# Radops 2000 Reference Manual

R.J.Barnes

# Index

# Introduction

# Radops 2000

The **Rad**ar **Op**erating **S**ystem (**Radops**) is a suite of software that forms the control system for the HF Radars of the **Super D**ual **A**uroral **R**adar **N**etwork (**SuperDARN**). It is written under the **QNX** Operating System, a modern micro-kernel OS designed with particular emphasis on real-time embedded control applications.

The Radops software is divided into separate tasks that are each responsible for a different aspect of controlling the Radar. They communicate with each other using the QNX method of **I**nter-**P**rocess **C**ommunication (**IPC**), message passing. Messages can be passed between tasks running on different computers over a **L**ocal **A**rea **N**etwork (**LAN**), allowing the software to be distributed across a network of machines.

The Radar consists of an array of Antennas which are electronically steered into one of sixteen directions or beams.

The radar operates by transmitting a complicated pattern of pulses and then sampling the reflected echoes. The raw data is then processed by calculating the **A**uto-**C**orrelation **F**unction (**ACF**) at seventy five (75) range gates or bins.

Natural and artificial noise is a particular problem at the frequencies the Radar operates at, and can drown out the echoes from the pulse sequence. So in the normal mode of operation the software searches for a “quiet” frequency with a low noise level to use at which to transmit the pulses. If the received signal is too powerful, it can be electronically attenuated within the receiver.

The timing of the pulse sequence must be very accurate and consequently a separate computer is dedicated to controlling it. When the pulse sequences is transmitted, all interrupts are disabled on this timing computer to help achieve the required accuracy. The computer is linked to the main computer using an Ethernet LAN connection.

The overall control of the Radar is provided by a program called a *Radar Control Program*. This is a task supplied by the operator that controls the other tasks and determines the mode of operation of the Radar. A set of C libraries provide the interface between the control program and the other parts of the Radar software.

The radar tasks can be divided into several groups: driver tasks, primary tasks, display tasks, summary tasks. diagnostic and test programs, software version control utilities, off-line data processing tasks, and Internet access tasks. The two most important groups are the hardware drivers and the primary tasks.

## Drivers

The driver tasks communicate directly with the Radar hardware interface cards.

The task `a_d_drive` programs the Analogue to Digital Converter (ADC) card which samples the signal received by the Radar. It uses shared memory and Direct Memory Transfer (DMA) to achieve the high time resolution required.

The task `radops_dio` runs on the timing computer and controls the Digital Input/Output board which transmits the pulse sequence. The card is also responsible for selecting the operating frequency, beam number and attenuation level.

The Radar network is synchronized to the **G**lobal **P**ositioning **S**ystem (GPS) clock by using a GPS satellite receiver card. The system clock on the main computer is periodically synchronized to the GPS clock by the driver `gps_clock`.

The driver `gbuf` controls the video hardware on the main computer and is essential for running the many graphical displays that are part of the Radops software.

## Primary Tasks

The primary tasks are the main programs responsible for processing and storing the data generated by the Radar.

The task `errlog` logs and records errors and messages from the other parts of the software in a plain text file. A new log is created at the start of each day.

The task `schedule`, as its name implies determines which control program should be running and at what time. A text file containing the start times and command lines for programs to run is periodically loaded by the task. When the start time of a program has elapsed the current control program is terminated and the new one started.

The task `echo_data` distributes data from the control program to the other parts of the Radar software. Using `echo_data`, tasks can be started and stopped without interrupting the current operation of the Radar

The task `raw_write` stores the data generated by the Radar in files on the hard disk. It compresses the data using a simple algorithm to reduce the amount of disk space required.

The task `fitacf` processes the raw ACFs by attempting to fit them to a known distribution. This allows parameters like the back-scattered power and doppler velocity to be derived.

The task is very computationally intensive and processes one block of raw data while the next is being generated. The task `fit_buffer` temporarily stores the data from fitacf before returning it to the control program.

The final task `alter`, allows the user to inspect and change the operating parameters of the control program.

# Software Organization

The Radar software is organized within the "`/radops`" directory:

| | |
|---|---|
| `/radops/bin` | binaries of all of the main Radar tasks excluding control programs. |
| `/radops/errlogs` | error logs produced by `errlog`. |
| `/radops/include` | header files for the Radar control library and the tasks library. |
| `/radops/lib` | task and control libraries. |
| `/radops/scdlogs` | logs produced by the Scheduler. |
| `/radops/scripts` | shell scripts for making the Radar software and for starting the tasks. |
| `/radops/src` | source code for the Radar software. |
| `/radops/tables` | hardware tables for the Radar. |
| `/radops/usr` | source and binaries for the control programs and the support library. |

The sub-directories in "`/radops/usr`" include:

| | |
|---|---|
| `/radops/usr/bin` | binaries for the control programs. |
| `/radops/usr/include` | headers for the support library. |
| `/radops/usr/lib` | support library for the control programs. |
| `/radops/usr/src/` | source code for the control programs. |

# System Requirements

The software is designed to run on a **QNX** network comprising of two or more machines. One machine contains the DIO card and acts as the timing computer, a second machine containing the A/D card and the GPS clock interface card is the main computer and runs the majority of the Radar tasks.

The software has been tested on QNX version 4.22-4.24 and Watcom C version 9.52-10.6 The operating system should be running in 32 bit mode.

The Radar software is supplied as a tar archive on either on diskette or via FTP. The archive contains the source code for the software which must be compiled before it can be used. Copy the archive "`radops2000.release.`*x.yy*`.tar.F`" into the root directory of your main computer and type:

```
install -u radops2000.release.x.yy.tar.F
```

This will backup any existing Radar software into the directory "`/radops.old`" and create the new directory structure for the Radar software.

The install script will check and if necessary attempt to modify the default login profile to include the environment variables used by the Radar software by adding the following lines to the file "`/etc/profile`":

```
. /radops/scripts/rad_export
. /radops/scripts/rad_path
```

The file "`/radops/scripts/rad_export`" contains the environment variables that define where the Radar will store its data files. You may wish to alter this file for your particular system.

The install script will then attempt to update the header and data files that are specific for a particular Radar.

You will then be asked if you wish to compile the Radar software. If you type "`n`" the script will terminate at this point. If you type "`y`" the script "`/radops/scripts/make_radar`" will be executed.

A title page showing the version number of the software will be displayed. The script then checks which station the software is to be compiled for and prompts you to confirm that this is correct. Type "`y`" if the station name is correct or "`n`" to exit the script.

The script will now ask if you wish to install the pre-compiled binaries. Only a few of the radar tasks contain code that is unique for each site and the archive contains a set of binaries that can be quickly copied to the "`/radops/bin`" directory. If you have problems installing the pre-compiled binaries then type "`n`" and the entire set of software will be compiled from scratch.

Once the software has been compiled or installed you must copy the program `radops_dio` to the timing computer. Assuming that the software has been installed on node one (1) of your QNX network and the timing computer is node two (2), enter the following on the node one (1) machine:

```
mkdir //2/radops
mkdir //2/radops/bin
cp /radops/bin/radops_dio //2/radops/bin/
```

You should alter the `sysinit` file of the timing computer so that `radops_dio` is automatically loaded when the machine is re-booted.

If you need to recompile the software, or you chose not to compile it when you first installed the system, you can use the script "`make_radar`". Type the command:

```
/radops/scripts/make_radar
```

The script will display a title page showing the software version number. It will then check which station the Radar software is to be compiled for and prompt you to confirm that this is correct. Type "`y`" if the station is correct or "`n`" to exit the script.

If the station is not correct then you must modify the header file "`/radops/include/radops/radar_id.h`" for your particular Radar. Un-comment the lines of code appropriate for your station and re-run the "`make_radar`" script.

The Radar tasks must be started in a specific order. Assuming that `radops_dio` is already running on the timing computer, the script "`/radops/scripts/start_radar`" can be used to start the rest of the software:

```
#
# start the tasks
#

ontty /dev/con2 /radops/bin/errlog
ontty /dev/con3 /radops/bin/a_d_drive
ontty /dev/con3 /radops/bin/gps_clock
ontty /dev/con4 /radops/bin/echo_data
ontty /dec/con3 /radops/bin/fit_buffer
ontty /dev/con3 /radops/bin/fitacf
ontty /dev/con3 /radops/bin/rawwrite
```

There are other Radar tasks, such as `sd_summary` and `vlptm`, that can be added to this script depending on the way you operate your Radar.

Usually control programs are started by the scheduler so the last line of your "`start_radar`" script should run the Scheduler:

```
ontty /dev/con5 /radops/bin/schedule radops.sched
```

The Radar can be stopped by creating a "`stop_radar`" script that kills all the Radar tasks in turn:

```
#
# stop the tasks
#

slay -f schedule
slay -f rawwrite
slay -f fitacf
slay -f fit_buffer
slay -f echo_data
slay -f a_d_drive
slay -f gps_clock
slay errlog
```

Under the scheduler the current control program is a child process of `schedule`, so killing the scheduler will also terminate the control program.

Once the rest of the Radar software has been started then a control program can be run. The programs are located in the directory "`/radops/usr/bin`". Your PATH environment variable will already include this directory so you should just be able to type the name of the program to get it to run.

# Command Line Options

Many Radar Control Programs support optional parameters that can be passed in on the command line. These follow the normal UNIX convention for optional command parameters:

```
normal_scan -sb 3 -eb 12
```

This would force `normal_scan` to start a scan at beam 3 and end the scan on beam 12. The program `normal_scan` is the program the Radar normally runs for SuperDARN common mode time.

Normally you can find out about the command line options by using the `use` command. The usage message produced by `normal_scan` is:

```
normal_scan [-dt day_time] [-nt night_time]
            [-df day_start_freq]
            [-nf night_start_freq]
            [-dr day_frang]
            [-nr night_frang] [-xcf xcount]
            [-dm day_mpinc] [-nm night_mpinc]
            [-sb start_beam] [-eb end_beam]
            [-af start_freq] [-st scat_thr]
            [-ft frequency table file]
            [option file]
```

Normally control programs are not started by an operator. Instead a task called `schedule` is responsible for starting and stopping programs. It uses a script called a schedule file that specifies what time and date a program is due to start and what command line to use to execute the program:

```
# test.sched
default /radops/usr/bin/normal_scan
1996 3 19 14 50 /radops/usr/bin/sdcusp_2200
1996 3 19 15 00 /radops/usr/bin/normal_scan
1996 3 19 15 30 /radops/usr/bin/sdcusp_2200
1996 3 19 15 40 /radops/usr/bin/normal_scan
```

Each line in the schedule file corresponds to a command to execute. Lines beginning with a "#" are treated as comments and are ignored.

If a line begins with the word `default`, then the rest of the line is treated as the command to execute if no other program is due to start. A schedule file must include a default program.

Other lines are interpreted as :

*<year> <month> <day> <hour> <minute> <command line>*

The time is specified in UTC format. After the time has been read, the remainder of the line is treated as the command line to execute.

The scheduler scans through the file and loads and runs the appropriate control program. Every thirty seconds the scheduler checks through the loaded schedule, if the start time of a new program has expired then the current program is stopped and the new one started.

To run the scheduler enter the command :

`schedule` *filename*

Where *filename* is the name of the schedule file to run.

The scheduler will then scan through the schedule file and load and run the appropriate Control Program. Every thirty seconds the scheduler checks through the loaded schedule; if the start time of a new program has expired then the current program is stopped and the new one started.

Periodically the scheduler reloads and re-processes the schedule file. This allows any alterations or additions to the schedule to be correctly identified and acted upon. By default the schedule file is reloaded every hour, however this can be changed by including an option flag when the scheduler is started :

| | |
|---|---|
| `schedule -h` *filename* | reload filename every hour |
| `schedule -d` *filename* | reload filename every day |
| `schedule -t` *filename* | eload filename every 10 minutes. |

The scheduler can also be operated in verbose mode by specifying the "`-v`" option on the command line. In this mode the scheduler will display the currently operating schedule every thirty seconds. This is useful in checking that the schedule file has been correctly interpreted.

The scheduler will record all of its actions in a special log file stored in the directory "`/radops/scdlogs`". Each day a new log file is created with the filename "`scdlog`.*ddd*" where *ddd* is the day of the year.

The scheduler will periodically check to see whether the control program is still running. If it finds that it has died, it will attempt to restart it. If the current control program does not restart then the scheduler will load the default program instead.

# Changing Radar Parameters

When a control program is running it is useful to be able to and view or alter the operating parameters. This is accomplished by using the task `alter`, which received the current radar parameters from the control program and then enters a command line shell denoted by the "`>`" prompt.

The shell supports three commands :

```
go
show
<variable> = <value>
```

`go`

Typing "`go`" will send the altered parameters back to the control program so that they will take affect at the start of the next integration period.

`show [`*variable_name..*`]`

Typing "`show`" will list the values of the specified variables. If no variable names are listed then the entire set will be shown.

*<variable> = <value>*

A new value is assigned to a variable by using the "`=`" sign:

```
bmnum=12
combf=Hello world
```

There should be no spaces between the value, the "`=`" sign, or the variable name.

# Changing Radar Parameters

The radar parameters that can be altered 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. |

The author of the control program can also specify other variables that can be altered. The names and values of these optional variables can be found by using the "`show`" command.

Using the debug modes of the driver tasks, the Radar software can be tested on a computer without requiring the hardware interface cards. This allows new control programs to be checked out before they are run on an actual Radar.

All the Radar tasks will behave exactly as they would on an actual Radar, `fitacf` will process raw data from the control program, `raw_write` and the summary tasks will open and write data files, and the display tasks can be run.

As the timing sequence is generated but not transmitted, the task `radops_dio` can be run on the same computer as the rest of the Radar software, so that the software in its entirety can be tested on a single computer.

The script "`start_debug`" will start up the software in debug mode.

Unfortunately, in debug mode, no data is actually produced, however there are functions in the control library that allow data to be read in from a file. This allows the control program to simulate an operational Radar by using the raw data taken from a file.The source code for Radar control programs is located in the directory "`/radops/usr/src`". Each program has its own sub-directory that contains the C source code for the program and any extra files that it requires.

Each source directory contains a `makefile` that is used to compile the program (see the QNX Utilities Reference guide for more about makefiles). A typical `makefile` looks like this:

```
CFLAGS = -Oneatx -l control.lib support.lib

normal_scan :  normal_scan.o
            $(LINK.c) normal_scan.o \
            -o normal_scan
            usemsg normal_scan normal_scan.c
```

The first line of the `makefile` ensures that the program is compiled using the two libraries needed to operate the Radar, "`control.lib`" and "`support.lib`".

The location of these two libraries are defined by the environment variable LIB used by the C compiler to locate library files.

To compile the program type:

```
make
```

The compiled program should then be copied to the directory "`/radops/usr/bin`".

# Hardware Drivers

# a_d_drive

## Syntax

```
a_d_drive [-d] [name]
```

## Options

| | |
|---|---|
| `-d` | run the driver in debug mode by disabling the hardware interface. |
| *name* | register the driver under this name with the Operating System. |

## Description

The driver `a_d_drive` provides the software interface to the dt2828 Analogue to Digital Converter card.

The card can be programmed to use either an internal or external trigger signal, it can sample over multiple channels and it can use Direct Memory Access to store the sampled data in memory.

The driver supports two memory buffers that are used for the DMA transfers. The buffers use shared memory so that the control program can access and process the raw data. Two buffers are used so that while one is being processed the other can be used to sample the next integration period.

By starting the driver using the "`-d`" option it will run in "debug" or simulation mode and will not attempt to program the dt2828 card. This allows the software to be tested without the card installed on the main computer.

The optional *name* string is the name used to register the task with the operating system. By default the driver is registered under the name "`a_d_drive`". When another task wishes to locate the driver it must use this name. This allows two Radars to be connected to the same network as each driver must have a unique name.

## Syntax

```
gbuf      [-v] console ...
gbuf      [-f]
```

## Options

| | |
|---|---|
| `-v` | create consoles using VGA graphics modes. |
| `-f` | free up shared memory and exit. |
| *console* | the qnx console numbers to use as graphics consoles. |

## Description

The driver `gbuf` controls the graphics hardware on the main computer. It converts one or more QNX text consoles into graphics display consoles in either SVGA or VGA modes.

For each graphics console the driver creates an area of shared memory to use as a frame buffer. Graphics operations are performed on the buffer and whenever the console becomes active it is displayed on the screen.

The console numbers to use must be included on the command line. Once a console has been claimed for graphics it cannot be used for command line input.

By default the driver uses an SVGA mode of 800x600 pixels and 256 colours. A VGA mode of 640x480 pixels and 256 colours can be used by including the "`-v`" command line option.

When the driver is terminated it will not automatically relinquish the shared memory it has claimed as other tasks may also be using it. To force the shared memory to be released, call the driver using just the "`-f`" option.

# gps_clock

## Syntax

```
gps_clock [-d] [name]
```

## Options

| | |
|---|---|
| `-d` | run the driver in debug mode by disabling the hardware interface. |
| *name* | record the status and diagnostic information in the file called *name*. |

## Description

The driver `gps_clock` controls the GPS receiver card and returns the time synchronized to the GPS master clock. The driver automatically re-calibrates the system clock every 500 seconds.

By starting the driver using the "`-d`" option it will run in "debug" or simulation mode and will not attempt to access the GPS receiver card. This allows the software to be tested without the card installed on the main computer.

The optional name string is the filename to use when recording the status and diagnostic information, by default it is called "`/radops/gpslog`". The status is recorded every 500 seconds.

## Syntax

```
radops_dio   [-d] [name]
```

## Options

| | |
|---|---|
| `-d` | run the driver in debug mode by disabling the hardware interface. |
| *name* | register the task under this name with the Operating System. |

## Description

The driver `radops_dio` is responsible for programming the DIO card which sets the radar operating frequency, the receiver attenuation, the beam number, reads the antenna status information, and most importantly, outputs the pulse pattern that the radar transmits. As this is a time critical operation, interrupts are disabled on the computer while the sequence is transmitted. Consequently the DIO card is installed on a secondary timing computer. As the driver requires direct access to the DIO card it must also be run on the timing computer and communicate with the rest of the radar software over the network.

