /* Illegal record type */
        detector->datapts=0;
        detector->lbksize=0;
        detector->detname=NULL;

        return(TRUE);

}
```

B.6 FFIRBB20.C – 20 SPS BB FIR filter

```
#include <detect.h>

/*-----------------------------------------------------------------*
 *    Function: BOOL FfirBB20(detector,points,lookback,data)       *
 *                                                - 20SPS BB fir   *
```

```
*-------------------------------------------------------------------*
*    Arguments:      struct detect_info *detector - context info    *
*                    WORD points - number of data points in the     *
*                                  the current record               *
*                    WORD lookback - amount of lookback required    *
*                                   for this filter                 *
*                    LONG *data - array that contains the seismic   *
*                                  samples to be processed by the   *
*                                  FIR filter                       *
*-------------------------------------------------------------------*
*    Returns:        Returns TRUE if job completed sucessfully      *
*                      (not used in this implementation)            *
*-------------------------------------------------------------------*
*    Lookback:       This filter has a lookback of 36               *
*-------------------------------------------------------------------*
*    This routine processes the current record, which has "points"  *
*    seismic samples, with an FIR filter.  The array "data"         *
*    contains these samples plus the last "lookback" values from    *
*    the previous record (the values from the previous record,      *
*    that are needed for the FIR filter, are stored in the 0th      *
*    through lookback-1 positions of the array).  Hence, upon       *
*    input, the first sample of the current record is stored at     *
*    index "lookback" of the array.                                 *
*                                                                   *
*    To conserve RAM, we use the same array (the array "data")      *
*    for both input to the filter and output of it:  Each time      *
*    we process a new sample, a slot is free above the lookback     *
*    area.  We fill this slot with the filtered data; hence,        *
*    when the record has been completely processed, the first       *
*    "points" slots of the array contain the filtered seismic       *
*    samples of the record.                                         *
*-------------------------------------------------------------------*/

BOOL FfirBB20(detector,point,lookback,data)
struct detect_info *detector;
WORD point,lookback;
LONG *data;
{

#define cSH_20 >> 5
#define filta(x) (- *(x-1) - *(x-2) - *(x-3) - *(x-4) - *(x-5) \
                  - *(x-6) - *(x-7) - *(x-8)) << 2
#define filtb(x) (- *(x-9) - *(x-10) - *(x-11) - *(x-12) + *(x-19) \
                  + *(x-20) + *(x-21)) << 1
#define filtc(x) (- *(x-13) - *(x-14) + *(x-22) + *(x-23) + *(x-24) \
```

```
                + *(x-25) + *(x-26))
#define filtd(x) (- *(x-15) + *(x-27) + *(x-28) + *(x-29)) >> 1
#define filte(x) (- *(x-16) + *(x-18) + *(x-30) + *(x-31) + *(x-32)) >> 2
#define filtf(x) ((*(x-33) + *(x-34))>>3) + (*(x-35)>>4) \
        + ((*(x-36) - *(x-17))>>5) + (*x<<5)

        LONG *ldp,result;
        WORD i;

        ldp= &data[lookback];
          /* the address of the first sample of the current record */

        for (i=0; i<point; i++) {
          /*process all of the data of the current record */

                result=filta(ldp);/* implemented on different lines because
                                     some compilers don't like long macros */
                result+=filtb(ldp);
                result+=filtc(ldp);
                result+=filtd(ldp);
                result+=filte(ldp);
                result+=filtf(ldp);

                result=result cSH_20;
                ldp++;                  /* address of next sample */

                data[i]=result; /* store the result in the slot just above
                                   the lookback of the next sample */

        }

        return(TRUE);

}
```

B.7 FFIRSP10.C – 10 SPS SP FIR filter

```
#include <detect.h>

/*-------------------------------------------------------------------*
 *    Function: BOOL FfirSP10(det,points,lookback,data) - 10SPS SP fir*
 *-------------------------------------------------------------------*
 *    Arguments:      struct detect_info *det - detector info        *
 *                    WORD points - number of data points            *
```

