The Common Operations and Development Environment (CODE) for the WSR-88D Open RPG

CODE B17.0r1.13: May 2016

Includes ORPG Build 17.0r1.13

Volume 3. WSR-88D Algorithm Programming Guide

- API Reference and Algorithm Structure Guidance -

The **U.S. Government Edition** of CODE is the complete version. Distribution is limited to within the United States Government.

The **Public Edition** of CODE is intended for public release. Certain Copyrighted material has been removed to permit release outside the U.S. Government.

CODE provides:

- Instructions for setting up the development environment (includes ORPG source code)
- Guidance for compiling software and configuring new ORPG tasks & products
- Instructions for definition and use of algorithm adaptation data and algorithm dependent parameters
- API Programming Guide and the structure of WSR-88D algorithms (with sample algorithms)
- WSR-88D specific analysis tools
- A set of WSR-88D Archive II Data files and other special test case data.

CODE User provides:

- An Intel PC with Red Hat Enterprise Workstation.

CODE Guide Volume 1. Guide to Setting Up the Development Environment
- Document 1. CODE Specific ORPG Installation Instructions
  - I - Preparation for Installation
  - II - Installation Instructions
  - III - Supplemental Information
  - IV - Running the ORPG
- Document 2. Installing CODE Software
  - I - Software Requisites for CODE Utilities
  - II - Instructions for CODE Utilities
  - III - Instructions for Sample Algorithms

CODE Guide Volume 2. ORPG Application Software Development Guide
- Document 1. The ORPG Architecture
- Document 2. The ORPG Development Environment
  - I - Integrating Development Software with ORPG Source Code
  - II - Compiling Software in the ORPG Environment
  - III - ORPG Configuration for Application Developers
  - IV - Configuring Site Specific Adaptation Data
- Document 3. WSR-88D Final Product Format
  - I - Product Block Structure
  - II - Traditional Product Data Packets
  - III - Generic Product Components
  - IV - ORPG Application Dependent Parameters
- Document 4. ORPG Internal Data for Algorithm Developers
  - I - Base Data Format
  - II - Algorithm Adaptation Data - Configuration & Use
  - III - Other Data Inputs

CODE Guide Volume 3. WSR-88D Algorithm Programming Guide
- Document 1. The WSR-88D Algorithm API Overview
- Document 2. The WSR-88D Algorithm API Reference
  - I - API Service Registration / Initialization
  - II - Control - Input/Output - Abort Services
  - III - Final Product Construction
  - IV - API Convenience Functions
- Document 3. The WSR-88D Algorithm Structure and Sample Algorithms
  - I - WSR-88D Algorithm Structure
  - II - Sample Algorithms
  - III - Writing Product Data Fields
- Document 4. Special Topics
  - I - Topics Related to Using the Development Environment
  - II - Topics Related to Reading Radial Base Data
  - III - Topics Related to Writing Algorithms

CODE Guide Volume 4. CODE Utility Guide
- Document 1. CODEview Text (CVT) - ASCII Product Display
- Document 2. CODEview Graphics (CVG) - Graphic Product Display
  - I - Displaying Products with CVG
  - II - Configuring Products for Display by CVG
- Document 3. Archive II Disk File Ingest - play_a2 Tool
- Document 4. Product Distribution with the nbtcp Tool
- Document 5. Additional CODE / ORPG Tools

Volume 3. WSR-88D Algorithm Programming Guide

Introduction

These documents provide a reference and tutorial on using the WSR-88D Algorithm API along with sample algorithms. Additional guidance for WSR-88D algorithm developers and documentation of ORPG internal data (including base data and final product format) is provided with CODE Guide Volume 2 - *ORPG Application Software Development Guide*.

Document 1. The WSR-88D Algorithm API Overview

A brief overview of the Algorithm API.

Document 2. The WSR-88D Algorithm API Reference

Section I API Service Registration / Initialization
Section II Control - Input/Output - Abort Services
Section III Final Product Construction
Section IV API Convenience Functions

Document 3. WSR-88D Algorithm Structure and Sample Algorithms

Section I Guidance for the Structure of Algorithms
Section II Sample Algorithms
Section III Writing Product Data Fields

Document 4. Special Topics

This document contains miscellaneous topics that have not been placed in other documents. As this type of information increases, a reorganization of documentation will be considered.

Section I Topics Related to Using the Development Environment
Section II Topics Related to Writing Algorithms

Appendices

Provides logic flow diagrams and main module source code listing for the sample algorithms.

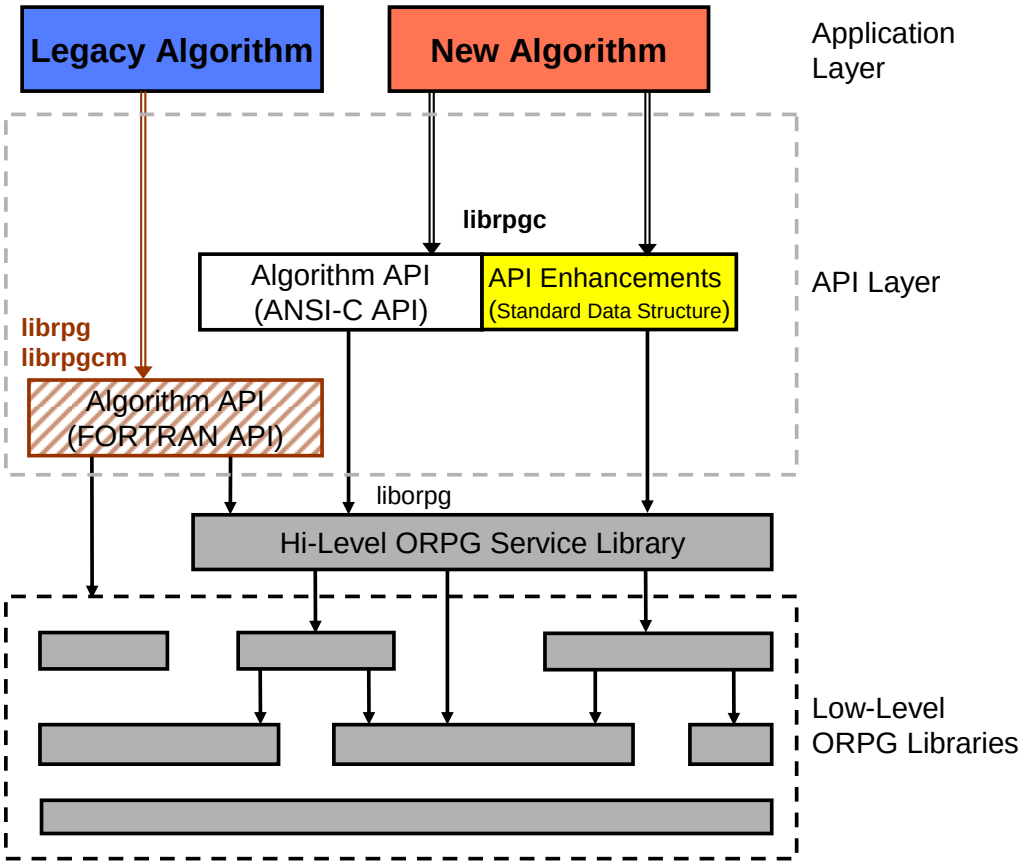
Volume 3. WSR-88D Algorithm Programming Guide

Document 1. WSR-88D Algorithm API Overview

Introduction

All legacy algorithms written in FORTRAN are being ported to C. The FORTRAN binding which provided the interface for the legacy algorithms will be eliminated in a future build. These services are contained in the shared libraries **`librpg`** and **`librpgcm`**.

An ANSI-C binding, provided by **`librpgc`**, is used to write all new algorithms. The relationship between these API libraries and ORPG infrastructure services is depicted by the following diagram. The diagram also indicates the continuing enhancements to the C API (including a standard data structure).



Functions in the API layer have a name beginning with one of the following prefixes: **RPGC_**, **RPGCS_**, or **RPGP_**. **Algorithms should not use lower level infrastructure services.**

Inappropriate Use of non-API Services

Some existing algorithms do not completely follow the guidance and use non-API functions from the main ORPG library of services. These services should not be used in development code and the ROC should not accept algorithms using non-API services. Many of these algorithms are referenced as examples (**cpc007/tsk003/basspect.c, cpc007/tsk013/bref8bit.c, cpc007/tsk014/bvel8bit.c, cpc007/tsk015/superes8bit.c,** etc.) and the use of the following functions in these algorithms should not be considered correct. Most of the ORPG library functions incorrectly used have a direct counterpart in the algorithm API.

Inappropriately Used Service Function	Corresponding Algorithm API Function
LE_send_msg	**RPGC_log_msg**
ORPGPAT_get_elevation_index	**RPGC_get_buffer_elev_index**
ORPGPAT_get_prod_id_from_code	
ORPGPAT_get_num_dep_prods	
ORPGDA_read	**RPGC_data_access_read**
ORPGDA_write	**RPGC_data_access_write**
ORPGDA_seek	**RPGC_data_access_seek**
ORPGDA_clear	**RPGC_data_access_clear**
ORPGDA_open	**RPGC_data_access_open**
ORPGDA_info	**RPGC_data_access_msg_info**
ORPGDA_list	**RPGC_data_access_list**
ORPGDA_lbfd	
ORPGRDA_get_rda_config	
ORPGVCP_BAMS_to_deg	**RPG_elev_angle_BAMS_to_deg**
ORPGVST_get_volume_time	

Summary of Deprecated API Functions

The algorithm API is continuously evolving. Often newly added functions do not add a new capability but rather a modified / improved capability. You will still see the deprecated functions in existing algorithms (including those recently ported from Fortran). However, they should not be used in new

algorithms. The following charts include the old deprecated functions highlighted in red. A list of deprecated functions is provided as a convenient reference.

Deprecated API Function	Replaced by new API Function
RPGC_abort_datatype_because	**RPGC_abort_dataname_because**
RPGC_check_data	**RPGC_check_data_by_name**
RPGC_get_current_elev_index	**RPGC_get_buffer_elev_index**
RPGC_get_current_vol_num	**RPGC_get_buffer_vol_num**
RPGC_get_inbuf	**RPGC_get_inbuf_by_name**
RPGC_get_outbuf_for_req	**RPGC_get_outbuf_by_name_for_req**
RPGC_get_outbuf	**RPGC_get_outbuf_by_name**
RPGC_in_data	**RPGC_reg_inputs** **RPGC_reg_io**
RPGC_in_opt	**RPGC_in_opt_by_name**
RPGC_out_data	**RPGC_reg_outputs** **RPGC_reg_io**
RPGC_out_data_wevent	**RPGC_out_data_by_name_wevent**
RPGCS_get_last_elev_index	**RPGC_is_buffer_from_last_elev**

Algorithm API Chart

Not all of the API functions have been documented. The functions not documented are not critical to algorithm development and many of the recently added functions were created to support the porting of legacy FORTRAN algorithms to ANSI-C.

- Some of the new functions are redundant with existing functions.
- Some are not generally useful for algorithm development.
- Some of the new functions have not yet been used in algorithms.

NOTE: Some previously deprecated functions are being used to port FORTRAN algorithms.

WSR-88D Algorithm API (Build 12)

Color Code	
	Deprecated Functions - Note 1
	Functions not needed for or not appropriate for new algorithms - Note 2
	Functions supporting a very specific purpose / not generally useful - Note 2
	New Functions for Build 12 **(or functions modified in Build 12)**

Note 1: Deprecated Functions. The new or existing function (in the following row) should be used for all new development in place of the deprecated function. Some functions recently deprecated (e.g., `RPGC_get_inbuf` , `RPGC_get_outbuf`) are still used by many algorithms. **This deprecated function will eventually be eliminated from the API.**

Note 2: The functions shaded in the color gray are not documented in Volume 3 Document 2. These functions are not needed to write new algorithms. There are several reasons for this.

a. *Functions not needed for or not appropriate for new algorithms.* Most of these functions will never be documented and some may be deleted from the API. Some of these functions were used to support mechanisms used in Legacy algorithms (for example the ITC blocks and timer support). Some were used in the porting of Fortran algorithms to C and support discontinued methods and should not be used in new algorithms (for example the 'adapt_block' functions). Some were used in the porting of Fortran algorithms to C and are redundant with existing functions.

b. Functions supporting a very specific purpose / not generally useful. These include functions supporting the NCDC model data and functions providing conversion to and from Lambert projections.

Build 12 API

Section I - API Service Registration / Initialization	
	ANSI-C (librpgc) see rpgc.h for prototype definition
A. Input / Output Data Registration	
	int RPGC_in_data(int datatype, **...**)
	int RPGC_out_data(int datatype, **...**)
	int RPGC_reg_io(int argc, char *argv[])
	int RPGC_reg_inputs(int argc, char *argv[])
	int RPGC_reg_outputs(int argc, char *argv[])
Additional Input Output Registration Functions	
	int RPGC_in_opt(int datatype, int block_time)
Build 9	int RPGC_in_opt_by_name(char *data_name, int block_time)
	int RPGC_out_data_wevent(int datatype, en_t event_id, **...**)
Build 9	int RPGC_out_data_by_name_wevent(char *dataname, en_t event_id)
Advanced Access of Replay Data	
	int RPGC_reg_volume_data(int data_id)
B. Adaptation Data Registration	
Supporting Current Adaptation Data	
	int RPGC_reg_ade_callback(int (*callback) (), void *blk_ptr, char *grp_name, int *timing, **...**)
	int RPGC_reg_site_info(void *struct_address)
	*int RPGC_is_ade_callback_reg(int (*update) (), int *status)*
Supporting Old Adaptation Data (used in porting Fortran Algorithms to C)	
Build 10	*int RPGC_reg_adpt(int id, char *buf, int timing, ...)*
Build 10	*int RPGC_is_adapt_block_registered(int block_id, int *status)*
Build 10	*int RPGC_read_adapt_block(int block_id, int *status)*
Build 10	*void RPGC_update_adaptation ( )*
C. Other Registrations	
	int RPGC_init_log_services(int argc, char *argv[])
Build 11	int RPGC_get_input_stream(int argc, char *argv[])
Access to Internal Data Tables	

	int RPGC_reg_scan_summary( )
Build 10	int RPGC_reg_volume_status(Vol_stat_gsm_t *vol_stat)
	int RPGC_reg_moments(int moments)
Using Command Line Parameters	
Build 10	int RPGC_reg_custom_options(const char *additional_options, RPGC_options_callback_t callback)
Other Functions Not Required for New Algorithms	
Build 10	*int RPGC_register_req_validation_fn(char *prod_name)*
Build 9	*int RPGC_reg_color_table(void *buf, int timing, **...**)*
Build 9	*int RPGC_reg_RDA_control(void *buf, int timing, **...**)*
D. Event Registration	
	int RPGC_reg_for_external_event(int event_code, void (*service_routine)(), int queued_parameter)
	int RPGC_reg_for_internal_event(int event_code, void (*service_routine)(), int queued_parameter)
	int RPGC_UN_register(int data_id, LB_id_t msg_id, void (*service_routine)())
E. Task Timing Initialization	
	int RPGC_task_init(int what_based, int argc, char *argv[])
	*int RPGC_reg_and_init(int what_based, int argc, char *argv[])*

Build 12 API

Section II - Control - Input/Output - Abort Services	
	ANSI-C (librpgc) see rpgc.h and rpgcs.h for prototype definition
A. Algorithm Control Loop	
	int RPGC_wait_act(int wait_for)
	int RPGC_wait_for_any_data(int wait_for)
	int RPGC_wait_for_event( )
B. Reading Input Data	
Reading Product Data Linear Buffers	
Build 9	void* RPGC_get_inbuf(int reqdata, int *opstat)
	void* RPGC_get_inbuf_by_name(char *reqname, int *opstat)
	void* RPGC_get_inbuf_any(int *datatype, int *opstat)
	int RPGC_get_inbuf_len(void *bufptr)
	int RPGC_rel_inbuf(void *bufptr)
	int RPGC_rel_all_inbufs( )
Reading Data from Product Database	
Build 10	int RPGC_find_pdb_product(char *name, RPGC_prod_rec_t *results, int max_results, ...)
Build 10	void* RPGC_get_pdb_product(char *name, LB_id_t msg_id)
Build 11	void RPGC_release_pdb_product(void *buf)
Advanced Access of Replay Data	
	int RPGC_check_volume_radial(unsigned int vol_seq_num, int elev_num, unsigned int type)
	char RPGC_read_volume_radial(unsigned int vol_seq_num, int elev_num, unsigned int *type, int *size)*
	*void RPGC_rel_volume_radial(char *bufptr)*
C. Writing Output Data	
Build 9	void* RPGC_get_outbuf(int datatype, int bufsiz, int *opstat)
	void* RPGC_get_outbuf_by_name(char *reqname, int bufsiz, int *opstat)
Build 9	void* RPGC_get_outbuf_for_req(int datatype, int bufsiz, User_array_t *user_array, int *opstat)
	void* RPGC_get_outbuf_by_name_for_req(char *dataname, int bufsiz, User_array_t *user_array, int *opstat)

Build 8	void* RPGC_realloc_outbuf(void *bufptr, int bufsiz, int *opstat)
	int RPGC_rel_outbuf(void *bufptr, int disposition, **...**)
	int RPGC_rel_all_outbufs(int disposition)
	int RPGC_abort_remaining_volscan( )
D. Getting Request Information	
	void *RPGC_get_customizing_data(int elev_index, int *n_requests)
Build 10	int RPGC_get_request_by_name(int elev_index, char *dataname, User_array_t *uarray, int max_requests)
Build 10	*int RPGC_get_customizing_info(int elev_index, User_array_t *user_array, int *num_requests)*
Build 10	*int RPGC_get_request(int elev_index, int prod_id, User_array_t *uarray, int max_requests)*
	int RPGC_check_data(int data_type)
Build 9	int RPGC_check_data_by_name(char *data_name)
E. Getting Information About the Current Volume / Elevation	
	void RPGC_what_moments(Base_data_header *hdr, int *ref_flag, int *vel_flag, int *wid_flag)
	int RPGC_get_current_vol_num( )
	int RPGC_get_buffer_vol_num(void *bufptr)
Build 8	unsigned int RPGC_get_buffer_vol_seq_num(void *bufptr)
	int RPGC_get_current_elev_index( )
	int RPGC_get_buffer_elev_index(void *bufptr)
	int RPGC_get_buffer_vcp_num(void *bufptr)
	int RPGCS_get_target_elev_ang(int vcp_num, int elev_ind)
Build 12	*short *RPGCS_get_elev_index_table(int vcp_num)* Note: for porting **VWINDPRO**
	int RPGCS_get_last_elev_index(int vcp_num)
Build 11	int RPGC_is_buffer_from_last_elev(void *bufptr, int *elev_index, int *last_elev_index)
Build 9	int RPGCS_get_wxmode_for_vcp(int vcp_num)
Build 10	int RPGC_bins_to_ceiling(void *bdataptr, int bin_size)
Build 9	*int RPGC_num_rad_bins(void *bdataptr, int maxbins, int radstep, int wave_type)*
	Scan_Summary *RPGC_get_scan_summary(int vol_num)
Build 9	Vcp_struct* RPGCS_get_vcp_data(int vcp_num)
Build 10	int RPGC_read_volume_status( )
Build 12	*int RPGCS_get_CMD_status( )*

Build 12	*int RPGCS_is_CMD_status_applied_this_elev(int vcp_num, int elev_ind)*
F. Reading Adaptation Data	
	int RPGC_ade_get_values(char *alg_name, char *value_id, double *values)
	int RPGC_ade_get_string_values(char *alg_name, char *value_id, char **values)
	int RPGC_ade_get_number_of_values(char *alg_name, char *value_id)
Functions Not Required for Algorithms	
	*int RPGC_read_ade(int *callback_id, int *status)*
	int RPGC_update_all_ade( )
G. Reading Base Data Messages	
Functions for Reading Basic Moments	
	void* RPGC_get_surv_data(void* bufptr, int* first_good_bin, int* last_good_bin)
	void* RPGC_get_vel_data(void* bufptr, int* first_good_bin, int* last_good_bin)
	void* RPGC_get_wid_data(void* bufptr, int* first_good_bin, int* last_good_bin)
Determining Basedata Message Type	
Build 10	unsigned short RPGC_check_radial_type(void *radptr, unsigned short check_type)
Build 10	unsigned short RPGC_check_cut_type(void *radptr, unsigned short check_type)
Reading Advanced Data Fields (Dual Polarization Data)	
Build 10	void* RPGC_get_radar_data(void* bufptr, int type, Generic_moment_t *mom)
H. Aborting Product Construction	
	int RPGC_abort( )
	int RPGC_abort_because(int reason)
	int RPGC_abort_datatype_because(int datatype, int reason)
Build 9	int RPGC_abort_dataname_because(char *dataname, int reason)
	int RPGC_abort_request(User_array_t *request, int reason)
	int RPGC_cleanup_and_abort(int reason)
Build 10	*int RPGC_product_replay_request_response(int pid, int reason, short *dep_params)*
I. Non-Product Data Access (Not used by legacy algorithms)	
Basic Linear Buffer Access Functions (Note 3)	
	int RPGC_data_access_open(int data_id, int flags)
	int RPGC_data_access_close(int data_id)
Build 9	int RPGC_data_access_list(int data_id, LB_info *list, int nlist)
	int RPGC_data_access_read(int data_id, void *buf, int buflen, LB_id_t msg_id)
	int RPGC_data_access_write(int data_id, void *buf, int buflen, LB_id_t msg_id)
Build 8	int RPGC_data_access_seek(int data_id, int offset, LB_id_t *msg_id)

Build 9	int RPGC_data_access_clear(int data_id, int msgs)
Build 8	int RPGC_data_access_UN_register(int data_id, LB_id_t msg_id, void (*callback) ())
Build 9	LB_id_t RPGC_data_access_previous_msgid (int data_id);
Build 9	int RPGC_data_access_msg_info(int data_id, LB_id_t id, LB_info *info)
Build 9	int RPGC_data_access_stat(int data_id, LB_status *status)
Database Linear Buffer Access Functions (Note 4)	
Build 10	int RPGC_DB_select(int data_id, char *where, void **result)
Build 10	int RPGC_DB_get_record(void *result, int ind, char **record)
Build 10	int RPGC_DB_get_header(void *result, int ind, char **hd)
Build 10	int RPGC_DB_insert(int data_id, void *record, int rec_len)
Build 10	int RPGC_DB_delete(int data_id, char *where)
Standard Disk File Support Functions	
	int RPGC_get_working_dir(char **path_name)
	int RPGC_construct_file_name(char *file_name, char **path_name)
Build 10	*void RPGCS_full_name(char *file_name, char *full_name)*
J. Miscellaneous Functions	
	void RPGC_log_msg(int code, const char *format, **...**)
	*int RPGC_monitor_input_buffer_load(char *dataname)*
	void RPGC_hari_kiri( )
	void RPGC_abort_task( )
Build 10	*void RPGC_kill_rpg( )*

Note 3: The purpose of these functions is to provide linear buffer data storage / access for algorithm data that is not part of the data "flow". Typically non-product data storage is used for state information and does not necessarily change every volume / elevation. Data stored and accessed in this manner are not defined as products. The operation of these functions depends upon the linear buffer being configured via the **`data_attr_table`** configuration file (see CODE Guide Volume 2 Document 2 Section III - *ORPG Configuration for Application Developers*).

Note 4: These functions support linear buffer access to non-product data as a database of records and provide the capability to search the records as well as a read / write / delete capability. Defining a new database which can be accessed by these functions requires modification of an infrastructure task. Use of this feature is restricted to data having numerous records which must be searched.

