Radops 2000 Support Library Reference Manual

R.J.Barnes

Index

Introduction

Introduction

This book contains a brief introduction to writing Radar control programs for the Radops2000 control software.

It also documents the functions available in the support library.

Writing Control Programs

Writing Control Programs

Radar control programs are written in the C programming language. A working but not extensive knowledge of C is required to write them. However, it is advisable to read a good introductory guide to C programming.

The Libraries

Two sets of C libraries are used in compiling control programs, `control.lib` and `support.lib`.

The first, `control.lib`, located in the directory "`/radops/lib`", contains all the low level functions required to communicate with the other Radar tasks. Using this library a program has complete control over the basic operation of the Radar, however much greater knowledge of the data structures and inner workings of the other Radar tasks is required to use them.

The second, `support.lib`, located in the directory "`/radops/usr/lib`", contains the high level functions, including the routines for performing a clear frequency search and a single integration. It also includes a number of functions that are designed to make writing a control program easier.

The library also includes a number of pre-defined variables that can be used to set the Radar operating parameters. The author of a Control Program can use these variables instead of having to implicitly set the values in the raw data block.

Almost all Radar control programs will require functions in both `control.lib` and `support.lib`. Full documentation of all the functions in the control library are given in the **Radops 2000 Reference Manual**. The support library is documented in the following chapters.

The Data Structures

Control programs use two C structures for manipulating Radar data. One is used for the raw data block produced by the normal integration algorithm, the other is used by `fitacf` for fitted data.

A structure in C is an abstract data type. Just as a variable of type `int` holds data, an integer number, a variable whose type is a structure also holds data. The difference is that a structure can hold many separate pieces of information, in the case of the Radar software, all of the data and parameters used by the Radar.

The raw data block is described by a structure called `rawdata`. To reserve memory to store a raw data block a variable is declared :

```
struct rawdata raw_block;
```

This will create a structure called *raw_block* which will hold the raw data. It is possible to write control programs without knowing anything about this structure and to treat it as just another simple variable.

For those who wish to access the parameters and data stored in the raw data block, the structure has the following members:

`struct radops_parms PARMS;`	radar parameter block.
`short int PULSE_PATTERN[PULSE_PAT_LEN];`	transmitted pulse pattern.
`short int LAG_TABLE[2][LAG_TAB_LEN];`	lag table.
`char combf[COMBF_SIZE];`	comment buffer.
`long pwr0[MAX_RANGE];`	lag-0 power.
`long acfd[MAX_RANGE][LAG_TAB_LEN][2];`	calculated raw ACF.
`long xcfd[MAX_RANGE][LAG_TAB_LEN][2];`	calculated raw XCF.

The values PULSE_PAT_LEN, LAG_TAB_LEN, COMBF_SIZE and MAX_RANGE correspond to:

PULSE_PAT_LEN	16
LAG_TAB_LEN	48
COMB_SIZE	80
MAX_RANGE	75

The structure `radops_parms` contains the Radar parameters, it has the following members.

`char MAJOR,MINOR;`	revision numbers.
`short int NPARM;`	total number of 16 bit words in the block.
`short int ST_ID;`	station ID.
`short int YEAR;`	year = 19XX
`short int MONTH;`	month.
`short int DAY;`	day.
`short int HOUR;`	hour.
`short int MINUT;`	minute.
`short int SEC;`	second.
`short int TXPOW;`	transmitted power (kW).
`short int NAVE;`	number of times pulse was transmitted.
`short int ATTEN;`	attenuation setting of receiver.
`short int LAGFR;`	the lag to the first range (microsecs.).
`short int SMSEP;`	the sample separation (microsecs.).
`short int ERCOD;`	error flag.
`short int AGC_STAT;`	AGC status word.
`short int LOPWR_STAT;`	low power status word.
`short int NBAUD;`	number of elements in a pulse code.
`long int NOISE;`	noise level.
`long int radops_sys_resL;`	reserved for future use.
`short int radops_sys_resS;`	reserved for future use.
`short int RXRISE;`	receiver rise time.
`short int INTT;`	integration period (secs.).
`short int TXPL;`	the pulse length (microsecs.).
`short int MPINC;`	the basic lag separation (microsecs.).