```
 *                    WORD lookback - amount of lookback present      *
 *                    LONG *data - array of data to be processed      *
 *--------------------------------------------------------------------*
 *    Returns:        Returns TRUE if job completed sucessfully       *
 *--------------------------------------------------------------------*
 *    Lookback:       This filter has a lookback of 3                 *
 *--------------------------------------------------------------------*
 *    This routine processes the current record, which has "points"   *
 *    seismic samples, with an FIR filter.  The array "data"          *
 *    contains these samples plus the last "lookback" values from     *
 *    the previous record (the values from the previous record,       *
 *    that are needed for the FIR filter, are stored in the 0th       *
 *    through lookback-1 positions of the array).  Hence, upon        *
 *    input, the first sample of the current record is stored at      *
 *    index "lookback" of the array.                                  *
 *                                                                    *
 *    To conserve RAM, we use the same array (the array "data")       *
 *    for both input to the filter and output of it:  Each time       *
 *    we process a new sample, a slot is free above the lookback      *
 *    area.  We fill this slot with the filtered data; hence,         *
 *    when the record has been completely processed, the first        *
 *    "points" slots of the array contain the filtered seismic        *
 *    samples of the record.                                          *
 *--------------------------------------------------------------------*/

BOOL FfirSP10(detector,point,lookback,data)
struct detect_info *detector;
WORD point,lookback;
LONG *data;
{

#define aSH_10  >>3     /* shift for 10 samples/sec filter */
#define filta(x) (*x<<2) - (*(x-2)) - (*(x-3))

        LONG *ldp,result;
        WORD i;

        ldp= &data[lookback];

        for (i=0; i<point; i++) {

                result=filta(ldp);

                result=result aSH_10;
                ldp++;
```

```
            data[i]=result;
        }
        return(TRUE);
}
```

B.8 FFIRSP20.C – 20 SPS SP FIR filter

```
#include <detect.h>

/*-------------------------------------------------------------------*
 *    Function: BOOL FfirSP20(det,points,lookback,data) - 20SPS SP fir*
 *-------------------------------------------------------------------*
 *    Arguments:      struct detect_info *det - detector information *
 *                    WORD points - number of data points in the     *
 *                                  the current record               *
 *                    WORD lookback - amount of lookback required    *
 *                                  for this filter                  *
 *                    LONG *data - array that contains the seismic   *
 *                                 samples to be processed by the    *
 *                                 FIR filter                        *
 *-------------------------------------------------------------------*
 *    Returns:        Returns TRUE if job completed sucessfully      *
 *                      (not used in this implementation)            *
 *-------------------------------------------------------------------*
 *    Lookback:       This filter has a lookback of 7                *
 *-------------------------------------------------------------------*
 *    This routine processes the current record, which has "points"  *
 *    seismic samples, with an FIR filter.  The array "data"         *
 *    contains these samples plus the last "lookback" values from    *
 *    the previous record (the values from the previous record,      *
 *    that are needed for the FIR filter, are stored in the 0th      *
 *    through lookback-1 positions of the array).  Hence, upon       *
 *    input, the first sample of the current record is stored at     *
 *    index "lookback" of the array.                                 *
 *                                                                   *
 *    To conserve RAM, we use the same array (the array "data")      *
 *    for both input to the filter and output of it:  Each time      *
 *    we process a new sample, a slot is free above the lookback     *
 *    area.  We fill this slot with the filtered data; hence,        *
 *    when the record has been completely processed, the first       *
```

```
 *    "points" slots of the array contain the filtered seismic        *
 *    samples of the record.                                          *
 *--------------------------------------------------------------------*/

BOOL FfirSP20(detector,point,lookback,data)
struct detect_info *detector;
WORD point,lookback;
LONG *data;
{

#define aSH_20 >>3     /* shift for 20 samples/sec filter */
#define filta(x) (*x<<1) + (*(x-1)<<2) + (*(x-2)<<1) \
                 - (*(x-5)) -(*(x-6)<<1) -(*(x-7))