When `radops_dio` is running it displays the current transmission frequency, the beam number, the attenuator setting, whether the radar is in test mode, AGC and LOW_PWR status bits and what operation is being executed.

The program has sixteen buffers that are used to store timing sequences. The control program downloads a timing sequence into one of these buffers prior to outputting it to the DIO card.

Two types of timing sequence can be used; one is output on a single beam, the other includes information about which beam each pulse should be transmitted on.

By starting the driver using the "`-d`" option it will run in "debug" or simulation mode and will not attempt to program the DIO card. This allows the software to be tested without the rest of the radar hardware, and as the interrupts are not disabled, the driver can be run on the main computer allowing all of the software to be tested on a single machine.

The optional name string is the name used to register the task with the operating system. By default the driver is registered under the name "`/radops_dio`". When another task wishes to locate the driver it must use this name. This allows two Radars to be connected to the same network as each driver must have a unique name.

# Primary Tasks

## Syntax

```
alter      [name]
```

## Options

| | |
|---|---|
| *name* | The name of the control program whose parameters are to be modified. |

The task `alter` allows the user to change the operating parameters of the radar.

When run the task receives the current operating parameters from the control program before entering a command shell shown by the "`>`" prompt. The Radar will continue to run while the user enters commands to examine and alter the parameters.

The shell supports three commands :

```
go
show
<variable> = <value>
```

`go`

Typing "`go`" will send the altered parameters back to the control program so that they will take affect at the start of the next integration period.

`show [variable_name..]`

Typing "`show`" will list the values of the specified variables. If no variable names are listed then the entire set will be shown.

`<variable> = <value>`

A new value is assigned to a variable by using the "`=`" sign:

```
bmnum=12
combf=Hello world
```

There should be no spaces between the value, the "`=`" sign, or the variable name.

The radar parameters that can be altered 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. |

The author of the control program can also specify other variables that can be altered. The names and values of these optional variables can be found by using the "show" command.

The optional *name* string on the command line is the name registered by the control program with the operating system, by default it is "/control_program".

# echo_data

## Syntax

```
echo_data [-s stime] [-t ttime] [-n
          echo_name] [-e err_name]
```

## Options

| | |
|---|---|
| `-s` *stime* | the time-out period for a single task. |
| `-t` *ttime* | the time-out period for all the tasks. |
| `-n` *echo_name* | register the task under this name with the Operating System. |
| `-e` *err_name* | sends errors to the task registered under the name *err_name*. |

## Description

The task `echo_data` distributes both the raw ACF data and the fitted data produced by `fitacf` to other tasks. Tasks that register themselves will automatically receive the next block of data as it becomes available.

When a task wishes to register itself it sends a message containing a text string to associate with it. The task `echo_data` adds the process id (***pid***) of the registering task and the string to a table stored in memory.

Each time `echo_data` receives a message from the control program it duplicates it and attempts to send it in turn to each task recorded in the table. If a task has died then `echo_data` will remove that entry from the table and move on to the next entry in the list.

If the task does not reply to the message within the time allocated for a single task as set by the "`-s`" option, then `echo_data` will time out and move onto the next task in the table. If the `echo_data` cannot send the message to all the tasks within the total allocated time as set by the "`-t`" option, then it will not send to the remaining tasks in the list. When the next block of data is received, the task will try sending to all the tasks again. By default, the time-out period for a single task is one (1) second, and the total time-out period is five (5) seconds.

The "`-n`" option specifies the name used to register the task with the operating system. When another task wishes to locate `echo_data` it must use this name. By default the task is registered under the name "`/echo_data`".

The "`-e`" option specifies the name of the error log that the task reports errors and warnings to. By default errors are sent to the task registered under the name "`/errlog`".

## Syntax

```
errlog    [name]
```

## Options

| | |
|---|---|
| *name* | register the task under this name with the Operating System. |

## Description

The `errlog` task reports and logs errors sent from the other radar tasks. The log records the time at which the message was sent, the process id of the task reporting the error, its name, and a text string that describes the error.

Errors are printed to the console on which the `errlog` task runs and also recorded in a file stored in the directory "/radops/errlogs". Each day a new log file is created with the filename `errlog`.*ddd*, where *ddd* is the day of the year.

The optional *name* string specifies the name used to register the task with the operating system. When a task wishes to locate the error log it must use this name. By default the task is registered under the name "`/errlog`".

## Syntax

```
fitacf    [-e err_name] [-f buf_name] [name]
```

## Options

| | |
|---|---|
| `-e` *err_name* | send errors to the task registered under the name *err_name*. |
| `-f` *buf_name* | send the output to the buffer task registered as *buf_name*. |
| *name* | register the task under this name with the Operating System. |

## Description

The task `fitacf` calculates the derived parameters such as velocity and spectral width.

The task receives the raw ACF data from the control program and attempts to fit it to the expected distribution. From this a number of parameters are calculated and stored in an output file. Files are opened and closed at times specified by the control program.

The data is also returned to the control program via the `fit_buffer` task for distribution to the other radar tasks.

The processing of the raw data requires a significant amount of CPU time and for this reason the calculations are performed during the next integration period. Consequently the fitted data lags one beam behind the raw data.

The program produces two output files, one has the suffix FIT and contains the data, the other has the suffix INX and contains an index to the data. The filenames are of the form:

*yymmdds*`.FIT`
*yymmdds*`.INX`

Where:

| | |
|---|---|
| *yy* | year XX*yy* |
| *mm* | month. |
| *dd* | day |
| *hh* | hour |
| *s* | station id e.g. g |

The "`-e`" option specifies the name of the error log task that the task reports errors and warnings to. By default errors are reported to the task registered under the name "`/errlog`".

The optional *name* string is the name used to register the task with the Operating System. When a control program wishes to communicate with `fitacf` it must use this name. By default the task is registered under the name "`/fitacf`".

The "`-f`" option specifies the name of the `fit_buffer` task that the processed data will be sent to.

# fit_buffer

## Syntax

```
fit_buffer   [name]
```

## Options

| | |
|---|---|
| *name* | register the task under this name with the Operating System. |

## Description

The task `fit_buffer` acts as a temporary storage buffer for the data produced by f`itacf`.

Both the raw ACFs and the fitted data are distributed to the other tasks by the control program which must get the processed data from `fitacf`. However `fitacf` processes the last block of data during the current integration period and the data must be stored until the control program is ready to receive it. The `fit_buffer` task receives the block of data from `fitacf` and passes it to the control program when required.

The optional name string specifies the name used to register the task with the operating system. When another `fitacf` wishes to locate the buffer it must use this name. By default the task is registered under the name "`/fit_buffer`".

## Syntax

`raw_write [-e` *err_name*`] [-t` *threshold*`] [`*name*`]`

## Options

| | |
|---|---|
| `-e` *err_name* | send errors to the task registered under the name *err_name*. |
| `-t` *threshold* | reject data with lag-zero power less than *threshold**`NOISE`/2. |
| *name* | register the task under this name with the Operating System. |

## Description

The task `raw_write` receives the raw ACF data from the control program, compresses it, and stores it on disk. Files are opened and closed at times specified by the control program.

The data is compressed from 32 bit integers into 16-bit pseudo floating point numbers. The compression does result in a loss of some accuracy as the low order bits of the original integers are lost, however the compression does give a 50% reduction in the output file size.

The filenames are of the form:

*yymmdds*`.DAT`

Where:

| | |
|---|---|
| *yy* | year XXyy |
| *mm* | month. |
| *dd* | day |
| *hh* | hour |
| *s* | atation id e.g. g |

By specifying the "`-t`" option a threshold value can be applied to the lag-zero power. Data below this threshold is not stored in the file providing a further reduction in the size of the output file. The threshold function is defined as:

*threshold**`NOISE`/2.

Data with lag-zero power less than the result of the above sum is rejected.

The "`-e`" option specifies the name of the error log task that the task reports errors and warnings to. By default errors are sent to the task registered under the name "`/errlog`".

The optional *name* string is the name used to register the task with the Operating System. When a control program wishes to communicate with `raw_write` it must use this name. By default the task is registered under the name "`/raw_write`".

# scheduler

## Syntax

```
schedule  [-t] [-h] [-d] [-v] [ -n name]
          sched_file
```

## Options

| | |
|---|---|
| `-t` | reload the schedule every ten minutes. |
| `-h` | reload the schedule every hour. |
| `-d` | reload the schedule every day. |
| `-v` | operate in verbose mode. |
| `-n` *name* | register the task under this name with the Operating System. |
| *sched_file* | the filename of the schedule file to load. |

## Description

The task `schedule` is responsible for scheduling when control programs are started and stopped. The task reads in a schedule file and extracts from this the names and start times for the programs to run.

A schedule file is a simple text file containing the start times and command line of the programs to run:

```
# test.sched
default /radops/usr/bin/normal_scan
1996 3 19 14 50 /radops/usr/bin/sdcusp_2200
1996 3 19 15 00 /radops/usr/bin/normal_scan
1996 3 19 15 30 /radops/usr/bin/sdcusp_2200
1996 3 19 15 40 /radops/usr/bin/normal_scan
```

Each line in the schedule file corresponds to a command to execute. Lines beginning with a "#" are treated as comments and are ignored.

If a line begins with the word `default`, then the rest of the line is treated as the command to execute if no other program is due to start. A schedule file must include a default program.

Other lines are interpreted as :

<*year*> <*month*> <*day*> <*hour*> <*minute*> <*command line*>

The time is specified in UTC format. After the time has been read, the remainder of the line is treated as the command line to execute.

The scheduler scans through the file and loads and runs the appropriate control program. Every thirty seconds the scheduler checks through the loaded schedule, if the start time of a new program has expired then the current program is stopped and the new one started.

Periodically the scheduler reloads and re-processes the schedule file. This allows any alterations or additions to the schedule to be correctly identified and acted upon. By default the schedule file is reloaded every hour, however this can be changed by using one of the option flags; "`-t`" will reload the schedule every 10 minutes, "`-h`" will reload it every hour, and "`-d`" will reload it once a day.

When the scheduler is started with the "`-v`" flag it operates in verbose mode displaying currently running schedule every thirty seconds. This is useful in checking that the schedule file has been correctly interpreted.

The "`-n`" option specifies the name used to register the task with the operating system. When another task wishes to locate the scheduler it must use this name. By default the task is registered under the name "`/schedule`".

The scheduler will record all of its actions in a special log file stored in the directory "`/radops/scdlogs`". Each day a new log file is created with the filename `scdlog`.*ddd* , where *ddd* is the day of the year.

# Display Tasks

# display

## Syntax

```
display   [-a [range]] [[-p] [-v] [-w]
          [scale]] [-m] [-e echo_name]
```

## Options

| | |
|---|---|
| `-a` *range* | plot the calculated ACF at the range gate given by *range*. |
| `-p` *scale* | plot the lambda power with a colour scale ranging between 0 and +*scale* dB. |
| `-v` *scale* | plot the velocity with a colour scale of range ±*scale* $ms^{-1}$. |
| `-w` *scale* | plot the lambda spectral width with a colour scale ranging between 0 and +*scale* $ms^{-1}$. |
| `-m` | plot the ACF with the highest lag-zero power. |
| `-e` *echo_name* | receive data from the version of `echo_data` registered under the name *echo_name.* |

## Description

The task `display` is a client of `echo_data` that displays a graphical representation of the radar data on the console.

The program provides a crude graphical display on a **QNX** or **X** terminal of the raw data generated by a control program and the fitted data produced by `fitacf.`

When the program is running it will display the current transmitted frequency, the noise level, the range separation, first range gate, time, and beam number. Depending on the display mode selected it will also plot the raw ACF at the specified range, the raw ACF with the largest power, or, the fitted velocity, the lambda power, or lambda width for all ranges and beams.

The fitted data displays have range gates running horizontally across the screen and beam number running vertically down the screen. On a **QNX** terminal radar data is plotted as coloured squares according to the colour bar shown at the top of the screen. On an X terminal the bar consists of the numbers from 0-9. The maximum value of the bar is displayed at the top right of the screen. For lambda power and spectral width the bar corresponds to values between zero and this value, for velocity it corresponds to values between plus and minus this value. The maximum value of the bar can be altered by pressing the up and down arrow keys, or by typing in the number and pressing return.

The ACF display shows the raw lag 0 power for all ranges at the top of the screen. Below that is the ACF plot for the specified range. The selected range numbered from zero can be altered by using the up and down arrow keys or by typing in the desired range and pressing return. Pressing "`m`" will plot the range with the maximum lag-0 power. The selected range is shown in the top right of the screen.

The selected mode can be changed by pressing the left and right arrow keys, or pressing "a" for ACF, "v" for velocity display, "p" for lambda power or "w" for lambda width. The program starts in the velocity mode.

The program can be stopped at any time by pressing <Ctrl> <c> or by typing "q".

The command line options control which mode the display task is started in; "-a" displays the ACF at the specified range, "-p" displays the lambda power, "-w" the lambda width, and "-v" shows the velocity. The optional scale value sets the limits of the colour scale. The "-m" option displays the ACF with the largest lag zero power.

The "-e" option specifies the name of the echo_data task to connect to. By default the task will connect to the program registered under the name "/echo_data".

## Syntax

```
fitdisp   [-p] [-w] [-v] [-q] [-t
          low_power] [-m max_val] [console]
```

## Options

| | |
|---|---|
| `-p` | plot lambda power. |
| `-w` | plot spectral width. |
| `-v` | plot velocity. |
| `-q` | ignore the quality flag in the data. |
| `-t` *low_power* | set the threshold of the lag zero power to plot. |
| `-m` *max_val* | set the limits of the colour scale to *max_val*. |
| `-e` *echo_name* | receive data from the version of `echo_data` registered under the name *echo_name.* |
| *console* | display the output on this graphics console. |

The task `fitdisp` receives data from `echo_data` and produces a real-time plot on a QNX machine running the `gbuf` graphics driver. The plot is a geographically accurate view of the fitted data produced by `fitacf`. The plot has a number of overlays including the outline of the continents, the field of views of the other Radars and a set of user defined text labels.

The data plotted is selected by the command line options; selecting "`-v`" plots velocity, "`-p`" plots lambda power, and "`-w`" plots spectral width. By default velocity is plotted. The maximum limits of the colour scale are set by using the "`-m`" flag.

Usually the task checks the fitted data and only plots values for which the quality flag has been set, this can be over-ridden by using the "`-q`" flag. The "`-t`" option applies a limiting threshold to the lag-zero power so that data with power below the threshold is ignored.

The "`-e`" option specifies the name of the echo_data task to connect to. By default the task registered under the name "`/echo_data`" is used.

The user defined labels are stored in a file pointed to by the environment variable `SD_OVERLAY`, usually this is set to "`/radops/tables/overlay`". The overlays are defined as a simple space separated text file with one label per line:

```
# Example of an overlay file
-90.0 0.0 South Pole
-98.0 45.0 AGO 1
90.0.0 0.0 North Pole
```

Lines beginning with a '#' are treated as comments and ignored. The first two entries on a line are the latitude and longitude of the label, the remainder of the line is taken to be the text label.

The last argument of the command line is the console number on which the plot will be displayed. If this argument is omitted then the plot will appear on console eight (8).

## Syntax

```
qltp      [-e echo_name] [console]
```

## Options

| | |
|---|---|
| `-e` *echo_name* | receive data from the version of `echo_data` registered under the name *echo_name.* |
| *console* | display the output plot on this graphics console. |

## Description

The task `qltp` receives data from `echo_data` and produces a summary plot that can be displayed on a QNX machine running the `gbuf` graphics manager. The data can be plotted in real time or taken from a summary file produced by `sd_summary`.

Plots are produced on console number eight (8) if no arguments are given or on console number specified by 1*console*.

The "`-e`" option specifies the name of the `echo_data` task to connect to. By default the task connects to the program registered under the name "`/echo_data`".

The summaries produced consist of a range-time plot of a single radar beam showing two of the parameters stored in the data produced by `fitacf`.

The task uses a menu system that can be controlled by either the mouse or the cursor keys, moving the mouse over an entry in the menu will hi-light it. Individual entries in the menu can be stepped through and hi-lighted by pressing the up and down cursor keys. Pressing return or clicking a mouse button over a hi-lighted entry will select it.

Some entries in the menu are switches that can be either on of off, these are shown in green when they are off, and white when they are on. Others are push buttons that trigger other operations such as loading and saving files, these are shown in yellow. Text fields where numbers and words can be typed in are shown in white with a blue background. Selecting one of these items will clear the text field and a new entry can be typed in; pressing enter or escape will store the new value. Number fields can also be altered by clicking on the yellow arrow buttons on either side of the field. The left pointing arrow will decrement the number, the right pointing arrow will increment it.

The task uses three sets of menus. They can be selected by clicking on the arrow buttons in the menu box at the bottom left of the screen; each menu relates to different parameters of the plot. The first shows the most commonly changed parameters such as the frame length and which item of data from `fitacf` to plot.

The plot window is divided into two and can display two different items of data from `fitacf`. The selected items are hi-lighted in white in the two lists at the top of the menu.

The scale to use for the two parameters is selected beneath them. The division marks of the key are set using "`Scale step`". Values of the parameter that lie outside the range of the scale are plotted in the appropriate colour for the limit of that scale.

The "`Threshold parameter`"and the cut-off "`Level`" are selected in the right hand column; range/beam points with values of the threshold parameter below the cut-off level will not be plotted.

The frame length is set in hours and minutes, up to 24 hours. The interval between division lines on the plot is set using "`Frame step`". The sub divisions along the bottom of the plot are set using "`Frame tick`".

The start and end ranges of the plot are set in kilometers from the radar site.

The plot consists of a range-time plot for a single beam selected using "`Beam`". The values of the two parameters selected from the f`itacf` data are plotted for this beam. If "`Beam`" is set to -1 then all the beams will be plotted. The "`Beam persistence`" sets the width of the bar to plot for a beam. Usually this is set to sixteen as all sixteen beams must be scanned before a new bar will be plotted.

The second menu shows more general parameters such as the noise and frequency scales. The entry "`Noise max`" is uses as the upper level of the noise scale and values are assumed to lie between zero and this value.

Usually data will not be plotted of the ground scatter flag is set or the quality flag is not set. This can be changed by selecting "`Ignore Quality Flag`" or "`Ignore Ground Scatter`".

