Document control number: TBD, Date: 1 July, 1997

Build 4.6

DRAFT

RADAR DATA SERVICE COMMUNICATION USER MANUAL

FOR THE

SURFACE SEARCH RADAR PROGRAM

CONTRACT NO. DTCG23-96-C-ASR009

CDRL SEQUENCE NO. 0021-002

Prepared for:

Commandant (G-ASM/SSR)
US COAST GUARD
2100 Second Street, SW
Washington, DC 20593-0001

Prepared by:

Maritime Systems Program Office
Hughes Naval and Maritime Systems
Building 617, MS B110
P. O. Box 3310
Fullerton, CA. 92634-3310

TABLE OF CONTENTS

1. Introduction 3

2. Referenced Documents 3

3. RDS COMM Description 3

3.1 RDS COMM Overview 3

3.2 RDS COMM Philosophy 4

3.3 Communications Protocol 4

3.4 Message System 5

3.4.1 Message Identification 5
3.4.2 Message Buffer Pool Creation 6
3.4.3 Attaching and detaching from the Buffer Pools 6
3.4.4 Message Registration and Deregistration 6
3.4.5 Message Transmission and Retrieval 6

3.5 RDS COMM User Interface 7

3.5.1 RDS COMM Startup 7
3.5.1.1 rds_startup() 7
3.5.2 RDS COMM Function Prototype Definitions 7
3.5.2.1 RDS Message Buffer Management Functions 7
3.5.2.1.1 AttachMessageBuffers( ) 7
3.5.2.1.2 DetachMessageBuffers( ) 8
3.5.2.1.3 GetMessageBuffer( ) 8
3.5.2.1.4 ReleaseMessageBuffer( ) 8
3.5.2.2 Registration Record Management Functions 8
3.5.2.2.1 CreateRegistrationRecord( ) 8
3.5.2.2.2 DeleteRegistrationRecord( ) 9
3.5.2.2.3 RegisterMessage( ) 9
3.5.2.2.4 DeregisterMessage( ) 9
3.5.2.2.5 DeregisterAllMessages( ) 10
3.5.2.2.6 RegisterForInput( ) 10
3.5.2.2.7 RdsCleanup( ) 10
3.5.2.3 Message Transmission and Retrieval Functions 11
3.5.2.3.1 ReadMessage( ) 11
3.5.2.3.2 OutputMessage( ) 11
3.5.2.3.3 OutputRadarImageMessage( ) 12
3.5.2.3.4 OutputControlMessage( ) 12

4. Appendices 14

4.1 Appendix A -Messaging Examples 14

4.1.1 Message Registration Example 14
4.1.2 Message Retrieval Example 15
4.1.3 Message Transmission Example 16
4.1.4 Read Example 17
4.1.5 Write Example 19

Acronym List

Acronym	Expansion
ANSI	American National Standards Institute
API	Application Program Interface
ETS	External Tactical System
FDDI	Fiber Distributed Data Interface
IEEE	Institute of Electrical and Electronic Engineers
IP	Internet Protocol
LAN	Local Area Network
RDS	Radar Data Services
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

1.

1. Introduction

The purpose of this document is to describe the Hughes Surface Search Radar System (AN/SPS-73(V)) Radar Data Service (RDS) Communication (COMM) Software. This document includes an introduction to the RDS COMM, its function, and provides an example of sending and receiving messages. The RDS COMM will be used by External Tactical System (ETS) to interface with an AN/SPS-73(V) system.

2. Referenced Documents

System Interface Design Document (SIDD) for the Surface Search Radar Program
DCN: 1721798

Version Description Document for the Surface Search Radar (SSR) RDS COMM
DCN: 1859403-2

ANSI X3T9.5, Fiber Distribution Data Interface (FDDI) Station Management

IEEE 802.3, Information Processing Systems, Local Area Networks - Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specification of 1993

3. RDS COMM Description

3.1 RDS COMM Overview

The RDS COMM is designed to enable applications residing on nodes[1] within a local area network (LAN) to exchange messages in a manner that is transparent to the individual applications. Applications may reside either within a single node or may be distributed across multiple nodes. Figure 1 shows a generalized view of the RDS COMM and its relationship to the system.