Build 12 API

Section III - Final Product Construction	
	ANSI-C (librpgc) see rpgc.h and rpgp.h for prototype definition
A. Final Product Construction (Not used by legacy algorithms)	
Functions for Writing and Reading 4-byte Data	
	int RPGC_set_product_int(void *loc, unsigned int value)
	int RPGC_set_product_float(void *loc, float value)
	int RPGC_get_product_int(void *loc, void *value)
	int RPGC_get_product_float(void *loc, void *value)
MISC Product Construction Helper Functions	
Build 8	float RPGC_NINT(float real)
Build 9	double RPGC_NINTD(double value)
Build 9	int RPGC_get_id_from_name(char *data_name)
Build 9	int RPGC_get_code_from_id(int prod_id)
Build 9	int RPGC_get_code_from_name(char *data_name)
	*int RPGC_compress_product(void *bufptr, int method)*
	void RPGC_decompress_product(void *bufptr)*
Data Packet Specific Helper Functions	
Supporting Packet 16 Header and Data Array (packet_16.h)	
Build 10	int RPGC_digital_radial_data_hdr(int first_bin_idx, int num_bins, int icenter, int jcenter, int range_scale_factor, int num_radials, void *output)
Build 8	int RPGP_set_packet_16_radial(unsigned char *packet, short start_angle, short angle_delta, unsigned char *data, int num_values)
Build 10	int RPGC_digital_radial_data_array(void *input, int input_size, int start_data_idx, int end_data_idx, int start_idx, int num_rad_bins, int binstep, int start_angle, int delta_angle, void *output)
Supporting Packet 17 Header and Data Array (for porting the DPA product)	
Build 10	int RPGC_digital_precipitation_data_hdr(int lfm_boxes_in_row, int num_rows, void *output) - *very specific use*
Build 10	int RPGC_digital_precipitation_data_array(void *input, int input_size, int lfm_boxes_in_row, int num_rows, void *output) - *very specific use*
Supporting AF1F and BA07 RLE data arrays	

Build 9	int RPGC_run_length_encode(int start, int delta, short *inbuf, int startix, int endix, int max_num_bins, int buffstep, short *cltab, int *nrleb, int buffind, short *outbuf)
Build 10	int RPGC_run_length_encode_byte(int start, int delta, unsigned char *inbuf, int startix, int endix, int max_num_bins, int buffstep, short *cltab, int *nrleb, int buffind, short *outbuf)
Build 9	int RPGC_raster_run_length(int nrows, int ncols, short *inbuf, short *cltab, int maxind, short *outbuf, int obuffind, int *istar2s)
Functions used to Assemble Header Blocks	
	int RPGC_prod_desc_block(void *ptr, int prod_id, int vol_num)
	int RPGC_set_dep_params(void *ptr, short *params)
	int RPGC_prod_hdr(void *ptr, int prod_id, int *length)
Build 10	int RPGC_set_prod_block_offsets(void *ptr, int sym_offset, int gra_offset, int tab_offset)
	*int RPGC_stand_alone_prod(void *ptr, char *string, int *length)*
B. Support for the Generic Product Format	
Functions Used For Product Construction	
Build 8	int RPGP_build_RPGP_product_t(int prod_id, int vol_num, char *name, char *description, RPGP_product_t *prod)
Build 8	int RPGP_finish_RPGP_product_t(RPGP_product_t *prod, int numof_prod_params, RPGP_parameter_t *prod_params, int numof_components, void **components)
Build 8	int RPGP_set_int_param(RPGP_parameter_t *param, char *id, char *name, int type, void *value, int size, int scale, **...**)
Build 8	int RPGP_set_float_param(RPGP_parameter_t *param, char *id, char *name, int type, void *value, int size, const int fld_width, const int precision, const double scale, **...**)
Build 8	int RPGP_set_string_param(RPGP_parameter_t *param, char *id, char *name, void *value, int size, **...**)
Build 8	int RPGP_product_serialize (RPGP_product_t *prod, char **serial_data)
Build 8	int RPGP_product_free (RPGP_product_t *prod)
Functions Used For Debugging Generic Products	
Build 8	int RPGP_product_deserialize (char *serial_data, int size, RPGP_product_t **prod)
Build 8	void RPGP_print_prod (RPGP_product_t *prod)
Build 8	void RPGP_print_components (int n_comps, char **comps)
Build 8	void RPGP_print_area (RPGP_area_t *area)
Build 8	void RPGP_print_points (int n_points, RPGP_location_t *points)
Build 8	void RPGP_print_params (int n_params, RPGP_parameter_t *params)
Build 8	void RPGP_print_event (RPGP_event_t *event)

Functions Specific to Using External Model Data (Not documented in API Reference, Vol 3 Doc 2) Note 5	
Build 9	int RPGCS_get_model_data(int model, char **data)
Build 9	void* RPGCS_get_model_attrs(int model, char *data)
Build 9	void* RPGCS_get_model_field(int model, char *data, char *field)
Build 9	double RPGCS_get_data_value(RPGCS_model_grid_data_t *data, int level, int i_ind, int j_ind, int *units)
Build 9	void RPGCS_free_model_field(int model, char *data)

Note 5: Functions supporting data access from a specific type of external data from NCDC models (the only "external" data at this time).

Build 12 API

Section IV - API Convenience Functions	
	ANSI-C (librpgc) rpgcs_data_conversion.h, rpgcs_time_funcs.h, and rpgcs_coordinates.h for prototype definition
A. Data Conversion Functions	
	RESO RPGCS_get_velocity_reso(int dop_res)
	RESO RPGCS_set_velocity_reso(int dop_res)
Base Data Value Decoding	
	float RPGCS_reflectivity_to_dBZ(int value)
	float RPGCS_velocity_to_ms(int value)
	float RPGCS_spectrum_width_to_ms(int value)
Base Data Value Encoding	
	int RPGCS_dBZ_to_reflectivity(float value)
	int RPGCS_ms_to_velocity(RESO reso, float value)
	int RPGCS_ms_to_spectrum_width(float value)
Decoding Generic Base Data	
Build 10	int RPGCS_radar_data_conversion(void* data, Generic_moment_t *data_block, float below_threshold, float range_folded, float **converted_data)
B. Date / Time Conversion Functions	
System Date / Time	
Build 9	int RPGCS_get_time_zone( )
Build 10	int RPGCS_get_date_time(int *ctime, int *cdate)
Julian Date Conversions	
	int RPGCS_julian_to_date(int julian_date, int *year, int *month, int *day)
	int RPGCS_date_to_julian(int year, int month, int day, int *julian_date)
	time_t RPGCS_time_span(int start_time, int start_date, int end_time, int end_date)
Unix Time Conversions	
	int RPGCS_unix_time_to_ymdhms(time_t time, int *year, int *month, int *day, int *hour, int *minute, int *second)
	int RPGCS_ymdhms_to_unix_time(time_t *time, int year, int month, int day, int hour, int minute, int second)
Radial Time Conversions	

	int RPGCS_convert_radial_time(unsigned int time, int *hour, int *minute, int *second, int *mills)
C. Coordinate Conversion Functions	
	void RPGCS_set_input_units(int units)
	void RPGCS_set_output_units(int units)
	int RPGCS_get_input_units( )
	int RPGCS_get_output_units( )
Single-Plane Cartesian x y Conversions	
	int RPGCS_xy_to_azran(REAL x, REAL y, REAL *range, REAL *azm)
	int RPGCS_xy_to_azran_u(REAL x, REAL y, int xy_units, REAL *range, int range_units, REAL *azm)
	int RPGCS_azran_to_xy(REAL range, REAL azm, REAL *x, REAL *y)
	int RPGCS_azran_to_xy_u(REAL range, int range_units, REAL azm, int azm_units, REAL *x, REAL *y, int xy_units)
Single-Plane Lat Lon Conversions	
	int RPGCS_xy_to_latlon(float x, float y, float *lat, float *lon)
	int RPGCS_latlon_to_xy(float lat, float lon, float *x, float *y)
	int RPGCS_azran_to_latlon(float rng, float azm, float *lat, float *lon)
	int RPGCS_latlon_to_azran(float lat, float lon, float *rng, float *azm)
Elevation Scan to Ground Cartesian Conversions	
	int RPGCS_azranelev_to_xy(REAL range, REAL azm, REAL elev, REAL *x, REAL *y)
	int RPGCS_azranelev_to_xy_u(REAL range, int range_units, REAL azm, int azm_units, REAL elev, int elev_units, REAL *x, REAL *y, int xy_units)
	int RPGCS_height(REAL range, REAL elev, REAL *height)
	int RPGCS_height_u(REAL range, int range_units, REAL elev, int elev_units, REAL *height, int height_units)
Build 10	int RPGCS_range(REAL height, REAL elev, REAL *range)
Build 10	int RPGCS_range_u(REAL height, int height_units, REAL elev, int elev_units, REAL *range, int range_units)
Window Product Helper	
Build 10	void RPGCS_window_extraction(float radius_center, float azimuth_center, float length, float *max_rad, float *min_rad, float *max_theta, float *min_theta) - *very specific use*
D. Lambert Projection / Grid Conversion Functions (Not documented in API Reference, Vol 3 Doc 2) Note 6	
Build 9	int RPGCS_lambert_grid_point_xy(int i_ind, int j_ind, double *x, double *y)

Build 9	int RPGCS_lambert_grid_point_latlon(int i_ind, int j_ind, double *lat, double *lon)
Build 9	int RPGCS_lambert_grid_point_azran(int i_ind, int j_ind, double *azm, double *ran)
Build 9	int RPGCS_lambert_latlon_to_grid_point(double lat, double lon, int *i_ind, int *j_ind)
Build 9	int RPGCS_lambert_xy_to_grid_point(double x, double y, int *i_ind, int *j_ind)
Build 9	int RPGCS_lambert_latlon_to_xy(double lat, double lon, double *x, double *y)
Build 9	int RPGCS_lambert_xy_to_latlon(double x, double y, double *lat, double *lon)
Build 9	void RPGCS_lambert_init(RPGCS_model_attr_t *model_attr)
Build 9	int RPGCS_lambertuv_to_uv(double ur, double vr, double lon, double *u, double *v)

Note 6: Functions supporting conversions between a specific Lambert projection coordinates to various grid coordinates are useful only for reading external model data that are in that specific projection.

Build 12 API

Section V - Legacy Algorithm Support	
	ANSI-C (librpgc) see rpgc.h for prototype definition
Inter-task common block (ITC) support (Not documented in API Reference, Vol 3 Doc 2) Note 7	
	*int RPGC_itc_in(int itc_id, void *first, unsigned int size, int sync_prod, **...**)*
	*int RPGC_itc_out(int itc_id, void *first, unsigned int size, int sync_prod, **...**)*
	*int RPGC_itc_callback(int itc_id, int (*func)())*
	*int RPGC_itc_read(int itc_id, int *status)*
	*int RPGC_itc_write(int itc_id, int *status)*
Timer Support (Not documented in API Reference, Vol 3 Doc 2)	
	*int RPGC_reg_timer(malrm_id_t id, void (*callback)())*
	int RPGC_set_timer(malrm_id_t id, int interval)
	int RPGC_cancel_timer(malrm_id_t id, int interval)

Note 7: Several Legacy Fortran algorithms used a mechanism of sharing persistent data called 'Inter-Task Communication (ITC) Blocks'. Support for this communication mechanism was implemented in the ORPG infrastructure. Even though the C Algorithm API includes functions to use ITC blocks, they are not recommended for use in new algorithms. The non-product data access functions should be used to store and share non-product data.

Volume 3. WSR-88D Algorithm Programming Guide

Document 2. The WSR-88D Algorithm API Reference

The following API discussion is oriented toward a developer writing a new algorithm in ANSI-C so topics such as ITC blocks and timer services (Legacy Algorithm Support) are not included.

> **New algorithm development in FORTRAN is no longer supported by the National Weather Service. The WSR-88D algorithms written in FORTRAN are being ported to ANSI-C during the next few development cycles.** Development activities must be coordinated with the Radar Operations Center Engineering Branch if modifying an existing FORTRAN algorithm. A list of recently ported algorithm tasks is provided in Appendix B.

Note: The reader should be familiar with the instructions for configuring new algorithms and the description of ORPG configuration files contained in CODE Guide Volume 2 - *ORPG Application Software Development Guide*. Some terms used here (e.g., **`<Prod_Buffer_Number>`**, **`<Prod_Buffer_Name>`**, and **`<Product_Code>`**) are defined in Document 2 Section III of that guide.

Section I API Service Registration / Initialization

The service registration / initialization routines are called only when the algorithm task is launched. These initialize internal data structures that enable the underlying infrastructure to function correctly.

Section II Control - Input/Output - Abort Services

After registrations and initialization, the algorithm enters into the processing phase. The algorithm remains in this phase until the ORPG is shut down or the task fails.

Section III Final Product Construction

This section documents functions that aid in the construction of final products (products distributed to external users).

Section IV API Convenience Functions

Currently the API convenience functions consist of standard services for encoding and decoding radar data moments, for accomplishing date and time conversions and for transforming polar coordinate data into rectangular coordinates.

Vol 3. Document 2 -
The WSR-88D Algorithm API Reference

Section I API Service Registration / Initialization

NOTE: Preliminary support for Dual Pol data was provided in Build 11 in order to support implementation. Actual Dual Polarization implementation is provided in Build 12.

The service registration / initialization routines are called only when the algorithm task is launched. These initialize internal data structures that enable the underlying infrastructure to function correctly. Note: These registration services do not completely "register" data stores and tasks with the ORPG. The configuration files **`product_attr_table`** and **`task_attr_table`** described in Volume 2, Document 2, Section III must be modified to actually register inputs and outputs.

NOTE

Not all of the API functions have been documented. The functions not documented are not critical to algorithm development and many of the recently added functions were created to support the porting of legacy FORTRAN algorithms to ANSI-C.

- Some of the new functions are redundant with existing functions.
- Some are not generally useful for algorithm development.
- Some of the new functions have not yet been used in algorithms.

NOTE: Some previously deprecated functions are being used to port FORTRAN algorithms.

BUILD 12 CHANGES:

- Updated discussion of event driven algorithms and the event registration functions**.** Reemphasized that normal meteorological algorithms usually have no need to register for events. Documented the **`ORPGEVT_START_OF_VOLUME_DATA`** event as the safest beginning volume event to use in a meteorological algorithm.

BUILD 11 CHANGES:

- Documented new function **`RPGC_get_input_stream`**.
- Documented the relationship between input product timing and output product timing and the two unique exceptions to the rule that output timing must be greater than the largest input timing.
- Expanded the description of mixing input timing types.

Input / output data registration, *task timing initialization*, and *summary scan registration* calls are required for all algorithms. *Input data registration* is not required for event driven algorithms not using product data as input. The *task timing initialization* must follow all others. See the discussion of *algorithm control loop* under Section II of this document (*Control - Input/Output - Abort Services*) for an explanation of how the registration / initialization services affect algorithm processing.

NOTE: This document makes multiple references to two forms of a data driven algorithm control loop. This is the continuous main loop of an algorithm that is entered after all registrations and initializations are complete.

1. The WAIT_ALL form of the control loop is the most common and is used in algorithms that have only one data input or need more than one data input to create a product. The loop control function used is **`RPGC_wait_act(WAIT_DRIVING_INPUT)`**
2. The WAIT_ANY form of the control loop is used in algorithms having multiple registered data inputs but need only one input to produce a product. There are two functions that can be used: **`RPGC_wait_act(WAIT_ANY_INPUT)`** and **`RPGC_wait_for_any_data(WAIT_ANY_INPUT)`**.

The contents of this section is organized as follows:

- Part A. Input / Output Data Registration
- Part B. Adaptation Data Registration
- Part C. Other Registrations (log services / internal data / command parameters)
- Part D. Event Registration
- Part E. Task Timing Initialization

Part A. Input / Output Data Registration

All input data (i.e., base data and any product data produced by another algorithm) and all output data for each algorithm task must be registered. Input and output data registration applies to *products* each of which has an entry in the product attribute table contained in the **product_attr_table** configuration file.

Input / Output Registration

Beginning with Build 9 the **RPGC_reg_io** function is used to register input and output data. The products listed in the **input_data** and **output_data** attributes of the task's entry in the **task_attr_table** file are registered. This function can be used in all data driven algorithm tasks.

The functions **RPGC_reg_inputs** and **RPGC_reg_outputs** could also be used. Their primary use would be in event driven tasks that have product input but no product output or vice versa.

> The data timing attribute of both input and output product data are determined by the value of the **type** attribute for that product entry in the **task_attr_table** file. The values for the **type** attribute are related to (but not the same as) the input/output data type definitions in **rpg_port.h**.. The input/output data type definitions in **rpg_port.h** are used by the infrastructure and some deprecated API registration functions. See the topic "Configuring a New Product" in Volume 2, Document 2, Section III, Part C.

.

value of the **prod_attr_table type** attribute	corresponds to	Data Timing Name This name was used with the deprecated registration functions.
type 0		**VOLUME_DATA**
type 1		**ELEVATION_DATA**
type 2		**TIME_DATA** **- not currently supported**
type 3		**DEMAND_DATA**
type 4		**REQUEST_DATA** **- special case**
type 5		**RADIAL_DATA**

> **Note**: The following functions have been deprecated and should not be used.
>
> - **RPGC_in_data**
> - **RPGC_out_data**
>
> They will be eliminated in a future build. If any development software uses these functions they should be replaced with **RPGC_reg_io**.

`int RPGC_reg_io( int argc, char *argv[] )`

PARAMETER:	**`argc`**	From task's main function, used by API infrastructure.
	`*argv[]`	From task's main function, used by API infrastructure.
RETURN VALUE:		With problem registering inputs/outputs, the task self terminates; returns 0 on success.

Registers all of the data inputs & outputs for the task based upon the **`task_attr_table`** and **`product_attr_table`** configuration.

EXAMPLE:

`int RPGC_reg_inputs( int argc, char *argv[] )`

PARAMETER:	**`argc`**	From task's main function, used by API infrastructure.
	`*argv[]`	From task's main function, used by API infrastructure.
RETURN VALUE:		With problem registering inputs/outputs, the task self terminates; returns 0 on success.

Registers all of the data inputs for the task based upon the **`task_attr_table`** and **`product_attr_table`** configuration.

EXAMPLE:

`int RPGC_reg_outputs( int argc, char *argv[] )`

PARAMETER:	**`argc`**	From task's main function, used by API infrastructure.
	`*argv[]`	From task's main function, used by API infrastructure.
RETURN VALUE:		With problem registering inputs/outputs, the task self terminates; returns 0 on success.

Registers all of the data outputs for the task based upon the **`task_attr_table`** and **`product_attr_table`** configuration files.

EXAMPLE:

Notes / Rules for use (for a data driven task):

Input Data Notes:

1. Multiple data inputs are allowed (8 maximum).
2. There can only be one **`RADIAL_DATA`** input. With respect to legacy algorithms, all algorithms that have a **`RADIAL_DATA`** input have no other product input. Now it is possible to mix non radial inputs with a **`RADIAL_DATA`** driving input. See rule 4 below.
3. Multiple data inputs - order of registration

- In the case where the algorithm blocks until a specific input is available (the WAIT_ALL form of the algorithm control loop), the first (or only) product listed in the **input_data** attribute of the **task_attr_table** entry for the algorithm task is the "driving" input. If there is more than one input, the driving input must be the first data input actually read in the body of the algorithm.
- In the case where the algorithm blocks until any one of several inputs are available (the WAIT_ANY form of the algorithm control loop), the order of products listed in the **input_data** attribute is not important.

4. Multiple data inputs - input timing type
   - In the case where the algorithm blocks until a specific input is available (the WAIT_ALL form of the algorithm control loop), inputs in addition to the "driving" are typically of the same timing (**ELEVATION_DATA, VOLUME_DATA, DEMAND_DATA**) or must be specified as a time-based input. Time based inputs are not yet supported by the Algorithm API. A mix of input timing types is allowed with restrictions:
     - **ELEVATION_DATA** and **VOLUME_DATA** can be mixed if the driving input is **ELEVATION_DATA**. In addition, the programmer must assure that the **VOLUME_DATA** being input is from the same volume as the driving input. One way to accomplish this is to read the volume input after reading all elevation inputs needed.
     - It is possible to mix **RADIAL_DATA** and **ELEVATION_DATA** inputs in the same manner (used in Sample Algorithm 4). The driving input must be **RADIAL_DATA**. When obtaining the **ELEVATION_DATA** input, the programmer must be sure it is asked for after the acquisition of the first radial of the expected elevation/volume and before the acquisition of the first radial of the next elevation/volume.
   - In the case where the algorithm blocks until any one of several inputs are available (the WAIT_ANY form of the algorithm control loop), the timing types can be any combination but cannot include **RADIAL_DATA**..
5. Only a non-driving input within a WAIT_ALL form of algorithm control loop can be designated as an optional input.

Output Data Notes:

1. Each algorithm task must have at least one registered output.
2. Multiple data outputs are allowed (8 maximum).
3. Most algorithms with multiple outputs have either all **ELEVATION_DATA** or all **VOLUME_DATA** outputs. **DEMAND_DATA** is used for data that is not directly related to the volumetric radar data. An algorithm can output **DEMAND_DATA** even if there is no request for it.
4. The following types have been mixed:
   - **VOLUME_DATA** and **ELEVATION_DATA**
   - **VOLUME_DATA** and **DEMAND_DATA**.

   In both cases the task timing is **VOLUME_BASED**.
5. There is no documented restriction on the order that multiple products are produced.

Input Timing related to Output Timing

1. Normally the timing of the input data must be equal to or less than the timing of the output data. For example, a **VOLUME_DATA** output could have any input timing with the restrictions noted above. But an **ELEVATION_DATA** output should not have a **VOLUME_DATA** input. There are two unique exceptions to this.

a. The **tdaruprod** task and the **mesoruprod** task. The only reason this makes sense is that these tasks have other **ELEVATION_DATA** inputs and also have a **WAIT_ANY** type of control loop. In these algorithms the **ELEVATION_DATA** input actually drives the production of the **ELEVATION_DATA** output. The **VOLUME_DATA** input only serves to store this data for use in the processing of the elevation data in the next volume scan.
b. The **alerting** task which is event driven and the **combattr** task also mix **ELEVATION_DATA** and **VOLUME_DATA** inputs but are not exceptions because the output is **VOLUME_DATA**.

Additional Input / Output Registration

There are two cases where an additional registration function is needed.

Designating an Input as Optional
After registering input data a subsequent call to **RPGC_in_opt_by_name** designates a specified input as optional and prevents the data read function from blocking indefinitely if the product is not available. There is another method for designating an input as optional using the **product_attr_table** described in Volume 2, Document 2, Section 3. As of Build 9, it is recommended that the **product_attr_table** method be used and the default block time be changed with **RPGC_in_opt_by_name**.

Posting an Event on Product Output
After registering output data a subsequent call to **RPGC_out_data_by_name_wevent** results in a specified event being posted when the product is output.

Note: The following functions have been deprecated and should not be used.

- **RPGC_in_opt**
- **RPGC_out_data_wevent**

They will be eliminated in a future build. If any development software uses these functions they should be replaced with **RPGC_in_opt_by_name** and **RPGC_out_data_by_name_wevent**.

```
int RPGC_in_opt_by_name( char *data_name, int timing )
```

PARAMETER:	**data_name**	The product name.
	timing	The maximum block time (in seconds) when the data is read.
RETURN VALUE:		Not Used

Designates the named registered input as optional and sets the maximum block time. Must be used after the input is registered with any of the registration functions. A product can also be designated as optional via an entry in the **product_attr_table** configuration file. However, this function must be used to set the block time if the default time is not desired. Only non-driving inputs with a WAIT_ALL form of control loop can be designated as optional. See *Reading Input Data* for an explanation of how this registration affects **RPGC_get_inbuf_by_name** function which reads the

input data.

EXAMPLE: cpc005/tsk002/pcipdalg.c

Parameter Descriptions

data_name

The name of the product as listed in the **input_data** attribute of the **task_attr_table** entry for the algorithm task. The **input_data** entry determines the number or ID of the linear buffer that is the source of the input data. See CODE Guide Volume 2, Document 2, Section III for a complete explanation of product and task configuration via the **product_attr_table** and **task_attr_table** configuration files.

timing

The maximum block time (in seconds) when the input is read by **RPGC_get_inbuf_by_name**. A typical block time is 4. The maximum recommended is 10.

Notes / Rules for use (for a data driven task):

1. Only a non-driving input within a WAIT_ALL form of algorithm control loop can be designated as an optional input.
2. The **opt_prods_list** attribute in the **product_attr_table** file can be used instead of this function to designate the input as optional. The default block time in this case is 0 seconds in Build 9 and increased to 5 seconds in Build 10. This function can be used to override the default block time.

```
int RPGC_out_data_by_name_wevent( char *dataname, en_t event_id )
```

PARAMETER: **dataname** The output product name.

event_id The event that is posted when the product is produced.

RETURN VALUE: Not Used

Designates an event that will be posted when the named product is written to the product linear buffer. One example of an event is **ORPGEVT_CPC4MSG** which is posted when the product **CPC4MSG** is generated. NOTE: There can be a maximum of 8 registered data outputs for an algorithm task.

EXAMPLE: cpc005/tsk002/pcipdalg.c
cpc008/tsk001/alerting.c

Parameter Descriptions

dataname

The name of the product as listed in the **output_data** attribute of the **task_attr_table** entry for the algorithm task. The **output_data** entry determines the number or ID of the linear buffer that will be associated with the posted event. See CODE Guide Volume 2, Document 2, Section III for a complete explanation of product and task configuration via the **product_attr_table** and **task_attr_table** configuration files.

event_id

The identity of the ORPG event that is posted when the product is produced. Available ORPG events are defined in **orpgevt.h**.

Notes / Rules for use (for a data driven task):

1.

Advanced Access of Replay Data

The following registration function, along with three access functions listed in Part B of Section II, support a more advanced method of accessing replay data. This function was added in Build 9 and is currently not used in any algorithm. If this capability is determined to be generally useful the function will be documented.

```
int RPGC_reg_volume_data( int data_id )
```

Part B. Adaptation Data Registration

Supporting Current Adaptation Data

From the application's point of view, the major purpose of adaptation data is to parameterize certain characteristics of an algorithm to permit changing its behavior using ORPG configuration files. Adaptation data also include site specific information and the default color table associated with certain products. A 'DEA access function' is written (using API functions to read individual data elements). This function can be part of the algorithm itself or included in a shared library.

With the previous adaptation data mechanism, the adaptation data had to be registered. With the new DEA mechanism, registration of the adaptation data is optional because the "DEA access function" can be implemented as part of the algorithm and be called from within the algorithm. Though not required, most adaptation data "access functions" in existing algorithms have been registered as callbacks. If an access function is registered as a callback, the function must have a single parameter which is the address of the data structure containing fields representing the individual data elements. The access function updates the fields in this structure.

RPGC_reg_ade_callback registers the 'DEA data access function' as a callback function which automatically updates the adaptation data as specified. The use of **RPGC_reg_ade_callback** is only required if you wish reduce resource use.

RPGC_reg_site_info reads the site specific adaptation data into a C structure for use by the algorithm.

For guidance on how to define, create, and install adaptation data for use by an algorithm, see the CODE Guide Volume 2, Document 3, Section III - *Algorithm Adaptation Data - Configuration & Use*.

```
int RPGC_reg_ade_callback( int (*callback) (), void *blk_ptr, char *grp_name,
                           int *timing, ... )
```

PARAMETER:	**callback**	Name of the callback function.
	blk_ptr	Pointer to adaptation data structure
	grp_name	Name of adaptation data group
	timing	Frequency of update.
	[*optional*]	The event id that triggers update for **ADPT_UPDATE_ON_EVENT** timing.
RETURN VALUE:		Returns an identifier for this registration. If error occurs, returns -1.