`short int MPPUL;`	the number of pulses in the pulse pattern.
`short int MPLGS;`	the number of lags in the pulse pattern.
`short int NRANG;`	the number of range gates.
`short int FRANG;`	distance to the first range (km.).
`short int RSEP;`	range separation (km.).
`short int BMNUM;`	beam number.
`short int XCF;`	cross-correlation flag.
`short int TFREQ;`	transmitted frequency (kHz).
`short int SCAN;`	scan mode flag.
`long int MXPWR;`	maximum power allowed.
`long int LVMAX;`	maximum noise allowed.
`long int usr_resL1;`	user defined long word 1.
`long int usr_resL2;`	user defined long word 2.
`short int CP;`	Program ID.
`short int usr_resS1;`	user defined short word 1.
`short int usr_resS2;`	user defined short word 2.
`short int usr_resS3;`	user defined short word 3.

The user can set the first range gate by specifying `FRANG` in kilometers. The libraries then use this value to set the lag to the first range in microseconds.

Similarly the user sets the range separation by specifying `RSEP` in kilometers. The libraries then use this value to calculate `SMSEP` in microseconds.

During the gain setting routine, the libraries will attempt to add enough attenuation so that the maximum reflected power is less than `MXPWR`. If this is not possible the error code (`ERCOD`) is set to indicate the receiver is overloaded.

During the clear frequency search, the library routine will find the clearest frequency in the range specified. The noise level determined for that frequency will be stored in the parameter `NOISE`. If `NOISE` is greater then `LVMAX`, the error code will be set to indicate that no clear frequency could be found.

The fitted data block is described by a C structure called `fitdata`. This structure has the following members:

`struct radops_parms prms;`	radar parameter block.
`struct range_data rng[MAX_RANGE]`	the fitted data.

The structure `range_data` has the following members:

`short int qflg;`	the quality flag.
`short int gsct;`	the ground scatter flag.
`double p_0;`	the lag 0 power.
`double p_l;`	the lambda power.
`double p_s;`	the sigma power.
`double w_l;`	the lambda width.
`double w_s;`	the sigma width.
`double v;`	the velocity.
`double v_err;`	the velocity error.
`double sdev_l;`	the standard deviation of the lambda fit.
`double sdev_s;`	the standard deviation of the sigma fit.
`double sdev_phi;`	the standard deviation of the phase fit.

The data recorded in this structure is a subset of the data recorded in the output files and is **NOT** the same as the data used by the analysis tasks.

The library `support.lib` predefines two variables to store raw and fitted data:

```
struct rawdata raw_dt_blk;
struct fitdata fit_dt_blk;
```

When writing a Radar Control Program you can use these two variables as buffers for the two data blocks by referencing them in the code.

An Example Program

The following example program is a simplified version of `normal_scan`, which demonstrates many of the library functions supplied by `support.lib` and `control.lib`. Each section of source code will be followed by an explanation of what it does.

```
#define CP_ID 150
```

First the program ID number is defined. This number is recorded in all the raw data blocks produced by the control program and is consequently recorded in all the data files produced.

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
```

Next a number of standard C library headers are included.

```
#include "message.h"
#include "radops.h"
#include "fitdata.h"

#include "task_write.h"
#include "user_int.h"
#include "get_status.h"
#include "read_raw.h"
#include "log_error.h"
```

The next set of headers are for the control library. They define the structures `rawdata` and `fitdata` and define the prototypes for the functions in the control library.

```
#include "support.h"
#include "sync.h"
#include "summary_control.h"
#include "default.h"
```

The next set of headers define the prototypes for the functions in the support library and defines some of the default parameters for the Radar.

```
#define FITACF "/fitacf"
#define RAWWRITE "/raw_write"
#define ECHO_DATA "/echo_data"
```

Next the names of the other tasks that the control program will communicate with are defined.

```
char prg_name[32];
int f=0;
int frame_counter=0;
```

After that a number of global variables are defined. These are used to store the name of the control program and to keep track of the number of blocks of fitted data that have been received.