Figure 1. Generalized view of the RDS API.

[1] A node refers to a single CPU that may share its IP address with several application tasks.

The RDS COMM exists as a set of processes that execute on each node in the local area network. Calls are made to the RDS COMM Application Programming Interface (API) functions from the application to interact with the RDS COMM to transmit and receive messages.

The RDS COMM and its API isolates applications from the details of the underlying network interface. Individual messages are routed to their destination using either Transport Control Protocol (TCP) (for messages that must have guaranteed delivery and order) or User Datagram Protocol (UDP) for other messages. The Internet Protocol (IP) is used to interface with the underlying physical transportation mechanism that can be either IEEE 802.3 Ethernet or ANSI X3T9.5 Fiber Distribution Data Interface (FDDI).

3.2 RDS COMM Philosophy

The general goal of the RDS COMM is to provide a single interface point for application-level network messaging traffic. The specific goals are:

- Provide an easy to use interface that enables applications to send and receive messages between processes executing on the same or different nodes on the network.
- Allow changes in the system architecture (i.e. Add/Remove Radar Processors or Operator Positions) to be made without requiring software changes.
- Isolate the applications programmer from the details of the inter-node message transfer process.

There are two central concepts within the RDS COMM. The first concerns the receipt of messages. Each application defines the information (messages) it wants to receive and then “registers” to receive those messages. The application is not concerned with which process or which node generates the messages. The registration definition is distributed throughout the network so that the RDS Server on each network node holds the destination address of all registered messages. Each application can register to receive specific messages or groups of messages.

The second concept concerns the sending of messages. If an application has information to send to other applications, it simply creates the message and sends it. The application is not concerned with who has requested the message, just that it has a message to send.

3.3 Communications Protocol

The RDS COMM makes use of the following communications protocols:

- Transport Control Protocol (TCP)
- User Datagram Protocol (UDP)

The RDS COMM utilizes the above communications protocols to provide the appropriate level of service (i.e. service reliability) for each type of message. The RDS COMM provides several functions that let the user specify which protocol to use to deliver a message. Certain messages utilize the reliable network protocol TCP to guarantee delivery, while others utilize the non-guaranteed network protocol UDP. For each message in the SSR system the communication protocol to be used is specified in the SIDD. See the

SSR SIDD RDS Message Set Summary for the details.

3.4 Message System

3.4.1 Message Identification

The RDS messages are uniquely identified by the following two fields:

Category type (Cat_Type)

Amplification type (Amp_Type).

The AN/SPS-73(V) system designer creates message groups by defining a general Cat_Type and then all Amp_Types for the messages associated with that Cat_Type.

An example of message grouping is the CAT_ANY_TRACK_DATA category, which encompasses all "track" related messages from the Radar Processors. Sample Amp_Types defined include:

- AMP_SYSTEM_TRACK_REPORT
- AMP_SENSOR_TRACK_REPORT

All registered messages are data structures identified by a Cat_Type/Amp_Type assignment pairing. The RDS API user can register for receipt of any individual message kind by specifying a Cat_Type/Amp_Type pair, or all messages of a Cat_Type by specifying a "wild card" for the Amp_Type.

The following header files (ANSI C format) specify the message interface to the Surface Search Radar (SSR) system:

msgbuf_api.h - This is the include file that contains the structure definitions used in the buffer management functions.

rds_api.h - This is an include file which contains the function prototypes for the RDS API and the data structures and constants required for the interface.

messages_api.h - This is an include file which contains enumeration values for all application defined message Cat_Types and Amp_Types.

For every message sent and/or received by the RDS API there is a defined data structure (each Cat_Type / Amp_Type pair has a data structure associated with it). The following files define the data structure within each message:

- alerts_api.h
- bits_api.h
- graphics_api.h
- iccontrol_api.h
- nsiu_api.h
- radar_api.h
- tracks_api.h

3.4.2 Message Buffer Pool Creation

Message buffer pool creation is done once, at system startup, by running the rds_startup program. The program takes five command line parameters that specify the number of buffers that will be created for the 128, 512, 2K, 9K, and 520K buffer pools used by the RDS API.