Registers a callback function which reads and updates adaptation data.

EXAMPLE:

```
rc = RPGC_reg_ade_callback( read_my_adapt,
          &sample1_adapt, "alg.sample1_dig.",
          ADPT_UPDATE_BOV );
if ( rc < 0 ) {
   RPGC_log_msg( GL_INFO,
       "SAMPLE1: cannot register callback function\n");
```

```
}
```

where **sample1_adapt** is the designated structure and the callback is of the form **int read_my_adapt(int).**

Parameter Descriptions

callback

The name of the data access function used to read and update the adaptation data. For development purposes, this function should be implemented as part of the algorithm. The data access function must have a return value of type **int** with one parameter that is the address of a data structure containing the fields to be updated.

blk_pointer

A pointer to a C structure that contains the adaptation data fields. This structure is used within the data access function to store updated adaptation data for local use by the algorithm.

grp_name

The adaptation group name. With sample algorithm 1 for example, the adaptation data file is named **sample1_dig.alg**. The group name is based on the inverse of the filename: "**alg.sample1_dig.**" (note the trailing period). This represents the upper level structure of an ORPG hierarchical database.

timing

Frequency of update. This parameter can have the following values defined in **rpg_port.h**: **ADPT_UPDATE_BOE** - beginning of elevation, **ADPT_UPDATE_BOV** - beginning of volume, **ADPT_UPDATE_ON_CHANGE** - when data changes, **ADPT_UPDATE_WITH_CALL**, **ADPT_UPDATE_ON_EVENT** - when event is posted.

[*optional*]

This is the ORPG event id, type **en_t** (which is an integer), that triggers an update of adaptation data. This parameter is used when the timing is **ADPT_UPDATE_ON_EVENT**. ORPG events (*external* events) are defined in **orpgevt.h**. See the next section for a definition of *internal* and *external* events.

Notes / Rules for use

1. The use of **RPGC_reg_ade_callback** is only required if you wish reduce resource use.
2. Most existing functions use **ADPT_UPDATE_BOV** and a few use **ADPT_UPDATE_ON_CHANGE**. Currently **ADPT_UPDATE_BOE** is not used for adaptation data.

int RPGC_reg_site_info(void *struct_address)

PARAMETER: **struct_address** Pointer to a **site_adapt** structure

RETURN VALUE: Returns >= 0 if successful, < 0 if not.

Registers a structure to get the latest site information adaptation data. The current version of the data are entered into the **site_adapt_t**.

EXAMPLE:

```
rc = RPGC_reg_site_info( &site_adapt );
if ( rc < 0 ) {
   RPGC_log_msg( GL_INFO,
      "SITE INFO: cannot register callback function\n");
}
```

where **site_adapt** is of type **struct Siteadp_adpt_t**

Parameter Descriptions

`struct_address`

Pointer to a **`site_adapt`** structure which is defined in **`siteadp.h`** to be populated.

Notes / Rules for use

1.

The following function is not required to implement an algorithm. If this capability is determined to be generally useful the function will be documented.

```
int RPGC_is_ade_callback_reg( int (*update) (), int *status )
```

Supporting Old Adaptation Data (used in porting Fortran to C)

The following functions use an older, unapproved mechanism for adaptation data. These functions are not approved for new algorithms and may be removed from the API at a later date.

```
int RPGC_reg_adpt( int id, char *buf, int timing, ...)

int RPGC_is_adapt_block_registered( int block_id, int *status )

int RPGC_read_adapt_block( int block_id, int *status )

void RPGC_update_adaptation ( )
```

Part C. Other Registrations (log services / internal data / command params)

Log Services Registration

The **RPGC_init_log_services** function initiates the internal ORPG log services. Since these services are initiated with the task initialization function, the only purpose this function serves is to initiate these services before other registration functions.

```
int RPGC_init_log_services( int argc, char *argv[] )
```

PARAMETER:	**argc**	Not used.
	***argv[]**	Not used.
RETURN VALUE:		Not Used

Initializes log services for the task. Currently the only benefit of using this function is to initialize log services before the **RPGC_task_init** statement. This would permit recording of messages during the buffer initialization and adaptation data initialization that is accomplished before proceeding to the algorithm control loop.
EXAMPLE:

Notes / Rules for use

1. Use of **RPGC_init_log_services** is not required. Currently the only advantage of using this function is to have log services running during the initialization section of the algorithm prior to entering the algorithm control loop.

Determining the Input Stream

The **RPGC_get_input_stream** function is used to determine the input stream (real-time or the alternate replay stream). This function is not required for the normal. data driven tasks having a second instance using the replay stream. It is required for event driven algorithms having a replay instance.

```
int RPGC_get_input_stream( int argc, char *argv[] )
```

PARAMETER:	**argc**	Not used.
	***argv[]**	Not used.
RETURN VALUE:		Either **PGM_REALTIME_STREAM** or **PGM_REPLAY_STREAM** on success, or -1 on error.

This function obtains the type of input data steam (the normal realtime stream or the alternate replay stream) the task is using. **PGM_REALTIME_STREAM** and **PGM_REPLAY_STREAM** are defined in `prod_gen_msg_h`.

EXAMPLE:

Notes / Rules for use

RPGC_get_input_stream is called during the registration / initialization part of the algorithm before reaching the main algorithm loop.

Access to Internal Data Tables

The function **RPGC_reg_scan_summary** registers the *Scan Summary Table* by obtaining a memory address to be referenced. The *Scan Summary Table* contains information about the current volume scan that is used by various parts of the ORPG infrastructure. This registration is no longer required with C algorithms if appropriate helper functions are used (**RPGC_prod_desc_block** and **RPGC_get_scan_summary**).

```
int RPGC_reg_scan_summary( )
```

PARAMETER: None

RETURN VALUE: Not Used (currently, always returns 0)

Registers the *Scan Summary Table* for those parts of the infrastructure that require access of this data.

EXAMPLE:

Notes / Rules for use

1. **RPGC_reg_scan_summary** can be used in all algorithms. However, if the supplied helper functions are used there is no need to register for scan summary in algorithms using the C API.

The following function registers and accomplishes the initial read of the volume status portion of the general status message. Another function, listed in Part E of Section II, accomplishes subsequent reading of the data. Most of the information is also available in the scan summary table or even the base data header.

However, this function is useful in event driven algorithms and must be used with the corresponding reading function if the even driven algorithm has no data input. It is the primary method of determining the volume number, sequence number, and the volume date - time if not registered for product inputs.

```
int RPGC_reg_volume_status( Vol_stat_gsm_t *vol_stat )
```

PARAMETER: **vol_stat** address of a volume status structure

RETURN VALUE: Always returns 0.

Description paragraph

EXAMPLE: cpc008/tsk001/alerting.c

Parameter Descriptions

vol_stat

The address of a pre-allocated volume status structure of type **Vol_stat_gsm_t.** See **gen_stat_msg.h.**

Notes / Rules for use (for a data driven task):

1. If the task has product inputs, the structure returned by this function will be automatically updated each volume and the associated read function need not be called. If the task has no product inputs the associated read function **RPGC_read_volume_status** must be called every volume.

The function **RPGC_reg_moments** is used to tell the API services which moments of basedata or rawdata are desired. If the moment is not available

- for the driving input in a WAIT_ALL form of control loop, the loop control function does not release if the registered moments are not available.
- for other inputs the get_inbuf_by_name function fails (does not return **RPGC_NORMAL**) if the registered moments are not available.

```
int RPGC_reg_moments( int moments )
```

PARAMETER: **moments** Desired basedata moments.

RETURN VALUE: Returns -1 if the specified moments do not correspond to the registered input data or if no basedata inputs are registered..

Registers the base data moments needed for correct algorithm function. On error, this functions has no affect other than to write an error message to the task's log file.

EXAMPLE: **RPGC_reg_moments(my_moments);**
where **int my_moments = (REF_MOMENT | VEL_MOMENT);** if interested in velocity and reflectivity.

Parameter Descriptions

moments

The base data moments required for this algorithm **OR**'d together. The three moments are **REF_MOMENT**, **VEL_MOMENT**, and **WID_MOMENT** (defined in **rpg_port.h**).

Notes / Rules for use

1. When needed moments are registered with RPGC_reg_moments, then **RPGC_what_moments** does not have to be used when reading the first base data radial / elevation message from a volume (because the call to "get_inbuf" will return an error).

Using Command Line Parameters

Though not generally useful, this function provides the capability of reading any custom command line parameters used to launch the algorithm task. The registered callback function accomplishes the processing desired in the algorithm.

```
int RPGC_reg_custom_options( const char *additional_options,
                             RPGC_options_callback_t callback )
```

PARAMETER:	**additional_options**	user-defined command line arguments
	callback	name of callback routine to process options
RETURN VALUE:		0 on success, -1 on error

Registers a user-supplied callback routine to process custom (user-defined) command line options. A user-defined option will not be processed if it is identical to a pre-defined command line option (currently: **T**, **l**, and **h**).

EXAMPLE: See **superres8bit_main.c** in cpc007/tsk015

Parameter Descriptions

additional_options

A string containing user-defined command line options that will be processed by the callback function **callback**. The options are in two possible forms:

- a single character, for example **"A"**
- a single character followed by a colon, for example **"v:"**

If the algorithm used both of these command line options the format would be **"v:A"** or **"Av:"**. Reviewing the man page of **getopt(3)** will aid in the understanding of the permissible format of command line options.

callback

The name of a user-supplied callback function. This function must be of type **RPGC_options_callback_t**.(defined in **rpgc.h**).

```
typedef int (*RPGC_options_callback_t)( int arg, char *optarg );
```

The first parameter of this function is passed the return value of **getopt**. The second parameter of this function will point to the argument (if any) for the defined option. The value of **optarg** is only meaningful if the option is a character followed by a colon. Reviewing the man page of **getopt(3)** will aid in the understanding of the parameter **optarg**.

Notes / Rules for use (for a data driven task):

1. This function should be call before any other registration function. If called after **RPGC_init_log_services** or **RPGC_task_init**, the custom options will not be processed.
2. Since the RPG uses **getopt** in the standard fashion when passing parameters to the defined **callback**, all options in the command line must begin with a **'-'** and all options must be before any element of **argv[]** that is not an option or argument for an option.

Other Functions Not Required for New Algorithms

The following function was added in Build 10 and is not yet used by any algorithm. If this capability is determined to be generally useful the function will be documented.

`int RPGC_register_req_validation_fn( char *prod_name )`

Two new registration functions were included with Build 9 to facilitate porting of the Legacy FORTRAN algorithms to C. These functions are not required for general algorithm development.

`int RPGC_reg_color_table( void *buf, int timing, ... )`

This function registers a predefined color table. A color table is actually a table that assigns a data value having up to 255 levels into one of 16 levels used for run length encoded (RLE) products. Since data levels are part of the product's specification, there is no need to implement the table in adaptation data.

`int RPGC_reg_RDA_control( void *buf, int timing, ... )`

Registers the **`RDACNT`** (RDA Control) block of adaptation data. This is the only block of adaptation data that remains in Legacy format. It is not needed by new product algorithms.

Part D. Event Registration

Virtually all algorithms should be data driven. That is the algorithm begins processing with the availability of input product data. Very few meteorological algorithms have a need to register for events. Most existing algorithms using events are involved in system monitoring and control, for example pcipdalg, cltutprod).

- The main reason for using an event driven algorithm would be if the task had non-product data inputs and no product inputs. In this case an alternative would be to have a driving product data input of the desired timing (elevation or volume) to act as a trigger to activate the algorithm. This alternative should be used if the task also has a replay version.
- Another reason for using an event driven algorithm is an algorithm that does not function with the data flow timing of the ORPG. In this case any output product produced is not **RADIAL_DATA**, **ELEVATION_DATA**, or **VOLUME_DATA**. The input product (if any) would be **DEMAND_DATA** and the output data also **DEMAND_DATA**.

The registering of events can be used to interrupt the algorithm and accomplish processing via a callback routine whenever an ORPG event is raised. This feature can be used with data driven algorithms (**RPGC_wait_act** or **RPGC_wait_for_any_data** used for the loop control) or with event driven algorithms (**RPGC_wait_for_event** used for loop control).

An event driven algorithm can be used for tasks that do not have any product input data. The algorithm begins processing upon posting of a registered event rather than the availability of input data.

External events are global to the ORPG. That is, any ORPG task can post an event and register for and respond to an event. ORPG algorithm tasks use this API to register and respond to events. Other ORPG tasks use lower level services. The most useful external event is **ORPGEVT_START_OF_VOLUME_DATA**.

Internal events only have meaning to ORPG algorithm tasks using the Algorithm API. They are actually defined within the algorithm API. The internal event that is generally useful is **EVT_ANY_INPUT_AVAILABLE**.

```
int RPGC_reg_for_internal_event( int event_code, void (*service_routine)(),
                                 int queued_parameter )
```

PARAMETER:	**event_code**	ID of event being registered.
	(*service_routine) ()	Address of a function **service_routine** which contains the logic to be executed when the event is posted.
	queued_parameter	A parameter that will be passed to the event service routine when called.
RETURN VALUE:		Not used.

Registers one of the two internal events that are defined.

EXAMPLE: **RPGC_reg_for_internal_event(EVT_ANY_INPUT_AVAILABLE, &ALG_Event_Handler, (int)0);**
where **ALG_Event_Handler** is a function with one parameter of type **int**, **EVT_ANY_INPUT_AVAILABLE** is an event that is posted when any one of the registered data inputs is available

```
int RPGC_reg_for_external_event( int event_code, void (*service_routine)(),
                                 int queued_parameter )
```

PARAMETER:	**event_code**	ID of event being registered.
	(*service_routine) ()	Address of a function **service_routine** which contains the logic to be executed when the event is posted.
	queued_parameter	A parameter that will be passed to the event service routine when called.
RETURN VALUE:		Not used.

Registers one of the external events that are defined.

EXAMPLE: **RPGC_reg_for_external_event(ORPGEVT_START_OF_VOLUME_DATA, &ALG_Event_Handler, (int)0);**
where **ALG_Event_Handler** is a function with one parameter of type **int**, **ORPGEVT_START_OF_VOLUME_DATA** is a globally defined event that is posted at every start of volume scan.

Parameter Descriptions

event_code

The identity of the registered event.

- The one *internal* event of general interest: **EVT_ANY_INPUT_AVAILABLE**. This event is posted by the API infrastructure when any of the algorithm's registered data inputs is available.
- *External* events are global to the ORPG. They are defined in **orpgevt.h**. The one external event of general interest is **ORPGEVT_START_OF_VOLUME_DATA** is at the beginning of the volume scan. **It is safer to use that the previously documented event because it is posted after the product generation table is complete for the current volume.**

(*service_routine) ()

Address of a service routine. This a function of the following type:
void defined_function (int); This function contains logic to be executed when the registered event is posted.

queued_parameter

A parameter passed to the event handler service routine. This parameter is useful if several events are registered all using the same event handler service routine. In this case a different parameter value is used for each registered event.

Notes / Rules for use

1. The only internal event that is generally useful is **EVT_ANY_INPUT_AVAILABLE**. With one or more registered data inputs, the event is posted when any input is available. The input must be WSR-88D data that is not **RADIAL_DATA**.

2. External events are events that are global to the ORPG; that is any ORPG task can respond to the event. An event designating the beginning of a volume is the most useful event for meteorological algorithms. Algorithm tasks should use **ORPGEVT_START_OF_VOLUME_DATA** because it is the safest beginning volume event to use. There is no corresponding elevation data event. The event **ORPGEVT_BEGIN_ELEV** could be used with caution because it is posted before the product generation table is complete. If used, the algorithm should sleep 5 seconds before beginning processing.
3. With more than one registered event, and if there is a need to respond differently to each event; either provide a unique event handler for each event, or use the **queued_parameter** to indicate which event is being responded to.
4. With more that one registered event in the queue, the event handlers are executed sequentially.

User Notification of an updated data store

A user of a data store (linear buffer) can be notified of an update to that store. The algorithm logic can respond when a particular buffer data store is updated.

Documentation of This Function is Not Complete

```
int RPGC_UN_register( int data_id, LB_id_t msg_id,
                      void (*service_routine)() )
```

PARAMETER:	**data_id**	The linear buffer ID
	msg_id	The LB message ID
	(*service_routine) ()	Address of a function **service_routine** which contains the logic to be executed when the event is posted.
RETURN VALUE:		Returns 0 on success. Negative return value indicates error.

Registers for the User Notification event posted when the specified data store is updated.

EXAMPLE: cpc006/tsk001/rpg_status_prod.c

Parameter Descriptions

data_id

The identity of the linear buffer. Non-product buffer ID's are defined in **orpgdat.h** / **orpgdat.inc**. If the header file has not been modified to include this new data ID, the **<Data_Buffer_Number>** must be entered rather than the **<Data_Buffer_Name>**. Whichever is used (name or number), it must be the same as the **data_id** attribute for the data store definition in the **data_attr_table** configuration file.

msg_id

The message ID of the message being written. The value **LB_ANY** can be used to react to any update to the buffer.

(*service_routine) ()

Address of a service routine. This a function of the following type:

void defined_function (int); This function contains logic to be executed when the registered event is posted.

Notes / Rules for use

1. TBD

Part E. Task Timing Initialization

The registering of task timing affects the behavior of the algorithm control loop. **RPGC_task_init** determines when a task will begin processing once the routine begins or after aborting an output product.

```
int RPGC_task_init( int what_based, int argc, char *argv[] )
```

PARAMETER: **what_based** The timing base for the task.

argc From task's main function, used by API infrastructure.

***argv[]** From task's main function, used by API infrastructure.

RETURN VALUE: Not Used

Registers the timing base for the task. In the case of aborted product output (that is one of the **RPGC_abort** functions called), this determines the earliest time the algorithm control loop will release. If **VOLUME_BASED**, the algorithm control resume time is "new-volume", otherwise the resume time is "new-elevation".

EXAMPLE: **RPGC_task_init(ELEVATION_BASED, argc, argv);**
where **argc** and **argv** are the standard parameters from the **main()** function.

Parameter Descriptions

what_based

Timing base for the task. The allowed values for this parameter (defined in **rpg_port.h**) are: **ELEVATION_BASED, VOLUME_BASED, RADIAL_BASED, EVENT_BASED**. The new form is: **TASK_ELEVATION_BASED, TASK_VOLUME_BASED, TASK_RADIAL_BASED, TASK_EVENT_BASED**. **Note that TIME_BASED / TASK_TIME_BASED are not supported**.

Notes / Rules for use

1. Must be called after all other registrations.
2. Event driven tasks (those that use **RPGC_wait_for_event** used for algorithm loop control) will have a timing input of **EVENT_BASED** with no registered data inputs. On the other hand, event driven algorithms with registered product data inputs should have a task timing basis compatible with the input data (typically **VOLUME_BASED**).
3. Data driven tasks typically have a task timing compatible with the output data (that is a task timing of **ELEVATION_BASED** for **ELEVATION_DATA** output and **VOLUME_BASED** for **VOLUME_DATA** output). **Normally the task timing is either the same as or greater than the largest input timing and the same as the output timing.** There are several algorithms where this is not the case.
    a. The **superob_vel** task and the **mdaprod** task have a task timing of **ELEVATION_BASED** and the output product is **VOLUME_DATA**. In other words, the volume product is produced even if there are missing elevations. These are not typical algorithms.
    b. The several tasks has a task timing of **VOLUME_BASED** and the output is **ELEVATION_DATA**. Most of these are of the Kinematic algorithm type. This configuration is used if the output products for later elevations in the volume should not be produced if there is any

problem with the lower elevations of input data. In other words the downstream tasks cannot handle a missing elevation.

4. (Expanded discussion of exception b above). Though not required by the infrastructure, if a task is producing an intermediate product, the task timing should be set to the greatest output timing of downstream tasks. Typically this means that if any downstream task produces **`VOLUME_DATA`** products, it is usually a good idea to register upstream tasks as **`VOLUME_BASED`** even if only producing **`ELEVATION_DATA`**. See the topic *Read All Input Data (If not Aborting)* in Part D. of Volume 3, Document 3, Section I.

The following function is not documented because it precludes use of some of the other registration functions. It is not recommended for general use.

`int RPGC_reg_and_init( )`

Vol 3. Document 2 -
The WSR-88D Algorithm API Reference

Section II Control - Input/Output - Abort Services

NOTE: Preliminary support for Dual Pol data was provided in Build 11 in order to support implementation. Actual Dual Polarization implementation is provided in Build 12.

NOTE

Not all of the API functions have been documented. The functions not documented are not critical to algorithm development and many of the recently added functions were created to support the porting of legacy FORTRAN algorithms to ANSI-C.

- Some of the new functions are redundant with existing functions.
- Some are not generally useful for algorithm development.
- Some of the new functions have not yet been used in algorithms.

NOTE: Some previously deprecated functions are being used to port FORTRAN algorithms.

BUILD 12 CHANGES:

- Modified the discussion of reading base data and determining radial size (number of bins) for clarity and safety.
- Emphasized that algorithms producing a product must either successfully produce that product or call the appropriate abort service.
- Added specific examples for when to use the **`PGM_PROD_NOT_GENERATED`** abort reason code (including the failure to read a non-product data store).
- Corrected an error in the documentation of **`RPGC_rel_outbuf`**. When using the optional output it must be OR the disposition with **`RPGC_EXTEND_ARGS`**, not **`EXTENDED_ARGS_MASK`**.
- Added that **`PGM_DISABLED_MOMENT`** is also used when a required advanced data field (Dual Pol data) is not in the radial message.
- Modified **`RPGC_get_customizing_data`** to provide the capability to set a specific limit to the number of requests for a product.

BUILD 11 CHANGES:

- Documented the differences in the product request lists obtained by replay instances of tasks.
- Behavior of replay tasks was described for both data driven and event driven algorithms.

- **RPGC_is_buffer_from_last_elev** was modified to also return the value of the last elevation index in addition to the whether the current index is the last.
- **RPGCS_get_last_elev_index** was deprecated.
- Bug fix documented for **RPGC_cleanup_and_abort.**
- Bug fix documented for **RPGC_release_pdb_product.**
- Noted that **RPGC_get_request_by_name** can also be used (instead of **RPGC_check_data_by_name**) to determine if a particular product has been requested.
- Documented the new opstat / opstatus values returned by the get_inbuf and get_outbuf functions. **RPGC_NORMAL** is preferred but **NORMAL** can still be used for the time being.
- Made it clear that Build 11 support for Dual Pol data is preliminary and only for development purposes.

After registrations and initialization, the algorithm enters into the processing phase. The algorithm remains in this phase until the ORPG is shut down or the task fails. The *algorithm control loop* service determines when the algorithm begins processing data. Other services described in this section involve data input / output and aborting product construction. Functions aiding in product construction are covered in Section III of this document. See Section IV - *API Convenience Functions*) for various data conversion / transformation functions.

NOTE: This document makes multiple references to two forms of a data driven algorithm control loop. This is the continuous main loop of an algorithm that is entered after all registrations and initializations are complete.

- The WAIT_ALL form of the control loop is the most common and is used in algorithms that have only one data input or need more than one data input to create a product. The loop control function used is **RPGC_wait_act(WAIT_DRIVING_INPUT)**
- The WAIT_ANY form of the control loop is used in algorithms having multiple registered data inputs but use only one input to produce a product. There are two functions that can be used: **RPGC_wait_act(WAIT_ANY_INPUT)** and **RPGC_wait_for_any_data(WAIT_ANY_INPUT).**

The contents of this section are organized as follows:

- Part A. Algorithm Control Loop
- Part B. Reading Input Data
- Part C. Writing Output Data
- Part D. Getting Request Information
- Part E. Getting Information About the Current Volume / Elevation
- Part F. Reading Adaptation Data
- Part G. Reading Base Data Messages
- Part H. Aborting Product Construction
- Part I. Non-Product Data Access
- Part J. Miscellaneous Functions

Part A. Algorithm Control Loop

The algorithm control loop is a continuous loop that holds processing until certain conditions are met. Most of the existing algorithms are data driven. Data driven algorithms begin processing only when required input is available. Algorithms can be event driven where processing is blocked until a particular ORPG event is posted.

Data Driven Algorithms

There are two types of data driven algorithms.

1. The most common type of data driven algorithm uses the WAIT_ALL form of the control loop with one or more registered data inputs. With multiple inputs registered, generally all are necessary though some can be designated as optional. The first registered input (called the driving input) cannot be optional. Algorithms using this form of the control loop read input(s) with individual calls to **`RPGC_get_inbuf_by_name`**.
2. The other type of data driven algorithm uses the WAIT_ANY form of loop control with more than one registered input. This type of algorithm produces a product from any one of several inputs. The available input is read with a call to **`RPGC_get_inbuf_any`**.

The release conditions of these two differ significantly.

1. The WAIT_ALL form of control loop releases when the registered driving input is available and there is a request for at least one of the registered output products.
2. The WAIT_ANY form of control loop releases when one of the registered inputs is available. It is recommended that an algorithm use **`RPGC_check_data_by_name`** to check that the output product is requested. Document 3 - *WSR-88D Algorithm Structure & Sample Algorithms* includes additional information.

The behavior of a replay task (a special second instance of a task) is covered in Document 3, Section I, Part C - *Algorithm Initialization and Control Loop*. A replay task can only be used with the WAIT_ALL form of control loop.