The frequency scale is divided into eight bins, frequencies below the value set for a specific bin will be plotted in the colour selected for that bin. The colour values range from -1 to 511, with -1 corresponding to black. Colour values less than or equal to 255 correspond to the first colour scale, and values greater than 255 to the second.

The "`Station Name`" string is the title used when labeling the plot.

The final menu controls the offline plotting of files produced by `sd_summary`.

The "`Start time`" is the offset from the start of the image file to begin plotting data.

Clicking on "`Plot File`" will bring up a filer window showing the currently available summary files. Selecting a file to plot will prompt the user to switch to the graphics console. At the end of each frame of data the program will pause until the user presses a key and then the next full frame of data is plotted until the end of the file is reached.

The four buttons at the bottom of all of the menus are used for loading and saving a set of options, entering the real time plot mode and for leaving the program.

Clicking on "Load" or "Save" will open a file selection window which is divided into two parts. The top part shows the full filename of the current configuration file and the bottom shows the contents of the directory where the file resides.

Selecting a file in the bottom part of the window using either the mouse or the cursor keys will change the file name in the top window. Text can be typed directly into the top window and will appear at the current text cursor which can be moved using the left and right cursor keys.

Pressing return will accept the file name in the top window as the name of the file to either load or save. If this name is a directory then the bottom half of the window will change to show the contents of the new directory.

Pressing the escape key will abort the load and save operation.

Clicking on the "Run" button will cause the program to enter the real time plot mode. The user is prompted to change to the graphics console to view the plot.

The plot mode is stopped by pressing a key on the console that the menu is displayed on.

The locations of files and the graphics mode used by `qltp` are all stored in the header "`/radops/src/qltp/config.h`":

```
/* configuration file for qltp */

#define QLTP_CONFIG_PATH "/radops/scripts"
#define QLTP_CONFIG_NAME "qltp.config"
#define QLTP_IMAGE_PATH "/summary"
#define QLTP_IMAGE_NAME "test.smr"
```

The directory path and initial filename for the `qltp` configuration file is specified with `QLTP_CONFIG_PATH` and `QLTP_CONFIG_NAME`. When a configuration file is loaded or saved for the first time, the file window will be opened using this directory and filename.

The initial name and location of the summary file used for off-line plotting is defined by `QLTP_IMAGE_PATH` and `QLTP_IMAGE_NAME`.

# Summary Data Tasks

# compress

## Syntax

```
compress  [-d echo_name] [-e errlog] [-l] [-v] [-w] [-y] [-x] [-p low_power] [-h hour] [-m minute] [-b bmnum] [name]
```

## Options

| | |
|---|---|
| `-d` *echo_name* | attaches to the version of `echo_data` registered under the name *echo_name*. |
| `-e` *err_name* | sends errors to the task registered under the name *err_name*. |
| `-l` | record lambda power data. |
| `-v` | record velocity data. |
| `-w` | record spectral width data. |
| `-y` | use the low compression rate. |
| `-x` | use the high compression rate. |
| `-p` *pwr* | reject data points with lambda-0 power less than *pwr*. |
| `-h` | set the number of hours of data recorded in a file. |
| `-m` | set the number of minutes of data recorded in a file. |
| -b *bmnum* | record summary information about beam *bmnum* in memory. |
| *name* | register the task under this name with the Operating System. |

## Description

The task `compress` is a client of `echo_data` that produces highly compressed summary files, called **C**olour **M**ap **F**iles (**CMP**), from the data generated by `fitacf`.

The program can be run in either high or low compression modes. In the high compression mode, set by the "`-x`" command line option, the program records data as an 8-bit number that is an index in the standard SuperDARN colour table. Files produced in this mode can be used to produce animation's of the observed scatter or time series plots of each radar beam much like those produced by `qltp` and `fitdisp`. In the low compression mode set by the "`-y`" command line option, the data is recorded as the full 64 bit floating point numbers.

The data files produced can contain the lambda power, spectral width and velocity parameters or any combination of the three. The parameters are selected on the command line using the "`-l`", "`-v`", or "`-w`" options.

Only data with the quality flag set is recorded in the file and a threshold limit, defined by the "`-p`" command line option, is applied to the lag zero power, data with power below this limit is ignored.

Unlike other summary tasks, which open and close files following requests from the control program, `compress` records files of a fixed length specified

by the "-h" and "-m" command line options. The "-h" option sets the number of hours to record in a file, and the "-m" option the number of minutes.

The task also maintains a buffer in memory that is used to record information about a single beam specified by the "-b" option. When the task receives a message from the ctrig task, the buffer is written to disk in the file "/cmp/delta".

The "-d" option selects which version of echo_data the task should attach to by default the task registered under the name "/echo_data" is used. The "-e" option selects which version of the error log errors should be reported to, by default errors are sent to the task registered under the name "/errlog".

The optional *name* string specifies the name used to register the task with the operating system. When a task wishes to locate compress it must use this name. By default the task is registered under the name "/compress".

# sd_summary

## Syntax

```
sd_summary   [-d echo_name] [-e errname] [-
          a beamA] [-b beamB] [-p pwr] [-n
          name]
```

## Options

| | |
|---|---|
| `-d` *echo_name* | attaches to the version of `echo_data` registered under the name *echo_name*. |
| `-e` *err_name* | sends errors to the task registered under the name *err_name*. |
| `-a` *beamA* | record information about *beamA* in the file. |
| `-b` *beamB* | record information about *beamB* in the file. |
| `-p` *pwr* | reject data points with lambda-0 power less than *pwr*. |
| `-n` *name* | register the task under this name with the Operating System. |

## Description

The task `sd_summary` is a client of `echo_data` that produces summary files from data produced by `fitacf`.

The program records in a text file the values of lambda power, lambda width and velocity for one or two beams per scan.

Only data with the quality flag set is recorded in the file and a threshold limit, defined by the "`-p`" command line option, is applied to the lag zero power, data with power below this limit is ignored.

By default the program records the data from beam eight (8) of the radar however the two command line options "`-a`" and "`-b`" can be used to select other beams. To record a single beam of data use the "`-a`" option on its own, to record two beams, use both "`-a`" and "`-b`".

The "`-d`" option selects which version of `echo_data` the task should attach to, by default the task registered under the name "`/echo_data`" is used. The "`-e`" option selects which version of the error log errors should be reported to, by default errors are reported to the task registered under the name "`/errlog`".

The optional name string specifies the name used to register the task with the operating system. When a control program wishes to locate `sd_summary` it must use this name. By default the task is registered under the name "`/sd_summary`".

## Syntax

```
vlptm      [-a echo_name] [-e errlog]
```

## Options

| | |
|---|---|
| `-a` *echo_name* | attaches to the version of `echo_data` registered under the name *echo_name*. |
| `-e` *err_name* | sends errors to the task registered under the name *err_name*. |

## Description

The task `vlptm` is a program for the estimation of two-dimensional velocity vectors on the basis of the line-of-sight velocity data from a single radar.

It consists of a set of C subroutines and look-up files containing the coordinate data for the particular radar. The task runs concurrently with `fitacf`; it receives velocity data from each scan and outputs the velocity vector (in geomagnetic coordinates) to a file. The file is the basis for the generation of the daily velocity clock-dial plot and is also the source for the transfer of the key parameter data to CDHF.

The logic of `vlptm` is based on the assumption of quasi-uniformity of the convection along a contour of constant magnetic latitude. The set of beam directions then samples the velocity at a given latitude from a variety of angles and the full two-dimensional vector can be estimated in an optimal sense be fitting a cosine dependence to the variation of the line-of-sight velocity with the look angle. The quality of the fitting varies. This is flagged by a quality index, *qflag*, defined by:

$$qflag = 2.5*(qn+qr)$$

Where:

$$qn = (1-(1/11)*(np-5)$$
$$qr = (1-(1/55)*(55-rms)$$

Where *np* is the number of points contributing to the fit and *rms* is the root-mean-square error on the fit. Normally *qflag* varies from (0) (the best) to (4) for acceptable fits.

Less satisfactory fits may also be listed, with *qflag* values that range from (5) to (*t*). In this case the lower limit for *np* (5) and/or the limit for dropping points (1/3 of the total number on the contour) has been reached but the fit may be unsatisfactory for the following reasons:

| | |
|---|---|
| qflag=5 | unable to meat criterion on rms (55 m/s). |
| qflag=6 | fit is not superior to fit that assumes constant velocity |
| qflag=7 | more than half the points are from beams >10. (GB Only). |

The `-a` option selects which version of `echo_data` the task should attach to, by default the task registered under the name "`/echo_data`" is used. The `-e` option selects which version of the error log errors should be reported to, by default errors are reported to the task registered under the name "`/errlog`".

# Internet Access Software

# client

## Syntax

```
client      host  port
```

## Options

| | |
|---|---|
| *host* | the name of the remote host to attach to. |
| *port* | the TCP/IP port number of the server task to attach to on the remote host. |

## Description

The `client` task is a simple diagnostic client of the `server` task. The source code can be used as a starting point for more sophisticated clients.

The task connects to the `server` task running on the specified host and port number and waits to receive blocks of data. When a block is received, the task prints out the ranges for which data is available and the velocities for those ranges.

# echo_dataIP

## Syntax

```
echo_dataIP  [-s stime] [-t ttime] [-n
          echo_name] [-e err_name] host  port
```

## Options

| | |
|---|---|
| `-s` *stime* | the time-out period for a single task. |
| `-t` *ttime* | the time-out period for all the tasks. |
| `-n` *echo_name* | register the task under this name with the Operating System. |
| `-e` *err_name* | sends errors to the task registered under the name *err_name*. |
| *host* | the name of the remote host to attach to. |
| *port* | the TCP/IP port number of the server task to attach to on the remote host. |

## Description

The task `echo_dataIP` is a version of `echo_data` that receives its data from a remote server task using TCP/IP protocols. The data received is a sub-set of that produced by `fitacf`. Tasks that register themselves will automatically receive the next block of data as it is received from the server.

When a task wishes to register itself it sends a message containing a text string to associate with it. The task `echo_dataIP` adds the process id (**pid**) of the registering task and the string to a table stored in memory.

Each time `echo_dataIP` receives a block of data from the server it transforms it into the standard message types used by the Radar software and attempts to send it in turn to each task recorded in the table. If a task has died then `echo_dataIP` will remove that entry from the table and move on to the next entry in the list.

If the task does not reply to the message within the time allocated for a single task as set by the "`-s`" option, then `echo_dataIP` will time out and move onto the next task in the table. If the `echo_dataIP` cannot send the message to all the tasks within the total allocated time as set by the "`-t`" option, then it will not send to the remaining tasks in the list. When the next block of data is received, the task will try sending to all the tasks again. By default, the time-out period for a single task is one (1) second, and the total time-out period is five (5) seconds.

The "`-n`" option specifies the name used to register the task with the operating system. When another task wishes to locate `echo_dataIP` it must use this name. By default the task is registered under the name "`/echo_dataIP`".

The "`-e`" option specifies the name of the error log that the task reports errors and warnings to. By default errors are sent to the task registered under the name "`/errlog`".

# server

## Syntax

```
server    [-n name] [-d echo_data]  [-e
          errlog] [port]
```

## Options

| | |
|---|---|
| `-n` *name* | register the task under this name with the Operating System. |
| `-d` *echo_name* | attaches to the version of `echo_data` registered under the name *echo_name*. |
| `-e` *err_name* | sends errors to the task registered under the name *err_name*. |
| *port* | the TCP/IP port number of the to attach to on the local host. |

## Description

The `server` task allows Radar data to be transmitted over the Internet in real time. The task receives data produced by `fitacf` from `echo_data` and passes it on to client tasks running on remote systems using the TCP/IP protocol.

The data transmitted consists of the radar parameter block and the lambda power, spectral width, and velocity components. A 3dB Threshold is applied to the lag zero power and data below this threshold is ignored.

When running, `server` listens on the specified port for connection requests from other programs. When a client is accepted, radar data will be compressed and transmitted to it after each integration period.

There is an upper limit of sixteen simultaneous clients that can be connected to the server at any one time.

The "`-d`" option selects which version of `echo_data` the task should attach to by default the task registered under the name "`/echo_data`" is used. The "`-e`" option selects which version of the error log errors should be reported to, by default errors are sent to the task registered under the name "`/errlog`".

The optional name string specifies the name used to register the task with the operating system. When a task wishes to locate `server` it must use this name. By default the task is registered under the name "`data_server`".

# Off-line Support Software

# close_file

## Syntax

`close_file` *task_name*

## Options

| | |
|---|---|
| *task_name* | send the close message to *task_name*. |

The `close_file` task sends a close message to the specified task forcing it to close any open files.

## Syntax

```
cmp_fit   [-l] [-v] [-w] [-y] [-x] [-p
          low_power] [-b bmnum] fit_file
```

## Options

| | |
|---|---|
| `-l` | store lambda power in the output file. |
| `-v` | store velocity in the output file. |
| `-w` | store spectral width in the output file. |
| `-y` | use the low compression format. |
| `-x` | use the high compression format. |
| `-p` *low_power* | apply a threshold of *low_power* to the lag-zero power. |
| `-b` *bmnum* | record only information about beam number *bmnum*. |
| *fit_file* | the filename of the fit file to read. |

The `cmp_fit` task reads the fit file *fit_file*, and produces a **C**olour **M**a**p** (**CMP**) file on the standard output, `stdout`.

The program can be run in either high or low compression modes. In the high compression mode, set by the "`-x`" command line option, the program transforms data into an 8-bit number that is an index in the standard SuperDARN colour table. Files produced in this mode can be used to produce animation's of the observed scatter or time series plots of each radar beam much like those produced by `qltp` and `fitdisp`. In the low compression mode set by the "`-y`" command line option, the data is recorded as the full 64 bit floating point numbers.

The data files produced can contain the lambda power, spectral width and velocity parameters or any combination of the three. The parameters are selected on the command line using the "`-l`", "`-v`", or "`-w`" options.

Only data with the quality flag set is recorded in the file and a threshold limit, defined by the "`-p`" command line option, is applied to the lag zero power, data with power below this limit is ignored.

By specifying the "`-b`" option an output file containing data for only the specified beam *bmnum*, can be produced.

# ctrig

## Syntax

```
ctrig      [cmp_name]
```

## Options

| | |
|---|---|
| *cmp_name* | send the trigger message to the `compress` task registered under the name *cmp_name*. |

The `ctrig` task sends a message to the `compress` task causing it to write the last few hours of data in a file. The file is called "`/cmp/delta`" and contains data for a single beam.

The file produced can be used to provide a quick "snap-shot" of the radar scatter without having to wait until the end of the day for the full summary data set.

The optional *cmp_name* string specifies the name of the `compress` task to send the signal to. By default the task registered under the name "`/compress`" is used.

# plot_cmp

## Syntax

```
plot_cmp  [-p] [-v] [-w] [-f] [-c console]
          [-d delay] cmp_file
```

## Options

| | |
|---|---|
| `-p` | plot lambda power data. |
| `-v` | plot velocity data. |
| `-w` | plot spectral width data. |
| `-f` | apply a simple filter to smooth the data. |
| `-c` *console* | plot the data on graphics console specified by *console*. |
| `-d` *delay* | specify the delay in milliseconds between frames. |
| *cmp_file* | the name of the cmp file to read. |

The task `plot_cmp` plots the contents of a **C**olour **Ma****p** file (**CMP**) as a simple animation. The plot has a number of overlays including the outline of the continents, the field of views of the other Radars and a set of user defined text labels.

The task uses the graphics console driver `gbuf` and the console to plot on is specified using the "`-c`" command line option. By default graphics console eight (8) is used.

The parameter that the program will plot, either lambda power, velocity or spectral width, can be set using the "`-p`", "`-v`", or "`-w`" command line option. By default velocity is plotted.

The time in milliseconds between successive frames of the animation can be set using the "`-d`" option.

# tplot_cmp

## Syntax

```
tplot_cmp [-b bmnum] [-c console] [-s start]
          [-e extent] cmp_file
```

## Options

| | |
|---|---|
| `-b` *bmnum* | plot data for beam number bmnum. |
| `-c` *console* | plot the data on graphics console specified by *console*. |
| `-s` *start* | start the plot at the time in hours and minutes given by *start*, expressed as *hh*:mm, where *hh* is the number of hours, and *mm* is the number of minutes. |
| `-e` *extent* | plot the period of time of length *extent*, expressed as *hh*:mm, where *hh* is the number of hours, and *mm* is the number of minutes |
| *cmp_file* | the name of the cmp file to read. |

The task `tplot_cmp` plots the contents of a **C**olour **M**a**p** file (**CMP**) as a time series plot on a graphics console.

The task uses the graphics console driver `gbuf` and the console to plot on is specified using the "`-c`" command line option. By default graphics console eight (8) is used.

The program will plot a 24 hour period starting at the first record encountered in the input file. The "`-s`", and "`-e`" options can be used to override the start time and the length of time to plot. The times should be expressed in the form:

*hh*`:`*mm*

Where:

| | |
|---|---|
| *hh* | hours |
| *mm* | minutes. |

The beam number to be plotted can be set using the `-b` option, by default beam eight (8) is plotted.

# Diagnostic Utilities

# a_d_test

## Syntax

```
a_d_test  [a_d_name]
```

## Options

| | |
|---|---|
| *a_d_name* | test the version of a_d_drive registered under the name *a_d_name*. |

The task `a_d_test` is a simple diagnostic task for the Analogue to Digital Converter (ADC) driver `a_d_drive`.

It performs some simple software triggered A/D conversions and prints the results on the display.

The optional `a_d_name` string specifies the name of the `a_d_drive` task to test. By default the task registered under the name "/`a_d_drive`" is tested.

## Syntax

`test_dio [`*dio_name*`]`

## Options

| | |
|---|---|
| *dio_name* | test the version of `radops_dio` registered under the name *dio_name*. |

The task `test_dio` is a very simple diagnostic program for the DIO driver `radops_dio`.

The task allows the user to send commands directly to `radops_dio`. When the program is started it displays a menu of the commands available:

```
1. reset_xt
2. send_tsg
3. set_beam
4. set_freq
5. set_gain
6. download timing sequence
7. verify_id
8. set antenna mode
9. set test mode
10. get status
11. set filter mode
99. EXIT
enter the function number :
```

The user types the number of the function to be performed, or 99 to exit the program, and presses `<enter>`. Depending on the command chosen they will then be prompted for another input.