```
void main() {

  char errbuf[32];
  int exit_poll=0;
  short int start_freq,end_freq,freq_range;
  short int start_beam=START_BEAM,
            end_beam=END_BEAM,
            skip_beam;
  short int day_start_hr=DAY_START;
  short int night_start_hr=NIGHT_START;

  short int day_start_freq=DAY_FREQ;
  short int night_start_freq=NIGHT_FREQ;
  short int day_frang=DAY_FRANG;
  short int night_frang=NIGHT_FRANG;
  short int day_mpinc=DAY_MPINC;
  short int night_mpinc=NIGHT_MPINC;
  int day_night_flag;
  int status;

  int count=0,xcount=XCF;
```

This is the start of the main body of the program. A number of variables are defined for the different Radar operating parameters. Some of the Radar parameters are changed during the night so there are two copies of many of the variables, one for night time operation, and one for the day time.

The values for these variables are defined in the header “`default.h`”.

```
  short int ptab[7] = {0,9,12,20,22,26,27};
  short int lags[2][18] ={
 {0,26,20,9,22,22,20,20,12,0,12,9,0,9,12,12,9,9},
 {0,27,22,12,26,27,26,27,20,9,22,20,12,22,26,27,
  26,27}};
```

The next section of code defines the pulse sequence and lag table to use.

```
  log_error(NULL,"control",
            "Registering control program.");

  register_program(NULL,NULL);

  log_error(NULL,"control","Finding drivers.");

start_up(NULL,NULL,NULL);
```

This section of code establishes communication with the hardware drivers and registers the control program with the Operating System.

```
  log_error(NULL,"control",
            "Setting pulse table.");
  set_pulse(&raw_dt_buf,ptab,7);
  set_lag_table(&raw_dt_buf,lags,18);
```

Next the pulse sequence and lag table are set up in the raw data block..

```
  log_error(NULL,"control",
            "Closing any existing data files.");
  read_clock(&yr,&mon,&day,
            &hr,&min,&sec,&msec,&usec);
  task_close(RAWWRITE,yr,mon,day,hr,min,sec);
  task_close(FITACF,yr,mon,day,hr,min,sec);
  task_close(ECHO_DATA,yr,mon,day,hr,min,sec);
```

Before the program opens any new files, a check is performed to make sure that any existing open files are closed. This ensures that the data files produced contain data from only one control program simplifying the problems of analysis.

```
  log_error(NULL,"control",
            "Opening first data files.");

  task_open(RAWWRITE,NULL,yr,mon,day,hr,min,sec);

  task_open(FITACF,NULL,yr,mon,day,hr,min,sec);

  task_open(ECHO_DATA,NULL,
            yr,mon,day,hr,min,sec);
```

Now, new files can be opened to receive data from this program.

```
  intt=7;
  rsep = 45;
  txpl = (rsep*20)/3;
  mpinc = day_mpinc;
  frang = day_frang;
  nrang = NRANG;
  max_atten = MAX_ATTEN;
  prot_atten= PROT_ATTEN;
  rxnarrow=RXNARROW;
  rxwide=RXWIDE;
  rsep_switch=RSEP_SWITCH;

  strcpy(combf,"test_scan");

  start_freq=day_start_freq;
  freq_range=300;
  cp = CP_ID;


  strcpy(prg_name,"test_scan");
```

The next section of code sets some of the globally defined variables for the Radar operating parameters.

```
 log_error(NULL,"control","Entering main
           loop.");
  do {
    read_clock(&yr,&mon,&day,&hr,
               &min,&sec,&msec,&usec);
```

Now the main loop of the control program is entered and the current UTC time is read from the system clock into the globally defined time variables.

```
    if (test_hour(2) !=0) {
      log_error(NULL,"control",
               "Opening new data files.");
      task_close(RAWWRITE,yr,mon,day,
                hr,min,sec);
      task_close(FITACF,yr,mon,day,
                hr,min,sec);
      task_close(ECHO_DATA,yr,mon,day,
                hr,min,sec);
      task_open(RAWWRITE,NULL,yr,mon,day,
                hr,min,sec);
      task_open(FITACF,NULL,yr,mon,day,
                hr,min,sec);
      task_open(ECHO_DATA,NULL,yr,mon,day,
                hr,min,sec);
    }
```

Next, a check is made to see if new files should be opened, under normal circumstances new files are created every few hours. This reduces the risk of data loss if a section of the hard disk becomes corrupt.