WARNING: Attempting to use the WAIT_ANY form of control loop in an algorithm requiring more than one input to produce an individual product can provide unexpected results. There is no synchronization of one input with the another. What is guaranteed is that a data message read will be from a volume / elevation subsequent to the last message (of that input type) read.

```
int RPGC_wait_act( int wait_for )
```

PARAMETER: **`wait_for`** Either **`WAIT_DRIVING_INPUT`** or **`WAIT_ANY_INPUT`**

RETURN VALUE: Not Used

Algorithm processing is blocked as follows. If the input parameter is **WAIT_DRIVING_INPUT** three conditions must be met. First, if the algorithm has just been launched or has just aborted processing, it blocks until the 'resume time' (either new-volume or new-elevation) determined by the **RPGC_task_init** parameter. Second, at least one of the registered output products must be requested. And third, the driving input (the first registered data input) must be available. If the parameter is **WAIT_ANY_INPUT** the blocking and limitations are identical to **RPGC_wait_for_any_data**.

EXAMPLE:

Parameter Descriptions

wait_for

The value **WAIT_DRIVING_INPUT** provides blocking for a driving input and the synchronization of multiple inputs. The value **WAIT_ANY_INPUT** provides for blocking for one of several inputs.

```
int RPGC_wait_for_any_data( int wait_for )
```

PARAMETER: **wait_for** Parameter must be set to **WAIT_ANY_INPUT** for normal functioning.

RETURN VALUE: Not Used

Continually checks the availability of all registered inputs. If any input has been updated, returns and allows processing to proceed. This loop control function is only valid for inputs other than **RADIAL_DATA** and supports **VOLUME_BASED** tasks and **ELEVATION_BASED** tasks.

EXAMPLE: cpc004/tsk007/recclprods_main.c
cpc018/tsk003/tdaru.c

Parameter Descriptions

wait_for

RPGC_wait_for_any_data can only be called with the value **WAIT_ANY_INPUT**. The value **WAIT_ANY_INPUT** provides for blocking for one of several inputs.

Notes / Rules for use

1. Almost all data-driven algorithms use the WAIT_ALL form of algorithm loop control.
2. For the WAIT_ANY form of algorithm loop control, input(s) must not be **RADIAL_DATA**.
3. Currently, only **VOLUME_BASED** tasks and **ELEVATION_BASED** tasks are supported with the WAIT_ANY form of loop control.
4. Care must be used with the WAIT_ANY form. While the algorithm may produce more than one output product, any product produced can only require one of the registered inputs. Attempting to synchronize multiple inputs will not work.

Event Driven Algorithms

Event driven algorithms use **RPGC_wait_for_event** version of the algorithm control loop. There are two types of event driven algorithms.

1. Event driven algorithms that are also registered for product input data use a task timing appropriate for the registered input, for example **RPGC_task_init(VOLUME_BASED, argc, argv)**. The available input data is read with **RPGC_get_inbuf_any**.
2. Event driven algorithms that are not registered for product input data use a task timing of **RPGC_task_init(EVENT_BASED, argc, argv)**

The event driven algorithm can be registered for more than one event. The release condition for an event driven algorithm is the handling of any of the registered events.

Regardless of where the algorithm is, when a registered event is posted, the logic in the callback function registered for that event is processed immediately. If there is more than one registered event posted to the event queue, the callbacks are executed sequentially

If the algorithm is holding in the loop control function (**RPGC_wait_for_event**), the call back is executed and the control function releases for the algorithm to begin processing the logic in the main loop. If the algorithm is already processing the main loop, when a registered event is posted, the loop is temporarily interrupted, the callback function is executed, and the main loop continues when the callback returns.

The behavior of this function is a little different if the task is configured as a replay task. The use of a replay task (a special second instance of a task) with an event driven algorithm is covered in Document 3, Section I, Part C - *Algorithm Initialization and Control Loop*.

int RPGC_wait_for_event( )

PARAMETER: None

RETURN VALUE: Not Used

Continually checks to see if any of the registered events are posted. When an event is posted, the logic in the event handler function is executed. If more than one event is in the queue, they are executed sequentially. After all events in the queue are handled, **RPGC_wait_for_event** returns.
NOTE: Even if no registered events are posted, **RPGC_wait_for_event** will automatically return at the end of volume.

EXAMPLE: cpc005/tsk002/pcipdalg.c
cpc008/tsk001/alerting.c

Notes / Rules for use

1. Very few algorithms are event driven. Normal algorithm tasks should be data driven.
2. One use for an event driven algorithm would be with a data stream not synchronized with the WSR-88D volume scans. Another is an algorithm that has no product data inputs.
3. With an event driven algorithm, normally all of the algorithm logic is in the event handler itself, rather than the body of the while loop containing **RPGC_wait_for_event**.

4. Input data for event driven algorithms that is not derived from WSR-88D base data should be configured with a **DEMAND_DATA** timing.
5. Output data for event driven algorithms having no product inputs should be **DEMAND_DATA.**
6. See Part D of this document for the types of events that are useful for algorithms.

Part B. Reading Input Data

Reading Product Data Linear Buffers

The API provides a high-level access for reading the input product linear buffer. When used correctly, all sequencing checks and (with multiple inputs) data synchronization checks are accomplished with these services.

Requirement to read all radials / elevations

- An algorithm reading base data (or any **RADIAL_DATA** input) and producing **ELEVATION_DATA** output must read until the *actual* end-of-elevation even if data only up to *pseudo* end-of-elevation are needed. **Do not read beyond the *actual* end-of-elevation.**
- An algorithm reading base data (or any **RADIAL_DATA** input) and producing **VOLUME_DATA** output must read until the *actual* end-of-volume even if data only up to *pseudo* end-of-volume are needed. **Do not read beyond the *actual* end-of-volume.**
- An algorithm reading elevation based input and producing a volume based output must read all elevations produced for that volume by the upstream task, even if fewer elevations are actually used.

The need to read all elevations / radials produced can be avoided only in algorithms that produce **VOLUME_DATA** output (and whose task timing is **VOLUME_BASED**). The function **RPGC_abort_remaining_volscan** can be used to accomplish this. LIMITATION: Currently this function does NOT work with the WAIT_ANY form of the control loop.

Algorithms blocking until a driving input is available

For algorithms that block until a particular input is available (the WAIT_ALL form of the algorithm control loop), the data are actually read (and synchronized) with **RPGC_get_inbuf_by_name** for each individual input.

The driving input must be read first. It is recommended that the optional inputs (if any) be read last.

Algorithms needing only one of several inputs

For algorithms that block until any one of several inputs is available (the WAIT_ANY form of the algorithm control loop), the data must be read with **RPGC_get_inbuf_any**.

The WAIT_ANY form of the control loop, using either **RPGC_wait_act(WAIT_ANY_INPUT)** or **RPGC_wait_for_any_data(WAIT_ANY_INPUT)**, releases when one of the registered inputs is available regardless whether the output product has been requested. Therefore, when using the WAIT_ANY control loop and the function **RPGC_get_inbuf_any**, it is recommended that **RPGC_check_data_by_name** be used to check to see if the registered output is requested.

Functions used with both forms of the control loop

- **RPGC_get_inbuf_len** is used to get the size of the input buffer message.
- **RPGC_rel_inbuf** is used to deallocate the memory that was obtained when the buffer was read.
- **RPGC_rel_all_inbufs** can used to release all acquired input buffers.

Note: The following function has been deprecated and should not be used.

- **RPGC_get_inbuf**

This will be eliminated in a future build. If any development software uses this function it should be replaced with **RPGC_get_inbuf_by_name**.

```
void* RPGC_get_inbuf_by_name( char *reqname, int *opstat )
```

PARAMETER:	**reqname**	The input product registration name.
	opstat	Address of the status of the operation. (output)
RETURN VALUE:		Pointer to the buffer containing input data or NULL if an error occurred. The returned pointer must be cast to the appropriate type for the product message being input.

Returns a pointer to a block of memory containing the input data message. The buffer is always allocated and must be cast to the proper type of an awaiting pointer. See the description of the parameter **opstat** for additional guidance on using this API function. **NOTE**: There have been situations where the returned pointer has been NULL even though the **opstat** was **RPGC_NORMAL**.

EXAMPLE: (Using Base Data)

```
basedataPtr=(Base_data_radial*)RPGC_get_inbuf_by_name("REFLDATA",
                                                      &opstatus);
```

where **struct Base_data_radial** is defined in **basedata.h**. The header information for the base data radial is **struct Base_data_header** also defined in **basedata.h**. See CODE Guide Volume 2 - *The ORPG Application Development Guide* for an introduction to the structure of the ORPG base data radial message.

Parameter Descriptions

reqname

The name of the product data to be read. **The value of this parameter is the same as the input product registered.** This name must be in the list contained in the **input_data** attribute of the **task_attr_table** entry for the algorithm task (the **input_data** entry determines the number or ID of the linear buffer that is the source of the input data). See CODE Guide Volume 2, Document 2, Section III for a complete explanation of product and task configuration via the **product_attr_table** and **task_attr_table** configuration files.

NOTE: Using base data (and raw data) as an input is a special case. **REFLDATA(79)** is used when only reflectivity data are needed and **COMBBASE(96)** is used when interested in Doppler data (velocity and spectrum width). Reflectivity data is also obtained when registered for

COMBBASE(96). Basedata is also available via complete elevations. For complete elevations of basedata either **REFLDATA_ELEV(302)** or **COMBBASE_ELEV(303)** must be registered See CODE Guide Volume 2, Document 4, Section I - *Base Data Format* for a description of both radial and elevation basedata messages.
It is also possible to register for **BASEDATA(55) BASEDATA_ELEV(301)**. See CODE Guide Volume 2, Document 4, Section I - *Base Data Format* for a complete explanation of all base data types.

opstat (output)
Address of the status of the operation. This is the status returned for the read operation. Return values are defined in **a309.h**. **NOTE**: There have been situations where the returned pointer has been NULL even though the **opstat** was **RPGC_NORMAL**. There are two cases for correctly handling the returned status. It is the programmers responsibility to ensure that the algorithm handles this correctly. CASE 1: For required inputs, with any value other than **RPGC_NORMAL**, the algorithm must be aborted. CASE 2: If the registered input has been configured as optional, the value **NO_DATA** indicates the read attempt has timed out. The algorithm must account for the missing data in this event.

Notes / Rules for use

1. For data driven algorithms: **RPGC_get_inbuf_by_name** cannot be used with the WAIT_ANY form of control loop, it must be used with the WAIT_ALL form.
2. **RPGC_get_inbuf_by_name** is not compatible with event driven algorithms. Use only with data driven algorithms.
3. For a data driven algorithm with a driving input, the first input product read must correspond to the first registered product (the first entry in the **input_data** attribute in the **task_attr_table** entry).
4. Optional inputs (if any) should be read last.
5. The first **RPGC_get_inbuf_by_name** call reads the driving input and accomplishes data sequencing checks. Additional **RPGC_get_inbuf_by_name** calls (if any) check to see if the next input data is synchronized with the driving input. The **opstat** must be checked after each call, and if any value other than **RPGC_NORMAL**, the algorithm must be aborted properly (for required inputs).
6. Normally, only one input buffer of any product type should be open (obtained) at any time. Actually, 2 buffers of a particular product type may be open (perhaps to facilitate comparison).
7. If **RPGC_get_inbuf_by_name** is successful, then **RPGC_rel_inbuf** must be called before returning to the beginning of the algorithm control loop (both for successful product generation and when aborting the product). Note that in the case of aborting product generation, calling **RPGC_cleanup_and_abort** also releases the input buffer.

```
void* RPGC_get_inbuf_any( int *datatype, int *opstat )
```

PARAMETER: **datatype** A pointer to the linear buffer id of the data read.

opstat Address of the status of the operation. (output)

RETURN VALUE: Pointer to the buffer containing input data or NULL if an error occurred. The returned pointer must be cast to the appropriate type for

the product message being input.

Returns a pointer to a block of memory containing the input data message. The buffer is always allocated and returned pointer must be cast to the proper type.

EXAMPLE: cpc004/tsk007/recclprods_main.c
cpc018/tsk003/tdaru.c

Parameter Descriptions

`datatype`

A pointer to the linear buffer id of the data actually read. This will be one of the types of input data registered (the **`input_data`** attribute in the **`task_attr_table`**). This value is used to determine the appropriate typecast for the return value.

`opstat` (output)

Address of the status of the operation. This is the status returned for the read operation. Return values are defined in **`a309.h`**. With any value other than **`RPGC_NORMAL`**, the algorithm must be aborted. It is the programmers responsibility to ensure that the algorithm handles this correctly.

Notes / Rules for use

1. For data driven algorithms: **`RPGC_get_inbuf_any`** must be used with the WAIT_ANY form of the algorithm control loop, it cannot be used with the WAIT_ALL form.
2. **`RPGC_get_inbuf_any`** is compatible with event driven algorithms.
3. If **`RPGC_get_inbuf_any`** is successful, then **`RPGC_rel_inbuf`** must be called before returning to the beginning of the algorithm control loop (both for successful product generation and when aborting the product). Note that in the case of aborting product generation, calling **`RPGC_cleanup_and_abort`** also releases the input buffer.

```
int RPGC_get_inbuf_len( void *bufptr )
```

PARAMETER: **`bufptr`** Pointer to the input buffer to be sized.

RETURN VALUE: -1 (on error) or the size of the input buffer in bytes (on success).

Returns the size of the input product in bytes. This is the size of the input product not the size of the buffer message which included the 96 byte internal header.

EXAMPLE: cpc014/tsk013/saaprods_main.c

Parameter Descriptions

`bufptr`

Pointer to the input buffer. This is the pointer returned by the corresponding **`RPGC_get_inbuf_by_name`** call cast to type **`void *`**.

Notes / Rules for use

1. WARNING: This function cannot be used after releasing the input buffer **`bufptr`** (returns -1).

`int RPGC_rel_inbuf( void *bufptr )`

PARAMETER: **`bufptr`** None

RETURN VALUE: Not Used

Deallocates buffer created by **`RPGC_get_inbuf_by_name`**.

EXAMPLE: **`RPGC_rel_inbuf((void*)basedataPtr);`**
where **`basedataPtr`** is the value returned by **`RPGC_get_inbuf_by_name`**.

`int RPGC_rel_all_inbufs( )`

PARAMETER: None

RETURN VALUE: Not Used

This function releases all input buffers that are currently open.

EXAMPLE: *This function is not yet used in any algorithm*

Parameter Descriptions

`bufptr`

Pointer to the input buffer. This is the pointer returned by the corresponding **`RPGC_get_inbuf_by_name`** call cast to type **`void *`**.

Notes / Rules for use

1. If an input buffer is acquired successfully (**`RPGC_get_inbuf_by_name`** or **`RPGC_get_inbuf_any`**), then the buffer must be released (**`RPGC_rel_inbuf`** or **`RPGC_rel_all_inbufs`**) before returning to the beginning of the algorithm control loop (both for successful product generation and when aborting the product). Note that in the case of aborting product generation, calling **`RPGC_cleanup_and_abort`** also releases the input buffer.
2. Normally, only one input buffer of any product type should be open (obtained) at any time. The absolute limit is 2 buffers of a particular product type may be open (perhaps to facilitate comparison).

Reading Data from Product Database

The following functions provide a method for reading product data directly from the product database. This includes warehoused intermediate products. The product must be a registered input.

`RPGC_find_pdb_product` is used to get a listing of products in the database for a specified registration name.

After reviewing the list, **`RPGC_get_pdb_product`** is used to get the product message from the database. Memory allocated for the product message must be freed using **`RPGC_release_pdb_product`**.

```
int RPGC_find_pdb_product( char *name, RPGC_prod_rec_t *results,
                           int max_results, ... )
```

PARAMETER:	**name**	registration name
	results	an array containing the results of the search (**OUTPUT**)
	max_results	the size of the input array **results**
	...	additional query fields
RETURN VALUE:		The number of product found in the query.

Searches the product database for a product code associated with the registration name provided in the first parameter **name**. The parameter **max_results** limits the number of results listed in the array represented by the second parameter **results**.

EXAMPLE: **clutprod_handle_volstart.c** in cpc004/tsk004

Parameter Descriptions

name

The registration name of the product to search for.

results (output)

A pointer to an array containing the results of the search. This is an array of **max_results** number of structures of type **RPGC_prod_rec_t** defined in **rpgc.h.** The structure contains the following fields:

int msg_id;	which is the internal message id
int vol_time;	which is the volume time in Julian seconds UTC
int elev;	which is the elevation angle in deg*10
int elev_ind;	which is the elevation index
short params[6];	which are the product request parameters.

max_results

The number of structures in the **results** array.

...

Variable arguments containing additional (optional) query fields. This list must always be terminated by **QUERY_END_LIST** even if no additional fields are specified. Each additional query field is either a 2-tuple or 3-tuple defined as follows:

- **QUERY_VOL_TIME** is followed by the UTC volume time in Julian seconds
- **QUERY_ELEV** is followed by elevation angle in deg*10
- **QUERY_VOL_TIME_RANGE** is followed by a beginning and end volume time
- **QUERY_ELEV_RANGE** is followed by a beginning and end elevation angle

Notes / Rules for use

1. The optional query fields have not been tested.

```
void* RPGC_get_pdb_product( char *name, LB_id_t msg_id )
```

PARAMETER: **name** product registration name

msg_id database message id

RETURN VALUE: Returns a pointer to the product message on success. Returns **NULL** on failure

Reads the product message with the internal message ID, **msg_id**, from the product database. The product is decompressed if required. If this product message has a product id or product code consistent with the registration name provided in the first parameter, **name**, a pointer to the product message is returned.

EXAMPLE:

Parameter Descriptions

name

The product registration name. This is used internally as a check.

msg_id

The database internal message id of the product message to read.. This is returned in the **results** parameter in **RPGC_find_pdb_product**.

Notes / Rules for use (for a data driven task):

1. *This function is not yet used in any algorithm.*

```
void RPGC_release_pdb_product( void *buf )
```

PARAMETER: **buf** product buffer to be freed

RETURN VALUE: NONE

This function frees the product buffer obtained with RPGC_get_pdb_product.

EXAMPLE:

Parameter Descriptions

buf

Pointer to the product buffer to be freed. This is the return value of **RPGC_get_pdb_product**.

.

Notes / Rules for use (for a data driven task):

1. *This function is not yet used in any algorithm.*

The following data access functions, along with a registration function listed in Part A of Section I, support a more advanced method of accessing replay data. These functions were added in Build 9 and are currently not used in any algorithm. If this capability is determined to be generally useful the functions will be documented.

```
int RPGC_check_volume_radial( unsigned int vol_seq_num, int elev_num,
                              unsigned int type )

char* RPGC_read_volume_radial( unsigned int vol_seq_num, int elev_num,
                               unsigned int *type, int *size )

void RPGC_rel_volume_radial( char *bufptr )
```

Part C. Writing Output Data

A high-level service is provided for product output (writing to the appropriate ORPG product data store). This output service also accomplishes necessary functions to announce product output to the ORPG infrastructure that handles product distribution.

With multiple product outputs, before acquiring the output buffer, it should be determined whether a request for that product exists using **RPGC_check_data_by_name** or **RPGC_get_request_by_name**. Though recommended, this is not mandatory since the function that obtains the output buffer will return an 'opstatus' other than RPGC_NORMAL if there is no request for the product.

RPGC_get_outbuf_by_name / **RPGC_get_outbuf_by_name_for_req** allocate a working space in which the product is constructed. The working space must accept the largest possible product.

RPGC_get_outbuf_by_name is the normal form of this function. **RPGC_get_outbuf_by_name_for_req** is used only for those products which can be modified via request message parameters (product dependent parameters). These request parameters are obtained with **RPGC_get_customizing_data**

RPGC_realloc_outbuf is called to increase the buffer size if needed.

RPGC_rel_outbuf is used to output the product and release memory obtained with the get_outbuf_by_name function. **RPGC_rel_outbuf** accepts an optional third parameter which is the actual size of the product being output. This significantly improves efficiency in some cases including product compression.

RPGC_abort_remaining_volscan does not abort product construction but is used to avoid having to read all input data in certain situations. This function is used in tasks producing a volume based output that read elevation based data and / or radial based data. The algorithm must use the "WAIT_ALL" form of the control loop.

> **Note**: The following functions have been deprecated and should not be used.
>
> - **RPGC_get_outbuf**
> - **RPGC_get_outbuf_for_req**
>
> They will be eliminated in a future build. If any development software uses these functions they should be replaced with **RPGC_get_outbuf_by_name** and **RPGC_get_outbuf_by_name_for_req**.

```
void* RPGC_get_outbuf_by_name( char *reqname, int bufsiz, int *opstat )
```

PARAMETER: **reqname** The output product registration name.

bufsiz The requested size of the output buffer (in bytes).

opstat Address of the status of the operation. (output)

RETURN VALUE: Pointer to buffer for the output product. This should be cast to type **short *** in order to use some of the common services related to product construction.

Returns a pointer to a block of memory in which the output product will be placed.

EXAMPLE:
```
buffer=(short *)RPGC_get_outbuf_by_name("DIGREFL", BUFSIZE,
&opstatus);
```

```
void* RPGC_get_outbuf_by_name_for_req( char *dataname, int bufsiz,
                                       User_array_t *user_array, int *opstat )
```

PARAMETER: **dataname** The output product registration name.

bufsiz The requested size of the output buffer (in bytes).

user_array Address of the user array entry containing the product request parameters for a specific product request

opstat Address of the status of the operation. (output)

RETURN VALUE: Pointer to buffer for the output product. This should be cast to type **short *** in order to use some of the common services related to product construction.

Returns a pointer to a block of memory in which the output product will be placed.

EXAMPLE:
```
int  opstatus; User_array_t *my_requests;
buffer=(short *)RPGC_get_outbuf_by_name_for_req("DIGREFL", BUFSIZE,
&my_requests[n-1], &opstatus);
```
Where **n** is request number.

Parameter Descriptions

reqname and **dataname**

The name of the product data to be output. **The value of this parameter is the same as the output product registered.** This name must be in the list contained in the **output_data** attribute of the **task_attr_table** entry for the algorithm task (the **output_data** entry determines the number or ID of the linear buffer containing the output data). See CODE Guide Volume 2, Document 2, Section III for a complete explanation of product and task configuration via the **product_attr_table** and **task_attr_table** configuration files.

bufsiz

Size of product in bytes (does NOT include the 96-byte internal header). This must be sufficient to accommodate the largest possible product. The programmer is responsible for allocating sufficient space in which to construct the product. The maximum buffer size that can be allocated is 8,000,000 bytes (increased from 2,000,000 bytes in Build 8).

user_array

Address of the user array entry that contains the request parameters (that is the customizing data) for a specific product request. The user array containing up to 10 product requests is obtained via

the **RPGC_get_customizing_data** call. The first user array entry is at **&user_array**, the second request (if there is one) is at **&user_array[1]**, and so forth.

opstat (output)
Address of the status of the operation. This is the status returned by the operation of allocating buffer space.

Notes / Rules for use

1. Only one output buffer of a specific product type can be open (obtained) at any time. However, if an algorithm task produces more than one product type (having different product id's), one buffer of each type may be opened an any given time.

```
void* RPGC_realloc_outbuf( void *bufptr, int bufsiz, int *opstat )
```

PARAMETER:	**bufptr**	Pointer to the output buffer to be resized.
	bufsiz	The requested size of the output buffer (in bytes).
	opstat	Address of the status of the operation. (output)
RETURN VALUE:		The pointer to the reallocated buffer on success, NULL on failure.

Reallocates the block of memory at the address returned by the associated RPGC_get_outbuf_by_name call. If opstat is RPGC_NORMAL the operation was successful. The contents of the original buffer are copied to the new location before the function returns. If the function fails, the original buffer pointer is valid.

EXAMPLE: cpc018/tsk001/buildDMD_PSB.c

Parameter Descriptions

bufptr
Pointer to the output buffer to be resized. This is the pointer returned by the corresponding **RPGC_get_outbuf_by_name** call.

bufsiz
Size of product in bytes (does NOT include the 96-byte internal header). The programmer is responsible for allocating sufficient space in which to construct the product. The maximum buffer size that can be allocated is 8,000,000 bytes (increased from 2,000,000 bytes in Build 8).

opstat (output)
Address of the status of the operation. This is the status returned by the operation of allocating buffer space.

Notes / Rules for use

1. Only one output buffer of a specific product type can be open (obtained) at any time. However, if an algorithm task produces more than one product type (having different product id's), one buffer of each type may be opened an any given time.
2. With multiple product outputs, before acquiring the output buffer, it should be determined whether a request for that product exists using **RPGC_check_data_by_name** or **RPGC_get_request_by_name.** This is optional since the function that obtains the output buffer returns an 'opstatus' other than RPGC_NORMAL if there is no request for the product.

```
int RPGC_rel_outbuf( void *bufptr, int disposition, ... )
```

PARAMETER: **bufptr** Pointer to the output buffer to be released.

disposition Flag to determine whether product is distributed (successful construction).

[*optional*] The size of the product being output.

RETURN VALUE: Not Used

Deallocates buffer created by **RPGC_get_outbuf_by_name** and **RPGC_get_outbuf_by_name_for_req**. With **disposition = FORWARD**, the product is written to the appropriate linear buffer and a successful product generation message is generated. If **disposition = DESTROY**, an unsuccessful product generation message is generated.

EXAMPLE: **RPGC_rel_outbuf((void*)buffer, FORWARD);**
where **buffer** is the value returned by **RPGC_get_outbuf_by_name**.

```
int RPGC_rel_all_outbufs( int disposition )
```

PARAMETER: **disposition** Flag to determine whether product is distributed (successful construction).

RETURN VALUE: Not Used

This function releases all output buffers that are currently open with the indicated disposition.

EXAMPLE: *This function is not yet used in any algorithm*

Parameter Descriptions

bufptr

Pointer to the output buffer to be released. This is the pointer returned by the corresponding **RPGC_get_outbuf_by_name** call.

disposition

Flag to determine whether product is distributed (successful construction). The value **FORWARD** is used when product construction has been successful. The value **DESTROY** is used when product generation has not been successful.