Once the command has been executed the a status code is printed on the console. If the command was successful this will be zero (0).

The optional *dio_name* string specifies the name of the `radops_dio` task to test. By default the task registered under the name "`/radops_dio`" is used.

# test_echo

## Syntax

`test_echo [-e` *echo_name*`] [-s` *station*`]` *fit_file*

## Options

| | |
|---|---|
| -e *echo_name* | send data to the version of echo_data registered under the name *echo_name*. |
| -s *station* | Change the Radar station identifier letter stored in the Radar parameter block to *station*. |
| *fit_file* | The filename of the fit file to read. |

The task `test_echo` allows `echo_data` and its client tasks to be tested.

The program reads in data records from a fit file generated by `fitacf` and passes them to `echo_data`. When the end of the file is reached the program will start at the beginning again in a continuous loop.

The "`-s`" option can be used to simulate data from a particular radar site. The single letter radar identifier stored in the parameter block is subsituted for the one supplied on the command line.

The "`-e`" option specifies which version of `echo_data` to send data to. By default data will be sent to the task registered under the name "`/echo_data`".

# test_gbuf

## Syntax

`test_gbuf [`*console*`]`

## Options

| | |
|---|---|
| *console* | plot graphics on the console number *console*. |

The task `gbuf_test` is a very simple diagnostic program for the graphics console driver `gbuf`.

The task draws some simple graphics on the screen.

The optional *console* number specifies the console on which the test should be performed. By default console number eight (8) is used.

# Software Version Control

## Syntax

`control_mod` *filename*

## Options

| | |
|---|---|
| *filename* | the filename of the module version file. |

## Description

The task `control_mod` is a utility used for checking the version numbers in the various modules in the Radar software source code.

It compares the **R**evision **C**ontrol **S**ystem (**RCS**), information stored in each module against a master list stored in the file *filename*. Each module must have the RCS keyword "`$Revision$`" somewhere in the file:

```
/* module.c
    ======= */

/* Insert RCS revision keyword here:

 $Revision$
 */

/* Program starts here */
.
.
```

The master list, usually called "`control.info`", is a list of modules together with the expected revision number:

```
# $Revision$
#

module.c    1.1
main.c      1.3
.
.
```

Lines starting with a '#' are treated as comments and ignored. Each line contains the name of the module followed by a space or tab and then the expected revision number.

The master list should also be maintained under RCS and include the "`$Revision$`" keyword, this is used to generate a master version number for the program.

The task scans the list and checks for the existence of each module and that the revision numbers agree. If a discrepancy is found the program will stop and report the error.

When all the modules have been checked, the task produces a master version header on stdout:

```
/*version.h*/
  ========

#define VERSION x.y
#define VSTRING "x.y"
#define VMAJOR x
#define VMINOR y
```

The output can be redirected to a file and included in the source code to access the version numbers.

# logo

## Syntax

```
logo [-f] [-t] [-d time]
```

## Options

| | |
|---|---|
| `-f` | fade the logo using a special effect. |
| `-t` | run in text mode only. |
| `-d` *time* | wait the specified number of seconds before returning control to the user. |

## Description

The task `logo` is a utility for displaying the Radar software title page, the radar station, and the master version number of the software. The master version number is defined in the header "`/radops/include/radops_version.h`".

The "`-f`" command line option will adds a special effect that fades the title page in. The "`-d`" option sets the length of time in seconds that the title page is displayed.

To display the version information in text mode only use the "`-t`" option.

# The Control Library

# Data Structures

## Syntax

```
#include  “radops.h”
```

## Description

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.

## Syntax

```
#include  “radops.h”
```

## Description

The structure `rawdata` 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 number of lags in the pulse pattern is the true number of lags which are present in the table `LAG_TABLE`. It is NOT the value of the maximum lag. If the maximum lag is 33 but only 22 of the 33 lags are actually calculated then `MPLGS` is 22.

## Syntax

```
#include  “fit_data.h”
```

## Description

The structure `fitdata` 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. |

a_d_drive.o

# do_scan

## Syntax

```
#include  “a_d_drive.h”
int do_scan( pid_t task_id,
             int buffer,
             int bytes,
             int mode,
             int channels);
```

## Description

The `do_scan` function sends a message to the `a_d_drive` task whose process id is *task_id*, requesting that an A/D scan should begin.

The transfer uses DMA buffer number *buffer* with zero being the first buffer. The number of bytes to transfer is *bytes*, using between 1 and 4 channels as specified by *channels*. If *mode* is equal to zero then the transfer is hardware triggered, otherwise software triggering is used.

## 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 | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_AD_FAIL | the task `a_d_drive` failed to complete this command. |

# get_buf_adr

## Syntax

```
#include  “a_d_drive.h”
void *get_buf_adr(pid_t task_id,
                  short int buffer);
```

## Description

The `get_buf_adr` function sends a message to the `a_d_drive` task whose process id is *task_id*, requesting the address of the DMA buffer numbered *buffer*.

## Returns

Returns a pointer to the requested DMA buffer, or (*NULL*) 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 | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_ECHO_ERR | the task `a_d_drive` failed to complete this command. |

# get_buf_num

## Syntax

```
#include  “a_d_drive.h”
int get_buf_num(  pid_t task_id);
```

## Description

The `get_buf_num` function sends a message to the `a_d_drive` task whose process id is *task_id*, requesting the number of DMA buffers the task has.

## Returns

Returns the number of DMA buffers that the `a_d_drive` task has, 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 | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_AD_FAIL | the task `a_d_drive` failed to complete this command. |

# get_buf_size

## Syntax

```
#include  “a_d_drive.h”
int get_buf_size( pid_t task_id);
```

## Description

The `get_buf_size` function sends a message to the `a_d_drive` task whose process id is *task_id*, requesting the size of the DMA buffers the task has.

## Returns

Returns the size in bytes of the DMA buffers that the `a_d_drive` task has, 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 | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_AD_FAIL | the task `a_d_drive` failed to complete this command. |

# get_scan_reset

## Syntax

```
#include  “a_d_drive.h”
pid_t get_scan_reset(  pid_t task_id);
```

## Description

The `get_scan_reset` function sends a message to the `a_d_drive` task whose process id is *task_id*, requesting the process id (**pid**) of the interrupt proxy.

Whenever a DMA transfer is completed the proxy is triggered allowing the driver to detect the end of the transfer.

## Returns

Returns the process id of the proxy 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 | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_ECHO_ERR | the task `a_d_drive` failed to complete this command. |

# get_scan_status

## Syntax

```
#include  “a_d_drive.h”
int get_scan_status(   pid_t task_id);
```

## Description

The `get_scan_status` function sends a message to the `a_d_drive` task whose process id is *task_id*, requesting the status of the last DMA transfer.

## Returns

Returns SCAN_OK if the transfer was successful or SCAN_ERROR if the transfer failed. If an error occurred then (-1) is returned and *raderr* is set.

## Errors

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

| | |
|---|---|
| RADERR_SIGNAL_FAIL | the 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 | The message was interrupted. |
| RADERR_TME_OUT | The message timed out. |
| RADERR_AD_FAIL | The task `a_d_drive` failed to complete this command. |

# scan_reset

## Syntax

```
#include  “a_d_drive.h”
int scan_reset(   pid_t task_id);
```

## Description

The `scan_reset` function kicks the proxy attached to the `a_d_drive` task whose process id is *task_id*. This has the affect of resetting the task when a DMA transfer fails.

## 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 | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_ECHO_ERR | the task `a_d_drive` failed to complete this command. |

dio.o

# init_xt_pid

## Syntax

```
#include  “radops_dio.h”
int init_xt_pid(  pid_t task_id);
```

## Description

The `init_xt_pid` function sends a message to the `radops_dio` task identified by the process id *task_id*, resetting the DIO card and clearing the timing sequence buffers.

## 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 | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_DIO_FAIL | the task `radops_dio` failed to complete this command. |

# send_fclr

## Syntax

```
#include  “radops_dio.h”
int send_fclr(    pid_t task_id,
                  unsigned char id,
                  short int frq_num,
                  short int *freq_table);
```

## Description

The `send_flcr` function sends a message to the `radops_dio` task identified by the process id *task_id*, telling it to perform a clear frequency search using the pulse sequence stored in the buffer *id*.

The sequence is transmitted at each frequency in the table pointed to by *frq_table*. The table has *frq_num* elements.

## 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 | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_DIO_FAIL | the task `radops_dio` failed to complete this command. |

# send_tsg

## Syntax

```
#include  “radops_dio.h”
int send_tsg(     pid_t task_id,
                  unsigned char id);
```

## Description

The `send_tsg` function sends a message to the `radops_dio` task identified by the process id *task_id*, telling it to transmit the pulse sequence stored in the buffer *id*.

## 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 | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_DIO_FAIL | the task `radops_dio` failed to complete this command. |

# send_tsg_no_stat

## Syntax

```
#include  “radops_dio.h”
int send_tsg_no_stat(  pid_t task_id,
                       unsigned char id);
```

## Description

The `send_tsg_no_stat` function sends a message to the `radops_dio` task identified by the process id *task_id*, telling it to transmit the pulse sequence stored in the buffer *id*. The status information is not updated.

## 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 | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_DIO_FAIL | the task `radops_dio` failed to complete this command. |

# set_antenna_pid

## Syntax

```
#include  “radops_dio.h”
int set_antenna_pid(   short int anum,
                   pid_t task_id);
```

## Description

The `set_antenna_pid` function sends a message to the `radops_dio` task identified by the process id *task_id*, setting the antenna number to *anum*.

## 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 | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_DIO_FAIL | the task `radops_dio` failed to complete this command. |

# set_beam_pid

## Syntax

```
#include  “radops_dio.h”
int set_beam_pid( unsigned char beam,
                  pid_t task_id);
```

## Description

The `set_beam_pid` function sends a message to the `radops_dio` task identified by the process id *task_id*, setting the beam number to *beam*.

## 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 | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_DIO_FAIL | the task `radops_dio` failed to complete this command. |

# set_filter_pid

## Syntax

```
#include  “radops_dio.h”
int set_filter_pid(unsigned char filter,
                   pid_t task_id);
```

## Description

The `set_filter_pid` function sends a message to the `radops_dio` task identified by the process id *task_id*, setting the filter mode to *filter*.

## 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 | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_DIO_FAIL | the task `radops_dio` failed to complete this command. |

# set_freq_pid

## Syntax

```
#include  “radops_dio.h”
int set_freq_pid( short int freq,
                  pid_t task_id);
```

## Description

The `set_freq_pid` function sends a message to the `radops_dio` task identified by the process id *task_id*, setting the frequency to *freq*.

## 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 | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_DIO_FAIL | the task `radops_dio` failed to complete this command. |

# set_test_mode_pid

## Syntax

```
#include  “radops_dio.h”
int set_test_mode_pid(
                  unsigned char mode,
                  pid_t task_id);
```

## Description

The `set_test_mode_pid` function sends a message to the `radops_dio` task identified by the process id *task_id*, switching the test mode on and off.

Setting *mode* to zero (0) turns the test mode off, setting it to one (1), turns it on.

## 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 | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_DIO_FAIL | the task `radops_dio` failed to complete this command. |

# set_tsg_pid

## Syntax

```
#include  “radops_dio.h”
int set_tsg_pid(  short int length,
                  unsigned char id,
                  char *code_byte,
                  char *rep_byte,
                  pid_t task_id);
```

## Description

The `set_tsg_pid` function sends a message to the `radops_dio` task identified by the process id *task_id*, downloading a timing sequence into the buffer *id*. Each entry in the array *code_byte* is the is a code to transmit, the corresponding entry in the array *rep_byte* is the number of times to repeat the code. Each array should be *length* bytes long.

## 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 | the 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 | the message was interrupted. |
| RADERR_TME_OUT | The message timed out. |
| RADERR_DIO_FAIL | the task `radops_dio` failed to complete this command. |

# verify_id_pid

## Syntax

```
#include  “radops_dio.h”
int verify_id_pid(unsigned char id,
                  pid_t task_id);
```

## Description

The `verify_id_pid` function sends a message to the `radops_dio` task identified by the process id *task_id*, verifying that the pulse sequence identified by *id* exits.

## Returns

Returns the length of the pulse sequence if it exists, zero (0) if it does not, 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 | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_DIO_FAIL | the task `radops_dio` failed to complete this command. |

get_fit.o

# get_fit

## Syntax

```
#include  “radops.h”
#include  “fitdata.h”
#include  “get_fit.h”
int get_fit( char *fit_buf_name,
             struct fitdata *fit_data);
```

## Description

The `get_fit` function sends a message to the `fit_buffer` task identified by *fit_buf_name*, requesting the most recent block of fitted data.

If the data is available it is stored in the structure pointed to by *fit_data*.

If *fit_buf_name* is NULL then the default name of “`/fit_buffer`” will be used.

## Returns

Returns the record number of the block of data returned. This corresponds to the total number of integration periods processed by `fitacf` since the last file was opened. If an error occurs (-1) is returned and *raderr* is set.

## Errors

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

| | |
|---|---|
| RADERR_SIGNAL_FAIL | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_FBUF_ERR | the task `fit_buffer` failed to complete this command. |

get_status.o

# get_status

## Syntax

```
#include  “radops.h”
#include  “get_status.h”
int get_status(struct rawdata*raw_data,
            int clear);
```

## Description

The `get_status` function sends a message to the `radops_dio` task requesting the AGC and low power status words.

The values are stored in the raw data structure pointed to by *raw_data*. If *clear* is not equal to zero then the status words are reset to zero once they have been read.

## 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 `radops_dio` is not running. |
| RADERR_SIGNAL_FAIL | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_DIO_FAIL | the task `radops_dio` failed to complete this command. |

# get_status_pid

## Syntax

```
#include  “radops.h”
#include  “get_status.h”
int get_status_pid(
            struct rawdata*raw_data,
            int clear,
            pid_t task_id);
```

## Description

The `get_status_pid` function sends a message to the `radops_dio` task identified by the process id *task_id*, requesting the AGC and low power status words.

The values are stored in the raw data structure pointed to by *raw_data*. If *clear* is not equal to zero then the status words are reset to zero once they have been read.

## 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 | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_DIO_FAIL | the task `radops_dio` failed to complete this command. |

log_error.o

# log_error

## Syntax

```
#include  “log_error.h”
int log_error(    char *errlog,
                  char *name,
                  char *buffer);
```

## Description

The `log_error` function sends a message to the `errlog` task identified by *errlog*, containing the error message string pointed to by *buffer*. If *name* is not NULL the error log will include the string together with the error message.

If *errlog* is NULL the default name of “`/errlog`” will be used.

## Returns

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

message.o

## Syntax

```
#include  “message.h”
int message( char *task,
             double time,
             void *smsg,
             void *rmsg,
             unsigned snbytes,
             unsigned rnbytes);
```

## Description

The `message` function sends a message pointed to by *smsg* to the task registered under the name *task*. Any reply is placed in the buffer *rmsg*. The size of the sent message will be *snbytes* while the size of the reply will be truncated to a maximum of *rnbytes*.

If *time* equals zero then the calling process will wait indefinitely for a reply. If time is greater than zero then the process will wait *time* seconds before returning.

The number of bytes send will be the minimum of that specified by the sender and receiver.

## Returns

Returns zero (0) on success. If an error occurs then (-1) is returned and *raderr* is set.

## Errors

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

| | |
|---|---|
| RADERR_NO_TASK | no task is registered with that name. |
| RADERR_SIGNAL_FAIL | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |

# message_array

## Syntax

```
#include  “message.h”
int message_array(char *task,
                  double time,
                  void **smsg,
                  void **rmsg,
                  unsigned *snbytes,
                  unsigned *rnbytes);
```

## Description

The `message_array` function sends an array of messages pointed to by the array *smsg* to the task registered under the name *task*. Any replies are placed in the buffers pointed to by the array *rmsg*. The size of each sent message will be taken from the corresponding entry in the array *snbytes* while the size of the reply will be truncated to a maximum of the corresponding entry in the array *rnbytes*.

If *time* equals zero then the calling process will wait indefinitely for a reply. If *time* is greater than zero then the process will wait time seconds before returning.

The number of bytes send will be the minimum of that specified by the sender and receiver.

The function scans the arrays *smsg* and *rmsg*, which must be NULL terminated to determine how many buffers to send and receive.

## Returns

Returns zero (0) on success. If an error occurs then (-1) is returned and *raderr* is set.

## Errors

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

| | |
|---|---|
| RADERR_NO_TASK | no task is registered with that name. |
| RADERR_SIGNAL_FAIL | the time out signal could not be claimed. The signal *SIGUSR1* is generated after a time-out to interrupt the message. |
| RADERR_NO_MEM | memory could not be allocated to store the messages. |
| 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 | the message was interrupted. |
| RADERR_TME_OUT | The message timed out. |

# message_pid

## Syntax

```
#include  “message.h”
int message_pid(  pid_t task_id,
                  double time,
                  void *smsg,
                  void *rmsg,
                  unsigned snbytes,
                  unsigned rnbytes);
```

## Description

The `message_pid` function sends a message pointed to by *smsg* to the task with process id *task_id*. Any reply is placed in the buffer *rmsg*. The size of the sent message will be *snbytes* while the size of the reply will be truncated to a maximum of *rnbytes*.

If *time* equals zero then the calling process will wait indefinitely for a reply. If time is greater than zero then the process will wait *time* seconds before returning.

The number of bytes send will be the minimum of that specified by the sender and receiver.

## Returns

Returns zero (0) on success. If an error occurs then (-1) is returned and *raderr* is set.

## Errors

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

| | |
|---|---|
| RADERR_SIGNAL_FAIL | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |

# message_pid_array

## Syntax

```
#include  “message.h”
int message_pid_array(
                  pid_t task_id,
                  double time,
                  void **smsg,
                  void **rmsg,
                  unsigned *snbytes,
                  unsigned *rnbytes);
```

## Description

The `message_array` function sends an array of messages pointed to by the array *smsg* to the task with process id *task_id*. Any replies are placed in the buffers pointed to by the array *rmsg*. The size of each sent message will be taken from the corresponding entry in the array *snbytes* while the size of the reply will be truncated to a maximum of the corresponding entry in the array *rnbytes*.