        LONG *ldp,result;
        WORD i;

        ldp= &data[lookback];  /* the address of the first sample of the
                                  current record */

        for (i=0; i<point; i++) {  /*process all of the data of the
                                  current record */

                result=filta(ldp); /* implemented on different lines because
                                  some compilers don't like long macros
*/

                result=result aSH_20;
                ldp++;                  /* address of next sample */

                data[i]=result; /* store the result in the slot just above
                                  the lookback of the next sample */

        }

        return(TRUE);

}
```

B.9 FFIRSP40.C – 40 SPS SP FIR filter

```
#include <detect.h>

/*--------------------------------------------------------------------*
```

```
 *    Function: BOOL FfirSP40(det,points,lookback,data) - 40SPS SP fir*
 *-------------------------------------------------------------------*
 *    Arguments:      struct detect_info *det - detector info        *
 *                    WORD points - number of data points            *
 *                    WORD lookback - amount of lookback present     *
 *                    LONG *data - array of data to be processed     *
 *-------------------------------------------------------------------*
 *    Returns:        Returns TRUE if job completed sucessfully      *
 *-------------------------------------------------------------------*
 *    Lookback:       This filter has a lookback of 16               *
 *-------------------------------------------------------------------*
 *    This routine processes the current record, which has "points"  *
 *    seismic samples, with an FIR filter.  The array "data"         *
 *    contains these samples plus the last "lookback" values from    *
 *    the previous record (the values from the previous record,       *
 *    that are needed for the FIR filter, are stored in the 0th      *
 *    through lookback-1 positions of the array).  Hence, upon       *
 *    input, the first sample of the current record is stored at     *
 *    index "lookback" of the array.                                 *
 *                                                                   *
 *    To conserve RAM, we use the same array (the array "data")      *
 *    for both input to the filter and output of it:  Each time      *
 *    we process a new sample, a slot is free above the lookback     *
 *    area.  We fill this slot with the filtered data; hence,        *
 *    when the record has been completely processed, the first       *
 *    "points" slots of the array contain the filtered seismic       *
 *    samples of the record.                                         *
 *-------------------------------------------------------------------*/

BOOL FfirSP40(detector,point,lookback,data)
struct detect_info *detector;
WORD point,lookback;
LONG *data;
{

#define bSH_40 >>4
#define filta(x) (- *(x-10) - *(x-12) - *(x-14) - *(x-16))
#define filtb(x) (*x + *(x-2) + *(x-4) + *(x-6) - *(x-11) - *(x-12) \
        - *(x-14) - *(x-15)) << 1
#define filtc(x) (*(x-1) + *(x-2) + *(x-4) + *(x-5) - *(x-13)) << 2
#define filtd(x) *(x-3) << 3

        LONG *ldp,result;
        WORD i;
```

```
        ldp= &data[lookback];

        for (i=0; i<point; i++) {

                result=filta(ldp);
                result+=filtb(ldp);
                result+=filtc(ldp);
                result+=filtd(ldp);

                result=result bSH_40;
                ldp++;

                data[i]=result;

        }

        return(TRUE);

}
```

B.10 FFPAV.C – 4 point running average filter

```
#include <detect.h>

/*-------------------------------------------------------------------*
 *    Function: BOOL Ffpav(det,points,lookback,data) - four point    *
 *                                    running average filter         *
 *-------------------------------------------------------------------*
 *    Arguments:      struct detect_info *det - detector context     *
 *                    WORD points - number of data points            *
 *                    WORD lookback - amount of lookback present     *
 *                    LONG *data - data array to be processed        *
 *-------------------------------------------------------------------*
 *    Returns:        Returns TRUE if job completed sucessfully      *
 *-------------------------------------------------------------------*
 *    Lookback:       This filter has lookback of 3                  *
 *-------------------------------------------------------------------*
 *    This routine processes the current record, which has "points"  *
 *    seismic samples, with an FIR filter.  The array "data"         *
 *    contains these samples plus the last "lookback" values from    *
 *    the previous record (the values from the previous record,      *
 *    that are needed for the FIR filter, are stored in the 0th      *
 *    through lookback-1 positions of the array).  Hence, upon       *
 *    input, the first sample of the current record is stored at     *
```