[*optional*] (for **RPGC_rel_outbuf** only)

The size in bytes of the product being released. In order to use this parameter, the disposition must be OR'd with **RPGC_EXTEND_ARGS**, for example:
RPGC_rel_outbuf(buf_id, FORWARD | RPGC_EXTEND_ARGS, buf_size)

Notes / Rules for use

1. The first parameter passed to **RPGC_rel_outbuf** must be the same as the buffer pointer returned by the corresponding **RPGC_get_outbuf_by_name** / **RPGC_get_outbuf_by_name_for_req** call.

2. An output buffer must be successfully obtained (with **RPGC_get_outbuf_by_name** or **RPGC_get_outbuf_by_name_for_req**) before releasing with a disposition of FORWARD (**RPGC_rel_outbuf** or **RPGC_rel_all_outbufs**).
3. If an output buffer is successfully acquired (with **RPGC_get_outbuf_by_name** or **RPGC_get_outbuf_by_name_for_req**), then the buffer must be released (**RPGC_rel_outbuf** or **RPGC_rel_all_outbufs**) before returning to the beginning of the algorithm control loop (both for successful product generation and when aborting product). Note that in the case of aborting product generation, calling **RPGC_cleanup_and_abort** also releases the output buffer (and generates an unsuccessful product generation message).
4. Only one output buffer of a specific product type can be open (obtained) at any time. However, if an algorithm task produces more than one product type (having different product id's), one buffer of each type may be opened an any given time.

RPGC_abort_remaining_volscan serves a different purpose than the abort routines in Part H. The purpose of this function is to avoid having to read input data until the end of the volume scan if no more data are needed to complete product construction. LIMITATION: Currently this function does not work with the WAIT_ANY form of the control loop.

```
int RPGC_abort_remaining_volscan( )
```

PARAMETER: None

RETURN VALUE: Not Used

Accomplishes internal house cleaning before returning to the *algorithm control loop* after successful product generation. ONLY USED IN VERY SPECIFIC SITUATIONS. SEE NOTES.

EXAMPLE: cpc013/tsk003/epre_main.c

Notes / Rules for use

1. For tasks using elevation based and / or radial based input data to create volume based output data: **RPGC_abort_remaining_volscan** must be called after successful product output (**RPGC_rel_outbuf(<xxxxx>, FORWARD);**) when not reading all input data available in the volume (i.e. all elevations and / or all radials). This is demonstrated in the second task of sample algorithm 3.

Part D. Getting Request Information

This section includes three functions that are needed in algorithms that either create more than one product type or create customized products based upon request information.

RPGC_check_data_by_name can be used to determine if a particular registered output is requested. It is useful in two situations:

1. For algorithms producing more than one product type, **RPGC_check_data_by_name** is used to see which are requested.
2. Algorithms that proceed after reading one of several registered inputs (algorithms using the WAIT_ANY form of control loop) must always check for product requests even when there is only one output product registered.

If not used, **RPGC_get_outbuf** / **RPGC_get_outbuf_by_name** also checks for a request and will return an opstatus other than RPGC_NORMAL if there is no request for the product.

RPGC_get_customizing_data is used in algorithms that use parameters in the product request message to control the algorithm logic. This function can only be used in algorithms having one output product. This function was modified in Build 12 to also provide the ability to limit the number of requests.

RPGC_get_request_by_name (added in Build 10) is a more flexible method of obtaining request information and can be used in any algorithm. With multiple product outputs, this function can also be used (instead of **RPGC_check_data_by_name**) to determine if a particular product has been requested.

The contents of the request list is different for replay tasks (a special second instance of a task covered in Document 3, Section I, Part C - *Algorithm Initialization and Control Loop*).

- The product request list obtained by the replay instance of a task contains only the one-time requests for the product.
- The product request list obtained by the normal real-time task has the routine requests and any one-time requests not serviced by the replay instance.
- If no replay task is configured, the product request lists contains all requests.

Since there can be more than one request for a product in this case, an alternate form of the function that obtains the output buffer must be used: **RPGC_get_outbuf_by_name_for_req**.

Note: The following function has been deprecated and should not be used.

- **RPGC_check_data**

This will be eliminated in a future build. If any development software uses this function it should be replaced with **RPGC_check_data_by_name**.

Modified in Build 12

```
void *RPGC_get_customizing_data( int elev_index, int *n_requests )
```

PARAMETER: **elev_index** The elevation index (-1 for volume products)

n_requests Address of the number requests for this product (output)

RETURN VALUE: Returns a pointer to the product request data (an array containing structures of type **User_array_t** defined in **rpgc_globals.h**). Returns **NULL** if no product requests are found.

Obtains an array of structures each of which contain the request parameters for a specific product request. For products that can be customized via product request parameters, there can be up to 20 requests for each product type. The number of requests used in Legacy algorithms is 10 and can be specified by setting `*n_requests to zero or 10`).
NOTE: This function allocates memory which must be freed after use.
ASSUMPTION: This function is only used in algorithms having one output.

EXAMPLE:

```
int  num_requests = 6;
User_array_t *request_list;
request_list = (User_array_t *)RPGC_get_customizing_data(
              -1,&num_requests);
```

The structure type **User_array_t** is defined in **rpg_globals.h**

Parameter Descriptions

elev_index

For elevation outputs, the current elevation index of the data being processed. -1 is used for all volume based outputs.

n_requests (output)

The number of requests for this product (the number of entries in the user_array).

Notes / Rules for use

1. Must deallocate memory at the address returned prior to returning to the beginning of the control loop.
2. If algorithm produces more than one product using customizing parameters, the function **RPGC_get_request_by_name** should be used to obtain the request parameters.
3. Inherent limit of 20 requests (Build 12).
4. Must initialize the second parameter to a value (Build 12).

```
int RPGC_get_request_by_name( int elev_index, char *dataname,
                              User_array_t *uarray, int max_requests )
```

PARAMETER: **elev_index** The elevation index (-1 for volume products)

dataname registration name for the output

uarray an array to receive the requests (**OUTPUT**)

max_requests maximum number of requests to be returned

RETURN VALUE: The number of requests for this product (the number of entries in the **uarray** array.

Returns the product request parameters for the registered output **dataname** and the specified elevations **elev_index**. If a negative number is supplied for **elev_index** with an elevation based product, requests for all elevations are returned.

EXAMPLE: cpc007/tsk015/superes8bit.c

Parameter Descriptions

elev_index

For elevation outputs, the current elevation index of the data being processed. The value. -1 is used for all volume based outputs and can be used to return request information for all elevations with elevation based products.

dataname

The registration name of the output product.

uarray (output)

A pre-allocated array containing structures of type **User_array_t** (defined in **rpgc_globals.h**) to receive the product request information. Must be sized to handle **max_requests** elements.

max_requests

The maximum number of requests to be returned. This must not be larger than the size of the **uarray** supplied to receive the output.

Notes / Rules for use

1. The **uarray** supplied must be sufficiently sized to hold **max_requests** number of **User_array_t** structures.

The following functions, added in Build 10, are not required to implement an algorithm and are not documented.

```
int RPGC_get_customizing_info( int elev_index, User_array_t *user_array,
                               int *num_requests )
```

This function is redundant with **RPGC_get_customizing_data**, which is preferred.

```
int RPGC_get_request( int elev_index, int prod_id, User_array_t *uarray,
                      int max_requests )
```

This function should be an internal function and is not recommended for use.

```
int RPGC_check_data_by_name( char *data_name )
```

PARAMETER: **data_name** The product registration name.

RETURN VALUE: Returns **NORMAL** if the product is requested, otherwise returns **NOT_REQD**.

Used with algorithms producing more than one product type to determine which product types are requested. Must be called for every output product registered.

EXAMPLE: cpc014/tsk003/hybrprod.c

Parameter Descriptions

data_name

The name of the product data who's request status is being checked. **The value of this parameter is the same as the output product registered.** This name must be in the list contained in the **output_data** attribute of the **task_attr_table** entry for the algorithm task (the **output_data** entry determines the number or ID of the linear buffer containing the output data). See CODE Guide Volume 2, Document 2, Section III for a complete explanation of product and task configuration via the **product_attr_table** and **task_attr_table** configuration files.

Notes / Rules for use

1. An output of **DEMAND_DATA** type is unique in that a product can be output even if there is no request for it.

Part E. Getting Information About the Current Volume / Elevation

This section includes a few helper functions that obtain information about the radar data being currently ingested by the ORPG. Most of the information obtained by these functions is contained within the header for the base data message. Some of the information is also contained in an internal table called the Scan Summary table. Since some of this information is needed when constructing final products, using these functions reduce the amount of information that needs to be passed with intermediate products.

Three helper functions have been replaced.

Note: The following functions do not work in all situations and have been deprecated.

- **RPGC_get_current_vol_num**
- **RPGC_get_current_elev_index**
- **RPGCS_get_last_elev_index**

If any development software uses these functions they should be replaced immediately with **RPGC_get_buffer_vol_num**, **RPGC_get_buffer_elev_index** and **RPGC_is_buffer_from_last_elev** respectively.

RPGC_what_moments is used only by algorithm tasks reading base data messages (radial and elevation). It is used to test the first base data message of each elevation for elevation products and the first message of the volume for volume products.

The following helper functions are available.

NOTE For Algorithms Having no Product Inputs (Event Driven Algorithms)

Functions marked with an * can only be used in algorithms having at least one product data input. Event driven algorithms are one example of algorithms having no product inputs. If there are no product inputs, the data obtained by these * functions can be obtained by using **RPGC_reg_volume_status** and **RPGC_read_volume_status** to obtain the volume date, volume time, volume number and volume sequence number. The volume number can be used to with **RPGC_get_scan_summary** to obtain the scan summary table and with **RPGCS_get_vcp_data** to obtain the vcp information structure.

VCP

RPGC_get_buffer_vcp_num* obtains the type of VCP currently running.

Volume Number

RPGC_get_buffer_vol_num* obtains the volume number which is used is used as an input to **RPGC_prod_desc_block**, **RPGP_build_RPGP_product_t**, and **RPGC_get_scan_summary**.

Volume Sequence Number

RPGC_get_buffer_vol_seq_num* reads the volume sequence number.

Elevation Index

RPGC_get_buffer_elev_index* provides the current elevation index.

Target Elevation Angle

RPGCS_get_target_elev_ang returns the target elevation angle which is used in the final product as well as in any algorithm needing height information.

Determining the end of the Volume:

RPGC_is_buffer_from_last_elev* provides both the current elevation index and the highest elevation index in the current volume. This should be used in algorithms reading elevation data to determine when the final elevation of a volume has been read. In the future, the RDA at times may not accomplish all scans defined in the VCP. After this modification, algorithms producing volume data from elevation data must use this function.

Weather Mode

RPGCS_get_wxmode_for_vcp returns the weather mode for the specified VCP sequence number.

The number of bins to 70,000 feet:

RPGC_bins_to_ceiling returns the number of bins to 70,000 feet based upon the bin resolution and elevation angle.

RPGC_num_rad_bins is used for porting Legacy algorithms from Fortran to C, can only be used with Legacy / recombined data, and is **not recommended for new algorithms**.

There are three functions which obtains a structure containing multiple data fields.

The contents of the structures obtained by these functions are documented in Part D - Miscellaneous Configuration / Status Data in Volume 2, Document 4, Section III.

- The scan summary table (obtained by **RPGC_get_scan_summary**) makes additional information available to the algorithm. Most of this information is available via other means. Information includes: volume start date/time, wx mode, vcp number, number of rpg elevation cuts, number of rda elevation cuts, and the spot blanking bitmap.
- **RPGCS_get_vcp_data** returns the structure containing VCP information. This includes: vcp number, number of elevations (rda), clutter map number, pulse width (long/short), velocity resolution, pulse width, sample resolution, reflectivity range resolution, velocity & spectrum width range resolution, and radial angular interval.
- **RPGC_read_volume_status** updates the contents of the structure obtained by the corresponding registration function. This is the primary method of obtaining the volume number and the volume time if not registered for input product data. The volume number obtained can be used to obtain the scan summary table with **RPGC_get_scan_summary**. Most of the additional information is also available elsewhere. Information includes: volume number, volume sequence number, volume date-time, vcp number, weather mode, number of elevation cuts and each elevation angle.

Many of these functions have related inputs and outputs:

- The output of **RPGC_get_buffer_vol_num** is used as an input to **RPGC_get_scan_summary**.

- The output of **RPGC_get_buffer_elev_index** is used as an input to **RPGCS_get_target_elev_ang** and **RPGC_is_buffer_from_last_elev**.
- The output of **RPGC_get_buffer_vcp_num** is used as an input to **RPGCS_get_target_elev_ang**, **RPGCS_get_wxmode_for_vcp**, and **RPGCS_get_vcp_data**.

```
int RPGC_what_moments( Base_data_header *hdr, int *ref_flag,
                       int *vel_flag, int *wid_flag )
```

PARAMETER: **hdr** Pointer to the Base Data Header portion of the radial message.

ref_flag Address of a flag indicating whether the reflectivity moment is enabled. (output)

vel_flag Address of a flag indicating whether the velocity moment is enabled. (output)

wid_flag Address of a flag indicating whether the spectrum width moment is enabled. (output)

RETURN VALUE: Not Used

Returns Boolean flags indicating which of the moments (reflectivity, velocity, and spectrum width) are currently enabled in the RDA.

EXAMPLE: cpc007/tsk014/bvel8bit.c
cpc007/tsk015/superes8bit.c

Parameter Descriptions

hdr

Pointer to the base data header portion of the radial message. The header is the first part of the base data radial message. **struct Base_data_header** is defined in **basedata.h**.

ref_flag (output)

Address of the reflectivity enabled flag. Flag set to **TRUE** if enabled.

vel_flag (output)

Address of the velocity enabled flag. Flag set to **TRUE** if enabled.

wid_flag (output)

Address of the spectrum width enabled flag. Flag set to **TRUE** if enabled.

Notes / Rules for use

1. **RPGC_what_moments** is used when base data is the registered input (see CODE Guide Volume 2, Document 4, Section I, *Base Data Format* for a description of all base data types). The flag value of the applicable moment(s) should be checked immediately after the first time a base data message is read (**RPGC_get_inbuf_by_name**) during the construction of a product. If a needed moment is disabled (flag not TRUE), the construction and output of the product must be aborted.
2. The use of this function is not required if the needed moments are registered with **RPGC_reg_moments**.

```
int RPGC_get_buffer_vol_num( void *bufptr )
```

PARAMETER: **bufptr** Pointer to the input buffer

RETURN VALUE: Volume scan sequence number.

Returns the sequence number of the current volume scan (the volume number of the product contained in the input buffer). This number cycles 1 through 80. This information is an input to the function **RPGC_prod_desc_block**.

EXAMPLE:

Parameter Descriptions

bufptr

Pointer to the input buffer (the return value of the get input buffer function). With multiple inputs having a driving input, use the buffer pointer to the driving input (the first call to **RPGC_get_inbuf_by_name**).

Notes / Rules for use

1. This function cannot be used in algorithms having no product data inputs.
2. WARNING: This function cannot be used after releasing the input buffer **bufptr** (returns -1).

```
unsigned int RPGC_get_buffer_vol_seq_num( void *bufptr )
```

PARAMETER: **bufptr** Pointer to the input buffer

RETURN VALUE: Volume scan sequence number.

Returns the sequence number of the current volume scan (the volume number of the product contained in the input buffer). This number increases monotonically (actually it increases to a very large value limited by the maximum value of a short).

EXAMPLE:

Parameter Descriptions

bufptr

Pointer to the input buffer (the return value of the get input buffer function). With multiple inputs having a driving input, use the buffer pointer to the driving input (the first call to **RPGC_get_inbuf_by_name**).

Notes / Rules for use

1. This function cannot be used in algorithms having no product data inputs.
2. WARNING: This function cannot be used after releasing the input buffer **bufptr** (returns -1).

```
int RPGC_get_buffer_elev_index( void *bufptr )
```

PARAMETER: **bufptr** Pointer to the input buffer

RETURN VALUE: The elevation index (1 through N, with N being the number of elevations in the VCP in use)

Returns the "RPG" elevation index of the current elevation (the elevation of the product contained in the input buffer). This is the ordinal of the elevation within the Volume Coverage Pattern in use.

EXAMPLE:

Parameter Descriptions

bufptr

Pointer to the input buffer (the return value of the get input buffer function). With multiple inputs having a driving input, use the buffer pointer to the driving input (the first call to **RPGC_get_inbuf_by_name**).

Notes / Rules for use

1. This function cannot be used in algorithms having no product data inputs.
2. WARNING: This function cannot be used after releasing the input buffer **bufptr** (returns -1).

```
int RPGC_get_buffer_vcp_num( void *bufptr )
```

PARAMETER: **bufptr** pointer to algorithm input buffer

RETURN VALUE: -1 on error or non-negative volume scan number on success

Returns the VCP number of the data used to produce the product contained in the input buffer.

ASSUMPTION:

EXAMPLE:

Parameter Descriptions

bufptr

Pointer to the input buffer (the return value of the get input buffer function). With multiple inputs having a driving input, use the buffer pointer to the driving input (the first call to **RPGC_get_inbuf_by_name**).

Notes / Rules for use

1. This function cannot be used in algorithms having no product data inputs.
2. WARNING: This function cannot be used after releasing the input buffer **bufptr** (returns -1).

```
int RPGCS_get_target_elev_ang( int vcp_num, int elev_ind )
```

PARAMETER: **vcp_num** Volume coverage pattern (VCP) number

elev_ind RPG elevation index within VCP

RETURN VALUE: Error code **RPGCS_ERROR** (defined in rpgcs.h) on error or elevation angle*10 on success (i.e., a return value of 54 is 5.4 degrees).

Returns the target elevation angle of the product contained in the input buffer. The return value is undefined if this function is used on a volume based product.

ASSUMPTION:

EXAMPLE:

Parameter Descriptions

vcp_num

Volume coverage pattern (VCP) number. This number identifies the type of volume scan that produced the current base data input. This value is obtained via the **RPGC_get_buffer_vcp_num** function.

elev_ind

The "RPG" elevation index. This is the ordinal of the elevation within the Volume Coverage Pattern in use. This value is obtained via the **RPGC_get_buffer_elev_index** function.

Notes / Rules for use

1.

```
int RPGCS_get_last_elev_index( int vcp_num )
```

PARAMETER: **vcp_num** Volume coverage pattern (VCP) number

RETURN VALUE: Error code **RPGCS_ERROR** (defined in rpgcs.h) on error or the elevation index.

Returns the last elevation index that can be expected from the VCP that produced the product in the input buffer (i.e., the size of the defined VCP currently being used).

ASSUMPTION:

EXAMPLE:

Parameter Descriptions

vcp_num

Volume coverage pattern (VCP) number. This number identifies the type of volume scan that produced the current base data input. This value is obtained via the **RPGC_get_buffer_vcp_num** function.

Notes / Rules for use

1. **Though currently not used by many algorithms. The following function is preferred. Beginning in Build 11, for algorithms having no product inputs, the last elevation index can be obtained from the scan summary table.**

```
int RPGC_is_buffer_from_last_elev( void *bufptr, int *elev_index,
    int *last_elev_index )
```

PARAMETER:	**bufptr**	Pointer to the input buffer
	elev_index	RPG elevation index within VCP
	last_elev_index	Index of the last elevation (output)
RETURN VALUE:		1 if the buffer data is from the last elevation index, 0 if the buffer data is not from the last elevation index, or -1 on error.

Determines whether the current elevation is the last elevation in the VCP being used for data ingest.

EXAMPLE: cpc008/tsk003/a30831.c
cpc018/tsk001/mdaprodMain.c
cpc018/tsk003/tdaru.c

Parameter Descriptions

bufptr
Pointer to the input buffer (the return value of the get input buffer function). With multiple inputs having a driving input, use the buffer pointer to the driving input (the first call to **RPGC_get_inbuf_by_name**).

elev_index
The "RPG" elevation index. This is the ordinal of the elevation within the Volume Coverage Pattern in use. This value is obtained via the **RPGC_get_buffer_elev_index** function.

last_elev_index (output)
The index of the last elevation in the volume.

Notes / Rules for use

1. This function cannot be used in algorithms having no product data inputs. **For algorithms having no product inputs, the last elevation index can be obtained from the scan summary table.**
2. WARNING: This function cannot be used after releasing the input buffer **bufptr** (returns -1).
3. **This is the preferred function for obtaining the last elevation index. After a future RDA modification, this function must be called every elevation by algorithms producing volume products from elevation input data**.

```
int RPGCS_get_wxmode_for_vcp( int vcp_num )
```

INPUT:	**vcp_num**	Volume coverage pattern (VCP) number

RETURN VALUE: -1 or error or 1 for clear air mode or 2 for precipitation mode.

Returns the weather mode in effect during data collection. **CLEAR_AIR_MODE** and **PRECIPITATION_MODE** are defined in **a309.h**.

EXAMPLE:

Parameter Descriptions

vcp_num

Volume coverage pattern (VCP) number. This number identifies the type of volume scan that produced the current base data input. This value is obtained via the **RPGC_get_buffer_vcp_num** function.

Notes / Rules for use

1.

```
int RPGC_bins_to_ceiling( void *bdataptr, int bin_size )
```

PARAMETER: **bdataptr** pointer to a base data radial

bin_size size of the range bins in meters

RETURN VALUE: the number of bins or **0** on error.

Calculates the number of range bins to 70,000 feet for the given the **bin_size**.

EXAMPLE: cpc007/tsk015/superes8bit.c

Parameter Descriptions

bdataptr

A pointer to a base data radial. This is used to obtain the target elevation angle.

bin_size

The size of the range bins in meters. For basic moment data this is obtained from either the **surv_bin_size** or the **dop_bin_size** field in the basedata header. For advanced Dual Pol data fields this is obtained from the **bin_size** field in the generic moment structure.

Notes / Rules for use (for a data driven task):

1. This function cannot be used in algorithms having no product data inputs.
2. WARNING: This function cannot be used after releasing the input buffer **bdataptr** (returns 0).

The function

```
int RPGC_num_rad_bins( void *bdataptr, int maxbins, int radstep,
                       int wave_type )
```

is used in several Legacy algorithms ported to C but should not be used in new algorithms.

The following functions obtain data structures containing several data fields.

`Scan_Summary* RPGC_get_scan_summary ( int vol_num )`

PARAMETER: **`vol_num`** Sequence number of the current volume scan.

RETURN VALUE: Returns NULL on error, or pointer to Scan_Summary structure for "vol_num" on success. The struct **`Scan_Summary`** is defined in **`orpgsum.h`**.

ASSUMPTION:

EXAMPLE: cpc007/tsk013/bref8bit.c
cpc007/tsk015/superes8bit.c
cpc014/tsk013/saaprods_main.c
cpc018/tsk003/tdaru.c

Parameter Descriptions

`vol_num`

The sequence number of the current volume scan (cycles 1 through 80). This value is obtained via the **`RPGC_get_buffer_vol_num`** function.

Notes / Rules for use

1. The structure of the returned scan summary table could be modified in a future build. This function should be used with caution and only if directly reading the scan summary table is the only source of needed data.

`Vcp_struct* RPGCS_get_vcp_data( int vcp_num )`

PARAMETER: **`vcp_num`** Volume coverage pattern (VCP) number

RETURN VALUE: Returns NULL on error, or pointer to Vcp_struct for "vol_num" on success. The structure **`Vcp_struct`** is defined in **`vcp.h`**.

EXAMPLE: CODE Sample Algorithm 1

Parameter Descriptions

`vcp_num`

Volume coverage pattern (VCP) number. This number identifies the type of volume scan that produced the current base data input. This value is obtained via the **`RPGC_get_buffer_vcp_num`** function.

Notes / Rules for use

1. *This function is not yet used in any algorithm.*

The following function reads the volume status portion of the general status message. Another function, listed in Part C of Section I, accomplishes the necessary registration and the obtains the pointer to the data. Much of the information is also available in the scan summary table or even the base data header.

However, this function is useful in event driven algorithms and must be used with the registration function if the even driven algorithm has no data input. It is the primary method of determining the volume number, sequence number, and the volume date - time if not registered for product inputs.

```
int RPGC_read_volume_status( )
```

PARAMETER: None

RETURN VALUE: Always returns 0.

Updates the volume status structure obtained by the registration function.

EXAMPLE: cpc008/tsk001/alerting_buffer_control.c

Notes / Rules for use (for a data driven task):

1. The registration function **`RPGC_reg_volume_status`** must be called before this function can be used.
2. If the task has product inputs, the structure returned by **`RPGC_reg_volume_status`** will be automatically updated each volume and this read function need not be called. If the task has no product inputs this function must be called every volume to update the data..

Part F. Reading Adaptation Data

The adaptation mechanism was replaced in Build 6. Some algorithms recently ported from Fortran to C use the old mechanism. This document only describes the current adaptation data system. See sample algorithm 1 for a simple example.

From the application's point of view, the major purpose of adaptation data is to parameterize certain characteristics of an algorithm to permit changing its behavior using ORPG configuration files.

With the previous adaptation data mechanism, a C structure was automatically populated with all of the algorithm's adaptation data fields. With the new DEA mechanism, each data field (data element) must be read individually from the database. The algorithm API provides two helper functions to read adaptation data. A 'DEA access function' must be written using the following API functions to read individual data elements.

`RPGC_ade_get_values` is used to read all integer and floating point data. This function is also used for enumerated types.

`RPGC_ade_get_string_values` is used to read all string data.

If the data element consists of a list of values (an array), the function **`RPGC_ade_get_number_of_values`** is used to determine the number of values defined.