If *time* equals zero then the calling process will wait indefinitely for a reply. If *time* is greater than zero then the process will wait time seconds before returning.

The number of bytes send will be the minimum of that specified by the sender and receiver.

The function scans the arrays *smsg* and *rmsg*, which must be NULL terminated to determine how many buffers to send and receive.

## Returns

Returns zero (0) on success. If an error occurs then (-1) is returned and *raderr* is set.

## Errors

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

| | |
|---|---|
| RADERR_SIGNAL_FAIL | the 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_NO_MEM | memory could not be allocated to store the messages. |
| RADERR_MSG_FAIL | the message was interrupted. |
| RADERR_TME_OUT | The message timed out. |

# task_id

## Syntax

```
#include  “message.h”
pid_t task_id(    char *task);
```

## Description

The `task_id` function returns the process id (**pid**) of the task registered under the name *task*.

## Returns

Returns the process id on success. If the task cannot be found then (-1) is returned and *raderr* is set.

## Errors

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

| | |
|---|---|
| RADERR_NO_TASK | no task is registered with that name. |

option.o

## Syntax

```
#include  “option.h”
void process_file(
            FILE *fp,
            struct option *opt,
            void(*opterr)(char *));
```

## Description

The `process_file` function reads option switched from the file pointed to by *fp*. The file should contain space separated command options.

A command line options consists of a list of option switches, strings that start with the character “`-`”. Each switch has an optional argument following it witch can be either an integer, floating point number, or another string:

```
-beam 8 -debug -freq 12.34 -filename test.dat
```

The structure `option` contains the following members:

| | |
|---|---|
| `char *optname;` | the option switch to identify. |
| `char type;` | the type of the argument following option. |
| `int set;` | a flag set when the option is located. |
| `void *ptr;` | the address to store the argument. |

Option switches are read from the file and checked against the array opt. If a match is found with an *optname* member of one of the structures in *opt*,The corresponding *set* member is set to 1 to indicate the option was found.

If the *type* character is one of the recognized types, the next string in the command line is assumed to be an argument for the option switch.

There are five types:

| | |
|---|---|
| ‘s’ | short int. |
| ‘l’ | long int. |
| ‘f’ | float. |
| ‘d’ | double. |
| ‘t’ | string. |

If the argument is one of the numeric types it is converted from the *argv* string and stored at the location pointer to by the *ptr* member. If the argument is a string, then ptr is set to point to the appropriate element of *argv*.

If *type* is not one of the recognized types, the option switch is assumed to take no arguments.

If an unrecognized option string is found it is passed to the function pointed to by *opterr*. This function should report the error to the user and take the appropriate action.

## Returns

None.

# process_option

## Syntax

```
#include  “option.h”
int process_option(
            int argc,
            char *argv[],
            struct option *opt,
            void(*opterr)(char *));
```

## Description

The `process_option` function processes the command line arguments according to the option table pointed to by *opt*.

The argument *argv* is an array of strings extracted from the command line, the first string is the program name and subsequent strings are the space separated options. The number of strings contained in the array is defined by *argc*.

A command line options consists of a list of option switches, strings that start with the character “`-`”. Each switch has an optional argument following it which can be either an integer, floating point number, or another string:

```
-beam 8 -debug -freq 12.34 -filename test.dat
```

If the final string on the command line is not part of an option switch it is treated at a filename of a file containing more option switches.

The structure `option` contains the following members:

| | |
|---|---|
| `char *optname;` | the option switch to identify. |
| `char type;` | the type of the argument following option. |
| `int set;` | a flag set when the option is located. |
| `void *ptr;` | the address to store the argument. |

The array *argv* is scanned for option switches which are checked against the array opt. If a match is found with the *optname* member of one of the structures in *opt*, The corresponding *set* member is set to 1 to indicate the option was found.

If the *type* character is one of the recognized types, the next string in the command line is assumed to be an argument for the option switch.

There are five types:

| | |
|---|---|
| `s` | short int. |
| `l` | long int. |
| `f` | float. |
| `d` | double. |
| `t` | string. |

If the argument is one of the numeric types it is converted from the *argv* string and stored at the location pointer to by the *ptr* member. If the argument is a string, then ptr is set to point to the appropriate element of *argv*.

If *type* is not one of the recognized types, the option switch is assumed to take no arguments.

If an unrecognized option string is found it is passed to the function pointed to by *opterr*. This function should report the error to the user and take the appropriate action.

## Returns

Returns the index of *argv* after the last successfully processed option switch. If the last string on the command line is a filename, this will be one (1) less than *argc*.

read_clock.o

# read_clock

## Syntax

```
#include  “read_clock.h”
void read_clock(  int *year,
                  int *mon,
                  int *day,
                  int *hour,
                  int *minute,
                  int *second,
                  int *msec,
                  int *usec);
```

## Description

The `read_clock` function reads the system clock. The system clock is automatically calibrated against the GPS clock by the driver `gps_clock`.

## Returns

The current time, accurate to the nearest second is returned in the variables pointed to by *year*, *mon*, *day*, *hour*, *minute*, *second*. At present, the values of *msec* and *usec* are set to zero.

read_fit.o

# read_fit

## Syntax

```
#include  “read_fit.h”
int read_fit(
          FILE *fp,
          struct fitdata *fit_data);
```

## Description

The `read_fit` function reads a block of fitted data from the file pointed to by *fp* into the structure pointed to by *fit_data*.

## Returns

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

read_raw.o

# read_raw

## Syntax

```
#include  “read_raw.h”
int read_raw(
          FILE *fp,
          struct rawdata *raw_data);
```

## Description

The `read_raw` function reads a raw ACF record from the file pointed to by *fp* into the structure pointed to by *raw_data*.

## Returns

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

# read_raw_data

## Syntax

```
#include  “read_raw.h”
int read_raw_data(
          FILE *fp,
          struct rawdata *raw_data);
```

## Description

The `read_raw_data` function reads a raw ACF record from the file pointed to by *fp* into the structure pointed to by *raw_data*.

Only the raw ACFs are read from the file and the Radar parameter block remains unaffected.

## Returns

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

sample.o

# add_data

## Syntax

```
#include  “sample.h”
int add_data(struct beam_list **table,
             int beam,
             int range,
             enum param_code);
```

## Description

The `add_data` function is used to create a table that can be used to sample the data produced by fitacf across a scan. The table contains a list of range-beam coordinates and a list of parameters to record.

Each time the `transform_data` function is called with a block of fitted data, the table is inspected and a record is made of the data for the appropriate ranges and parameters.

To construct a table, multiple calls are made to `add_data` with the sampling coordinates specified by *beam* and *range*, and the parameter to record with *param_code*, which can be one of :

| | |
|---|---|
| PARAM_qflg | quality flag. |
| PARAM_gsct | ground scatter flag. |
| PARAM_p_0 | lag 0 power. |
| PARAM_p_s | sigma power. |
| PARAM_p_l | lambda power. |
| PARAM_w_l | lambda width. |
| PARAM_w_s | sigma width. |
| PARAM_v | velocity. |
| PARAM_v_err | velocity error. |
| PARAM_sdev_l | standard deviation of lambda fit. |
| PARAM_sdev_s | standard deviation of sigma fit. |
| PARAM_sdev_phi | standard deviation of phase fit. |

The structure `beam_list` is a linked list of all the beams to sample across a scan. It has the following members:

| | |
|---|---|
| `short int beam_no;` | beam number to sample. |
| `short int range_max;` | maximum range for this beam. |
| `struct range_list *table;` | pointer to a table of ranges. |
| `struct beam_list *next;` | pointer to the next entry in the linked list, NULL terminated. |

The structure `range_list` is a linked list of all the ranges to sample within a beam. It has the following members:

| | |
|---|---|
| `short int range;` | the range gate to sample. |
| `long int distance;` | the range in kilometers. |
| `struct param_list *table;` | pointer to a table of parameters. |
| `struct range_list *next;` | pointer to the next entry in the linked list, NULL terminated. |

The structure `param_list` is a list of the parameters to sample at a particular range. It has the following members:

| | |
|---|---|
| `enum param_code code;` | the parameter to sample. |
| `int total;` | total number of times a sample has been taken. |
| `struct time_list *table;` | pointer to a list of samples. |
| `struct param_list *next;` | pointer to the next entry in the linked list, NULL terminated. |

The structure `time_list` is a list of samples. It has the following members:

| | |
|---|---|
| `union {` | |
| `  double value;` | a floating point parameter. |
| `  int flag;` | a boolean flag. |
| `} data;` | union to store the sampled parameter. |
| `struct time_list *next;` | pointer to the next entry in the linked list, NULL terminated. |

Each member of the `time_list` linked list is a sample from one scan. The list is arranged in time order with the first entry being from the most recent scan. The length of the list, and consequently how many scans are stored, is dependent on the number of times `add_data` is called with that particular combination of beam, range and parameter.

## Returns

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

# remove_table

## Syntax

```
#include  “sample.h”
void remove_table(
             struct beam_list **table);
```

## Description

The `remove_table` function frees the memory uses by the sampling table pointed to be *table*.

## Returns

None.

# transform_data

## Syntax

```
#include  “sample.h”
int transform_data(
            struct fitdata *fit_data,
            struct beam_list **table);
```

## Description

The `transform_data` function extracts the appropriate parameters from the fitted data structure pointed to by *fit_data* and insert them into the sampling table pointed to by *table*.

## Returns

Returns zero (0) on success, or (-1) if the record cannot be processed occurs.

task_write.o

# task_close

## Syntax

```
#include  “task_write.h”
int task_close(   char *task,
                  short int year,
                  short int month,
                  short int day,
                  short int hour,
                  short int minute,
                  short int second);
```

## Description

The `task_close` function sends a message to the program registered under the name *task*, requesting that any open files are closed.

The task will close the files when it receives data with a time stamp later than the date and time specified by *year*, *month*, *day*, *hour*, *minute* and *second*.

## Returns

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

## Errors

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

| | |
|---|---|
| RADERR_NO_TASK | no task is registered with that name. |
| RADERR_SIGNAL_FAIL | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_TASK_FAIL | the task failed to complete this command. |

# task_open

## Syntax

```
#include  “task_write.h”
int task_open(    char *task,
                  short int year,
                  short int month,
                  short int day,
                  short int hour,
                  short int minute,
                  short int second);
```

## Description

The `task_open` function sends a message to the program registered under the name *task*, requesting that files should be opened.

The task will open the files when it receives data with a time stamp later than the date and time specified by *year*, *month*, *day*, *hour*, *minute* and *second*.

## Returns

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

## Errors

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

| | |
|---|---|
| RADERR_NO_TASK | no task is registered with that name. |
| RADERR_SIGNAL_FAIL | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_TASK_FAIL | the task failed to complete this command. |

# task_quit

## Syntax

```
#include  “task_write.h”
int task_quit(char *task);
```

## Description

The `task_quit` function sends a message to the program registered under the name *task*, requesting it to shut down and exit.

## Returns

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

## Errors

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

| | |
|---|---|
| RADERR_NO_TASK | no task is registered with that name. |
| RADERR_SIGNAL_FAIL | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_TASK_FAIL | the task failed to complete this command. |

# task_write_aux

## Syntax

```
#include  “task_write.h”
int task_write_aux(
          char *task,
          void *block,
          int length);
```

## Description

The `task_write_aux` function sends a message to the program registered under the name *task,* containing a block of auxiliary data, pointed to by *block*, of length *size* bytes.

The contents of the block of data depends on the receiving task. It is used to pass parameters and data that are note part of either the `rawdata` or `fitdata` structures to those tasks that require it.

## Returns

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

## Errors

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

| | |
|---|---|
| RADERR_NO_TASK | no task is registered with that name. |
| RADERR_SIGNAL_FAIL | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_TASK_FAIL | the task failed to complete this command. |

# task_write_fit

## Syntax

```
#include  “task_write.h”
int task_write_fit(
          char *task,
          struct fitdata *fit_data,
          short int flag);
```

## Description

The `task_write_fit` function sends a message to the program registered under the name *task,* containing the fitted data pointed to by *fit_data*.

If *flag* is set to one (1) then the data will be stored in a file, otherwise it will be processed but not recorded.

## Returns

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

## Errors

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

| | |
|---|---|
| RADERR_NO_TASK | no task is registered with that name. |
| RADERR_SIGNAL_FAIL | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_TASK_FAIL | the task failed to complete this command. |

# task_write_raw

## Syntax

```
#include  “task_write.h”
int task_write_raw(
          char *task,
          struct rawdata *raw_data,
          short int flag);
```

## Description

The `task_write_raw` function sends a message to the program registered under the name *task,* containing the raw ACF data pointed to by *raw_data*.

If *flag* is set to one (1) then the data will be stored in a file, otherwise it will be processed but not recorded.

## Returns

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

## Errors

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

| | |
|---|---|
| RADERR_NO_TASK | no task is registered with that name. |
| RADERR_SIGNAL_FAIL | the 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 | the message was interrupted. |
| RADERR_TME_OUT | the message timed out. |
| RADERR_TASK_FAIL | the task failed to complete this command. |

user_int.o

# scheduled

## Syntax

```
#include  “user_int.h”
int scheduled(void);
```

## Description

The `scheduled` function tests whether the control program was started by the scheduler.

## Returns

Returns one (1) if the program was started by the scheduler, or zero (0) otherwise.

# register_program

## Syntax

```
#include  “user_int.h”
int register_program(
          char *schedule_name,
          char *control_name);
```

## Description

The `register_program` attempts to register the control program with the operating system under the name *control_name*. It also attempts to locate the scheduler registered under the name *schedule_name*.

If *schedule_name* or *control_name* is NULL the appropriate default names of “`/schedule`”, and “`/control_program`” will be used.

## Returns

Returns one (1) if successful, or (-1) if an error occurred.

## Syntax

```
#include  “user_int.h”
int user_int(
            struct rawdata *raw_data,
            char *variables,
            …);
```

## Description

The `user_int` function checks to see if the task alter is running and wishes to change the radar parameters stored in the structure pointed to by *raw_data*.

An extra set of user defined parameters can also be sent to the task using the *variables* string and the optional arguments.

Within alter commands can be entered at the shell shown by the “>” prompt.

The shell supports four commands :

The shell supports three commands :

```
go
show
<variable> = <value>
```

`go`

Typing “`go`” will send the altered parameters back to the control program so that they will take affect at the start of the next integration period.

`show [`*variable_name..*`]`

Typing “`show`” will list the values of the specified variables. If no variable names are listed then the entire set will be shown

`<`*variable*`> = <`*value*`>`

A new value is assigned to a variable by using the “=” sign:

```
bmnum=12
combf=Hello world
```

There should be no spaces between the value, the “=” sign, or the variable name.

The radar parameters that can be altered 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. |

The string *variables* is an identifier used to classify the optional argument list that follows it. The string consists of a set of space separated labels and codes to identify the name and type of the pointers that make up the variable argument list.

Four codes are recognized :

| | |
|---|---|
| i | short int. |
| l | long int. |
| f | float. |
| d | double. |
| s | string. |

Each label is followed by a type code :

```
"a_short_integer i  a_string s a_float f"
```

Would specify that the three pointers following the variables string were of types : *short int **, *char **, and *float **. They would be given the labels : "a_short_integer", "a_string", and "a_float".

Typing "show" in the shell of the alter task would list these three extra variables as well as the radar parameters.

The function user_int is also used to detect when the scheduler wishes to terminate the control program. If the function returns one (1), the control program should shut down as soon as possible.

## Returns

Returns one (1) if the control program should shut down, or zero (0) otherwise.

# The Task Library

add_point.o

# add_point

## Syntax

```
#include “util/add_point.h”
int add_point(    struct parray *a,
                  unsigned char op,
                  int x,
                  int y,
                  int wdt,
                  int hgt);
```

## Description

The function `add_point` adds a point to a clipping table.

A clipping table is a list of plotting commands for drawing complex outlines. Each entry in the table has an associated action, either plot or move. If the action is to plot, a line is drawn from between the last point and the current point.

When a point is added to the table by the function `add_point` a check is performed to ensure that the coordinates are within the boundaries of the screen. If they are not the point is automatically converted to a move. If the previous point was also off screen then it is replaced by the current point.

The coordinates of the point are defined by *x* and *y*. If *op* is set to zero then the action to associate with the point is to plot a line, otherwise a move is performed. The coordinate limits range from zero (0) to the width and height of the screen as defined by *wdt* and *hgt*.

The clipping table is stored in the structure pointed to by *a*. The structure `parray` has the following members:

| | |
|---|---|
| `long int nelem;` | the number of points in the table. |
| `struct apnt *pnt;` | a pointer to an array containing the points. |

The structure `apnt` has the following members:

| | |
|---|---|
| `unsigned char op;` | the action to perform: 0=draw a line, otherwise move. |
| `short int x;` | the x coordinate. |
| `short int y;` | the y coordinate. |

## Returns

Returns zero (0) if the point was successfully added to the table or (-1) if an error occurred.

cnv_time.o

# get_time

## Syntax

```
#include “util/cnv_time.h”
void get_time(    long int time,
                  int yr,
                  int *mo,
                  int *dy,
                  int *hr,
                  int *mn,
                  int *sc);
```

## Description

The `get_time` function converts the time expressed as seconds passed the start of the year into day, month, hour, minute, and second.

The variable *time* is the number of seconds that have elapsed since midnight on the first of January in the year *yr*. The converted time, expressed as month, day, hour minute and second, is stored in the variables pointed to by *mo*, *dy*, *hr*, *mn*, and *sc*.

## Returns

None.

# year_sec

## Syntax

```
#include “util/cnv_time.h”
long int year_sec(int yr,
                  int mo,
                  int dy,
                  int hr,
                  int mn,
                  int sec);
```

## Description

The `year_sec` function converts the time expressed as year, month, day, hour, minute, and second into the number of seconds that have elapsed from the start of the year.

The time expressed as month, day, hour minute and second is defined by the variables *yr*, *mo*, *dy*, *hr*, *mn*, and *sec*.