```
    day_night_flag=day_or_night(day_start_hr,
                        night_start_hr);
    if (day_night_flag == NIGHT_FLAG) {
      log_error(NULL,"control","Night time.");
      start_freq=night_start_freq;
      frang = night_frang;
      mpinc = night_mpinc;
    } else {
      log_error(NULL,"control","Day time.");
      start_freq=day_start_freq;
      frang = day_frang;
      mpinc = day_mpinc;
    }
```

This section of code tests to see if the Radar is operating during the day or night and sets the radar operating parameters accordingly.

```
    if (xcount > 0) {
      ++count;
      if(count == xcount) {
        xcf = 1;
        count = 0;
      } else xcf= 0;
    } else xcf = 0;
```

The next section of code checks to see if a Cross-Correlation should be performed.

```
  scan=1;

    log_error(NULL,"control","Starting beam
            scan.");
    for(bmnum = start_beam; bmnum <= end_beam;
        ++bmnum) {
```

Now the integration loop is started. The Radar will scan across the beam numbers , integrating for the specified number of seconds along each beam.

```
       sprintf(errbuf,
               "Integrating beam:%d",bmnum);
       log_error(NULL,"control",errbuf);

       end_freq=start_freq+freq_range;

       set_block(&raw_dt_buf);

       set_time(&raw_dt_buf);
```

The `set_block` and `set_time` functions set up the raw data block with the current Radar operating parameters and the correct time.

```
      status=fclr(&raw_dt_buf,
                  start_freq,end_freq,5);
      if (status==0) status=radar(&raw_dt_buf);
      get_status(NULL,&raw_dt_buf,0);
```

The next section of code performs the clear frequency search and the full integration.

```
      task_write_raw(RAWWRITE,&raw_dt_buf,1);
      task_write_raw(FITACF,&raw_dt_buf,1);
      task_write_raw(ECHO_DATA,&raw_dt_buf,1);

      task_write_aux(ECHO_DATA,prg_name,
                     strlen(prg_name)+1);
```

Once the integration is complete the raw data is distributed to the other Radar tasks using the `task_write` functions. The call to `task_write_aux` passes the name of the control program to the display tasks.

```
      if ((f=get_fit(NULL,&fit_dt_buf))
           !=frame_counter) {
        sprintf(errbuf,
                "Received fit data block %d",f);
        log_error(NULL,"control",errbuf);

        task_write_fit(ECHO_DATA,&fit_dt_buf,1);

        frame_counter=f;
      } else log_error(NULL,"control",
                       "No fit data waiting.");
```

Next the last block of data processed by `fitacf` is retrieved from `fit_buffer`. Checks are made using the *frame_counter* variable to ensure that the correct block of data is read in.

```
      exit_poll=user_int(&raw_dt_buf,
      "start_beam i end_beam i day_start_freq i \
       night_start_freq i day_frang i \
       night_frang i day_mpinc i \
       night_mpinc i start_freq i end_freq i",
       &start_beam,&end_beam,
       &day_start_freq,&night_start_freq,
       &day_frang,&night_frang,
       &day_mpinc,&night_mpinc,
       &start_freq,&end_freq);
```

This section of code checks to see if the user wishes to change any of the Radar operating parameters. In addition if the scheduler wishes to stop the program the variable *exit_poll* is set to a non-zero value.

```
      set_vars(&raw_dt_buf);

      if (exit_poll !=0) break;

      scan=0;
    }
  } while (exit_poll==0);
```

Finally in the integration loop, the global paramaters are updated from the raw data block and checks are made to see if the control program should be terminated.

```
  log_error(NULL,"control",
           "Exiting control program.");

  if ((f=get_fit(NULL,&fit_dt_buf))
            !=frame_counter) {

    sprintf(errbuf,
           "Received fit data block %d",f);
    log_error(NULL,"control",errbuf);

    task_write_fit(ECHO_DATA,&fit_dt_buf,1);

    frame_counter=f;
  } else log_error(NULL,"control",
                   "No fit data waiting.");

  log_error(NULL,"control",
            "Closing data files.");

  task_close(RAWWRITE,yr,mon,day,hr,min,sec);
  task_close(FITACF,yr,mon,day,hr,min,sec);
  task_close(ECHO_DATA,yr,mon,day,hr,min,sec);
  exit(0);
}
```

The last section of code waits for the final block of data from `fitacf` before closing any open data files.

The Support Library

calc_skip

Syntax