```
 *    index "lookback" of the array.                                    *
 *                                                                      *
 *    To conserve RAM, we use the same array (the array "data")         *
 *    for both input to the filter and output of it:  Each time         *
 *    we process a new sample, a slot is free above the lookback        *
 *    area.  We fill this slot with the filtered data; hence,           *
 *    when the record has been completely processed, the first          *
 *    "points" slots of the array contain the filtered seismic          *
 *    samples of the record.                                            *
 *----------------------------------------------------------------------*/

BOOL Ffpav(detector,point,lookback,data)
struct detect_info *detector;
WORD point,lookback;
LONG *data;
{

#define SHFT_DEF >>2    /* Shift for default filter */
#define fil_def(x) (*x) + (*(x-1)) + (*(x-2)) + (*(x-3))

        LONG *ldp,result;
        WORD i;

        ldp= &data[lookback];

        for (i=0; i<point; i++) {

                result=fil_def(ldp);

                result=result SHFT_DEF;
                ldp++;

                data[i]=result;

        }

        return(TRUE);

}
```

B.11 FNULL.C – Dummy filter

```
#include <detect.h>
```

```
/*-------------------------------------------------------------------*
 *    Function: BOOL Fnull(det,points,lookback,data) - Null filter   *
 *-------------------------------------------------------------------*
 *    Arguments:      struct detect_info *det - detector context     *
 *                    WORD points - number of data points            *
 *                    WORD lookback - amount of lookback present     *
 *                    LONG *data - array of data to be processed     *
 *-------------------------------------------------------------------*
 *    Returns:        Returns TRUE if job completed sucessfully      *
 *-------------------------------------------------------------------*
 *    Lookback:       This Filter has a lookback of 0                *
 *-------------------------------------------------------------------*
 *    This routine processes the current record, which has "points"  *
 *    seismic samples, with a dummy FIR filter.  The array "data"    *
 *    contains these samples plus the last "lookback" values from    *
 *    the previous record (the values from the previous record,      *
 *    that are needed for the FIR filter, are stored in the 0th      *
 *    through lookback-1 positions of the array).  Hence, upon       *
 *    input, the first sample of the current record is stored at     *
 *    index "lookback" of the array.                                 *
 *                                                                   *
 *    To conserve RAM, we use the same array (the array "data")      *
 *    for both input to the filter and output of it:  Each time      *
 *    we process a new sample, a slot is free above the lookback     *
 *    area.  We fill this slot with the filtered data; hence,        *
 *    when the record has been completely processed, the first       *
 *    "points" slots of the array contain the filtered seismic       *
 *    samples of the record.                                         *
 *-------------------------------------------------------------------*
 *    This routine will be used if no filter has been specified to   *
 *    E_Create().                                                    *
 *-------------------------------------------------------------------*/

BOOL Fnull(detector,point,lookback,data)
struct detect_info *detector;
WORD point,lookback;
LONG *data;
{

        LONG *ldp,result;
        WORD i;

        ldp= &data[lookback];

        for (i=0; i<point; i++) {
```

```
                result= *ldp;
                ldp++;

                data[i]=result;

        }

        return(TRUE);

}
```

B.12 DISP_PAR.C – Display event parameters

```
#include <detect.h>

/*-------------------------------------------------------------------*
 *    Function: VOID Disp_par(detector) - display detector parameters *
 *-------------------------------------------------------------------*
 *    Arguments:      struct detect_info *detector - call information *
 *-------------------------------------------------------------------*
 *    Returns:        Nothing (VOID)                                 *
 *-------------------------------------------------------------------*
 *    Fatal Errors:   This routine does not have fatal exits         *
 *-------------------------------------------------------------------*
 *    Called by user programs to display the currently set detector  *
 *    parameters.                                                    *
 *-------------------------------------------------------------------*/