3.4.3 Attaching and detaching from the Buffer Pools

Before an application can get a message buffer, the application must first attach to the message buffer pool created by rds_startup program. After the message buffer pool has been attached, the application can get/release message buffers to/from the message buffer pools.

Before exiting an application must detach form the message buffer pool to properly free up operating system.

3.4.4 Message Registration and Deregistration

Message registration allows an application to identify the messages it would like to receive from the other processes on the network. To register for a message or a group of messages the application first creates a registration record, then fills the registration record with registration requests and then sends it to the RDS API, using the RegisterForInput function. The RDS API returns a channel identifier, which the application uses to receive messages. The process of registering to receive messages may be performed at any time, it is not necessary for the sending process(s) to be operational. Once the RegisterForInput function has been called the registration record is freed by the RDS and a new record is required for new registration requests.

Message deregistration allows the application to identify the messages it would like to stop receiving from the other processes on the network. Message registration and deregistration may be done using the same registration record. To deregister for a message or a group of messages the application first creates a registration record, then deregisters the message(s) it wants to stop receiving, and then sends it to the RDS API, using the RegisterForInput function. The RDS API returns a channel identifier to the application, which the application uses to retrieve messages. Deregistration for a group of messages may be performed at any time, it is not necessary for the sending process(s) to be operational.

The registration process is used by applications wanting to receive RDS API messages. If an application only wants to send messages, it is not necessary to register for messages. The application can send messages without registering the message type.

3.4.5 Message Transmission and Retrieval

Message transmission allows an application to send messages to other processes on the network. To send messages to other processes, the application first gets a message buffer, then fill the buffer with data, and then calls the RDS API output message function to send the message. The message buffer is released by the RDS API.

Message retrieval allows the application to receive messages from other processes on the network. To receive messages sent from other processes, the application calls the RDS API read message routine with the channel identifier created by the registration process. Message buffers received by the application must

be released by the application.

3.5 RDS COMM User Interface

3.5.1 RDS COMM Startup

The following section describes how to startup the RDS COMM tasks on a HPUX workstation.

3.5.1.1 rds_startup()

This program is used on a UNIX system to specify the number of message buffers in the buffer pools to be created and to start the RDS processes. The command line parameters for this program are:

```
int rds_startup ( int buffer1qty, /* 128  byte */
                  int buffer2qty, /* 512  byte */
                  int buffer3qty, /* 2K   byte */
                  int buffer4qty, /* 9K   byte */
                  int buffer5qty) /* 520K byte */
```

This program should be called during system startup (for example from the rc.local script) and will initialize the message buffer pools and then start the RDS processes. The five buffer pool are 128 bytes, 512 bytes, 2K bytes, 9K bytes, and 520K bytes in size. The 128 byte, 512 byte, and the 2K byte buffers are used for the typical messages that are sent and received by RDS API applications. The 9K byte buffers are used internally by the RDS API, and the 520K byte buffers are used for Radar Image buffers.

This program will exit with a status of 0 on success, or -1 if an error is detected.

3.5.2 RDS COMM Function Prototype Definitions

The following section defines the RDS COMM function prototypes that are used to interface with the SSR system. See Appendix A for examples of their usage.

3.5.2.1 RDS Message Buffer Management Functions

3.5.2.1.1 AttachMessageBuffers( )

This function attaches to the message buffer pool that was created by the rds_startup program. The function prototype is as follows:

```
int AttachMessageBuffers (void)
```

This function must be called before using the message buffer management functions of the RDS API. This function should be called once in the initialization portion of an applications startup. This function is used by the RDS to gain access the message buffer pool.

This function returns 0 on success, or -1 if an error is detected.

3.5.2.1.2 DetachMessageBuffers( )

This function detaches the application from the message buffer pool. The function prototype is as follows:

```
int DetachMessageBuffers (void)
```

Application processes should call this function when they are done using the RDS API messages.

This function returns 0 on success, or -1 if an error is detected.

3.5.2.1.3 GetMessageBuffer( )

This function allocates a message buffer that can be used to create and send messages. The function prototype is as follows:

```
void *GetMessageBuffer (int size)
```

The size of the required message buffer is passed as the only parameter.