```
#include  “support.h”
int calc_skip(int bnd);
```

Description

The `calc_skip` function determines how many beams should be skipped at the beginning of the current scan to ensure that the next scan will begin at the start of an interval of *bnd* minutes. The function uses the globally defined variables *yr*, *mon*, *day* ,*hr*, *min*, *sec*, and *msec*. These should be set to the current time using the function `read_clock`.

Returns

Returns the number of beams to skip at the start of a scan.

day_or_night

Syntax

```
#include  “support.h”
int day_or_night( int day_hr,
                  int night_hr);
```

Description

The `day_or_night` function tests to see whether the current time is during the day or the night. The function uses the globally defined variables *yr* ,*mon*, *day*, *hr*, *min*, *sec*, and *msec*. These should be set to the current time using the function `read_clock`. The start of day and night are defined by the arguments *day_hr* and *night_hr*.

Returns

Returns DAY_FLAG if the time is during the day, otherwise returns NIGHT_FLAG.

Syntax

```
#include  “support.h”
int fclr( struct rawdata *raw_data,
          short int start,
          short int end
          short int step);
```

Description

The `fclr` function will perform a clear frequency search starting at frequency *start* and ending at *end* in intervals of *step*. When completed the selected frequency is stored in the *PARMS.TFREQ* member of the `rawdata` structure pointed to by *raw_data*.

Any errors that occur during the frequency search are printed on the stream *stderr* and reported to the error log.

Returns

Returns zero (0) on success, or (-1) if an error occurs and raderr is set.

Errors

When an error occurs, *raderr* contains a value indicating the type of error that occurred.

RADERR_SIGNAL_FAIL	a time out signal could not be claimed. The signal *SIGUSR1* is generated after a time-out to interrupt the message.
RADERR_TIMER_FAIL	a timer could not be created. A timer is set to trigger a signal after the required time-out period.
RADERR_MSG_FAIL	a message was interrupted.
RADERR_TME_OUT	a message timed out.
RADERR_AD_FAIL	the task `a_d_drive` failed to complete a command.
RADERR_DIO_FAIL	the task `radops_dio` failed to complete a command.

get_summary

Syntax

```
#include  "summary_control.h"
int get_summary(  char *sd_name,
                  long int *beamA,
                  long int *beamB,
                  long int *power);
```

Description

The `get_summary` function will request from the task `sd_summary` the two beam numbers and the threshold power used in recording summary information. The values are stored in the variables pointed to be *beamA*, *beamB*, and *power*.

If the string *sd_name* is NULL, then the message will be sent to the task registered under the name "`/sd_summary`", otherwise the task will be searched for under the name *sd_name*.

Returns

Returns zero (0) on success, or (-1) if an error occurs and raderr is set.

Errors

When an error occurs, *raderr* contains a value indicating the type of error that occurred.

RADERR_NO_TASK	the task `sd_summary` could not be located.
RADERR_SIGNAL_FAIL	a time out signal could not be claimed. The signal *SIGUSR1* is generated after a time-out to interrupt the message.
RADERR_TIMER_FAIL	a timer could not be created. A timer is set to trigger a signal after the required time-out period.
RADERR_MSG_FAIL	a message was interrupted.
RADERR_TME_OUT	a message timed out.

init_proxy

Syntax

```
#include  "support.h"
void init_proxy();
```

Description

The `init_proxy` function creates a proxy attached to the University of Leicester Micro-controller monitor task. When this proxy is triggered, the status of the micro-controllers is read.

Returns

None.

Syntax

```
#include  "support.h"
int print_param(
             FILE *fp,
             struct  rawdata *raw_data);
```

Description

The `print_param` function will print a plain text table showing the current radar parameters to the file pointed to by *fp*. The parameters are taken from the `rawdata` structure pointed to by *raw_data*.

Returns

Returns zero (0) on success, or (-1) if an error occurred.

pulse_code

Syntax