Typically an adaptation data "access function" using these calls is written as part of the algorithm. Though not required, the ROC has included these access functions as part of **`libadaptstruct`** for C algorithms and **`libadaptcomblk`** for Fortran algorithms. This facilitates sharing of adaptation data by making the access function easily available to all algorithms. However, an algorithm can access any adaptation data by using the full ID of the data element (including the algorithm group name) with the API data read functions.

Modification of these libraries is not recommended for development activity that is not directly involved with the ROC in integrating new algorithms into the operational ORPG. If modifying an existing algorithm having a data access function already integrated into the shared library, an alternative to modifying the original access function is to develop a new access function that reads the new adaptation data elements.

```
int RPGC_ade_get_values( char *alg_name, char *value_id, double *values )
```

INPUT:	**`alg_name`**	The name of the algorithm "owning" this data.
	`value_id`	The variable name in the DEA file.
	`values`	Pointer to a temporary variable to hold the numeric value.

RETURN VALUE: Returns -1 on error, 0 on success.

Retrieves a value of a data element from the database given "alg_name", "value_id", and a pointer to a variable to receive the value(s).

EXAMPLE:
```
ret = RPGC_ade_get_values("alg.sample1_dig.",
                          "db_Element_1",&get_value);
if(ret == 0) {
     struct_ptr->my_element_1 = (short)get_value;
}
```
where **struct_ptr** is the address of the C structure and **get_value** is type **double**.

Parameter Descriptions

alg_name

A string containing the name of the algorithm "owning" this data. This is identical to the group name parameter used in the callback registration function. The group name is based on the inverse of the filename of the adaptation data DEA file. With sample algorithm 1 for example, The adaptation data file is named **sample1_dig.alg**.
The resulting alg_name parameter is "**alg.sample1_dig.**" (note the trailing period).

value_id

A string containing the variable name as it appears in the DEA file.

values

Pointer to a temporary variable to hold the numeric value(s). This must be cast to the correct type of the variable representing the data filed. See example.

Notes / Rules for use

1.

```
int RPGC_ade_get_string_values( char *alg_name, char *value_id, char **values )
```

PARAMETER:

alg_name	The name of the algorithm "owning" this data.
value_id	The variable name in the DEA file.
values	Pointer to a temporary variable to hold the string.

RETURN VALUE: Returns -1 on error, 0 on success.

Retrieves string(s) from the database given "alg_name", "valid_id", and a pointer to string to receive the string(s).

EXAMPLE: cpc101/lib004/rpgc_site_info_callback_fx.c

Parameter Descriptions

alg_name

A string containing the name of the algorithm "owning" this data. This is identical to the group name parameter used in the callback registration function. The group name is based on the inverse of the filename of the adaptation data DEA file. With sample algorithm 1 for example, The adaptation data file is named **sample1_dig.alg**.
The resulting alg_name parameter is "**alg.sample1_dig.**" (note the trailing period).

value_id
A string containing the variable name as it appears in the DEA file.

values
Pointer to a temporary variable to hold the string.

Notes / Rules for use

1. *This function is not yet used in any algorithm.*

```
int RPGC_ade_get_number_of_values( char *alg_name, char *value_id )
```

PARAMETER:	**alg_name**	The name of the algorithm "owning" this data.
	value_id	The variable name in the DEA file.
RETURN VALUE:		The number of data values in the list / array or with failure, a negative error code.

Returns the number of value(s) or negative error code.

EXAMPLE:

Parameter Descriptions

alg_name
A string containing the name of the algorithm "owning" this data. This is identical to the group name parameter used in the callback registration function. The group name is based on the inverse of the filename of the adaptation data DEA file. With sample algorithm 1 for example, The adaptation data file is named **sample1_dig.alg**.
The resulting alg_name parameter is "**alg.sample1_dig.**" (note the trailing period).

value_id
A string containing the variable name as it appears in the DEA file.

Notes / Rules for use

1. *This function is not yet used in any algorithm.*

Functions Not Required for Algorithms

The following functions are not required by normal algorithms and are not documented.

```
int RPGC_read_ade( int *callback_id, int *status )
```

```
int RPGC_update_all_ade()
```

Part G. Reading Base Data Messages

Guidance for Determining the Number and Size of Radial Data Arrays

The best approach to take in determining the number of radials and sizing of data arrays is described in is Part B of Volume 3, Document 4, Section II. To summarize:

- **Determining the maximum size of the data array (number of bins)** (Data above 70,000 feet MSL are not valid).
  CAUTION: If either statically allocating arrays for the radials or allocating a standard size at the beginning of an elevation, the method used must be conservative. It is always possible for the first radial in an elevation to be spot blanked and contain no data bins. The reference above includes a discussion of attempting to preserve resources if pre-allocating all of the radial arrays at the beginning of an elevation.
- **Determining the size of an individual radial** (Data above 70,000 feet MSL are not valid).
  - When reading basic moments (R, V, SW) from one of the Legacy resolution data types (**BASEDATA**, **REFLDATA**, or **COMBBASE**) the maximum size of the reflectivity array is 460. Beginning with Build 12 the maximum size of the velocity and spectrum width arrays is 1200 at lower elevations and 920 at higher elevations. **This must be checked every elevation.** Prior to Build 12 the maximum size was 920 bins. The reference above contains details and sample code that includes consideration of the 70,000 foot limit and product range limit.
  - When reading basic moments (R, V, SW) from one of the super resolution data types (**SR_COMBBASE**, etc.) or the dual pol data types (**DUALPOL_COMBBASE**, etc.) the size of the reflectivity array can be either 460 or 1840. Beginning with Build 12 the maximum size of the velocity and spectrum width arrays is 1200 at lower elevations and 920 at higher elevations. **This must be checked every elevation.**
  - When reading one of the Dual Polarization fields the number of valid data values within the array is determined by the value of **no_of_gates** field in the generic moment structure. The standard sizes of the Dual Pol field arrays are either 1200 or 1840. The reference above contains details and sample code that includes consideration of the 70,000 foot limit and product range limit. **The presence of the Dual Pol fields must be checked every elevation.**
- **Determining the radial spacing**. When reading the first radial of an elevation, the field **azm_reso** is used obtain the radial spacing. If the value is 1, the radials are in the new higher resolution of half degree spacing. **This must be accomplished every elevation if reading one of the Super Resolution data types (SR_COMBBASE, etc.).**
- **Determining the data sample bin size (in range)**. When reading the first radial of an elevation:
  - For the basic moments (R, V, SW) the field **surv_bin_size** is used to obtain the size of the surveillance bins and determine the range of the data. If the value is 250, the surveillance data is in the higher resolution of 250 meters.
  - For the Dual Pol data the **bin_size** field in the generic moment structure is used to obtain the size of the Dual Pol data bins and determine the range of the data.

In addition,

- **For basic moments, the fields `n_dop_bins` / `n_surv_bins` must be checked when reading each radial** to ensure data beyond the last good bin is not used (and the algorithm's data array padded with zero's). This can eliminate checking for the spot blank field in the basedata header.
- **For Dual Pol data, the field `no_of_gates` must be checked when reading each radial** to ensure data beyond the last good bin is not used (and the algorithm's data array padded with zero's).

Following this procedure will accommodate any future changes to the design of the VCPs.

The contents of the base data header and the structure and contents of the radial and elevation base data messages is covered in Volume 2, Document 4, Section I, *Base Data Format*.

Reading Basic Moments (R, V, SW)

Reading the data arrays contained in base data messages must be accomplished by using the correct offsets into the base data radial / elevation. The base data header fields **`ref_offset`**, **`vel_offset`**, and **`spw_offset`** give the offset for the reflectivity array, velocity array, and the spectrum width array.

1. **Base Data Radial Messages** - These offsets are in bytes from the beginning of the header.
2. **Base Data Elevation Messages** - **Prior to Build 10**, these offsets are in bytes relative to the beginning of the base data header. **After Build 10**, these offsets are in bytes relative to the beginning of the first data array (e.g., just after the end of the basedata header).

The size of the data arrays depend upon the type of base data registered. When registering for the original types (**`REFLDATA`**, **`COMBBASE`**, **`BASEDATA`**), the reflectivity arrays are 460 and the Doppler arrays are 920. Beginning with Build 10, registering for the new types (Super Resolution and Dual Pol) the reflectivity arrays can be either 460 or 1840 and the Doppler arrays either 920 or 1200. The following are defined in **`basedata.h`**.

BASEDATA_REF_SIZE	460	Size of the reflectivity array
MAX_BASEDATA_REF_SIZE	1840	Super Res reflectivity array
BASEDATA_VEL_SIZE	920	Size of the Doppler arrays
BASEDATA_DOP_SIZE	1200	Super Res Doppler arrays

The size of the arrays for the current elevation are obtained indirectly from related base data header fields:. The value of these fields must be tested with every elevation if registered for **`SR_`** and **`DUALPOL_`** data types for two reasons: (1) the volume coverage patterns can be redefined and (2) operationally the Super Resolution capability can be disabled in the RDA.

The position of the first good data value is determined by the fields: **`surv_range`** and **`dop_range`**. This value is usually 1 which means the first position in the array has the first good data value.

If statically allocating space for a complete elevation of radial base data, the following macros should be used in order to handle extreme conditions in the WSR-88D radar. Also note that in Build 10 the **`Compact_basedata_elev`** structure in **`basedata_elev.h`** was modified to require allocation of memory for each **`Compact_radial`** structure.

BASEDATA_MAX_RADIALS	400	Maximum number or radials with a 1.0 degree azimuth interval
BASEDATA_MAX_SR_RADIALS	800	Maximum number of radials with a 0.5 degree azimuth interval

NOTE: Current basic moment data arrays are arrays of shorts in the radial message and arrays of bytes in the elevation message. **At some point in the future the radial message arrays may be converted to bytes.**

Functions for Reading Basic Moments

The base data moments (velocity, spectrum width, and reflectivity) are no longer guaranteed to be in a specific order. The data offsets in the radial header or the provided helper functions must be used to access the data arrays.

For radial messages only, three helper functions can be used rather than the offsets. Using a pointer to the base data radial message obtained when reading the linear buffer, these functions return the pointer to the beginning of the array containing the desired moment. These functions cannot be used with elevation basedata messages.

`RPGC_get_surv_data`
Returns a pointer to the surveillance data array.

`RPGC_get_vel_data`
Returns a pointer to the radial velocity data array.

`RPGC_get_wid_data`
Returns a pointer to the spectrum width data array.

`RPGC_get_radar_data` (Available with Build 10)
Though designed to read the future Dual Polarization data fields, this function can also read the basic moments. Returns a pointer to the designated data array.

The function **`RPGC_bins_to_ceiling`** provides a convenient means of determining the last good data point in the array based upon the 70,000 ft MSL data cut-off. The function **`RPGC_num_rad_bins`** accomplished the same but is only valid for Legacy recombined data. These functions are described in Part E of this document.

```
void* RPGC_get_surv_data( void* bufptr, int* first_good_bin,
                          int* last_good_bin )
```

PARAMETER:	**`bufptr`**	Pointer to the radial message
	`first_good_bin`	Pointer to the index of the first bin (output)
	`last_good_bin`	Pointer to the index of the last bin (output)
RETURN VALUE:		Pointer to start of surveillance data within a radial, or NULL on failure.

EXAMPLE:

```
void* RPGC_get_vel_data( void* bufptr, int* first_good_bin,
                         int* last_good_bin )
```

PARAMETER:	**bufptr**	Pointer to the radial message
	first_good_bin	Pointer to the index of the first bin (output)
	last_good_bin	Pointer to the index of the last bin (output)
RETURN VALUE:		Pointer to start of velocity data within a radial, or NULL on failure

EXAMPLE:

```
void* RPGC_get_wid_data( void* bufptr, int* first_good_bin,
                         int* last_good_bin )
```

PARAMETER:	**bufptr**	Pointer to the radial message
	first_good_bin	Pointer to the index of the first bin (output)
	last_good_bin	Pointer to the index of the last bin (output)
RETURN VALUE:		Pointer to start of spectrum width data within a radial, or NULL on failure.

EXAMPLE: *This function is not yet used in any algorithm.*

Parameter Descriptions

bufptr

A pointer to the input buffer containing radial data.

first_good_bin (output)

A pointer to the value of the index to first good data bin. The index can be used with the returned pointer to the surveillance data (almost always 0). That is the index is intended for use in a C array (i.e., beginning with 0). NOTE: This value is 1 less than the contents of the base data header field **surv_range** or **dop_range** which were array indexes originally intended for FORTRAN (i.e., beginning with 1).

last_good_bin (output)

A pointer to the value of the index to last good data bin. The index can be used with the returned pointer to the surveillance data (almost always 0). That is the index is intended for use in a C array (i.e., beginning with 0). This value is calculated from the base data header fields **dop_range** & **num_dop_bins** for velocity and spectrum width and from **surv_range** & **num_surv_bins** for reflectivity data. This value reflects the 70,000 feet MSL cutoff.

Notes / Rules for use

1. It is important that the return value always be tested for NULL. Though rare, it is possible to have a NULL pointer to the data array even though that moment is enabled in the RDA.
2. It is critical that the algorithm does not read and use data that exceed the limits of the first and last "good" bin. See item 2 - *Processing Individual Radials* in Volume 3, Document 4, Section II Part B. If the algorithm internal arrays or product arrays exceed this, they should be padded with '0'.

Determining Basedata Message Type

The following convenience functions provide a method of determining the type of basedata radial message. They provide a method of reading the cut type bits and the radial type bits in the **msg_type** field in the base data header.

RPGC_check_radial_type can be used to determine two factors.

- **In Build 12.1 the meaning of the SUPERRES_TYPE bit reflects the azimuth resolution.**
- **In Build 12 the meaning of the HIGHRES_REFL_TYPE bit reflects the horizontal range of the surveillance bins.** The information about the size and resolution of the data can be obtained via other basedata header fields.
- The values of the other radial type bits are directly related to the type of basedata being used so as long as the algorithm registers for the correct type data these fields do not need to be tested.

```
unsigned short RPGC_check_radial_type( void *radptr,
                                       unsigned short check_type )
```

PARAMETER: **radptr** address of the basedata message

check_type bit mask for radial type desired

RETURN VALUE: The result of AND'ing the bit mask provided in the **check_type** parameter with the value of the radial type bits in the basedata header **msg_type** field. These are bits 8, 9, 10, 11, and 12.

Used with the appropriate radial type masks to determine the radial type of the basedata message.

EXAMPLE: cpc007/tsk015/superes8bit.c

Parameter Descriptions

radptr

The address of the basedata radial message (or at least the basedata header).

check_type

A bit mask for the radial type desired. Possible values are: **SUPERRES_TYPE**, **DUALPOL_TYPE**, **RECOMBINED_TYPE**, and **PREPROCESSED_DUALPOL_TYPE** or any combination OR'd together. Using the **RADIAL_TYPE_MASK** passes the value of all of the radial type bits which can be subsequently tested. These masks are defined in **basedata.h**.

Notes / Rules for use (for a data driven task):

1. WARNING: The return value is undefined if the **radptr** is NULL.
2. The use of this function is not recommended until Build 12.

RPGC_check_cut_type is useful if registered for one of the **BASEDATA** types instead of a **REFLDATA** or **COMBBASE** type in determining whether the radial is desired.

```
unsigned short RPGC_check_cut_type( void *radptr, unsigned short check_type )
```

PARAMETER:	**radptr**	address of the basedata message
	check_type	bit mask for cut type desired
RETURN VALUE:		The result of AND'ing the bit mask provided in the **check_type** parameter with the value of the cut type bits in the basedata header **msg_type** field. These are bits 5, 6, and 7.

Used with the appropriate cut type masks to determine the cut type of the basedata message..

EXAMPLE: cpc007/tsk015/superes8bit.c

Parameter Descriptions

radptr

The address of the basedata radial message (or at least the basedata header).

check_type

A bit mask for the cut type desired. Possible values are: **REFLDATA_TYPE**, **COMBBASE_TYPE**, and **BASEDATA_TYPE** or any combination OR'd together. Using the **CUT_TYPE_MASK** passes the value of all of the cut type bits which can be subsequently tested. These masks are defined in **basedata.h**.

Notes / Rules for use (for a data driven task):

1. WARNING: The return value is undefined if the **radptr** is NULL.
2. To select the desired radials for products needing only surveillance data, a bit mask of **REFLDATA_TYPE | BASEDATA_TYPE** would return a value of TRUE for desired radials.
3. To select the desired radials for products needing Doppler data. a bit mask of **COMBBASE_TYPE | BASEDATA_TYPE** would return a value of TRUE for desired radials

---

Reading Dual Polarization Data (in the generic moment)

NOTE: Preliminary support for Dual Pol data was provided in Build 12 in order to support implementation. Actual Dual Polarization implementation is provided in Build 12.1 and Dual Pol data is available in data produced by a Build 12 RDA.
Some details concerning Dual Pol data as described in this document are subject to change.

Beginning with Build 10, support is provided for advanced data fields that will be contained in the series of generic moment structures in the basedata radial message. This data will not be available until Build 12. Only preliminary development support for these Dual Pol fields is provided in Build 11. See Volume 2, Document 4, Section I, *Base Data Format*, for a description of these data types and the structure of the basedata radial message.

These data fields can be accessed by using the offsets provided in the **`offsets`** array in the base data header. The number of offsets in the array is given by the **`no_moments`** field. Each offset must be tested for a value of 0, meaning no data.

The offset obtains the generic structure address. Reading the advanced data fields contained in the generic moment structures is more involved than the original basedata moments and requires:

- Reading the **`name`** field in the structure to determine the data type.
- Reading the **`data_word_size`** field in the structure to determine whether the **`gate union`** field is an **`unsigned char`** array, **`unsigned short`** array, **`unsigned int`** array, or **`float`** array.

The position of the first good data value is assumed to be 1, which means the first position in the array has the first good data value. The number of valid data values within the array is determined by the value of **`no_of_gates`** field.

The Dual Pol data fields available to algorithms are either 1200 or 1840. The maximum number of data bins in each type of array is:

`1200 for ZDR, RHO, PHI, KDP, SDP`
`1840 for SNR, SMZ, SDZ` (related to number of Z gates)
`1200 for SMV` (related to number of V gates)

Currently it is unclear whether in Build 12 the number of gates could be 900 instead of 1200 and 460 instead of 1840. This could occur at higher elevations or if 'Super Resolution' is turned off at the RDA. If preallocating arrays use 1200 or 1840 as appropriate. Even though the following are defined in **`basedata.h`**, as you can see this list is not complete.

BASEDATA_RHO_SIZE	1200	Size of the DRHO Correlation Coefficient array (shorts)
BASEDATA_PHI_SIZE	1200	Size of the DPHI Differential Phase array (shorts)
BASEDATA_SNR_SIZE	1840	Size of the DSNR Signal to Noise Ratio array (bytes)
BASEDATA_ZDR_SIZE	1200	Size of the DZDR Differential Reflectivity array (bytes)
BASEDATA_RFR_SIZE	240	Ignore for now.

Function for Reading Generic Moment Data Fields (Dual Pol Data) and Basic Moments

This helper function can be used to obtain the pointer to the array containing the additional desired data fields present in the base data message instead of using the **no_moments** field and the **offsets** array field in the basedata header. This function reads radar data contained in the Generic moment structure. This structure contains advanced data fields including Dual Polarization data fields targeted for Build 12 Only preliminary development support for Dual Pol data is provided in Build 11. In addition, this function can read the basic data moments (reflectivity, velocity, and spectrum width).

```
void* RPGC_get_radar_data( void* bufptr, int type, Generic_moment_t *mom )
```

PARAMETER: **bufptr** Pointer to the radial message

type The Type of Data Desired

mom A pointer to a generic moment structure

RETURN VALUE: Returns a pointer to the desired data array in the base data message. Returns NULL if data is not present or message is not a radial message.

Given the address of the radial message and the type of data desired, this function returns a pointer to the start of the desired data and populates a generic structure with appropriate header information.

EXAMPLE:

```
s_data = (short *) RPGC_get_radar_data( (void *)radial,
                                        RPGC_DREF, &gen_moment );
```

where

```
Generic_moment_t gen_moment;
Generic_moment_t *my_gen_moment = &gen_moment;
```

Parameter Descriptions

bufptr

A pointer to the basedata message obtained by **RPGC_get_inbuf**.

type

The type of data field to read are defined in **rpgc.h** (ignore **RPGC_DRF2** and **RPGC_DRFR**).

RPGC_DZDR - Differential Reflectivity
RPGC_DPHI - Differential Phase
RPGC_DRHO - Correlation Coefficient
RPGC_DSNR - Signal-to-Noise Ratio
RPGC_DSMZ - Smoothed Reflectivity
RPGC_DSMV - Smoothed Velocity
RPGC_DKDP - Specific Differential Phase
RPGC_DSDZ - Texture (standard deviation) of Reflectivity
RPGC_DSDP - Texture (standard deviation) of Differential Phase

For data types that have not been predefined, use the type **RPGC_DANY**. When **RPGC_DANY** is used the desired type must be entered in the 'name' field of the structure passed in the **mom** parameter. Even though the original base moments are not contained in a generic moment structure. they can be read with this function using the types **RPGC_DREF**, **RPGC_DVEL**, and **RPGC_DSW**..

mom

A pointer to a generic moment structure, type **Generic_moment_t** defined in **generic_basedata.h**. You must provide the address of a declared variable of that type. A NULL pointer cannot be used. The header fields in this structure are populated with the information required to interpret the data array. In the case that the type passed is **RPGC_DANY**,

the 'name' field in the generic moment structure must be set to the desired 4 character name before calling this function.

Notes / Rules for use

1. *This function has been tested in a sample algorithm and is available for use.*

Part H. Aborting Product Construction

This guidance is based upon changes in the ORPG algorithm abort services introduced after the initial deployment of the ORPG. However, some algorithms have a different pattern of usage based upon the legacy algorithms (which were not completely consistent).

The typical reason an algorithm is aborted is an inability to read input data or input data that are out of sequence. All algorithms must do one of the following:

1. **Successfully output a product, or**
2. **Call the appropriate abort function.**

The algorithm developer is responsible for accomplishing an abort. When aborting product generation:

- All output buffers successfully obtained must be released with **`RPGC_rel_outbuf`** (or **`RPGC_rel_all_outbufs`**) using the value **`DESTROY`** for the **`disposition`** parameter.
- All input buffers successfully obtained must be released with **`RPGC_rel_inbuf`** (or **`RPGC_rel_all_inbufs`**).
- One of the RPGC abort routines must be called. Finally, control must be returned to the *algorithm control loop* function.

Note: The following function has been deprecated and should not be used.

- **`RPGC_abort_datatype_because`**

This will be eliminated in a future build. If any development software uses this function it should be replaced with **`RPGC_abort_dataname_because`**.

`int RPGC_abort( )`

PARAMETER: None

RETURN VALUE: Not Used

Accomplishes infrastructure house cleaning before returning to the *algorithm control loop*. An abort reason code is generated internally based upon the failure to obtain either an input buffer or an output buffer.

EXAMPLE:

`int RPGC_abort_because( int reason )`

PARAMETER: **reason** Abort reason code.

RETURN VALUE: Not Used

Accomplishes infrastructure house cleaning before returning to the *algorithm control loop*. Passes an abort reason code specified by the **reason** parameter.

EXAMPLE:

```
int RPGC_abort_dataname_because( char *dataname, int reason )
```

PARAMETER: **dataname** Name of the output product not being generated.

reason Abort reason code.

RETURN VALUE: Not Used

Accomplishes infrastructure house cleaning before returning to the *algorithm control loop*. Passes an abort reason code specified by the **reason** parameter. This abort function is used by algorithms producing more than one product type. This does not include products requiring customizing data in the product request parameters (other than the standard elevation parameter). Allows aborting a request for one product type while continuing to generate other products.

EXAMPLE: cpc007/tsk001/basrflct.c
cpc014/tsk003/hybrprod.c

```
int RPGC_abort_request( User_array_t *request, int reason )
```

PARAMETER: **request** Address of the user array entry containing a specific product request.

reason Abort reason code.

RETURN VALUE: Not Used

Accomplishes infrastructure house cleaning before returning to the *algorithm control loop*. Passes an abort reason code specified by the **reason** parameter. This abort function is used by algorithms producing products requiring customizing data in the product request parameters. Allows aborting one request for a product type while satisfying other requests for that product type.

EXAMPLE: cpc007/tsk015/superes8bit.c
cpc014/tsk011/user_sel_LRM.c
cpc014/tsk014/saausers_main.c

```
int RPGC_cleanup_and_abort( int reason )
```

PARAMETER: **reason** Abort reason code.

RETURN VALUE: Not Used

Releases all acquired input buffers that are not already released, releases all output buffers still open using a DESTROY disposition (see **RPGC_rel_outbuf**), and accomplishes infrastructure house cleaning before returning to the *algorithm control loop*. Passes an abort reason code specified by the **reason** parameter.

EXAMPLE: cpc013/tsk005/prcprate_RateAlg_buffctrl.c
cpc013/tsk008/saa_main.c

Parameter Descriptions

reason

Abort reason code. Abort reason codes are defined in **prod_gen_msg.h** (prior guidance stipulated use of codes in **a309.h**). In some cases the reason code is simply the value of *opstat* returned by *get_inbuf* and *get_outbuf* calls. In other cases the codes **PGM_MEM_LOADSHED**, **PGM_DISABLED_MOMENT**, **PGM_INVALID_REQUEST**, **PGM_PROD_NOT_GENERATED**, or **PGM_INPUT_DATA_ERROR** are explicitly passed. See the rules for use below.

dataname

The name of the product not being produced. **The value of this parameter is the same as the output product registered.** This name must be in the list contained in the **output_data** attribute of the **task_attr_table** entry for the algorithm task (the **output_data** entry determines the number or ID of the linear buffer containing the output data). See CODE Guide Volume 2, Document 2, Section III for a complete explanation of product and task configuration via the **product_attr_table** and **task_attr_table** configuration files.

request

The individual request being aborted. The user array contains up to 10 product requests and is obtained via the **RPGC_get_customizing_data** call. The first user array entry is at **&user_array**, the second request (if there is one) is at **&user_array[1]**, and so forth.