## Returns

Returns the number of seconds that have elapsed since midnight on the first of January.

cnvt_coord.o

## Syntax

```
#include “geo/geo.h”
void cubic(  int center,
             int bcrd,
             int rcrd,
             struct rpos *pos,
             int lagfr,
             int smsep,
             int height,
             double *x,
             double *y,
             double *z);
```

## Description

The function `cubic` converts a Radar range/beam coordinate pair into a geographic location expressed in normalized Cartesian coordinates.

If variables *bcrd* and *rcrd* define the beam and range that the conversion is to be performed on. If the flag center is not equal to zero the position is calculated for the center of the range cell, otherwise the left hand lower edge of the cell is used.

The location of the Radar is defined by the structure pointed to by *rpos*. In addition, the calculation requires the lag to first range in milliseconds as defined by *lagfr*, and the sample separation, also in milliseconds as defined by *smsep*.

The variable *height* defines the height above in kilometers above the Radar at which the position is to be calculated. However, if height is less than 90, it is assumed to be an angle of elevation from the Radar site.

The converted position expressed as normalized Cartesian coordinates is stored in the variables pointed to by *x*, *y*, and *z*. The coordinate system maps positions on the Earth’s surface to a sphere of radius one (1).

## Returns

None.

# geographic

## Syntax

```
#include “geo/geo.h”
void geographic(  int center,
                  int bcrd,
                  int rcrd,
                  struct rpos *pos,
                  int lagfr,
                  int smsep,
                  int height,
                  double *rho,
                  double *lat,
                  double *lng);
```

## Description

The function `geographic` converts a Radar range/beam coordinate pair into a geographic location expressed in longitude and latitude.

If variables *bcrd* and *rcrd* define the beam and range that the conversion is to be performed on. If the flag center is not equal to zero the position is calculated for the center of the range cell, otherwise the left hand lower edge of the cell is used.

The location of the Radar is defined by the structure pointed to by *rpos*. In addition, the calculation requires the lag to first range in milliseconds as defined by *lagfr*, and the sample separation, also in milliseconds as defined by *smsep*.

The variable *height* defines the height in kilometers above the Radar at which the position is to be calculated. However, if height is less than 90, it is assumed to be an angle of elevation from the Radar site.

The converted position expressed as distance from the center of the Earth and latitude and longitude are stored in the variables pointed to be *rho*, *lat*, and *lng*.

## Returns

None.

# load_rpos

## Syntax

```
#include “geo/geo.h”
int load_rpos(    char *fname,
                  int year,
                  long int yr_sec);
```

## Description

The function `load_rpos` loads a set of Radar positions from the file identified by *fname*. The position of the Radar at year *year*, and *yr_sec* seconds within that year are loaded.

The position file is a plain text file with each line containing a single entry for a Radar at a specific time and date. If a line begins with the character “#” it is treated as a comment and ignored.

Each line consists of a space separated list of data that defines the Radar position. The data are:

| | |
|---|---|
| Station ID | the station identifier code. |
| Year | date of the position. |
| Year Second | seconds passed year of the position. |
| Latitude | latitude of the center tower of the Radar. |
| Longitude | longitude of the center tower of the Radar. |
| Altitude | altitude in kilometers of the Radar above sea-level. |
| Bore Site | line of sight of the radar. |
| Beam Separation | angular separation of the beams in degrees. |
| V direction | |
| Attenuation | |
| Interferometer Position | |
| Receiver Rise Time | rise time of the receiver in micro-seconds. |

## Returns

Returns zero (0) on success, or (-1) otherwise.

## Syntax

```
#include “geo/geo.h”
struct rpos *radar_pos(     int station);
```

## Description

The function `radar_pos` returns a pointer to a structure of the type `rpos` containing the position of the Radar with station identifier *station*.

The structure `rpos` has the following members:

| | |
|---|---|
| `double gdlat;` | latitude of the center tower of the Radar. |
| `double gdlon;` | longitude of the center tower of the Radar. |
| `double boresite;` | line of sight of the radar. |
| `double bmwidth;` | angular separation of the beams in degrees. |
| `double rxris;` | rise time of the receiver in micro-seconds. |

## Returns

Returns a pointer to a structure of type `rpos` containing the location of the Radar, or `NULL` if an error occurred.

echo_util.o

# echo_register

## Syntax

```
#include “echo/echo_util.h”
int echo_register(     char *name,
                       char *echo_name,
                       char flag);
```

## Description

The function `echo_register` registers a task with `echo_data` so that it will receive Radar data.

The function will attempt to connect with the version of `echo_data` registered under the name *echo_name*. If this string is NULL, then the default name of “`/echo_data`” is used.

The string *name* is used by `echo_data` when it reports activities involving the task.

The variable flag defines what kind of data will be forwarded to the task. Different combinations of data can be forwarded by combining the flags using a bitwise OR:

```
echostat=echo_register( “mytask”,
   NULL,
   PASS_RAW  | PASS_FIT );
```

The flags are:

| | |
|---|---|
| PASS_AUX | pass auxiliary data. |
| PASS_FIT | pass data from `fitacf`. |
| PASS_RAW | pass the raw ACF data. |

## Returns

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

file_io.o

## Syntax

```
#include “file_io.h”
unsigned char *decode_msg(int tid,char
          msg,char *rmsg,int *flag);
```

## Description

The `decode_msg` function decodes one of the standard data transfer messages sent from a control program by the `task_write` library.

The function decodes the data component of a message given the header character, *msg*, and the task id (**tid**) of the task that sent it, *tid*.

The following code demonstrates how to extract the header when a message is received:

```
  while (1) { /* start of loop */

    /* when a message is received extract
       the first byte */

    tid=Receive(0,&msg,sizeof(char));

    /* decode the message */

    data=decode_msg(tid,msg,&rmsg,&flag);


   /* process the data (if any) and reply to
      the message */
```

After the function has been called, the character pointed to by *rmsg* contains the message to send in reply. The possible replies are:

| | |
|---|---|
| TASK_OK | the message was successfully decoded. |
| TASK_ERR | an error occurred when decoding the message. |
| UNKNOWN_MESSAGE | the message could not be recognized. |

If the message could not be recognized or an error occurred, the calling task should reply to the message immediately and return:

```
  /* process the data (if any) and reply to the
     message*/

  if ( (rmsg==UNKNOWN_TYPE) ||
       (rmsg==TASK_ERR) ) {
    Reply(reply_tid,&rmsg,sizeof(char));
    continue; /* skip to the end of the loop */
  }


  /* Process the data here */
```

If the message contains a block of data then the function will return a pointer to it, otherwise `NULL` is returned.

After the message has been decoded the integer pointed to by *flag* contains a set of Boolean flags that indicate the file operations to perform.

The flags are:

| | |
|---|---|
| OPEN_BIT | open a new file before processing the block of data. |
| CLOSE_BIT | if a file is open then close it before processing this block of data. |
| WRITE_BIT | if a file is open then write the processed data to it.. |

The `task_write` controls file operations by sending a message containing the time at which files should be opened and closed. The `decode_msg` function records these times and compares them with the time-stamps associated with the Radar data blocks. At the appropriate time the function sets the flags in the *flag* variable.

The calling program should test for each flag and take the appropriate actions:

```
if ((fp !=NULL) && (flag & CLOSE_BIT))
   fclose(fp) /* close the file */

if (flag & OPEN_BIT)
   fp=fopen(fname,"w"); /* open a new file */

/* process data */

if ((flag & WRITE_BIT) && (fp !=NULL))
  /* write record */
```

## Returns

Returns a pointer to a block of data decoded from the message or NULL. The message to send in reply is stored at *rmsg*, and the file operations to perform are stored at *flag*.

# open_file

## Syntax

```
#include “file_io.h”
char *open_file(char *pathenv,struct
          radops_parms *prm,char *ext,int
          mode,char *sfx,int flag);
```

## Description

The `open_file` function attempts to create an empty file with a unique filename according to the SuperDARN naming convention of :

yymmddhhs[x].eee

Where:

| | |
|---|---|
| yy | year XXyy. |
| mm | month. |
| dd | day. |
| hh | hour. |
| s | station identifier letter. |
| x | optional suffix to preserve unique filenames (A-Z,a-z). |
| eee | file extension. e.g. FIT,DAT. |

The string *pathenv* specifies the pathname of the directory to store the file in and the string *ext* contains the extension or file-type to apply.

The date to associate with the file is taken from the radar parameter block pointed to by *prm*.

If the parameter *mode* is set to zero (0), the file mode bits will be overwritten so that the file can be read and written by all.

The character pointer *suffix* is used to store the suffix character that must sometimes be applied to the filename to ensure a unique name.

The *flag* parameter controls the operation of the function, it has the possible values of:

| | |
|---|---|
| 0 | check for the existence of a file with this filename. If it already exists step through the possible suffixes until a unique name is found. |
| 1 | force the use of the supplied suffix and if the file already exists overwrite it with an empty file. |
| 2 | force the use of the supplied suffix, and if the file already exists leave it intact. |

## Returns

Returns the full pathname of the file created or `NULL` if an error occurred.

# filer.o

## Syntax

```
#include  “filer/filer.h”
int filer(   char *title,
             char *path,
             char *dir_name,
             char *file_name);
```

## Description

The `filer` function draws and maintains a file selection window on the terminal. The string *title* is printed above the window. The initial directory displayed in the window is specified by *dir_name*, the initial file selected is defined by *file_name*. The final complete path name of the file is stored in the string *path* when the function returns.

The file selection window is split into two, the top half of the window shows the full path name of the selected file. The bottom half of the window shows the contents of the directory that the file is in.

Files can be selected by pressing the up and down arrow keys to scroll through the contents of the directory, or by clicking on a file with the mouse. The name of the file can be changed by typing a new name in the top half of the window.

A different directory can be selected by typing its name in the top half of the window and pressing return.

You must call the `term_load` function before attempting to call this function.

## Returns

Returns the key code that caused the filer to terminate, either “<**enter**>” or “<**escape**>”. The string path will contain the complete *pathname* of the selected file.

# leaf_name

## Syntax

```
#include  “filer/filer.h”
char *leaf_name(char *path);
```

## Description

The `leaf_name` function extracts the leaf name of a complete file path.

`“/radops/src/task_lib/filer.c”` would be truncated to `“filer.c”`.

## Returns

Returns a pointer to the extracted leaf name.

gbuf_util.o

# get_display

## Syntax

```
#include “gbuf/gbuf_util.h”
int get_display(char *fname,struct gbuf
          **g,struct image_header **hdr);
```

## Description

The `get_display` function attempts to claim one of the frame buffers used by the `gbuf` driver.

The driver maintains a set of frame buffers in shared memory. Each buffer is identified by a filename under the “`/dev/shmem`” directory. The filenames are of the form “`Display.con`*x*”, where *x* is the console number for the display.

When the function is called it will attempt to access the frame buffer identified by *fname*. If successful a pointer to a structure containing information about the display is returned in *hdr*, and the pointer to the actual frame buffer is returned in *g*.

The structure image_header has the following members:

| | |
|---|---|
| `long int size;` | the total size of the shared memory buffer, including this header. |
| `pid_t pid;` | the process ID of the `gbuf` driver that created the buffer. |
| `short int pflag;` | the palette flag. If `pflag` is not equal to zero then the palette registers will be set whenever the console becomes active. |
| `long int pal_reg[256];` | the contents of the palette registers for this display. |
| `short int numxpixels;` | the width of the display in pixels. |
| `short int numypixels;` | the height of the display in pixels. |
| `short int bp;` | the number of bits per pixel for the display. |

## Returns

Returns zero (0) on success and sets the pointer hdr and g. If an error occurs (-1) is returned.

# refresh_display

## Syntax

```
#include “gbuf/gbuf_util.h”
int refresh_display(struct image_header
          *hdr);
```

## Description

The function `refresh_display` signals the `gbuf` driver that the display identified by *hdr* should be redisplayed on the console.

The `gbuf` program maintains a frame buffer on which all graphics operations are performed. However, operations are only reflected on the console screen when it first becomes active and after the `refresh_display` function is called.

## Returns

Returns zero (0) on success, or (-1) if the refresh failed.

graph_lib.o

# bgcolor

## Syntax

```
 #include “graph_lib.h”
void bgcolor(     int c);
```

## Description

The function `bgcolor` sets the current background colour to *c*.

## Returns

None.

# clg

## Syntax

```
 #include “graph_lib.h”
void clg();
```

## Description

The function `clg` clears the contents of the currently active graphics buffer to the background colour.

## Returns

None.

# cnv_to_ppm

## Syntax

```
 #include “graph_lib.h”
int cnv_to_ppm(   FILE *fp);
```

## Description

The function `cnv_to_ppm` converts the current graphics buffer inta a **P**ortable **P**ix**M**ap (**PPM**) image which is written to the open file pointed to by fp.

The PPM format is recognized by a large number of graphics packages.

## Returns

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

## Syntax

```
 #include “graph_lib.h”
void color(  int c);
```

## Description

The function `color` sets the current foreground colour to *c*.

## Returns

None.

# copy_gbuf

## Syntax

```
 #include “graph_lib.h”
int copy_gbuf(    struct gbuf *ga,
                  struct gbug *gb);
```

## Description

The `copy_gbuf` function copies the contents of the graphics buffer pointed to by *gb* to the one pointed to by *ga*. The buffers must be of the same size.

## Returns

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

# copy_pixel

## Syntax

```
 #include “graph_lib.h”
void copy_pixel(  struct gbuf *ga,
                  struct gbuf *gb,
                  int x,
                  int y);
```

## Description

The `copy_pixel` function copies a single pixel from the graphics buffer pointed to by *gb* to the one pointed to by *ga*. The buffers must be of the same size.

The pixel is taken from the location specified by *x* and *y*.

## Returns

None.

# copy_polygon

## Syntax

```
 #include “graph_lib.h”
void copy_polygon(     struct gbuf *ga,
                       struct gbuf *gb,
                       int num,
                       int *x,
                       int *y);
```

## Description

The `copy_polygon` function copies a polygon from the graphics buffer pointed to by *gb* to the one pointed to by *ga*. The buffers must be of the same size.

The arrays *x* and *y* define the vertices of the polygon. The number of elements in each array must be equal to *num* which is the number of vertices in the polygon.

## Returns

None.

# draw

## Syntax

```
 #include “graph_lib.h”
void draw(   int x,int y);
```

## Description

The function `draw` plots a line from the current graphics cursor to the position specified by *x* and *y*. The current graphics cursor is set to the end of the line as defined by *x* and *y*.

## Returns

None.

# draw_ellipse

## Syntax

```
 #include “graph_lib.h”
void draw_ellipse(int fill,
                  int x,
                  int y,
                  int w,
                  int h);
```

## Description

The draw_ellipse function draws an ellipse.

The center of the ellipse is defined by *x* and *y*, and the x and y radii are defined by *w* and *h*. If *fill* is equal to zero (0) then only the outline of the ellipse is drawn, otherwise it is filled.

## Returns

None.

# draw_polygon

## Syntax

```
 #include “graph_lib.h”
void draw_polygon(     int num,
                       int *x,
                       int *y);
```

## Description

The `draw_polygon` function plots a filled polygon.

The arrays *x* and *y* define the vertices of the polygon. The number of elements in each array must be equal to *num* which is the number of vertices in the polygon.

The current graphics cursor is set to the coordinate of the first vertex of the polygon.

## Returns

None.

# draw_rectangle

## Syntax

```
 #include “graph_lib.h”
void draw_rectangle(   int fill,
                       int x,
                       int y,
                       int w,
                       int h);
```

## Description

The function `draw_rectangle` draws a rectangle with bottom left coordinate defined by *x* and *y*, and with width and height defined by *w*, and *h*.

If *fill* is equal to zero (0) then only the outline of the ellipse is drawn, otherwise it is filled.

## Returns

None.

# draw_text

## Syntax

```
 #include “graph_lib.h”
void draw_text(   char *text);
```

## Description

The function `draw_text` plots the text string *text* at the current graphics cursor.

## Returns

None.

# free_gbuf

## Syntax

```
 #include “graph_lib.h”
void free_gbuf(struct gbuf *gf);
```

## Description

The function `free_gbuf` releases the memory claimed for the graphics buffer pointed to by *gf*.

## Returns

None.

# make_gbuf

## Syntax

```
 #include “graph_lib.h”
struct gbuf *make_gbuf(int wdt,int
          hgt,unsigned char *pal_reg);
```

## Description

The `make_gbuf` function reserves memory for a graphics buffer. The buffer is the same format as that used by the `gbuf` driver and can be used as a frame store.

The size of the buffer is defined by *wdt* and *hgt*.

The parameter *pal_reg* points to an array of palette registers that will be used when displaying the image or when it is saved in a graphics file. The two dimensional array has 256x3 elements:

| | |
|---|---|
| `pal_reg[n][0];` | red component. |
| `pal_reg[n][1];` | green component. |
| `pal_reg[n][2];` | blue component. |

The function returns a pointer to a structure of the type gbuf which has the following members:

| | |
|---|---|
| `int wdt;` | width of the buffer in pixels. |
| `int hgt;` | height of the buffer in pixels. |
| `int x;` | the X coordinate of the graphics cursor. |
| `int y;` | the Y coordinate of the graphics cursor. |
| `char c;` | the foreground colour palette index. |
| `char bc;` | the background colour palette index. |
| `char pal_reg[256][3];` | the palette registers. |
| `char *buf;` | the memory to store the image in. |

## Returns

Returns a pointer to the graphics buffer created, or if an error occurred `NULL` is returned.

## Syntax

```
 #include “graph_lib.h”
void move(   int x,
             int y);
```

## Description

The function `move` sets the current graphics cursor to the position specified by *x* and *y*.

## Returns

None.

# set_gbuf

## Syntax

```
 #include “graph_lib.h”
void set_gbuf(struct gbuf *gf);
```

## Description

The function `set_gbuf` sets the active graphics area to the buffer pointed to by *gf*. All subsequent graphics operations will be performed on this buffer.

## Returns

None.

# write_pixel

## Syntax

```
 #include “graph_lib.h”
void write_pixel( int x,
                  int y);
```

## Description

The function `write_pixel` plots a single pixel at the position specified by *x* and *y* in the current foreground colour.