VOID Disp_par(detector)
struct detect_info *detector;
{
        register struct con_sto *continu;
        continu=detector->incontd;

        printf("Parameters of detector %s:\n",continu->ch_name);
        printf(" filhi=%d; fillo=%d; iwin=%d; n_hits=%d;\n",
                continu->filhi,continu->fillo,continu->iwin,continu->n_hits);
        printf(" xth1=%o; xth2=%o; xth3=%o; xthx=%o;\n",
                continu->xth1,continu->xth2,continu->xth3,continu->xthx);
        printf(" def_tc=%ld; wait_blk=%d; val_avg=%d\n\n",
                continu->def_tc,continu->wait_blk,
                                continu->val_avg);
        return;
```

```
}
```

Appendix C

Example Driver

Here is the code of an example driver to detect various data streams on an input data tape. This may be adapted for custom situations.

```
#include <detect.h>

/*-------------------------------------------------------------------*
 *    Function: VOID main() - main section for detecting data streams *
 *-------------------------------------------------------------------*
 *    Arguments:      Arguments are taken from STDIN                 *
 *-------------------------------------------------------------------*
 *    Returns:        Nothing (VOID)                                 *
 *-------------------------------------------------------------------*
 *    Fatal Errors:   If any mallocs fail, or user types an          *
 *                    interrupt, the program will exit abnormally    *
 *-------------------------------------------------------------------*
 *    This is an example of how to define a main or other subroutine *
 *    to setup and call the event detector.  This particular example *
 *    defines an off-line detector which will read through as many   *
 *    different input devices as required and will do event          *
 *    detection on as many different channels, as desired. It is     *
 *    even possible to run multiple detections on the same channel,  *
 *    so as to test different parameters.  Processing will continue  *
 *    until there is an end of file on all input devices.            *
 *                                                                   *
 *    The information used to set up the various event detectors     *
 *    is entered in STDIN as a parameter file, here is an example:   *
 *                                                                   *
 *    /dev/rmtb bb_z 1 0 12 4 40 200 77 17 10 30 5 500 1014 16       *
 *    /dev/rmtb sp_z 2 0 12 8 80 160 20 15 10 15 4 500 0507 8        *
 *                                                                   *
 *    The parameters are in the following order separated by spaces: *
 *                                                                   *
 *    input_device detector_name rec_type component block_factor     *
 *    filhi fillo iwin xth1 xth2 xth3 xthx n_hits def_tc wait_blk    *
 *    val_avg                                                        *
```

```
 *-----------------------------------------------------------------*/

#define MAXDET 12              /* Maximum number of event detected channels */
#define DBF_SIZ 2048

struct detstr {
        TEXT detnam[20];          /* Place to store detector name */
        TEXT devnam[30];          /* Device for this detector */
        WORD dev_nbr;            /* Number in device array for this detector */
        WORD recotyp;            /* Record type */
        WORD compon;             /* Component to use */
        WORD block_f;            /* Blocking factor */
        LONG s_filhi, s_fillo, s_iwin, s_xth1, s_xth2, s_xth3, s_xthx,
        s_n_hits, s_def_tc, s_wait_blk, s_val_av;
                                 /* User's detector parameters */
}               dtectr[MAXDET]; /* One for each detector */

struct detect_info  edete[MAXDET];/* Detector setup data */

struct devlst {
        TEXT lstnam[30];          /* The name of the device */
        WORD filnum;             /* The file number where it is open()'d */
        WORD d_block;            /* The size of the block */
        WORD b_fact;             /* The blocking factor */
}               dvice[MAXDET];  /* One per device */

UBYTE * in_rec;                 /* Our read in buffer */
LONG * mainarray;              /* Main data array */

VOID main() {

        WORD det_nbr, dv_nbr, i, k, maxb;
        WORD det_no, dev_no;
        WORD ikount;
        BOOL any, ev, (*filt)(), FfirBB20(), FfirSP40(), Ffpav();