```
#include  "support.h"
int pulse_code(
```

Description

The `radar` function will perform an integration over a single beam using the radar parameters stored in the raw data structure *raw_data*. When completed the structure contains the calculated ACF and XCF of the integrated data.

This version of the algorithm uses the phase-coding technique to improve the resolution of the observed scatter. It should only be used on a Radar with the appropriate hardware modifications for phase-coding.

Any errors or warnings that occur during the integration are printed on the stream *stderr*, and reported to the error log.

Returns

Returns zero (0) on success, or (-1) if an error occurs and raderr is set.

Errors

When an error occurs, *raderr* contains a value indicating the type of error that occurred.

RADERR_SIGNAL_FAIL	a time out signal could not be claimed. The signal *SIGUSR1* is generated after a time-out to interrupt the message.
RADERR_TIMER_FAIL	a timer could not be created. A timer is set to trigger a signal after the required time-out period.
RADERR_MSG_FAIL	a message was interrupted.
RADERR_TME_OUT	a message timed out.
RADERR_AD_FAIL	the task `a_d_drive` failed to complete a command.
RADERR_DIO_FAIL	the task `radops_dio` failed to complete a command.

put_summary

Syntax

```
#include  “summary_control.h”
int put_summary(  char sd_name,
                  long int beamA,
                  long int beamB,
                  long int power);
```

Description

The `put_summary` function will set the two beam numbers and threshold power used in recording summary information by the task `sd_summary` to the values *beamA*, *beamB*, and *power*.

If the string *sd_name* is NULL, then the message will be sent to the task registered under the name “`/sd_summary`”, otherwise the task will be searched for under the name *sd_name*.

Returns

Returns zero (0) on success, or (-1) if an error occurs and raderr is set.

Errors

When an error occurs, *raderr* contains a value indicating the type of error that occurred.

RADERR_NO_TASK	the task `sd_summary` could not be located.
RADERR_SIGNAL_FAIL	a time out signal could not be claimed. The signal *SIGUSR1* is generated after a time-out to interrupt the message.
RADERR_TIMER_FAIL	a timer could not be created. A timer is set to trigger a signal after the required time-out period.
RADERR_MSG_FAIL	a message was interrupted.
RADERR_TME_OUT	a message timed out.

Syntax

```
#include  "support.h"
int radar(   struct rawdata *raw_data);
```

Description

The `radar` function will perform an integration over a single beam using the radar parameters stored in the raw data structure *raw_data*. When completed the structure contains the calculated ACF and XCF of the integrated data.

Any errors or warnings that occur during the integration are printed on the stream *stderr*, and reported to the error log.

Returns

Returns zero (0) on success, or (-1) if an error occurs and raderr is set.

Errors

When an error occurs, *raderr* contains a value indicating the type of error that occurred.

RADERR_SIGNAL_FAIL	a time out signal could not be claimed. The signal *SIGUSR1* is generated after a time-out to interrupt the message.
RADERR_TIMER_FAIL	a timer could not be created. A timer is set to trigger a signal after the required time-out period.
RADERR_MSG_FAIL	a message was interrupted.
RADERR_TME_OUT	a message timed out.
RADERR_AD_FAIL	the task `a_d_drive` failed to complete a command.
RADERR_DIO_FAIL	the task `radops_dio` failed to complete a command.

read_uconts

Syntax

```
#include  "support.h"
void read_uconts(
```

Description

The `read_uconts` function triggers the proxy attached to the University of Leicester Micro-Controller monitor task causing the status of the controllers to be read.

Returns

None.

set_block

Syntax

```
#include  "support.h"
void set_block(struct raw_data *rawdata)
```

Description

The `set_block` function copies the globally defined variables that correspond to the radar parameters into the raw data structure pointed to by *raw_data*.

The companion of this function is `set_vars` which performs the reverse operation of copying the parameters in the `rawdata` structure into the globally defined variables.

The globally defined variables are:

`intt`	the integration period.
`txpl`	the pulse length.
`mpinc`	the lag separation in micro seconds.
`mppul`	the number of pulses in a pulse pattern.
`mplgs`	the number of lags in the lag table.
`nrang`	the number of range gates.
`frang`	the distance in kilometers to the first range gate.
`rsep`	the range separation in kilometers.
`bmnum`	the current beam number.
`xcf`	the cross correlation flag.
`tfreq`	the transmitted frequency.
`scan`	the scan mode.
`mxpwr`	the maximum power allowed.
`lvmax`	the maximum noise level allowed.
`cp`	the program id.
`usr_resS1`	user defined short variable 1.
`usr_resS2`	user defined short variable 2.
`usr_resS3`	user defined short variable 3.
`usr_resL1`	user defined long variable 1.
`usr_resL2`	user defined long variable 2.
`combf`	the comment buffer.

Returns

None.

set_lag_table

Syntax