This function returns a pointer to the message buffer on success, or a NULL if a message buffer can not be allocated.

3.5.2.1.4 ReleaseMessageBuffer( )

This function deallocates a message buffer that was received. The function prototype is as follows:

```
int ReleaseMessageBuffer (void *pMsgBuf)
```

This function is called to deallocate a message buffer and returns a 0 on success, or a -1 on error.

3.5.2.2 Registration Record Management Functions

3.5.2.2.1 CreateRegistrationRecord( )

This function creates a message registration record. The function prototype is as follows:

```
REG_RECORD *CreateRegistrationRecord (void)
```

The message registration record is used to control, which messages are to be placed into the application's communication channel. An application wishing to receive certain messages types uses this function to get a registration record which is used by the RegisterForInput function to control which messages are to be place on it's input channel, and returned by the ReadMessage function.

This function returns a pointer to a registration record, or NULL if an error is detected when creating the registration record. The pointer to a registration record is used in subsequent calls to the RegisterMessage( ), DeregisterMessage( ), and the RegisterForInput( ) functions.

3.5.2.2.2 DeleteRegistrationRecord( )

This function deletes a message registration record. The function prototype is as follows:

```
int DeleteRegistrationRecord (REG_RECORD *regptr)
```

This function is only called if an error occurs when calling RegisterMessage, DeregisterMessage, or RegisterForInput. Normally RegisterForInput will delete the registration record on a successful return.

This function returns a 0 on success, or -1 if an error is detected when deleting the registration record.

3.5.2.2.3 RegisterMessage( )

This function specifies to the RDS COMM that the application would like to receive messages of the specified Amp_Type and Cat_Type. The function prototype is as follows:

```
int RegisterMessage(REG_RECORD *regptr,
                    MESSAGE_SUBTYPE subtype,
                    MESSAGE_AMPTYPE amptype)
```

REG_RECORD is the pointer returned by the CreateRegistrationRecord function. MESSAGE_SUBTYPE is the Cat_Type of the message to be received. MESSAGE_AMPTYPE is the Amp_Type of the message to be received, or WILDCARD if all messages of the Cat_Type are to be received.

The maximum number of registration and/or deregistration requests that can be made at one time (using a REG_RECORD), is specified by MAXCOMPONENTS (currently set to 50).

Note: you can use the same registration record to register for new messages and deregister for old messages.

This function returns 0 if the registration record is modified successfully, or -1 if an error is detected when modifying the registration record.

3.5.2.2.4 DeregisterMessage( )

This function specifies to the RDS COMM that the application no longer needs to receive the specified message. The function prototype is as follows:

```
int DeregisterMessage(REG_RECORD *regptr,
                      MESSAGE_SUBTYPE subtype,
                      MESSAGE_AMPTYPE amptype)
```

REG_RECORD is the value returned by CreateRegistrationRecord( ). MESSAGE_SUBTYPE is the Cat_Type of the message that is no longer needed. MESSAGE_AMPTYPE is the Amp_Type of the message that is not longer needed.

This function returns 0 if the registration record is modified successfully, or -1 if an error is detected when modifying the registration record.

3.5.2.2.5 DeregisterAllMessages( )

This function requests the RDS COMM to deregister all the messages that the application has previously registered for. The function prototype is as follows:

```
int DeregisterAllMessages(CHANNEL_ID channel_id)
```

The DeregisterAllMessages( ) function deregisters for all registered messages and then reads and release the message buffer for any messages that may still be on the applications message channel. The channel_id is the value returned by RegisterForInput( ).

This function returns 0 if the registration record is modified successfully, or -1 if an error is detected when modifying the registration record.

3.5.2.2.6 RegisterForInput( )

This function activates the registration record which was built with previous calls to CreateRegistrationRecord( ), DeregisterMessage( ), and RegisterMessage( ) and begins the receipt of messages. The function prototype is as follows:

```
CHANNEL_ID RegisterForInput(REG_RECORD *regptr,
                            CHANNEL_ID channel_id)
```

The application provides a pointer to a registration record. If input messages are to be placed onto an existing communications channel, the existing channel identifier is passed as the second parameter. If the second parameter is NULL, a new communication channel will be created and the channel identifier returned by the RegisterForInput( ) function. The channel identifier is used by the ReadMessage( ) function.