                                                        /****Note 1****/
/*---------------Let interupts get us out under UNIX-------------------*/

#ifdef UNIX                     /* Only meaningful on UNIX */
        extern  onintr();
        if (signal(SIGINT, SIG_IGN) != SIG_IGN)
            signal(SIGINT, onintr);
#endif
```

```
/*--------------Initialize variables for loops-------------------------*/

        ikount = 0;                     /* counts number of devices closed, for
                                           normal exit */
        maxb = 0;                       /* the maximum blocking factor used
                                           herein */
        det_nbr = 0;                    /* number of the detector, incremented
                                           in while loop */
        dv_nbr = (-1);                  /* number of the device, incremented in
                                           while loop */

        mainarray = E_buffer(338, 40);/* Max lookback 40 */

                                                        /*****Note 2*****/
/*---------Loop through to get each of our detector parameter lines------*/

        while (!feof(stdin)) {

            i = scanf("%s %s %d %d %d %d %d %d %o %o %o %o %d %d %d %d\n",
                    dtectr[det_nbr].devnam,
                    dtectr[det_nbr].detnam,
                    &dtectr[det_nbr].recotyp,
                    &dtectr[det_nbr].compon,
                    &dtectr[det_nbr].block_f,
                    &dtectr[det_nbr].s_filhi,
                    &dtectr[det_nbr].s_fillo,
                    &dtectr[det_nbr].s_iwin,
                    &dtectr[det_nbr].s_xth1,
                    &dtectr[det_nbr].s_xth2,
                    &dtectr[det_nbr].s_xth3,
                    &dtectr[det_nbr].s_xthx,
                    &dtectr[det_nbr].s_n_hits,
                    &dtectr[det_nbr].s_def_tc,
                    &dtectr[det_nbr].s_wait_blk,
                    &dtectr[det_nbr].s_val_av);

#ifdef DISP_P
            printf("Detector #%d Device %s Channel %s #%d (rec %d) ", det_nbr,
                    dtectr[det_nbr].devnam, dtectr[det_nbr].detnam,
                dtectr[det_nbr].compon, dtectr[det_nbr].recotyp);
            printf("Blocking Factor %d\n",
                    dtectr[det_nbr].block_f);
            printf("filhi=%d,fillo=%d,iwin=%d,xth1=%o,xth2=%o,xth3=%o,\
               xthx=%o,n_hits=%d,\ndef_tc=%dms,\ wait_blk=%d,val_avg=%d\n",
                    dtectr[det_nbr].s_filhi, dtectr[det_nbr].s_fillo,
```

```
                dtectr[det_nbr].s_iwin, dtectr[det_nbr].s_xth1,
            dtectr[det_nbr].s_xth2,
                dtectr[det_nbr].s_xth3, dtectr[det_nbr].s_xthx,
                dtectr[det_nbr].s_n_hits, dtectr[det_nbr].s_def_tc,
                dtectr[det_nbr].s_wait_blk, dtectr[det_nbr].s_val_av);
#endif

                                                        /******Note 3*****/
/*-----------   Calculate maxb for malloc ------------------------------- */

            if (dtectr[det_nbr].block_f > maxb)
                maxb = dtectr[det_nbr].block_f;

/*--------------More than one detector per device is permitted--------------*/

            for (i = 0; i <= dv_nbr; i++)
                if (!strcmp(dvice[i].lstnam, dtectr[det_nbr].devnam)) {

/*--------------New detector, same device----------------------------------*/

                    i = (-1);
                    break;
                }

            if (i != -1) {

                                                        /****Note 4****/
/*-------------New device-------------------------------------------------*/

                strcpy(dvice[++dv_nbr].lstnam, dtectr[det_nbr].devnam);
                dvice[dv_nbr].filnum = (-1);
                if ((dvice[dv_nbr].filnum = open(dvice[dv_nbr].lstnam,
                                000)) >= 0) {
                    printf("Device %s opened\n",
                            dvice[dv_nbr].lstnam);
                }

                else
                    printf("Device %s cannot be opened\n",
                            dvice[dv_nbr].lstnam);