```
#include  "support.h"
void set_lag_table(
          struct rawdata *raw_data,
          short int *lag_table,
          short int mplgs);
```

Description

The `set_lag_table` function will set the lag table in the raw data structure pointed to by *raw_data* to the lag table defined by the array *lag_table*. The table should be an array of 2 by *mplgs*:

```
short int lag_table[2][mplgs];
```

The *LAG_TABLE* and *PARMS.MPLGS* members of the `rawdata` structure are set.

Returns

None.

Syntax

```
#include  "support.h"
void set_pulse(struct rawdata *raw_data,
               short int *pattern,
               short int mmpul);
```

Description

The `set_pulse` function will set the pulse pattern in the raw data structure pointed to by *raw_data* to the pulse pattern defined by the array *pattern*. The pattern should have *mppul* elements.

The *PULSE_PATTERN* and *PARMS.MPPUL* members of the `rawdata` structure are set.

Returns

None.

set_time

Syntax

```
#include  "support.h"
int set_time(struct rawdata *raw_data);
```

Description

The `set_time` function will read the system clock and record the time in the raw data structure pointer to by *raw_data*.

Returns

Returns zero (0) on success, or (-1) if an error occurred.

set_vars

Syntax

```
#include  "support.h"
void set_vars(struct rawdata *raw_data);
```

Description

The `set_vars` function copies the radar parameters in the raw data structure pointed to by *raw_data* to the corresponding globally defined variables.

The companion of this function is `set_block` which performs the reverse operation of copying the globally defined variables into the `rawdata` structure.

The globally defined variables are:

`intt`	the integration period.
`txpl`	the pulse length.
`mpinc`	the lag separation in micro seconds.
`mppul`	the number of pulses in a pulse pattern.
`mplgs`	the number of lags in the lag table.
`nrang`	the number of range gates.
`frang`	the distance in kilometers to the first range gate.
`rsep`	the range separation in kilometers.
`bmnum`	the current beam number.
`xcf`	the cross correlation flag.
`tfreq`	the transmitted frequency.
`scan`	the scan mode.
`mxpwr`	the maximum power allowed.
`lvmax`	the maximum noise level allowed.
`cp`	the program id.
`usr_resS1`	user defined short variable 1.
`usr_resS2`	user defined short variable 2.
`usr_resS3`	user defined short variable 3.
`usr_resL1`	user defined long variable 1.
`usr_resL2`	user defined long variable 2.
`combf`	the comment buffer.

Returns

None.

Syntax

```
#include  "support.h"
int start_up(     char *ad_driver,
                  char *radops_name,
                  char *err_name);
```

Description

The `start_up` function sets up communication with the two driver tasks, `radops_dio` and `a_d_drive`. If either of these tasks cannot be found then the function will report an error and exit the program immediately.

The strings *ad_driver*, *radops_name*, and *err_name* can be used to specify the names of the two driver tasks and the error log to search for. If the strings are NULL then the default names of "`/a_d_drive`", "`/radops_dio`", and "`/errlog`" will be used.

Returns

Returns zero (0) on success, or (-1) if an error occurred.

test_hour

Syntax

```
#include  "sync.h"
int test_hour(hr_bnd);
```

Description

The `test_hour` function will checks to see if an interval of *hr_bnd* hours has elapsed. The function uses the globally defined variables *yr*, *mon*, *day* ,*hr*, *min*, *sec*, and *msec*. These should be set to the current time using the function `read_clock`.

Returns

Returns a non zero value if a two hour boundary has passed.

wait_boundary

Syntax

```
#include  "sync.h"
int wait_boundary(int sync);
```

Description

The `wait_boundary` function divides the day into intervals of *sync* minutes. It will wait until the boundary of one of these intervals has expired before returning.

The function uses the globally defined variables *yr*, *mon*, *day*, *hr*, *min*, *sec*, and *msec*. These should be set to the current time using the function `read_clock`.

Returns

None.