The CHANNEL_ID may be typecast as an int and used as a file descriptor (fd) in a select( ) function call.

This function returns a CHANNEL_ID on success, or NULL if an error is detected when creating a communication channel.

3.5.2.2.7 RdsCleanup( )

This function is called by an RDS COMM application prior to exiting in order to free up system resources. The function prototype is as follows:

```
int RdsCleanup(CHANNEL_ID channel_id)
```

The RdsCleanup( ) function deregisters for all registered messages, reads and release the message buffer for any messages that may still be on the applications message channel, and then closes all system resources (i.e. file descriptors) that have been created by the RDS COMM for that application.

For RDS COMM applications that only send data and do not have a channel_id (i.e. the application never called RegisterForInput), this function should be called with a channel_id of –1.

This function returns 0 if the closing of file descriptors is successful, or -1 if an error is detected.

3.5.2.3 Message Transmission and Retrieval Functions

3.5.2.3.1 ReadMessage( )

This function reads messages from a communication channel which has been created by RegisterForInput( ). The function prototype is as follows:

```
int ReadMessage(CHANNEL_ID,  struct timeval waitind, void **message)

waitind  = NULL                          /* means WAIT_ON_DATA */

waitind.tv_sec=0,
and waitind.tv_usec=0                    /* means NOWAIT_ON_DATA */

waitind.tv_sec=positive integer
or waitind.tv_usec=positive integer      /* means time to wait in sec
                                            and /or microseconds */
```

If a message is waiting on the communication channel when called, ReadMessage( ) will read the message from the communication channel, and return a value of 1 (indicating that *message* contains a valid message pointer). If there is not a message waiting on the communication channel, ReadMessage( ) will react based on the *waitind* parameter. If *waitind* is a NULL pointer (WAIT_ON_DATA), ReadMessage( ) will block until a message is available. If *waitind.tv_sec* is 0 and *waitind.tv_usec* is 0 (NOWAIT_ON_DATA), ReadMessage( ) will return 0.

When the *waitind.tv_sec and/or waitind.tv_usec* is an integer greater than 0, it represents the amount of time to wait. In that case, ReadMessage will suspend for up to the amount of time to wait that *waitind* specified, or the receipt of a message, which ever occurs first. If a message is received within the time to wait integer, ReadMessage( ) will return a 1 and *message* will contain a valid message pointer. If not, 0 will be returned.

This function returns a value of 0 if there was no message available and the application requested no wait, a value of 1 if a message was read, or -1 if an error is detected when reading the message (an error code is reported in *errno)*.

Note: It is the application's responsibility to release the message buffer returned by ReadMessage().

Note: It is the application's responsibility to read the message(s) it has registered for at a rate greater than the sending applications transmission rate, otherwise the message channel will fill up and new messages will be dropped.

3.5.2.3.2 OutputMessage( )

This function sends a message to other RDS COMM application processes. The function prototype is as follows:

```
int OutputMessage(HEADER *header,
                  MESSAGE_SUBTYPE subtype,
                  MESSAGE_AMPTYPE amptype,
                  int size)
```

The application wanting to send data via UDP, gets a message buffer of the appropriate size for it's message, fills the buffer with the data to be send, and then calls OutputMessage( ) to send it. Receiving processes may be either local or remote nodes on the network.

Note: OutputMessage( ) releases the message buffer allocated by the application.

This function returns a value greater than or equal to 0 if the data is successfully sent to the RDS API or -1 if an error is detected while sending the message (an error code is reported in *errno)*.

3.5.2.3.3 OutputRadarImageMessage( )

This function sends a message to other processes. The function prototype is as follows:

```
int OutputRadarImageMessage(HEADER *header ,
                            MESSAGE_SUBTYPE subtype,
                            MESSAGE_AMPTYPE amptype,
                            int size)
```

The application wanting to send Radar Image data, gets a message buffer of the appropriate size for it's message, fills the buffer with the data to be sent, and then calls OutputRadarImageMessage( ) to send it. Receiving processes may be either local or remote nodes on the network.