            }

            if (dvice[dv_nbr].filnum >= 0) {
```

```
                                                        /****Note 5****/
/*--------Put user's parameters into the continuity structure------------*/

            switch (dtectr[det_nbr].recotyp) {
                case 1:
                    k = 18;
                    filt = FfirSP40;
                    break;
                case 2:
                    k = 38;
                    filt = FfirBB20;
                    break;
                default:
                    k = 3;
                    filt = Ffpav;
            }

            E_create(&edete[det_nbr], k, dtectr[det_nbr].recotyp,
                    dtectr[det_nbr].detnam, mainarray, filt);

            setallparams(&edete[det_nbr],
                    dtectr[det_nbr].s_filhi,
                    dtectr[det_nbr].s_fillo,
                    dtectr[det_nbr].s_iwin,
                    dtectr[det_nbr].s_n_hits,
                    dtectr[det_nbr].s_xth1,
                    dtectr[det_nbr].s_xth2,
                    dtectr[det_nbr].s_xth3,
                    dtectr[det_nbr].s_xthx,
                    dtectr[det_nbr].s_def_tc,
                    dtectr[det_nbr].s_wait_blk,
                    dtectr[det_nbr].s_val_av);

            Disp_par(&edete[det_nbr]);

            dvice[dv_nbr].d_block =
                dtectr[det_nbr].block_f * DBF_SIZ;
            dvice[dv_nbr].b_fact = dtectr[det_nbr].block_f;
            dtectr[det_nbr].dev_nbr = dv_nbr;

        }

        det_nbr++;

    }
```

```
        if (!(in_rec = malloc((UWORD) maxb * DBF_SIZ))) {
            printf("failure on malloc of rec_str\n");
            exit(1);
        }

        printf("Detectors created\n");

        any = FALSE;

                                                        /****Note 6****/
/*---------------------Main loop - process input data--------------------*/
        do {
            for (dev_no = 0; dev_no <= dv_nbr; dev_no++) {
            /* check all devices */
                if (dvice[dev_no].filnum >= 0) {
                    any = TRUE;
                    if (k = (read(dvice[dev_no].filnum, (UBYTE *) in_rec,
                                dvice[dev_no].d_block)) !=
                                    dvice[dev_no].d_block) {
                        dvice[dev_no].filnum = (-1);
                                    /* error condition */
                        if (k == 0)
                            printf("Drive %s at EOF\n",
                                    dvice[dev_no].lstnam);
                        if (k < 0)
                            printf("Drive %s got error %d\n",
                                    dvice[dev_no].lstnam, k);
                        if (k > 0)
                            printf("Drive %s read short record (%d)\n",
                                    dvice[dev_no].lstnam, k);
                        close(dvice[dev_no].filnum);
                        dtectr[det_nbr].dev_nbr = -1;
                        if (ikount++ == dv_nbr)
                            exit(0);
                        else
                            continue;       /* try another device */
                    }

                                                        /****Note 7****/
/*---------------------Process and detect each data----------------------*/

                    for (det_no = 0; det_no < det_nbr; det_no++) {

                        if (dtectr[det_no].dev_nbr == dev_no)
```

```
                        for (k = 0; k < dvice[dev_no].b_fact; k++) {
                            if (E_cdsnload(&edete[det_no],
                                        (UBYTE *) in_rec + (k * DBF_SIZ),
                                        dtectr[det_no].compon)) {

                                E_filter(&edete[det_no]);
                                E_detect(&edete[det_no]);
                            }
                        }
                    }
                }
            }
        } while (any == TRUE);        /* True while valid file number exists
*/
}

#ifdef UNIX
onintr() {                      /* the sole purpose of this routine is to indicate
                                   that an interrupt has occurred.  It is called
                                   by the UNIX utility signal  in the routine
                                   Detmain */
        printf("processing interrupt\n");
        exit(1);
}
#endif