Notes / Rules for use

1. If all products cannot be successfully completed, at least one abort service must be called.
2. The appropriate abort function is called after releasing the appropriate open output buffer (if any) and prior to returning to the loop control function.
3. The current guidance is to use **RPGC_abort** when possible. This basic function can be used if the failure is due to the inability to obtain either an input or output buffer and the task only produces one product.
4. With algorithms producing more than one product type, **RPGC_abort_dataname_because** is used when unable to complete the specified product type but able to continue the generation of other products.
5. With algorithms producing products that use customizing data from the product request message, **RPGC_abort_request** is used when unable to complete generation of a product for a specific request message but able to continue the generation of that product for other requests.
6. The convenience function **RPGC_cleanup_and_abort** can be used in situations where no remaining requested products can be completed. **This function has resulted in task failures in some cases.**
7. A reason code must be passed when the applicable abort function requires a code and when aborting for reasons other than the failure to obtain an input or output buffer. The correct reason code to be passed is:
    - Typical Algorithms (those NOT producing **RADIAL_DATA**)

        - If the abort is because of an inability to get a buffer with *get_inbuf* or *get_outbuf*, the **opstatus** returned is used as the reason code.
        - If the abort is because not all required base data moments are enabled (in the case where the **RPGC_what_moments** test was used), **PGM_DISABLED_MOMENT** is the reason code passed. Note: The above test is not required if the required basic moments are registered (see **RPGC_reg_moments**).
        - Beginning with Build 12, **PGM_DISABLED_MOMENT** is also used if a needed advanced data field (Dual Pol in the generic moment structure) is not available.
        - If the abort is because of the failure to obtain memory via **malloc** or **calloc**, **PGM_MEM_LOADSHED** is passed.
        - When the current elevation cut does not support the product being generated in algorithms ingesting one of the super resolution data types use **PGM_PROD_NOT_GENERATED.**
        - When reading a non-product data store fails use **PGM_PROD_NOT_GENERATED.**
        - If there is some algorithm unique reason for being unable to complete product generation, then
            - Use the **PGM_INPUT_DATA_ERROR** code if problem is generally related to the nature of the input data.
            - Use the **PGM_INVALID_REQUEST** code if the problem is related to the nature of the product request parameters.
        - DEFAULT: If the reason for aborting the product generation does not clearly fall into one of the above cases, **PGM_PROD_NOT_GENERATED** is a safe default reason code that can be used.
    - Algorithms Producing **RADIAL_DATA**
        - The only time an algorithm producing **RADIAL_DATA** aborts is for the failure to obtain an input buffer or output buffer. The status returned by the API function is used as the abort reason so **RPGC_abort** is used if the radial output is the only output.
        - For other conditions (disabled basic moment, an advanced Dual Pol data missing, etc.) the radial is passed along. See *Special Instructions for Radial Producers* in Volume 3 Document 3 Section I Part D.

8. Some abort functions permit continued processing while others do not
    - After RPGC**_abort**, **RPGC_abort_because** or **RPGC_cleanup_and_abort** are called, processing (reading input, writing output, etc.) must not continue. The algorithm must accomplish any necessary cleanup and return to the loop control function.
    - After **RPGC_abort_dataname_because** is called, processing of other registered output products (if any are requested) may continue.
    - After **RPGC_abort_request** is called, processing of additional requests for that product (if any) may continue. The correct reason code would be **PGM_INVALID_REQUEST** if there is a problem with the request message.

The following function was not intended for normal replay product request responses and should not be used by normal algorithms. This function is not documented.

```
int RPGC_product_replay_request_response( int pid, int reason,
                                          short *dep_params )
```

Part I. Non-product Data Access

There are situations where an algorithm can use a persistent data store (on disk) that does not need to be configured as an intermediate product. Part D of CODE Guide Vol 2, Document 2, Section III - *ORPG Configuration for Application Developers* provides guidance in deciding when to use an intermediate product and when to use a non-product data store.

> Several Legacy algorithms ported from FORTRAN use another mechanism of sharing persistent data called 'Inter-Task Communication (ITC) Blocks'. Even though the C Algorithm API includes functions to use ITC blocks, they are not recommended for use in new algorithms. For new algorithms, non-product data stores are used in place of 'ITC Blocks'

There are 11 functions involved in access of a linear buffer non-product data store. Currently some these functions are used by the snow product algorithm in **`cpc013/tsk008`** (saa_compute_products.c) and in **`cpc014/tsk014`** (saausers_main.c & saausers_io_functions.c)

- **`RPGC_data_access_open`** opens a linear buffer that has been configured in the **`data_attr_table`** configuration file.
- **`RPGC_data_access_close`** closes a linear buffer.
- **`RPGC_data_access_list`** - obtains a list of information about linear buffer messages. This is a convenient means of getting a list of available message IDs.
- **`RPGC_data_access_read`** reads a data message from a linear buffer non-product data store.
- **`RPGC_data_access_write`** writes a data message to a linear buffer non-product data store.
- **`RPGC_data_access_seek`** repositions the read pointer. Normally used with a *message queue* type linear buffer.
- **`RPGC_data_access_clear`** erases a specified number of older messages in a buffer. Used in specific situations to make room for new messages.

The following function provides a means of reacting to an update to a specific message buffer.

- **`RPGC_data_access_UN_register`** is used to register a callback handler function that is executed when the specified buffer is updated.

The following functions are used to obtain information about a buffer or a particular message in a buffer.

- **`RPGC_data_access_previous_msgid`** obtains the message ID of the last message read from or written to the buffer.
- **`RPGC_data_access_msg_info`** obtains information about the linear buffer message specified.
- **`RPGC_data_access_stat`** obtains several kinds of status information about a linear buffer

There are 5 function that provide database style access of a linear buffer non-product data store. Currently these functions are not used in any operational algorithm.

- **RPGC_DB_select** obtains the 'location' of messages that match the query provided
- **RPGC_DB_get_record** obtains the entire message (record) based upon the 'location'.
- **RPGC_DB_get_header** obtains a reference to just the query header portion of the message (record).
- **RPGC_DB_insert** inserts a new record into the database linear buffer.
- **RPGC_DB_delete** permanently removes a message (record) from the database.

There are two functions that support use of a standard disk file non-product data store. Currently used by the snow algorithm in **cpc013/tsk008** (saa_file_io.c).

- **RPGC_get_working_dir** obtains the current ORPG working directory.
- **RPGC_construct_file_name** creates complete-path filename for the disk file non-product data store.

Basic Linear Buffer Access Functions

These functions support access to *Public* non-product data stores which are implemented as linear buffers and configured in the **data_attr_table** configuration file. Functions are provided to open and close the buffer files and to read a message from the buffer and write a message into the buffer.

Currently, the *Public* non-product linear buffers are frequently configured as 'replaceable' (**LB_REPLACE**) which creates a non-sequential linear buffer where messages are written with a specific user provided message ID. If the user writes a message with a previously used message ID, it replaces the message with the same ID. If a user writes a message with an ID not being used, the message is added to the buffer.

A *message database* type buffer (**LB_DB**) has been defined that provides either the behavior of the **LB_REPLACE** type or the **LB_MSG_POOL** type. The purpose of this type of buffer is to provide a non-sequential message set. The algorithm is responsible for making room for more messages when the buffer is full. With **LB_DB** buffers, the behavior is determined by the method of specifying the user ID when adding messages to the linear buffer. The method chosen must be used consistently. With this type of linear buffer the user has more control because any specific message can be read or updated (replaced). Each message could even contain different types of data.

- If messages are added by writing with a user specified message ID (a non-zero positive integer), the subsequent behavior is like **LB_REPLACE**. It is the users' responsibility to use the IDs uniquely. The advantage of this method is each specific message ID can contain an algorithm specific type of data.
- If messages are added by writing with a message ID of **LB_ANY**, the subsequent behavior is the **LB_MSG_POOL** or database type. The user must subsequently obtain the message ID chosen by the infrastructure in order to access the message.

Public non-product linear buffers can also be configured as a *message queue* type buffer (the default **LB_QUEUE**). The purpose of this type of buffer is to write message sequentially and read message sequentially. When full, the older messages are automatically deleted. This type of data store might be advantageous for use even with product data (associated with the radar scan) if there is a need to read

previous messages. The **RPGC_data_access_seek** function permits moving the read pointer to provide access to previous messages.

> NOTE: These linear buffer access functions require that the linear buffer be managed by the ORPG by being defined in the **data_attr_table** configuration file.

Basic Access Functions

The **RPGC_data_access_open** function opens the buffer with the desired access privileges. Normally **RPGC_data_access_close** is not used unless the access privileges must be changed.

```
int RPGC_data_access_open( int data_id, int flags )
```

PARAMETER:	**data_id**	The linear buffer ID
	flags	Access permission flags
RETURN VALUE:		The LB descriptor on success or a negative number for indicating an error condition (codes defined in **orpgda.h**)

Opens the linear buffer **data_id** for access permissions **flags**. If buffer is already open no action is taken.

EXAMPLE: cpc005/tsk002/pcipdalg.c
cpc013/tsk008/saa_compute_products.c
cpc014/tsk014/saausers_main.c

Parameter Descriptions

data_id

The identity of the linear buffer. Non-product buffer id's are defined in **orpgdat.h** / **orpgdat.inc**. If the header file has not been modified to include this new data id, the **<Data_Buffer_Number>** must be entered rather than the **<Data_Buffer_Name>**. Whichever is used (name or number), it must be the same as the **data_id** attribute for the data store definition in the **data_attr_table** configuration file.

flags

Read / write permission flags. These are '**LB_READ**' for read only, '**LB_WRITE**' for write only, or '**LB_READ | LB_WRITE**' for both.

Notes / Rules for use

1. **RPGC_data_access_open** must be called with the required permissions. If the linear buffer is opened read-only, in order to write to the buffer it would have to be closed with **RPGC_data_access_close** then reopened with write permission.
2. If a file based linear buffer (typically used by algorithms) is already open, no action is taken by this function. However, if a shared memory linear buffer is already open, **RPGC_data_access_open** may fail.

```
int RPGC_data_access_close( int data_id )
```

PARAMETER: **data_id** The linear buffer ID

RETURN VALUE: Not Used

Closes the linear buffer **data_id**.

EXAMPLE: *This function is not yet used in any algorithm*

Parameter Descriptions

data_id

The identity of the linear buffer. Non-product buffer id's are defined in **orpgdat.h** / **orpgdat.inc**. If the header file has not been modified to include this new data id, the **<Data_Buffer_Number>** must be entered rather than the **<Data_Buffer_Name>**. Whichever is used (name or number), it must be the same as the **data_id** attribute for the data store definition in the **data_attr_table** configuration file.

Notes / Rules for use

1. Since algorithms are persistent tasks, there is generally no need to close a linear buffer unless
   - the buffer is SINGLE_WRITER and other tasks also need to update the buffer or
   - the buffer was initially opened as read only and write permissions are needed.

A common use of the **RPGC_data_access_list** function is to provide a list of valid message IDs that are currently in the linear buffer.

```
int RPGC_data_access_list( int data_id, LB_info *list, int nlist )
```

PARAMETER: **data_id** The linear buffer ID

list pointer to a list of information structures **(OUTPUT)**

nlist maximum number of items in the list

RETURN VALUE: the number of item is the list or a negative error number.

Returns a list of information structures about the latest **nlist** messages in the linear buffer.

EXAMPLE: cpc005/tsk002/write_gagedb.c

Parameter Descriptions

data_id

The identity of the linear buffer. Non-product buffer IDs are defined in **orpgdat.h** / **orpgdat.inc**. If the header file has not been modified to include this new data ID, the **<Data_Buffer_Number>** must be entered rather than the **<Data_Buffer_Name>**. Whichever is used (name or number), it must be the same as the **data_id** attribute for the data store definition in the **data_attr_table** configuration file.

list (output)

A pointer to a pre-allocated array of information structures (**LB_info** defined in **lb.h**). The size of this pre-allocated array cannot be less than the **nlist** parameter. The structure contains the following fields:

LB_id_t id;	which is the internal message id
int size;	which is the size of the message
int mark;	which is the mark value of the message ?????

nlist

The maximum number of items to include in the list.

Notes / Rules for use

1.

The following functions provide the capability of reading and writing messages.

CASE 1: Message Database Buffers.

- Method 1 (Message Replace). A specific message ID is used with **RPGC_data_access_read** and **RPGC_data_access_write**. This is because the data content do not have to be similar from message to message. Each message ID is typically associated with specific data or message type content. This is one reason a *message database* linear buffer useful. The user of the data must know the meaning of the message IDs.
- Method 2 (Message Pool) A specific message ID is used with **RPGC_data_access_read** and a message ID of **LB_ANY** is used with **RPGC_data_access_write**. This creates a message pool or database of messages for random access. The messages may contain the same type of data. If the message contain different type of data, the type must be identified by information in the message itself, not the message ID. To read the message, the message ID must be obtained using **RPGC_data_access_previous_msgid** immediately after it is written to the buffer.

CASE 2: Message Queue Buffers.

It is best to NOT specify a specific message ID when writing messages to a *message queue* linear buffer. In this type buffer the data are similar in content from message to message, but in a specific sequence (for example elevation index or time). The sequence is determined solely by the order the messages are written (added). Normally a message is added sequentially by using **LB_ANY** as the message ID with **RPGC_data_access_write**. The messages can be read sequentially by using **LB_NEXT** as the message ID with **RPGC_data_access_read**. After reading messages, earlier messages can be read again by moving the read pointer with **RPGC_data_access_seek**.

```
int RPGC_data_access_read( int data_id, void *buf, int buflen,
                           LB_id_t msg_id )
```

PARAMETER:	**data_id**	The linear buffer ID
	buf	Pointer to the buffer to receive the message read (output)
	buflen	The number of bytes to read or a flag to read the entire message.

msg_id The LB message ID

RETURN VALUE: The message length on success; **0** for message not yet received (**LB_TO_COME**); or a negative number for indicating an error condition (error codes are defined in **orpgda.h**).

Reads the message **msg_id** from the linear buffer **data_id** into the allocated buffer **buf**.

EXAMPLE: cpc013/tsk008/saa_compute_products.c
cpc014/tsk014/saausers_io_functions.c

Parameter Descriptions

data_id

The identity of the linear buffer. Non-product buffer ID's are defined in **orpgdat.h** / **orpgdat.inc**. If the header file has not been modified to include this new data ID, the **<Data_Buffer_Number>** must be entered rather than the **<Data_Buffer_Name>**. Whichever is used (name or number), it must be the same as the **data_id** attribute for the data store definition in the **data_attr_table** configuration file.

buf (output)

A pointer to a buffer to hold the message read.
Option 1: If reading the entire message, space is allocated and the caller is responsible to free the buffer when no longer needed. In this case the **buf** must be type **char****.
Option 2: If reading a portion of the message, the function copies the indicated number of bytes, **buflen**, into the address provided.

buflen

Option 1: To read the entire message, a flag value **LB_ALLOC_BUF** is used. In this case the function allocates the required space in the location **buf**. Program logic must free this memory when no longer needed.
Option 2: To read just a portion of a message (useful for example to read a message header), the number of bytes to be read is specified. No memory is allocated in this case.

msg_id

The message ID is used to determine which message in the linear buffer is read.
CASE 1: For a *message database* type linear buffer, this is a user-specified message ID. This permits a user to read and write to a specific message ID in the linear buffer.
CASE 2: The *message queue* linear buffer permits retrieval of a series of message via a LB read pointer. This could be **LB_NEXT** to simply read the next message in sequence (the location of the read pointer. The function **RPGC_data_access_seek** can be used to place the read pointer at the desired location before reading.

Notes / Rules for use

1. If **RPGC_data_access_read** is called with **LB_ALLOC_BUF**, the caller is responsible for freeing memory allocated to the buffer **buf**.
2. Unlike reading product data with **RPGC_get_inbuf_by_name**, no synchronization is accomplished when reading a series of messages with **RPGC_data_access_read** from a *message queue* type buffer. The caller is responsible for obtaining the correct message.

```
int RPGC_data_access_write( int data_id, void *buf, int buflen,
                            LB_id_t msg_id )
```

PARAMETER:	**data_id**	The linear buffer ID
	buf	Pointer to the buffer containing the data to be written
	buflen	The number of bytes to be written
	msg_id	The LB message ID
RETURN VALUE:		The message length on success; **0** for a full buffer (**LB_FULL**); or a negative number for indicating an error condition (error codes are defined in **orpgda.h**).

Writes data contained in the buffer **buf** into the message **msg_id** of linear buffer **data_id**.

EXAMPLE: cpc013/tsk008/saa_compute_products.c
cpc014/tsk014/saausers_io_functions.c

Parameter Descriptions

data_id

The identity of the linear buffer. Non-product buffer ID's are defined in **orpgdat.h** / **orpgdat.inc**. If the header file has not been modified to include this new data ID, the **<Data_Buffer_Number>** must be entered rather than the **<Data_Buffer_Name>**. Whichever is used (name or number), it must be the same as the **data_id** attribute for the data store definition in the **data_attr_table** configuration file.

buf

A pointer to the buffer containing the data to be written into the linear buffer.

buflen

The number of bytes to be written into the linear buffer message.

msg_id

The message ID of the message being written.

CASE 1: For a *message database* type linear buffer, there are two methods. Which ever is chosen, must be used consistently:

- Method 1: A user-specified message ID (a non-zero positive integer). This permits a user to read and write to a specific message ID in the linear buffer and associate a type of data with a specific message ID. This is the **LB_REPLACE** behavior.
- Method 2.:With the value of **LB_ANY** , the message ID is chosen by the infrastructure. The user must obtain the ID to access the message after written. This is the **LB_MSG_POOL** behavior.

CASE 2: The *message queue* linear buffer permits storage and retrieval of a series of messages. The value of **LB_ANY** should always be used to allow the IDs to be assigned automatically (an ID that is one larger than the previous message ID).

Notes / Rules for use

1. Must have write permission.
2. The return value **LB_FULL** can be a result of several conditions. You cannot overwrite a message before reading it in an **LB_MUST_READ** type buffer. You can run out of space in **LB_NO_EXPIRE** and **LB_REPLACE** type buffers. Older messages can be removed with **RPGC_data_access_clear**.
3. Immediately after writing a message using **LB_ANY**, **RPGC_data_access_previous_msgid** can be called to obtain the message ID assigned by the infrastructure.

The function **RPGC_data_access_seek** provides a means of setting the read pointer to the desired location in a *message queue* type linear buffer.

Documentation of This Function is Not Complete

```
int RPGC_data_access_seek( int data_id, int offset, LB_id_t *msg_id )
```

PARAMETER: **data_id** The linear buffer ID

offset The offset (+/-) from the input message **msg_id**

msg_id Input: The buffer message ID from which the offset is applied.
Output: The message ID of the moved read pointer or error code.

RETURN VALUE: the size of the message on success, **0** for message expired or not yet available, or a negative error condition (error codes defined in **orpgda.h**)

Resets the location of the buffer read pointer with respect to the input **msg_id**.

EXAMPLE: *This function is not yet used in any algorithm*

Parameter Descriptions

data_id

The identity of the linear buffer. Non-product buffer ID's are defined in **orpgdat.h** / **orpgdat.inc**. If the header file has not been modified to include this new data ID, the **<Data_Buffer_Number>** must be entered rather than the **<Data_Buffer_Name>**. Whichever is used (name or number), it must be the same as the **data_id** attribute for the data store definition in the **data_attr_table** configuration file.

offset

The number of steps to move the read pointer from the input **msg_id**. A negative value moves the pointer to an earlier message and a positive value moves the pointer to a later message.

msg_id (input / output)

Input: The ID of the message from which the offset is applied. This can be a specific message ID or one of the following: **LB_FIRST**, **LB_CURRENT**, or **LB_LATEST** which is the first message in the buffer, the message at the current read pointer, or the last message in the buffer.
Output: The message ID to where the read pointer has been moved or **LB_SEEK_TO_COME** if message not in buffer or **LB_SEEK_EXPIRED** if message is expired.

Notes / Rules for use

1. WARNING: This function can place the read pointer on an EXPIRED message or a message that has not yet been received. Always check the return value.

The following function permits removal of a specified number of oldest messages. Currently there is no 'delete' function to remove any single message.

Documentation of This Function is Not Complete

```
int RPGC_data_access_clear( int data_id, int msgs )
```

PARAMETER: **data_id** The linear buffer ID

msgs Number of messages to remove

RETURN VALUE: The number of message removed or a negative error (message error codes defined in **orpgda.h**).

Removes the oldest **msgs** number of messages from the buffer. If the buffer has fewer than **msgs** messages, all messages are cleared.

EXAMPLE: *This function is not yet used in any algorithm*

Parameter Descriptions

data_id

The identity of the linear buffer. Non-product buffer ID's are defined in **orpgdat.h** / **orpgdat.inc**. If the header file has not been modified to include this new data ID, the **<Data_Buffer_Number>** must be entered rather than the **<Data_Buffer_Name>**. Whichever is used (name or number), it must be the same as the **data_id** attribute for the data store definition in the **data_attr_table** configuration file.

msgs

The number of messages to remove from the buffer.

Notes / Rules for use

1. Must have write permission (**LB_WRITE**) for the buffer.
2. This function can be used with a *message database* type buffer to remove messages that are no longer needed or desired to make room for new messages. Not needed with a *message queue* type buffer.

Responding to an Update of a Specific Buffer

Documentation of This Function is Not Complete

```
int RPGC_data_access_UN_register( int data_id, LB_id_t msg_id,
                                  void (*callback) () )
```

PARAMETER: **data_id** The linear buffer ID

msg_id The LB message ID triggering the handler function.

(*callback) () Address of a function **callback** which contains the logic to be executed when the buffer is updated.

RETURN VALUE: **LB_SUCCESS** or a negative error number (error codes defined in **orpgda.h**).

Registers the handler function for the linear buffer update for **msg_id.**

EXAMPLE: cpc005/tsk002/pcipdalg.c

Parameter Descriptions

data_id

The identity of the linear buffer. Non-product buffer ID's are defined in **orpgdat.h** / **orpgdat.inc**. If the header file has not been modified to include this new data ID, the **<Data_Buffer_Number>** must be entered rather than the **<Data_Buffer_Name>**. Whichever is used (name or number), it must be the same as the **data_id** attribute for the data store definition in the **data_attr_table** configuration file.

msg_id

The message ID who's update triggers the handler function. There are two special values of this parameter. If **LB_ANY**, the handler is executed if any message is updated. If **LB_EXPIRED**, the handler is executed if any message is cleared, expired, or deleted.

(*callback) ()

Address of a function **callback** which contains the logic to be executed when the buffer is updated. .....

Notes / Rules for use

1.

Obtaining Information About a Buffer or a Specific Message

```
LB_id_t RPGC_data_access_previous_msgid ( int data_id )
```

PARAMETER: **data_id** The linear buffer ID

RETURN VALUE: The message ID of the last read / write action on the buffer.

Obtains the message ID of the last read / write action on the buffer.

EXAMPLE: *This function is not yet used in any algorithm*

Parameter Descriptions

data_id

The identity of the linear buffer. Non-product buffer ID's are defined in **orpgdat.h** / **orpgdat.inc**. If the header file has not been modified to include this new data ID, the **<Data_Buffer_Number>** must be entered rather than the **<Data_Buffer_Name>**. Whichever is used (name or number), it must be the same as the **data_id** attribute for the data store definition in the **data_attr_table** configuration file.

Notes / Rules for use

1. This function should be called immediately after writing a message with **LB_ANY** to obtain the message ID assigned by the infrastructure.

```
int RPGC_data_access_msg_info( int data_id, LB_id_t id, LB_info *info )
```

PARAMETER: **data_id** The linear buffer ID

id The LB message ID

info address of the information structure obtained

RETURN VALUE: **LB_SUCCESS**, **LB_NOT_FOUND**, **LB_TO_COME**, or **LB_EXPIRED**

Obtains information about the linear buffer message specified.

EXAMPLE: *This function is not yet used in any algorithm*

Parameter Descriptions

data_id

The identity of the linear buffer. Non-product buffer ID's are defined in **orpgdat.h** / **orpgdat.inc**. If the header file has not been modified to include this new data ID, the **<Data_Buffer_Number>** must be entered rather than the **<Data_Buffer_Name>**. Whichever is used (name or number), it must be the same as the **data_id** attribute for the data store definition in the **data_attr_table** configuration file.

id

The message ID for the information to be obtained. This can be a specific message ID or one of the following: **LB_FIRST**, **LB_CURRENT**, or **LB_LATEST** which is the first message in the buffer, the message at the current read pointer, or the last message in the buffer.

info

The address of the structure **LB_info** containing the data obtained about the specified message. **LB_info** is defined in **lb.h** The structure contains the following fields:

LB_id_t id;	which is the internal message id
int size;	which is the size of the message
int mark;	which is the mark value of the message ?????

Notes / Rules for use

1.

The **RPGC_data_access_stat** function can obtain several kinds of status information about a linear buffer.