This function returns a value greater than or equal to 0 if the data is successfully sent to the RDS API or -1 if an error is detected while sending the message (an error code is reported in *errno)*.

Note: OutputRadarImageMessage( ) releases the message buffer allocated by the application.

For Build 4.6, the data is being sent via TCP/IP.

3.5.2.3.4 OutputControlMessage( )

This function sends a message to other processes. The function prototype is as follows:

```
int OutputControlMessage(HEADER *header,
                         MESSAGE_SUBTYPE subtype,
                         MESSAGE_AMPTYPE amptype,
                         int size,
                         unsigned int destination_address)
```

The application wanting to send data via TCP, gets a message buffer of the appropriate size for it's message, fills the buffer with the data to be send, and then calls OutputControlMessage( ) to send it. The destination_address is the IP address of the node to which the message is to be sent. Receiving processes may be either local or remote nodes on the network.

This function returns a value greater than or equal to 0 if the data is successfully sent to the RDS API or -1 if an error is detected while sending the message (an error code is reported in *errno)*.

Note: OutputControlMessage( ) releases the message buffer allocated by the application.

1.1.1.1

4. Appendices

4.1 Appendix A -Messaging Examples

4.1.1 Message Registration Example

The following code snippet is an example of how an RDS COMM application could register for the system track report message.

```
#include “rds_api.h”
#include “messages_api.h”


CHANNEL_ID RegisterForMessages(void)
{
REG_RECORD *regPtr;
in status;

regPtr = CreateRegistrationRecord();
if(regptr == NULL)
{
   printf(“RegisterForMessage(): CreateRegistrationRecord Failed.\n”);
   exit(-1);
}

status = RegisterMessage(regPtr,
                         CAT_ANY_TRACK_DATA,
                          AMP_SYSTEM_TRACK_REPORT);
if(status < 0)
{
   printf(“ERROR in Registering for message.\n”);
   DeleteRegistrationRecord(regPtr);
   /* continue or break here depending on logic flow */
}

return RegisterForInput(regPtr, NULL);

if(status < 0)
{
   printf(“ERROR in Registering for Input.\n”);
   DeleteRegistrationRecord(regPtr);
   /* continue or break here depending on logic flow */
}

}
```

4.1.2

Message Retrieval Example

This is an example of a function designed to process one specific type of message (a system track report message). The application will wait on the communication receive channel until a message is received. When the message is received, its Cat and Amp Types are checked and the message is processed based on the Cat and Amp_Type. The function then waits for the next message to be received. Notice that the function releases the message buffer allocated for the message when it has finished processing the message.

```
void ProcessMessages(void)
{
SYSTEM_TRACK_RECORD *headerPtr;
SYSTEM_TRACK_TYPE *system_track;

while(1)
   {
   /*  Read the message  */
   /*  The message has already been registered to receive  */
   if (ReadMessage(CHANNEL_ID channel_id, NULL , &he aderPtr) == 1)
      {

      /*  Check the message type  */
      if ((headerPtr->subtype == CAT_ANY_TRACK_DATA) &&
        (headerPtr->amptype == AMP_SYSTEM_TRACK_REPORT))
      {
      SYSTEM_TRACK_RECORD *sys_trk_rec;

      sys_trk_rec = (SYSTEM_TRACK_RECORD *) headerPtr;
      if (sys_trk_rec->num_of_tracks > 0)
         {
         system_track = &(sys_trk_rec->systemTracks[0]);
         }
      }
      ReleaseMessageBuffer(headerPtr);
      }
   }  /*  while  */
```