#ifdef SHCOMMENT
```

Program Notes (see inline code)

1. On UNIX, the program ocassionally had difficulty being interrupted and closing down correctly. This signal was inserted to assure that this problem would not occur. It might not be necessary on other versions of UNIX (not BSD 2.9), or on other operating systems. In any case, an interrupt here forces an exit(), which is useful when doing a profile.

2. The event detector information is read from standard in with scanf(). No attempts are made to assure data accuracy or format. Please keep your input data accurate and always provide the correct number of parameters.

3. The maximum buffer size is computed. This will be the size of the largest input block.

4. The devices requested in the input parameters are compared which the current table. Device names which are not identical to the names in the table are declared new devices, and are placed in the tables. Do not use alias device names for the same

device, and be sure that the names are spelled the same, and are in the same case, for the same device.

Here, an attempt is made to open the device, however if it cannot be opened, the detectors for this device are automatically eliminated.

5. The filtering and lookback information are determined based on the record type, and then the detector structure is built via E_create(). The event detector parameters are then stowed away in the detector's continuity structure with setallparams() (a macro).

6. All parameters are loaded, and all devices are opened. This is the main processing loop. Each device is read round-robin fashion, and one device block is read on each device. This means that large blocked devices will process more records than smaller ones. Take this into account when planning the data flow through the detector.

7. The above block is deblocked. All of the event detectors which use this device are checked against each block. If the data are pertinent, the data are filtered, and the event detector is run. Event onset information is captured via standard out. If the user wishes more complex onset information processing, the user's onset routine can be pointed to by the continuity structure (see Onset()).

Explanation of Variables

dtectr – An array of structures whose purpose is to keep track of the input/output (I/O) details of each detector. These data are stored when the configuration/parameters data are read in.

edete – An array of detector structures which will contain all of the actual logistical information required by each detector. This is originally set up by e_create, and updated and refered to by all of the e_* routines called by the event detector. It contains the continutity structure for each detector, and contains the lookback data be used for filtering. It also contains pointers to the data storage arrays plus information which is used by the loader/decoder to ensure that the detector will process the proper data.

dvice – An array of structures which contain information for each discrete I/O device which is read to get data for the detector. Housekeeping information for general device I/O is kept here.

det_nbr – This is the current count of detectors in the dtectr and edete arrays. It is incremented and used as an index as information is loaded, and is used as an index maximum in the read loops.

dv_nbr – This is the current count of devices, and it is also the index to the dvice structure array. It is used similarly to det_nbr. The use of dv_nbr is similar to that of det_nbr.

i, k – These are temporary variables for looping and indexing.

maxb – This is used to contain the largest buffer size requested in the input parameters, so that an input buffer of sufficient size can be malloc()'d.

det_no, dev_no – variables corresponding to the detector and device arrays, which are used as indexes during the read loops.

ikount – This contains the current count of devices. It is set to the number of devices that are to be opened and it is decremented as devices reach end of file. When ikont is zero, all data has been processed, and the program will terminate.

any – A boolean flag.

filt – A pointer to a filter function. The pointer for the filter function is passed on to E_Create(), and the pointer is stored in the detector structure for that particular detector.

mainarray – An array of LONG (32-bit) values that is used for storage of the decoded input data, and it is also used for output of the filtered data. This multiple use of buffers is efficient in terms of time and memory.

If one does not desire filtering, be sure to specify a lookback of 0 when the filter is set up. If one wishes a lookback to be stored for the user's own filter, specify Fnull() to E_create().

in_rec – This is the buffer which is maxb chars long. The buffer is used to load the raw unconverted data from the various input devices. The decoder then processes this data.

```
#endif
```

```
********************** psprint 1.1 **********************
USER     : SCOTT
ACCOUNT  : ASL-BFEC
JOB      : MAIN
NODE     : ASLE
FILE     : ASL$MANAGERS:[SCOTT.DOCS.MURDOCK]MAIN.DVI;42
FORMAT   : DVI
QUEUED   :  5-JUN-1991 11:06:54.84
PAGES    : 136
NOTE     :
```