## Returns

None.

log_error.o

# log_error

## Syntax

```
#include  “log_error.h”
int log_error(    char *errlog,
                  char *name,
                  char *buffer);
```

## Description

The `log_error` function sends a message to the `errlog` task identified by *errlog*, containing the error message string pointed to by *buffer*. If *name* is not NULL the error log will include the string together with the error message.

If *errlog* is NULL the default name of “`/errlog`” will be used.

## Returns

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

message.o

## Syntax

```
#include  “message.h”
int message( char *task,
             double time,
             void *smsg,
             void *rmsg,
             unsigned snbytes,
             unsigned rnbytes);
```

## Description

The `message` function sends a message pointed to by *smsg* to the task registered under the name *task*. Any reply is placed in the buffer *rmsg*. The size of the sent message will be *snbytes* while the size of the reply will be truncated to a maximum of *rnbytes*.

If *time* equals zero then the calling process will wait indefinitely for a reply. If time is greater than zero then the process will wait *time* seconds before returning.

The number of bytes send will be the minimum of that specified by the sender and receiver.

## Returns

Returns zero (0) on success. If an error occurs then (-1) is returned and *msgerr* is set.

## Errors

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

| | |
|---|---|
| MSGERR_NO_TASK | no task is registered with that name. |
| MSGERR_SIGNAL_FAIL | the time out signal could not be claimed. The signal *SIGUSR1* is generated after a time-out to interrupt the message. |
| MSGERR_TIMER_FAIL | a timer could not be created. A timer is set to trigger a signal after the required time-out period. |
| MSGERR_MSG_FAIL | the message was interrupted. |
| MSGERR_TME_OUT | the message timed out. |

# message_array

## Syntax

```
#include  “message.h”
int message_array(char *task,
                  double time,
                  void **smsg,
                  void **rmsg,
                  unsigned *snbytes,
                  unsigned *rnbytes);
```

## Description

The `message_array` function sends an array of messages pointed to by the array *smsg* to the task registered under the name *task*. Any replies are placed in the buffers pointed to by the array *rmsg*. The size of each sent message will be taken from the corresponding entry in the array *snbytes* while the size of the reply will be truncated to a maximum of the corresponding entry in the array *rnbytes*.

If *time* equals zero then the calling process will wait indefinitely for a reply. If *time* is greater than zero then the process will wait time seconds before returning.

The number of bytes send will be the minimum of that specified by the sender and receiver.

The function scans the arrays *smsg* and *rmsg*, which must be NULL terminated to determine how many buffers to send and receive.

## Returns

Returns zero (0) on success. If an error occurs then (-1) is returned and *msgerr* is set.

## Errors

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

| | |
|---|---|
| MSGERR_NO_TASK | no task is registered with that name. |
| MSGERR_SIGNAL_FAIL | the time out signal could not be claimed. The signal *SIGUSR1* is generated after a time-out to interrupt the message. |
| MSGERR_NO_MEM | memory could not be allocated to store the messages. |
| MSGERR_TIMER_FAIL | a timer could not be created. A timer is set to trigger a signal after the required time-out period. |
| MSGERR_MSG_FAIL | the message was interrupted. |
| MSGERR_TME_OUT | The message timed out. |

# message_pid

## Syntax

```
#include  “message.h”
int message_pid(  pid_t task_id,
                  double time,
                  void *smsg,
                  void *rmsg,
                  unsigned snbytes,
                  unsigned rnbytes);
```

## Description

The `message_pid` function sends a message pointed to by *smsg* to the task with process id *task_id*. Any reply is placed in the buffer *rmsg*. The size of the sent message will be *snbytes* while the size of the reply will be truncated to a maximum of *rnbytes*.

If *time* equals zero then the calling process will wait indefinitely for a reply. If time is greater than zero then the process will wait *time* seconds before returning.

The number of bytes send will be the minimum of that specified by the sender and receiver.

## Returns

Returns zero (0) on success. If an error occurs then (-1) is returned and *msgerr* is set.

## Errors

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

| | |
|---|---|
| MSGERR_SIGNAL_FAIL | the time out signal could not be claimed. The signal *SIGUSR1* is generated after a time-out to interrupt the message. |
| MSGERR_TIMER_FAIL | a timer could not be created. A timer is set to trigger a signal after the required time-out period. |
| MSGERR_MSG_FAIL | the message was interrupted. |
| MSGERR_TME_OUT | the message timed out. |

# message_pid_array

## Syntax

```
#include  “message.h”
int message_pid_array(
                  pid_t task_id,
                  double time,
                  void **smsg,
                  void **rmsg,
                  unsigned *snbytes,
                  unsigned *rnbytes);
```

## Description

The `message_array` function sends an array of messages pointed to by the array *smsg* to the task with process id *task_id*. Any replies are placed in the buffers pointed to by the array *rmsg*. The size of each sent message will be taken from the corresponding entry in the array *snbytes* while the size of the reply will be truncated to a maximum of the corresponding entry in the array *rnbytes*.

If *time* equals zero then the calling process will wait indefinitely for a reply. If *time* is greater than zero then the process will wait time seconds before returning.

The number of bytes send will be the minimum of that specified by the sender and receiver.

The function scans the arrays *smsg* and *rmsg*, which must be NULL terminated to determine how many buffers to send and receive.

## Returns

Returns zero (0) on success. If an error occurs then (-1) is returned and *msgerr* is set.

## Errors

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

| | |
|---|---|
| MSGERR_SIGNAL_FAIL | the time out signal could not be claimed. The signal *SIGUSR1* is generated after a time-out to interrupt the message. |
| MSGERR_TIMER_FAIL | a timer could not be created. A timer is set to trigger a signal after the required time-out period. |
| MSGERR_NO_MEM | memory could not be allocated to store the messages. |
| MSGERR_MSG_FAIL | the message was interrupted. |
| MSGERR_TME_OUT | The message timed out. |

# task_id

## Syntax

```
#include  “message.h”
pid_t task_id(    char *task);
```

## Description

The `task_id` function returns the process id (**pid**) of the task registered under the name *task*.

## Returns

Returns the process id on success. If the task cannot be found then (-1) is returned and *msgerr* is set.

## Errors

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

| | |
|---|---|
| MSGERR_NO_TASK | no task is registered with that name. |

radar_name.o

# radar_code

## Syntax

```
#include  “util/radar_name.h”
char *radar_code( int station_id);
```

## Description

The `radar_code` function returns the identifying letter of the station with the identifier code *station_id*.

## Returns

Returns a character containing the identifier letter of the Radar station or the string “`x`” if the station is unknown

# radar_name

## Syntax

```
#include  “util/radar_name.h”
char *radar_name( int station_id);
```

## Description

The `radar_name` function returns the name of the station with the identifier code *station_id*.

## Returns

Returns a text string containing the name of the Radar station or the string “`Unknown`” if the station is unknown.

read_clock.o

# read_clock

## Syntax

```
#include  “read_clock.h”
void read_clock(  int *year,
                  int *mon,
                  int *day,
                  int *hour,
                  int *minute,
                  int *second,
                  int *msec,
                  int *usec);
```

## Description

The `read_clock` function reads the system clock. The system clock is automatically calibrated against the GPS clock by the driver `gps_clock`.

## Returns

The current time, accurate to the nearest second is returned in the variables pointed to by *year*, *mon*, *day*, *hour*, *minute*, *second*. At present, the values of *msec* and *usec* are set to zero.

read_data.o

# read_double

## Syntax

```
#include  “util/read_data.h”
int read_double(  FILE *fp, double *val);
```

## Description

The `read_double` function reads an number of type `double` from the file pointed to by *fp*. The result is stored at the address pointed to by *val*.

## Returns

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

# read_float

## Syntax

```
#include  “util/read_data.h”
int read_float(   FILE *fp, float *val);
```

## Description

The `read_float` function reads an number of type `float` from the file pointed to by *fp*. The result is stored at the address pointed to by *val*.

## Returns

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

# read_long

## Syntax

```
#include  “util/read_data.h”
int read_long(    FILE *fp,
                  long int *val);
```

## Description

The `read_long` function reads an number of type `long` from the file pointed to by *fp*. The result is stored at the address pointed to by *val*.

## Returns

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

# read_short

## Syntax

```
#include  “util/read_data.h”
int read_short(   FILE *fp,
                  short int *val);
```

## Description

The `read_short` function reads an number of type `short` from the file pointed to by *fp*. The result is stored at the address pointed to by *val*.

## Returns

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

read_fit.o

# read_fit

## Syntax

```
#include  “read_fit.h”
int read_fit(
          FILE *fp,
          struct fitdata *fit_data);
```

## Description

The `read_fit` function reads a block of fitted data from the file pointed to by *fp* into the structure pointed to by *fit_data*.

## Returns

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

read_raw.o

# read_raw

## Syntax

```
#include  “read_raw.h”
int read_raw(
          FILE *fp,
          struct rawdata *raw_data);
```

## Description

The `read_raw` function reads a raw ACF record from the file pointed to by *fp* into the structure pointed to by *raw_data*.

## Returns

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

# read_raw_data

## Syntax

```
#include  “read_raw.h”
int read_raw_data(
          FILE *fp,
          struct rawdata *raw_data);
```

## Description

The `read_raw_data` function reads a raw ACF record from the file pointed to by *fp* into the structure pointed to by *raw_data*.

Only the raw ACFs are read from the file and the Radar parameter block remains unaffected.

## Returns

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

sample.o

# add_data

## Syntax

```
#include  “sample.h”
int add_data(struct beam_list **table,
             int beam,
             int range,
             enum param_code);
```

## Description

The `add_data` function is used to create a table that can be used to sample the data produced by fitacf across a scan. The table contains a list of range-beam coordinates and a list of parameters to record.

Each time the `transform_data` function is called with a block of fitted data, the table is inspected and a record is made of the data for the appropriate ranges and parameters.

To construct a table, multiple calls are made to `add_data` with the sampling coordinates specified by *beam* and *range*, and the parameter to record with *param_code*, which can be one of :

| | |
|---|---|
| PARAM_qflg | quality flag. |
| PARAM_gsct | ground scatter flag. |
| PARAM_p_0 | lag 0 power. |
| PARAM_p_s | sigma power. |
| PARAM_p_l | lambda power. |
| PARAM_w_l | lambda width. |
| PARAM_w_s | sigma width. |
| PARAM_v | velocity. |
| PARAM_v_err | velocity error. |
| PARAM_sdev_l | standard deviation of lambda fit. |
| PARAM_sdev_s | standard deviation of sigma fit. |
| PARAM_sdev_phi | standard deviation of phase fit. |

The structure `beam_list` is a linked list of all the beams to sample across a scan. It has the following members:

| | |
|---|---|
| `short int beam_no;` | beam number to sample. |
| `short int range_max;` | maximum range for this beam. |
| `struct range_list *table;` | pointer to a table of ranges. |
| `struct beam_list *next;` | pointer to the next entry in the linked list, NULL terminated. |

The structure `range_list` is a linked list of all the ranges to sample within a beam. It has the following members:

| | |
|---|---|
| `short int range;` | the range gate to sample. |
| `long int distance;` | the range in kilometers. |
| `struct param_list *table;` | pointer to a table of parameters. |
| `struct range_list *next;` | pointer to the next entry in the linked list, NULL terminated. |

The structure `param_list` is a list of the parameters to sample at a particular range. It has the following members:

| | |
|---|---|
| `enum param_code code;` | the parameter to sample. |
| `int total;` | total number of times a sample has been taken. |
| `struct time_list *table;` | pointer to a list of samples. |
| `struct param_list *next;` | pointer to the next entry in the linked list, NULL terminated. |

The structure `time_list` is a list of samples. It has the following members:

| | |
|---|---|
| `union {` | |
| `  double value;` | a floating point parameter. |
| `  int flag;` | a boolean flag. |
| `} data;` | union to store the sampled parameter. |
| `struct time_list *next;` | pointer to the next entry in the linked list, NULL terminated. |

Each member of the `time_list` linked list is a sample from one scan. The list is arranged in time order with the first entry being from the most recent scan. The length of the list, and consequently how many scans are stored, is dependent on the number of times `add_data` is called with that particular combination of beam, range and parameter.

## Returns

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

# remove_table

## Syntax

```
#include  “sample.h”
void remove_table(
             struct beam_list **table);
```

## Description

The `remove_table` function frees the memory uses by the sampling table pointed to be *table*.

## Returns

None.

# transform_data

## Syntax

```
#include  “sample.h”
int transform_data(
            struct fitdata *fit_data,
            struct beam_list **table);
```

## Description

The `transform_data` function extracts the appropriate parameters from the fitted data structure pointed to by *fit_data* and insert them into the sampling table pointed to by *table*.

## Returns

Returns zero (0) on success, or (-1) if the record cannot be processed occurs.

terminal.o

## Syntax

```
#include  “terminal/terminal.h”
void centre_text( int row,
                  unsigned attr,
                  char *text);
```

## Description

The `centre_text` function displays the string *text* centered on the middle of the screen, on line *row*, with the terminal attributes *attr*.

You must call the `term_load` function before attempting to call this function.

## Returns

None.

# confirm_prompt

## Syntax

```
#include  “terminal/terminal.h”
int confirm_prompt(    char *text);
```

## Description

The `confirm_prompt` function displays a box with the message in the string *text* on the screen together with two buttons marked “`<yes>`” and “`<no>`”. The function waits until the user selects one of the buttons.

You must call the `term_load` function before attempting to call this function.

## Returns

Returns one (1) if “`<no>`” is selected, or zero (0) otherwise.

# draw_menu

## Syntax

```
#include  “terminal/terminal.h”
int draw_menu(
            struct menu_entry *menu,
            int cursor);
```

## Description

The `draw_menu` function displays a menu on the screen with entry *cursor* hi-lighted.

You must call the `term_load` function before attempting to call this function.

## Returns

Returns the number of entries in the menu.

# draw_menu_item

## Syntax

```
#include  “terminal/terminal.h”
void draw_menu_item(
          struct menu_entry *menu,
          int index,
          int hilighted);
```

## Description

The `draw_menu_item` function prints the entry *index* of the menu *menu* on the screen. If *hilighted* is not equal to zero then the entry will be hi-lighted.

You must call the `term_load` function before attempting to call this function.

## Returns

None.

## Syntax

```
#include  “terminal/terminal.h”
void menu_handler(
            struct menu_entry *menu,
            int cursor);
```

## Description

The `menu_handler` function waits until an entry in the array *menu* is selected and will return the index of the selected entry. The entry with index *cursor* is hi-lighted initially.

The menu can be operated by either the mouse or the keyboard. Menu entries are hi-lighted by either moving the mouse cursor over them, or by pressing the up and down cursor keys. Control is returned to the task by clicking a mouse button or by pressing the return key.

Two types of menu entry are supported, buttons or fields.

Clicking or pressing return when a button is hi-lighted will immediately return control to the task. A button can be switched between two states, de-selected and selected, by setting the appropriate entry the menu structure. This allows both push buttons and switches to be implemented. The menu handler does not automatically select and de-select buttons, and it is the responsibility of the calling task to update the menu entry as appropriate for the type of button.

Hi-lighting a field and pressing any key other than return will enter the edit mode and text can be typed into the field. Pressing “<**enter**>” or “<**escape**>” will leave the edit mode and return control to the task.

The elements of the array *menu* should be structures of the type `menu_entry`. The array should be terminated with a zero initialized element:

```
struct menu_entry menu[]={
  {4,2,MENU_BUTTON,SLC_ON,SLC_OFF,
   CRS_ON,CRS_OFF,0,0,”No”},
  {4,2,MENU_BUTTON,SLC_ON,SLC_OFF,
   CRS_ON,CRS_OFF,0,0,”No”},
   0};
```

The structure `menu_entry` has at least the following members :

| | |
|---|---|
| `int row;` | screen row to display entry |
| `int col;` | screen column to display entry |
| `enum menu_type;` | type of menu entry |
| `unsigned attr_slct_on;` | text attribute for selected and hilighted |
| `unsigned attr_slc_off;` | text attribute for selected and not hilighted |
| `unsigned attr_crs_on;` | text attribute for not selected and hilighted |
| `unsigned attr_crs_off;` | text attribute for not selected and not hilighted |
| `int select;` | flag for when entry is selected |
| `int len;` | length of text field |
| `char *text;` | text string for entry |

The *row* and *col* entries refer to the screen position at which to display the entry.

The type can be of either MENU_BUTTON or MENU_WRITE, implying either a button or a text field that can be edited.

The four attributes are the attributes used by `term_type` to display the text of the menu entry for each of the four conditions.

If the member *select* is not equal to zero then the menu entry is selected and the attribute *attr_slc_off* or *attr_slc_on* is used to display the text when the entry is hi-lighted or not hi-lighted by the cursor, otherwise the attribute *attr_crs_on* or *attr_crs_off* is used.

The string *text* contains the text that will be displayed as the menu entry. For writable fields this should point to the text array into which the text should be written.

The member *len* is used to fix the width of the fields that can be edited, if this is set to zero then the field width will be calculated from the initial string stored in the array *text*.

You must call the `term_load` function before attempting to call this function.

## Returns

Returns the array index of the menu item hi-lighted when the mouse button was clicked or return was pressed.

# report_error

## Syntax

```
#include  “terminal/terminal.h”
void report_error(     char *text);
```

## Description

The `report_error` function displays a box with the message in the string *text* on the screen together with a button marked “`<Continue>`”. The function waits until the user clicks on the button or presses the return key.

You must call the `term_load` function before attempting to call this function.

## Returns

None.

# setup_mouse

## Syntax

```
#include  “terminal/terminal.h”
void setup_mouse( void);
```

## Description

The `setup_mouse` function sets up the mouse to work with the menu system.

You must call the `term_load` function before attempting to call this function.

## Returns

None.

# show_message

## Syntax

```
#include  “terminal/terminal.h”
void show_message(     char *text);
```

## Description

The `show_message` function displays a box with the message in the string *text* on the screen.

You must call the `term_load` function before attempting to call this function.

## Returns

None.

test_key.o

# free_key

## Syntax

```
#include  “util/test_key.h”
void free_key(void);
```

## Description

The `free_key` function releases the proxy created by the `register_key` function.

## Returns

None.

# register_key

## Syntax

```
#include  “util/test_key.h”
int register_key(void);
```

## Description

The `register_key` function creates a proxy that is triggered whenever a key press occurs. The proxy can be tested for using the `test_key` function.