4.1.3

Message Transmission Example

This is an example of a function designed to send one specific type of message. The application sends a message with a specific CAT and AMP type. Notice the application does not release the message buffer since that is done by the RDS API.

```
void SendMessage(long x, long y, float duration, unsigned long radius)
{
TRACK_ACQUIRE_REQUEST *track_acq_req;
in status;

/*  This code assumes the message buffers have already been initialized  */
/*  Allocate a message buffer  */
track_acq_req = (TRACK_ACQUIRE_REQUEST *)
                     GetMessageBuffer(sizeof(TRACK_ACQUIRE_REQUEST));

/*  Assign message info  */
track_acq_req->x_pos = x;
track_acq_req->y_pos = y;
track_acq_req->duration = duration;
track_acq_req->search_radius = radius;

/*  Send the message  */
status = OutputMessage(track_acq_req,
                   CAT_OP_RP_OPERATOR_ENTRY,
                   AMP_ EXECUTE_MANUAL_SENSOR_TRACK_ACQUIRE,
                   sizeof(TRACK_ACQUIRE_REQUEST))
if(status < 0)
{
   printf("SendMessage(): Failed in outputing the message.\n");
   ReleaseMessageBuffer(track_acq_req);
}

}
```

4.1.4

Read Example

This is an example of how to read a Radar Image message.

```
int Reader(void)
{
int status;
int done = 0;
REG_RECORD *regRecord;
CHANNEL_ID channel_id;
RP_RADAR_IMAGE_MSG *message;

/*  The Message Buffers were created by the RDS startup function, so just attach
*/
status = AttachMessageBuffers();

if (status == -1)
{
   printf("Error Attaching Message Buffers\n");
   exit(-1);
}

/*  Create a message registration record
*/
regRecord = CreateRegistrationRecord();

if (regRecord == NULL)
{
   printf("Error Creating Registration Record\n");
   exit(-2);
}

/*  Register to receive Primary Radar Images messages from Radar 1.
    If there is a failure, then delete registration record
*/
status = RegisterMessage(regRecord, CAT_RADAR_1_ANY_IMAGE,
                                    AMP_PRIMARY_IMAGE);
if (status == -1)
{
   printf("Error registering for Primary Radar Image\n");
   DeleteRegistrationRecord(regRecord);
   exit(-3);
}

/*  Begin the receipt of messages.
    If there is a failure, then delete the registration record
*/
channel_id = RegisterForInput(regRecord, NULL);

if (channel_id == NULL)
{
   printf("Error Registering for Input\n");
   DeleteRegistrationRecord(regRecord);
   exit(-4);
}

while (!done)
```

```
{
   status = ReadMessage(channel_id, NULL , &message);
   if (status  == 1)
   {
      switch (message->subtype)
      {
         case CAT_RADAR_1_ANY_IMAGE:
            if (message->header.amptype == AMP_PRIMARY_IMAGE)
            {
                  processRadarImage(message);
            }
            break;
         default:
            printf("Invalid message subtype received\n");
            done = 1;
            break;
      }  /*  switch subtype  */
      ReleaseMessageBuffer(message);
   }  /*  if a message was received  */
}  /*  while  */

/**********************/
/*  Clean up and exit  */


status = RdsCleanup( channel_id);

if (status ==  -1)
{
   printf("Error in cleaning up file descriptors.\n");
}

status = DetachMessageBuffers( );

if (status ==  -1)
{
   printf("Error in detaching from the message buffer pool.\n");
}

printf("Thats all folks...\n");
return(status);
}  /*  Reader()  */
```

4.1.5

Write Example

This is an example of how to write a message. This can be used in the simulation of the Radar Image. The output rates of the Radar Image are 24, 26 and 60 RPM.

```
int Writer()
{
int status;
int done = 0;
RP_RADAR_IMAGE_MSG* pUncompressedImage;

while (!done)
{
   pUncompressedImage = (RP_RADAR_IMAGE_MSG*)
                           GetMessageBuffer( sizeof(RP_RADAR_IMAGE_MSG) );
   if ( pUncompressedImage == NULL )
   {
      printf("Error Getting Message buffer\n");
      done = 1;
   }

   /*  Read Image from Radar Scan Converter  */
   AquireRadarImage(pUncompressedImage);

   status = OutpuRadarImageMessage(pUncompressedImag,
                                   CAT_RADAR_1_ANY_IMAGE,
                                   AMP_PRIMARY_IMAGE,
                                   sizeof(RP_RADAR_IMAGE_MSG));
   if (status == -1)
   {
      printf("Error Outputting Radar Image Message\n");
      ReleaseMessageBuffer(pUncompressedImage);
      done = 1;
   }
}  /*  while  */

return(0);
}  /*  Writer()  */
```