## Returns

Returns one (1) if the proxy was successfully created, or zero (0) if an error occurred.

# test_key

## Syntax

```
#include  “util/test_key.h”
int test_key(     pid_t);
```

## Description

The `test_key` function tests whether a message received from the process identified by *pid*, was a proxy created by the function `register_key` and triggered as a result of the user pressing a key.

## Returns

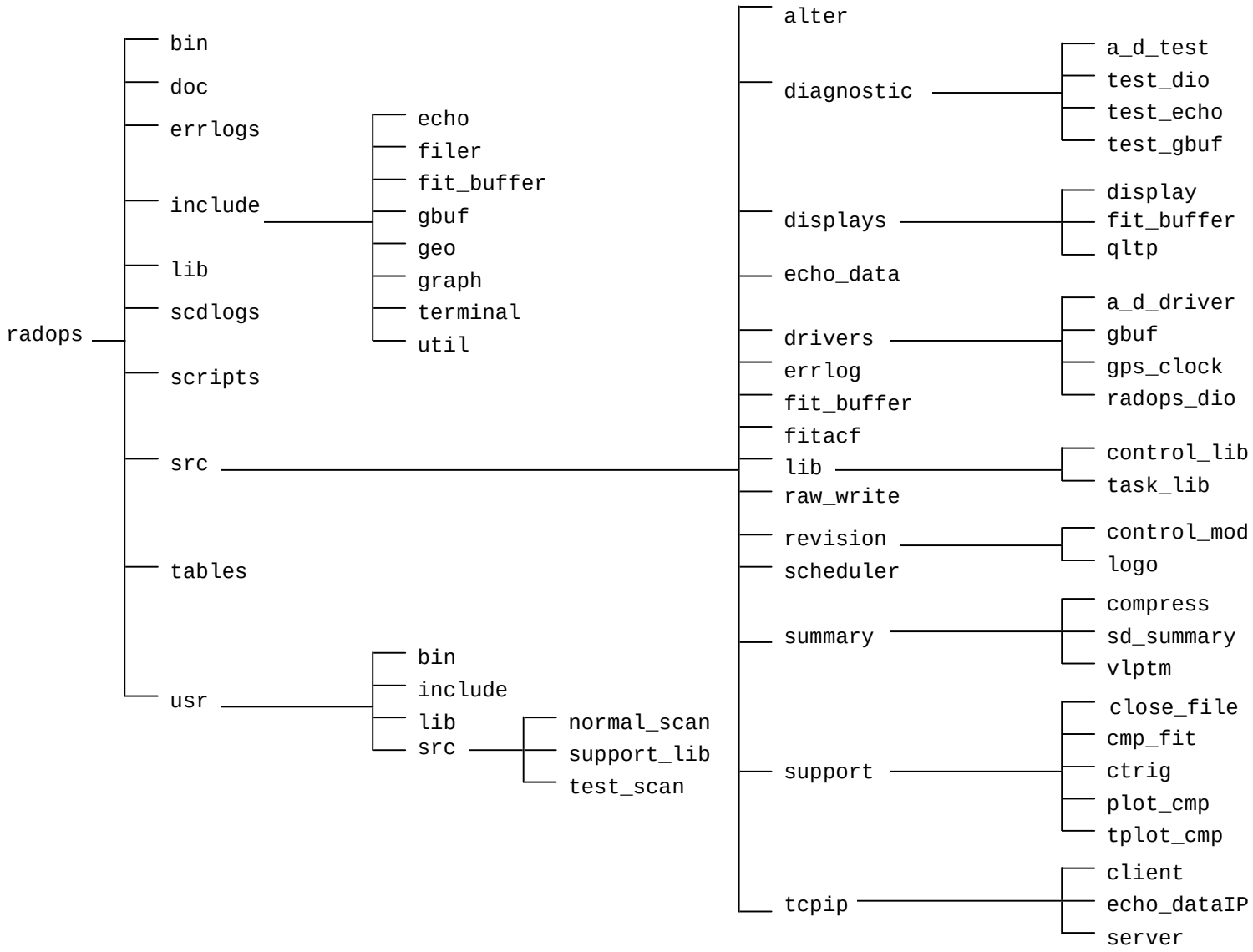
Returns one (1) if the message was the result of a key press, or zero (0) otherwise.

# Appendix A

## Software Organization Chart

# Appendix B

## Directory Structure


radops
bin
doc
errlogs
include
lib
scdlogs
scripts
src
tables
usr
echo
filer
fit_buffer
gbuf
geo
graph
terminal
util
bin
include
lib
src
normal_scan
support_lib
test_scan
alter
diagnostic
displays
echo_data
drivers
errlog
fit_buffer
fitacf
lib
raw_write
revision
scheduler
summary
support
tcpip
a_d_test
test_dio
test_echo
test_gbuf
display
fit_buffer
qltp
a_d_driver
gbuf
gps_clock
radops_dio
control_lib
task_lib
control_mod
logo
compress
sd_summary
vlptm
close_file
cmp_fit
ctrig
plot_cmp
tplot_cmp
client
echo_dataIP
server

# Appendix C

## File List

## radops/:

```
./                doc/              make_radar.log     tables/
../               errlogs/          scdlogs/           usr/
bin/              include/          scripts/
demo.dat          lib/              src/
```

## radops/bin:

```
./             cmp_fit*       errlog*        plot_cmp*      test_dio*
../            compress*      fit_buffer*    qltp*          test_echo*
a_d_drive*     control_mod*   fitacf*        radops_dio*    test_gbuf*
a_d_test*      ctrig*         fitdisp*       raw_write*     tplot_cmp*
alter*         display*       gbuf*          schedule*      vlptm*
client*        echo_data*     gps_clock*     sd_summary*
close_file*    echo_dataIP*   logo*          server*
```

## radops/doc:

```
./                history.radops
../               readme.radops
```

## radops/errlogs:

```
./    ../
```

## radops/include:

```
./                      geo/                    read_clock.h
../                     get_fit.h               read_fit.h
RCS/                    get_status.h            read_raw.h
a_d_drive.h             graph/                  sample.h
dio.h                   log_error.h             station.h
dma-addr.h              message.h               task_msg.h
dma-alloc.h             name.h                  task_write.h
echo/                   option.h                terminal/
file_io.h               radar_id.h              types.h
filer/                  raderr.h                user_int.h
fit_buffer/             raderr.txt              util/
fitdata.h               radops.h
gbuf/                   radops_version.h
```

## radops/include/RCS:

```
./             ../            radops.h,v
```

## radops/include/echo:

```
./             ../            echo_util.h
```

## radops/include/filer:

```
./          ../         filer.h
```

## radops/include/fit_buffer:

```
./             ../            fit_util.h
```

## radops/include/gbuf:

```
./             ../            gbuf_util.h
```

## radops/include/geo:

```
./       ../      geo.h
```

## radops/include/graph:

```
./            ../           graph_lib.h
```

## radops/include/terminal:

```
./            ../           terminal.h
```

## radops/include/util:

```
./               add_point.h        radar_name.h       test_key.h
../              cnv_time.h         read_data.h
```

## radops/lib:

```
./            ../           control.lib    task.lib
```

## radops/scdlogs:

```
./    ../
```

## radops/scripts:

```
./               rad_export         start_radar*
../              rad_path           stop_debug*
debug.sched      radops.sched       stop_radar*
make_radar*      start_debug*       tidy_up*
```

## radops/src:

```
./            displays/      fit_buffer/    revision/      support/
../           drivers/       fitacf/        sbin/          tcpip/
alter/        echo_data/     lib/           scheduler/
diagnostic/   errlog/        raw_write/     summary/
```

## radops/src/alter:

```
./               alter.c            control.info       makefile
../              alter.h            main.c
```

## radops/src/diagnostic:

```
./          a_d_test/    test_dio/    test_gbuf/
../         sbin/        test_echo/
```

## radops/src/diagnostic/a_d_test:

```
./               atest.c            makefile
../              control.info
```

## radops/src/diagnostic/sbin:

```
./          a_d_test*    test_echo*
../         test_dio*    test_gbuf*
```

## radops/src/diagnostic/test_dio:

```
./               control.info       dio_test.h         test_dio.c
../              dio_test.c         makefile
```

## radops/src/diagnostic/test_echo:

```
./               control.info       test_echo.c
../              makefile           version.h
```

## radops/src/diagnostic/test_gbuf:

```
./                control.info       test_gbuf.c
../               makefile           version.h
```

## radops/src/displays:

```
./         ../        display/   fitdisp/   qltp/      sbin/
```

## radops/src/displays/display:

```
./                control.info       main.c
../               display.c          makefile
colours.h         display.h          version.h
```

## radops/src/displays/fitdisp:

```
./                build_table.h      graphics.c         transform.h
../               control.info       makefile           version.h
build_table.c     fit_disp.c         transform.c
```

## radops/src/displays/qltp:

```
./                control.info       plot.c             qltp.h
../               graphics.c         plot_file.c        version.h
config.h          makefile           qltp.c
```

## radops/src/displays/sbin:

```
./         ../        display*   fitdisp*   qltp*
```

## radops/src/drivers:

```
./             a_d_driver/    gps_clock/     sbin/
../            gbuf/          radops_dio/
```

## radops/src/drivers/a_d_driver:

```
./                      dt.ext                  dt_strig.c
../                     dt.h                    main.c
control.info            dt2828.c                makefile
dma-alloc.c             dt2828.h                pragma.h
dma.dec                 dt_etrig.c              set_clock.c
dma.h                   dt_int.c                version.h
dma_init.c              dt_int_handler.c
dt.dec                  dt_reset.c
```

## radops/src/drivers/gbuf:

```
./                control.info       makefile           version.h
../               gbuf.c             pal_table.c
```

## radops/src/drivers/gps_clock:

```
./                bc620.h            gps_time.c
../               control.info       makefile
bc620.c           gps_clock.c        version.h
```

## radops/src/drivers/radops_dio:

```
./                control.info       logger.h           tsg.h
../               display.c          main.c             version.h
ASM/              do_op.c            main.h             watchdog.c
DIO.c             do_op.h            makefile           watchdog.h
DIO.h             forbid_freq.c      port.h
PIO48.h           forbid_freq.h      reset.c
bcd.c             logger.c           tsg.c
```

## radops/src/drivers/radops_dio/ASM:

```
./            out_tsg.c      out_tsg_b.a    readme
../           out_tsg.l      out_tsg_b.l
out_tsg.a     out_tsg.o      out_tsg_b.o
```

## radops/src/drivers/sbin:

```
./          ../         a_d_drive*   gbuf*        gps_clock*
```

## radops/src/echo_data:

```
./              control.info      echo.h            version.h
../             echo.c            makefile
```

## radops/src/errlog:

```
./              control.info      makefile
../             errlog.c
```

## radops/src/fit_buffer:

```
./              control.info      makefile
../             fit_buffer.c      version.h
```

## radops/src/fitacf:

```
./                      fitacf.c                rang_badlags.c
../                     fitacf.ext              remove_noise.c
acf_preproc.c           fitfile.h               sd_swab.c
acf_preproc.h           ground_scatter.c        swap_data.c
badlags.c               hardware.c              swap_parms.c
calc_phi_res.c          inx_close.c             transmit_data.c
control.info            makefile                uname.h
dbl_cmp.c               math_handler.c          version.h
elev_goose.c            more_badlags.c          write_header.c
elevation.c             my_math.h               xfer.c
endian.h                noise_acf.c             xfer.h
fit_acf.c               noise_stat.c
fit_noise.c             omega_guess.c
fit_prio.h              proc_rec.c
```

## radops/src/lib:

```
./            control_lib/   task_lib/
../           slib/
```

## radops/src/lib/control_lib:

```
./              dma-addr.c        message.c         read_raw.c
../             get_fit.c         option.c          sample.c
a_d_drive.c     get_status.c      raderr.c          task_write.c
control.info    log_error.c       read_clock.c      user_int.c
dio.c           makefile          read_fit.c
```

## radops/src/lib/slib:

```
./            ../           control.lib    task.lib
```

## radops/src/lib/task_lib:

```
./              file_io.c         makefile          read_raw.c
../             filer.c           message.c         sample.c
add_point.c     filer.ext         pal_table.c       terminal.c
cnv_time.c      gbuf_util.c       radar_name.c      terminal.ext
cnvt_coord.c    gfont.c           read_clock.c      test_key.c
control.info    graph_lib.c       read_data.c
echo_util.c     log_error.c       read_fit.c
```

## radops/src/raw_write:

```
./                compress.h         makefile           version.h
../               control.info       raw_write.c
compress.c        header.c           record.h
```

## radops/src/revision:

```
./             control_mod/   sbin/
../            logo/
```

## radops/src/revision/control_mod:

```
./                control_mod.c
../               makefile
```

## radops/src/revision/logo:

```
./           ../          logo.c       makefile
```

## radops/src/revision/sbin:

```
./             check_id*      logo*
../            control_mod*
```

## radops/src/sbin:

```
./             alter*         errlog*        fitacf*        schedule*
../            echo_data*     fit_buffer*    raw_write*
```

## radops/src/scheduler:

```
./                execute.c          schedule.c         version.h
../               main.c             schedule.h
control.info      makefile           test.sched
```

## radops/src/summary:

```
./             compress/      sd_summary/
../            sbin/          vlptm/
```

## radops/src/summary/compress:

```
./                buffer.h           control.info
../               colours.h          makefile
buffer.c          compress.c         version.h
```

## radops/src/summary/sbin:

```
./             compress*      vlptm*
../            sd_summary*
```

## radops/src/summary/sd_summary:

```
./                makefile           sd_summary.c
../               print_list.c       version.h
control.info      print_val.c        write_header.c
```

## radops/src/summary/vlptm:

```
./                control.info       read_datrec.c      write_header.c
../               linreg.c           scntabl.c
bigtabl.c         lshell.c           version.h
bigtabl.h         makefile           vlptm.c
bigtabl_def.c     medfilter.c        vlptm.h
```

## radops/src/support:

```
./            close_file/    ctrig/         sbin/
../           cmp_fit/       plot_cmp/      tplot_cmp/
```

## radops/src/support/close_file:

```
./                close_file.c       makefile
../               control.info       version.h
```

## radops/src/support/cmp_fit:

```
./                cmp_fit.c          control.info       version.h
../               colours.h          makefile
```

## radops/src/support/ctrig:

```
./                control.info       makefile
../               ctrig.c            version.h
```

## radops/src/support/plot_cmp:

```
./                control.info       graphics.c         transform.c
../               decode.c           makefile           transform.h
build_table.c     decode.h           plot_cmp.c         version.h
build_table.h     do_filter.c        size.h
```

## radops/src/support/sbin:

```
./            close_file*    ctrig*         plot_cmp*
../           cmp_fit*       fitfile*       tplot_cmp*
```

## radops/src/support/tplot_cmp:

```
./                decode.c           makefile
../               decode.h           tplot_cmp.c
control.info      graphics.c         version.h
```

## radops/src/tcpip:

```
./            client/        sbin/
../           echo_dataIP/   server/
```

## radops/src/tcpip/client:

```
./                connex.h           makefile~
../               decode_msg.c       msg_io.c
client.c          fit_data.h         read_data.c
connex.c          makefile           read_data.h
```

## radops/src/tcpip/echo_dataIP:

```
./                decode_msg.h       msg_io.c           version.h
../               echo.h             msg_io.h
cmp_fit.h         echoIP.c           socket.c
decode_msg.c      makefile           socket.h
```

## radops/src/tcpip/sbin:

```
./            client*        server*
../           echo_dataIP*
```

## radops/src/tcpip/server:

```
./                makefile          server.c          timer.c
../               msg_io.c          socket.c          version.h
cmp_fit.c         msg_io.h          socket.h
cmp_fit.h         process_msg.c     srv_prio.h
```

## radops/tables:

```
./                        cnvtabl_h.dat             invtabl_t.dat
../                       cnvtabl_j.dat             invtabl_w.dat
bmrtabl_d.dat             cnvtabl_k.dat             lamda.dat
bmrtabl_e.dat             cnvtabl_n.dat             logo.img
bmrtabl_f.dat             cnvtabl_t.dat             map_data
bmrtabl_g.dat             cnvtabl_w.dat             overlay
bmrtabl_h.dat             gb_doppler_gates.dat      pal_reg.16
bmrtabl_j.dat             hardware.dat              pal_reg.256
bmrtabl_k.dat             invtabl_d.dat             palette
bmrtabl_n.dat             invtabl_e.dat             qltp_drpl.dat
bmrtabl_t.dat             invtabl_f.dat             restrict.freq
bmrtabl_w.dat             invtabl_g.dat             search.freq
cnvtabl_d.dat             invtabl_h.dat             sigma.dat
cnvtabl_e.dat             invtabl_j.dat             sigma.dat.b
cnvtabl_f.dat             invtabl_k.dat             sy_doppler_gates.dat
cnvtabl_g.dat             invtabl_n.dat
```

## radops/usr:

```
./         ../        bin/       include/   lib/       src/
```

## radops/usr/bin:

```
./                        normal_scan.debug*        test_scan.debug*
../                       normal_scanD*
normal_scan*              test_scan*
```

## radops/usr/include:

```
./                        freq_band.h               support.h
../                       report_error.h            sync.h
default.h                 summary_control.h
```

## radops/usr/lib:

```
./             ../            support.lib
```

## radops/usr/src:

```
./             normal_scan/   test_scan/
../            support_lib/
```

## radops/usr/src/normal_scan:

```
./                freq_band.h        normal_scan.c
../               makefile           pulse_code.h
```

## radops/usr/src/support_lib:

```
./                        forbid_freq.h             read_uconts.c
../                       init_proxy.c              summary_control.c
blkin.c                   makefile                  support.c
blkin.h                   new_tsg.c                 sync.c
control.info              phase_decode.c            tmseq.c
core_math.c               pulse_code.c              tmseq.dec
core_math.h               pulse_code.h              tmseq.ext
define.h                  pulse_math.c              ucont.h
fclr.c                    pulse_math.h              version.h
fclr.h                    radar.c
forbid_freq.c             radar.h
```

## radops/usr/src/test_scan:

```
./              ../             makefile        test_scan.c
```