Documentation of This Function is Not Complete

```
int RPGC_data_access_stat( int data_id, LB_status *status )
```

PARAMETER: **data_id** The linear buffer ID

status TBD

RETURN VALUE: TBD

FUNCTION DESCRIPTION - TBD

EXAMPLE: *This function is not yet used in any algorithm*

Parameter Descriptions

data_id

The identity of the linear buffer. Non-product buffer ID's are defined in **orpgdat.h** / **orpgdat.inc**. If the header file has not been modified to include this new data ID, the **<Data_Buffer_Number>** must be entered rather than the **<Data_Buffer_Name>**. Whichever is used (name or number), it must be the same as the **data_id** attribute for the data store definition in the **data_attr_table** configuration file.

status

TBD

Notes / Rules for use

1. TBD

Database Linear Buffer Access Functions

These functions support database style access to *Public* non-product data stores which are implemented as linear buffers and configured in the **data_attr_table** configuration file. In addition, an ORPG service that manages database access must be modified.

IMPORTANT: These functions should not be used unless the data are made up of numerous records that must be searched and retrieved via specified key fields. The basic non-product data access functions described earlier are preferred because they do not require modification of an ORPG infrastructure service.

The *Public* non-product linear buffers must be configured as 'replaceable' (using **LB_DB** or **LB_REPLACE**) which creates a *message database* type buffer. These functions provide a database record search capability in addition to the ability to add, delete, and read message (records). **WARNING:** It is the user's responsibility to not exceed the number of messages configured for the data store in the **data_attr_table**.

Defining the Database

In order to use these functions a query structure (called a 'query record' in ORPG documentation) must be defined. The fields in this query structure can be used to accomplish an SQL-like search of the records (messages) in the linear buffer. **For algorithm non-product data stores, the query structure should always be the first part of the linear buffer message.** To create the query structure:

1. The query structure is defined in a header file which included by the algorithm source code.
2. The file **include/rpgdm.h** is modified to include this new file.

The RPG database manager must be modified to recognize the new database being defined for the algorithm.

1. The file **rpg_sdqs.conf** in the **cpc002/tsk006** directory is edited to define the new database. This includes the data_id of the non-product data store, the data type of the structures making up the message, and a list of the names of the structure fields which can be used in a query.
2. Then the task is recompiled by executing '**make clean**', '**make all**', then '**make install**' in the **cpc002/tsk006** directory.

A more detailed description of defining the database will be included in a future version of this document.

Defining the Query Text

The query text string uses a syntax similar to that used in the SQL "where" clause. This query text specifies the search criteria for selecting messages in the non-product data store and is the **where** parameter for the **RPGC_DB_select** and **RPGC_DB_delete** functions.

A more detailed description of defining the query text will be included in a future version of this document.

NOTE: These linear buffer access functions require that the linear buffer be managed by the ORPG by being defined in the **data_attr_table** configuration file.

Documentation of This Function is Not Complete

```
int RPGC_DB_select( int data_id, char *where, void **result )
```

PARAMETER:	**data_id**	The linear buffer ID
	where	text string describing the "where" portion of an SQL query
	***result**	address of a pointer to the result of the query (**OUTPUT**)
RETURN VALUE:		the number of records found or a negative error code

Obtains a list of messages meeting the criteria of the query defined by the **where** parameter from the non-product data store **data_id**.

EXAMPLE: cpc013/tsk012/dp_dua_accum_func.c (Build 12)

Parameter Descriptions

data_id

The identity of the linear buffer. Non-product buffer ID's are defined in **orpgdat.h** / **orpgdat.inc**. If the header file has not been modified to include this new data ID, the

<Data_Buffer_Number> must be entered rather than the **<Data_Buffer_Name>**. Whichever is used (name or number), it must be the same as the **data_id** attribute for the data store definition in the **data_attr_table** configuration file.

where
The text string describing the "where" portion of an SQL query. The data fields in the query text must be present in the query structure defined for the data store.

***result** (output)
The address of a pointer to the result of the query. The user should not attempt to decode or read the result. It's only purpose is to be used as in input parameter to other functions. If not **NULL**, ***result** must be freed by the user when no longer needed.

Notes / Rules for use (for a data driven task):

1. If not **NULL**, ***result** must be freed by the user when finished.

Documentation of This Function is Not Complete

int RPGC_DB_get_record(void *result, int ind, char **record)

PARAMETER:	**result**	pointer to the result returned by **RPGC_DB_select**
	ind	index into **result** to retrieve
	***record**	pointer to the retrieved record. (**OUTPUT**)
RETURN VALUE:		Returns the size of the record on success or a negative error code.

Retrieves one of the records (linear buffer message) specified by the index **ind** from the result of a query.

EXAMPLE: cpc013/tsk012/dp_dua_accum_func.c (Build 12)

Parameter Descriptions

result
A pointer to the result returned by **RPGC_DB_select**.

ind
The index into the list **result** to retrieve. **(is this 0-based or 1-based?)**

record (output)
A pointer to the retrieved record. This pointer (if not **NULL**) must be freed by the user when no longer needed.

Notes / Rules for use (for a data driven task):

1. If not **NULL**, ***record** must be freed by the user when finished

Documentation of This Function is Not Complete

`int RPGC_DB_get_header( void *result, int ind, char **hd )`

PARAMETER:	**`result`**	pointer to the result returned by **`RPGC_DB_select`**
	`ind`	index into **`result`** to retrieve
	`*hd`	address of the query structure of the record (**`OUTPUT`**)
RETURN VALUE:		Returns the size of the query structure on success or a negative error code.

Obtains a reference to the query structure portion of the record specified by the index **`ind`**.

EXAMPLE: *This function is not yet used in any algorithm*

Parameter Descriptions

`result`

A pointer to the result returned by **`RPGC_DB_select`**.

`ind`

The index into the list **`result`** to retrieve. **(is this 0-based or 1-based?)**

`*hd` (output)

The address of the query structure of the record. This is a reference to a memory location that must NOT be freed.

Notes / Rules for use (for a data driven task):

1. *This function is not yet used in any algorithm.*

Documentation of This Function is Not Complete

`int RPGC_DB_insert( int data_id, void *record, int rec_len )`

PARAMETER:	**`data_id`**	The linear buffer ID
	`record`	message to be inserted into the non-product linear buffer
	`rec_len`	length of the message being inserted
RETURN VALUE:		Returns the size of the record inserted or a negative error code.

Inserts a new message into the non-product linear buffer. This function will fail if the linear buffer is full.

EXAMPLE: *This function is not yet used in any algorithm*

Parameter Descriptions

`data_id`

The identity of the linear buffer. Non-product buffer ID's are defined in **`orpgdat.h`** / **`orpgdat.inc`**. If the header file has not been modified to include this new data ID, the

<Data_Buffer_Number> must be entered rather than the **<Data_Buffer_Name>**. Whichever is used (name or number), it must be the same as the **data_id** attribute for the data store definition in the **data_attr_table** configuration file.

record

The message to be inserted into the non-product linear buffer.

rec_len

The length of the message being inserted.

Notes / Rules for use (for a data driven task):

1. *This function is not yet used in any algorithm.*

Documentation of This Function is Not Complete

int RPGC_DB_delete(int data_id, char *where)

PARAMETER:	**data_id**	The linear buffer ID
	where	text string describing the "where" portion of an SQL query
RETURN VALUE:		Returns the number of records deleted on success or a negative error code.

Deletes a message from the non-product linear buffer. This function is used to removed record no longer needed.

EXAMPLE: *This function is not yet used in any algorithm*

Parameter Descriptions

data_id

The identity of the linear buffer. Non-product buffer ID's are defined in **orpgdat.h** / **orpgdat.inc**. If the header file has not been modified to include this new data ID, the **<Data_Buffer_Number>** must be entered rather than the **<Data_Buffer_Name>**. Whichever is used (name or number), it must be the same as the **data_id** attribute for the data store definition in the **data_attr_table** configuration file.

where

The text string describing the "where" portion of an SQL query. This defines the records to be deleted. The data fields in the query text must be present in the query structure defined for the data store.

Notes / Rules for use (for a data driven task):

1. *This function is not yet used in any algorithm.*

Standard Disk File Support Functions

The API provides little support for *Private* non-product data stores which are implemented as standard disk files. These files are not managed by the ORPG but controlled by the algorithm. The *Private* non-product data stores should be placed in the current ORPG working directory. The function **RPGC_get_working_dir** returns the path to the working directory that includes a trailing '/'. The function **RPGC_construct_file_name** creates the fully qualified path name using the input filename.

```
int RPGC_get_working_dir( char **path_name )
```

PARAMETER: **path_name** The address of a character string to hold path (output)

RETURN VALUE: The length of the string containing the working directory path or a non-positive error code.

Obtains the full path of the current working directory (including a trailing '/') being used by the ORPG.

EXAMPLE: *This function is not yet used in any algorithm*

Parameter Descriptions

path_name (output)
The address of the character string containing the full path of the current ORPG working directory (including the trailing '/') as determined by the function. **path_name** will be NULL upon failure.

```
int RPGC_construct_file_name( char *file_name, char **path_name )
```

PARAMETER: **file_name** The file name to be used for the data store

path_name The address of a character string to hold path (output)

RETURN VALUE: The length of the string containing the fully qualified file name or a non-positive error code.

Determines the fully qualified filename for the data store based upon the current working directory being used by the ORPG and the file name string **file_name**.

EXAMPLE: cpc013/tsk005/prcprtac_file_io.c
cpc013/tsk008/saa_file_io.c

Parameter Descriptions

file_name
The file name to be used for the private data store.

path_name (output)
The address of the character string containing the fully qualified filename for the data store as determined by the function. **path_name** will be NULL upon failure.

Notes / Rules for use

1. Caller is responsible for freeing memory allocated for **`path_name`** when no longer needed.

The following function, added in Build 10, is completely redundant with **`RPGC_construct_file_name`** and is not documented.

```
void RPGCS_full_name( char *file_name, char *full_name )
```

Part J. Miscellaneous Functions

Log Services

```
void RPGC_log_msg( int code, const char *format, ... )
```

PARAMETER:

code	The type of error message.
***format**	error message and text format specifier
...	variable number of arguments depending upon the content of the error message.

RETURN VALUE: None

Writes an error message to the task's log file. This file (named <task_name>.log) is in the **$ORPGDIR/logs** directory.

EXAMPLE:

Parameter Descriptions

code

A valid ORPG log services code. **GL_INFO** should be used for all algorithm log messages. If aborting due to a system failure (e.g., failure of **malloc** or **calloc)**, **GL_ERROR** is also acceptable. The inability to obtain in input or output buffer (product and non-product data store) is not an error and GL_INFO should be used.

format

a string representing the error message and text format specifiers for following arguments (identical to the common **printf** format specifier). See **printf** man page.

...

Additional arguments upon which the format specifiers operate. See **printf** man page.

Notes / Rules for use

1. For operational algorithms, the **required** method of logging diagnostic information in the event of algorithm failures is using **RPGC_log_msg** to write the message to the task's log file. Using **stdout** / **stderr** via **printf** and **fprintf** statements has a disadvantage of writing to an output file with no mechanism for limiting the file size and therefore should not be used.

The following function is not required for normal algorithms and is not documented.

```
int RPGC_monitor_input_buffer_load( char *dataname )
```

On rare occasions, there may be a need to completely terminate the algorithm task rather than abort the product and return to the control loop.

`void RPGC_abort_task( )`

PARAMETER: None

RETURN VALUE: Not Used

Completely terminates the algorithm task.

EXAMPLE: cpc007/tsk001/basrflct.c
cpc016/tsk002/srmrmrv_main.c

Notes / Rules for use

1. Rarely used. Another example is the **`hiresreet`** task in **`cpc014/tsk012`** uses an older version of this function: **`RPGC_hari_kiri`**.

The following function is intended for special testing and serves no purpose in an algorithm. It shuts down the RPG software.

`void RPGC_kill_rpg( )`

Vol 3. Document 2 -
The WSR-88D Algorithm API Reference

Section III Final Product Construction

NOTE

Not all of the API functions have been documented. The functions not documented are not critical to algorithm development and many of the recently added functions were created to support the porting of legacy FORTRAN algorithms to ANSI-C.

- Some of the new functions are redundant with existing functions.
- Some are not generally useful for algorithm development.
- Some of the new functions have not yet been used in algorithms.

NOTE: Some previously deprecated functions are being used to port FORTRAN algorithms.

This section documents functions that aid in the construction of final products (products distributed to external users). Helper functions are provided to assist in filling standard fields in the product headers and to write 4-bit data fields in a manner consistent with providing products in network format (Big Endian). Other functions assist in constructing selected data packets. In addition, functions are provided to support construction and debugging of the generic product data structure. See Section IV - *API Convenience Functions*) for various data conversion / transformation functions.

BUILD 12 CHANGES:

- Improved the introduction to the API support for Traditional Final Products and the API support for the Generic Data packet. More general information was added and additional references to the details contained in Volume 2 Document 3 were provided.

The contents of this section is organized as follows:

- Part A. Final Product Construction
- Part B. Support for the Generic Product Data Packet

Part A. Final Product Construction

The functions described here assist in setting fields in the initial header block portions of the product, correctly writing integer fields into the product, and support construction of several data packets.

API support for the construction of the generic product data packet is provided in Part B. of this document.

The Algorithm API has some support for the construction of ICD format final products. The assembly of the complete product is complex and is demonstrated in the sample algorithms provided in the next section.

Major Block Structure of a Final Product

There are several principles for the overall structure of WSR-88D final products.

1. One *Message Header Block* (MHB) and one *Product Description Block* (PDB) always precede the product data blocks. These two blocks are sometimes called the "product header" in the *ICD for the RPG to Class 1 User*.

2. The remaining product blocks are called optional blocks and data blocks. Product messages usually (but not always) include the *Product Symbology Block*, and may include the *Graphic Alphanumeric Block* (GAB) and the *Tabular Alphanumeric Block* (TAB).

3. Though not all products require all blocks; the blocks are always assembled in the order described above.

A detailed description of the block structure of a final product is provided in CODE Guide Volume 2, Document 3, Section I.

Structure of Symbology Block in Traditional Products

The following guidelines are not formal 'rules' contained in the Class 1 User ICD or the Product Specification ICD. However they describe how traditional products typically use data packets within the symbology block. If followed, unexpected impacts on users of new products can be minimized.

1. It is recommended that a symbology block not be empty. It should contain at least one data packet in the first layer.

2. It is recommended that a layer in a symbology block not be empty. A layer should contain at least one data packet.

3. For the two dimensional data arrays (packets AF1F, BA07, and 16) intended for graphical display

    - The 2-D data array must be in layer 1 of the symbology block.
    - Only one 2-D array should be included in any product. *There is one legacy product that has multiple 2-D arrays in an LFM grid: the Hourly Digital Precipitation Array (DPA).*

- Data packets for any additional data (whether text, vector graphics, or special symbols) should be in subsequent layers.

4. Though not explicitly stated in the *ICD for the RPG Class 1 User* (mandatory requirements):

   - All 2-byte and 4-byte data fields must begin at an even numbered byte offset from the beginning of the product message.
   - All series of 1-byte data fields must begin at an even numbered byte offset from the beginning of the product message. This implies that 1-byte data fields must be used in pairs.
   - All data length fields used in data packets containing 1-byte data fields must account for an even number of 1-byte data fields. These fields may represent integer data or character data.

A detailed description of the structure of the symbology block is provided in CODE Guide Volume 2, Document 3 Section 1.

CODE Guide Volume 2, Document 3, Section II provides a description and use of the traditional data packets within the product symbology block. Currently there is API support for the construction of several data packets.

CODE Guide Volume 2, Document 3, Section III provides a description and purpose for the generic components within the generic product structure. API support for the construction of the generic product data packet is provided in Part B. of this document.

Special Considerations

There are two factors that must be considered when constructing final product messages.

1. **Affects of the Byte-Swapping Infrastructure.**
   The data fields within a final product message cannot always be written in a straight forward manner. This is due to the byte-swapping accomplished by the ORPG infrastructure. This byte-swapping is a result of the evolution of the ORPG from a Big Endian architecture to a Little Endian architecture. This topic is covered in detail in Part B of Volume 3, Document 3, Section III.

2. **The Alignment of Data Fields within a Product Message**.
   The structure of the final product message and the individual data packets (described in Volume 2, Document 3 and the RPG to Class 1 User ICD) is only partially aligned. Because the 4-byte integers are not always offset by a number of bytes that is evenly divisible by 4 and the size of logical portions of the product is not always evenly divisible by 4, care must be used in assembly of the product message. This topic is covered in detail in Part C of Volume 3, Document 3, Section III.

Functions for Writing and Reading 4-byte Data

The WSR-88D ORPG was initially implemented on a big Endian architecture, the Sun Solaris platform. Beginning with Build 9 the Linux PC (little Endian architecture) is the new operational platform. Final

products are distributed in network format (big Endian). In order for algorithms to function correctly, data fields in final products must be written in a controlled manner. A complete guide for writing final product data fields is provided in CODE Guide Volume 3, Document 3, Section III.

The functions **RPGC_set_product_int** and **RPGC_set_product_float** must be used to correctly write 4-byte data fields to the final products. Subsequently, the functions **RPGC_get_product_int** and **RPGC_get_product_float** can be used to read the 4-byte fields.

```
int RPGC_set_product_int( void *loc, int value )
```

PARAMETER: **loc** The starting address of data field in product buffer.

value Integer value to be written in the product.

RETURN VALUE: Not Used

Writes a 4-byte integer data field into the product buffer at address "loc"

EXAMPLE:

```
RPGC_set_product_int( (void *) &hdr->msg_len,
    (unsigned int) prod_len );
```

where **hdr** is type **struct Graphic_product**

```
int RPGC_set_product_float( void *loc, float value )
```

PARAMETER: **loc** The starting address of data field in product buffer.

value Floating point value to be written in the product.

RETURN VALUE: Not Used

Writes a 4-byte integer data field (type float) into the product buffer at address "loc"

EXAMPLE: CODE Sample Algorithm 2
cpc007/tsk001/basrflct.c

Parameter Descriptions

loc

The starting address of data field in product buffer. This is the address of the first halfword or 2-byte integer (short) representing the floating point or 4-byte integer data . The offset must be an even number of bytes. If a C structure is properly aligned, it can be used to specifically the location (see Part C of Vol 3, Document 3, Section III).

value

Value to be written in the product (int or float)

Notes / Rules for use

1. **RPGC_set_product_int** and **RPGC_set_product_float** must be used for the algorithm to correctly write the data on a Linux PC platform.

`int RPGC_get_product_int( void *loc, void *value )`

PARAMETER: **`loc`** Pointer to the product data field to be retrieved

`value` Pointer to the retrieved value (int)

RETURN VALUE: Not Used

Returns in "value" an integer or unsigned integer that can be read by the current platform architecture (big or little Endian).

EXAMPLE: **`RPGC_get_product_int( (void *) &short_buf[62], &len);`**
where **`short_buf`** is an array of shorts containing the final product.
---
cpc014/tsk003/hybrprod.c
cpc018/tsk003/tdaru.c
cpc018/tsk006/buildRapidUpdateTAB.c

`int RPGC_get_product_float( void *loc, void *value )`

PARAMETER: **`loc`** Pointer to the product data field to be retrieved

`value` Pointer to the retrieved value (float)

RETURN VALUE: Not Used

Returns in "value" a float that can be read by the current platform architecture (big or little Endian).

EXAMPLE: **`RPGC_get_product_int( (void *) &short_buf[54], &caleb_const);`**
where **`short_buf`** is an array of shorts containing the final product.

This function is not yet used in any algorithm

Parameter Descriptions

`loc`
Pointer to the starting address of the product data field to be retrieved. This is the address of the first halfword or 2-byte integer (short) representing the floating point or 4-byte integer data .

`value` (output)
Value to be retrieved (int or float)

Notes / Rules for use

1. **`RPGC_set_product_int`** and **`RPGC_set_product_float`** must be used for the algorithm to correctly write the data on a Linux PC platform.

MISC Product Construction Helper Functions

`float RPGC_NINT( float real )`

PARAMETER: **real** floating point number to be rounded

RETURN VALUE: The closest integer value.

The function returns the closest integer value to the input IEEE floating point number.

EXAMPLE: cpc007/tsk001/basrflct.c

```
double RPGC_NINTD( double real )
```

PARAMETER: **real** floating point number to be rounded

RETURN VALUE: The closest integer value.

The function returns the closest integer value to the input IEEE floating point number.

EXAMPLE: *This function is not yet used in any weather algorithm*

Parameter Descriptions

real

IEEE floating point number to be rounded.

Notes / Rules for use

1.

This function obtains the product id from the product name.

```
int RPGC_get_id_from_name( char *data_name )
```

PARAMETER: **data_name** The product name

RETURN VALUE: The product ID. If not found, returns -1

EXAMPLE: cpc007/tsk001/basrflct_main.c
cpc007/tsk013/bref8bit_main.c

Parameter Descriptions

data_name

The name of the product. This name must be in the list contained in either the **input_data** attribute or the **output_data** attribute of the **task_attr_table** entry for the algorithm task (this entry determines the number or ID of the linear buffer containing the data).

Notes / Rules for use

1. LIMITATION: This function only works if the **data_name** is one of the registered inputs or registered outputs.

The following functions obtain the product code (the external user id for the product) from the product id and product name.

```
int RPGC_get_code_from_id( int prod_id )
```

PARAMETER: **prod_id** The product ID

RETURN VALUE: The product code, or a negative number on error.

The product code returned is a positive number if prod_id refers to the final product and is 0 if data_name refers to an intermediate product.

EXAMPLE: cpc007/tsk015/superes8bit_main.c

Parameter Descriptions

prod_id

The product ID. This ID is determined by contents of the corresponding **input_data** / **output_data** attribute in the **task_attr_table** entry for the task.

Notes / Rules for use

1.

```
int RPGC_get_code_from_name( char *data_name )
```

PARAMETER: **data_name** The product name

RETURN VALUE: The product code, or a negative number on error.

The product code returned is a positive number if data_name refers to the final product and is 0 if data_name refers to an intermediate product.

EXAMPLE: cpc014/tsk003/hybrprod.c
cpc016/tsk002/srmrmrv_main.c

Parameter Descriptions

data_name

The name of the product. This name must be in the list contained in the **output_data** attribute of the **task_attr_table** entry for the algorithm task (the **output_data** entry determines the number or ID of the linear buffer containing the data).

Notes / Rules for use

1. LIMITATION: This function only works if the **data_name** is one of the registered outputs.

The following functions which compress and decompress products are not generally needed by algorithms. They may be useful in debugging products.

```
int RPGC_compress_product( void *bufptr, int method )

void* RPGC_decompress_product( void *bufptr )
```

Data Packet Specific Helper Functions

Supporting Packet 16 Header and Array

```
int RPGC_digital_radial_data_hdr( int first_bin_idx, int num_bins,
                                  int icenter, int jcenter,
                                  int scale_factor, int num_radials,
                                  void *output )
```

PARAMETER:	**first_bin_idx**	index of the first range bin
	num_bins	number of range bins
	icenter	icenter of sweep
	jcenter	jcenter of sweep
	scale_factor	range scale factor
	num_radials	number of radials in sweep
	output	location of the completed header
RETURN VALUE:		Currently always returns 0.

Populates the header portion (first 7 shorts) of data packet 16.

EXAMPLE: cpc007/tsk013/bref8bit.c
cpc007/tsk014/bvel8bit.c

Parameter Descriptions

first_bin_idx

The index of the first range bin. This should be `0` to indicate that the first data item in the array is valid.

num_bins

The number of range bins in the data array. In WSR-88D products the number of bins reflects the 70,000 ft MSL cutoff.

icenter

The icenter of sweep is always 0. Used in the Legacy display device.

jcenter

The jcenter of sweep is always 0. Used in the Legacy display device.

scale_factor

The range scale factor. Regardless of how this field is described in the Class 1 ICD, existing products have not followed a consistent definition. The best choice is to enter the cosine of the target elevation angle times 1000. For 0.5 degrees this is **999**.

num_radials

The number of radials in this product.

output

A pointer to the beginning address in which the header fields are to be populated

Notes / Rules for use

1. Must include packet_16.h.

There are two functions can be used in filling out each radial in the digital radial data array packet (data packet 16).

RPGP_set_packet_16_radial

is limited to an input array of 1-byte data. In addition, the input and output data arrays must always start and end at the same bin number and the bin step is assumed to be 1. Must also include **rpgp.h** in order to use this function. **NOTE: the return value of this function was inadvertently changed in Build 10.**

RPGC_digital_radial_data_array

is more flexible. The most useful option is the ability to accept input arrays of 1-byte, 2-byte, and 4-byte data. Other options include specifying the index of the starting bin for both the input and output data array. One advantage is that if the input data is smaller than the output data array, the output array is padded with the value 0.

```
int RPGP_set_packet_16_radial( unsigned char *packet, short start_angle,
                               short angle_delta, unsigned char *data,
                               int num_values )
```

PARAMETER:	**packet**	pointer to the buffer to hold the radial data
	start_angle	radial start angle (deg*10)
	angle_delta	radial angle delta (deg*10)
	data	pointer to the radial data array values
	num_values	number of items in radial data
RETURN VALUE:		number of bytes in this radial. **In Build 9 this included the 6 byte radial header. The build 10 version of this function does not include the 6 byte radial header.** Includes a padded byte at the end if required.

Constructs a single radial portion of data packet 16. Function returns number of bytes in this radial (does NOT include the header portion of the radial). The radial header fields: number of bytes in the radial (including pad), radial start angle, and radial delta angle are filled along with the radial data array values. As required for the ORPG byte-swapping infrastructure, the function writes all data as shorts, including the 1-byte data fields. The function also pads a 0 if there is an odd number of data fields.

EXAMPLE: CODE Sample Algorithm 1
cpc007/tsk006/itwsdbv.c

Parameter Descriptions

`packet`

A pointer to the buffer to hold the radial data. Within the buffer into which the product is being built, this is the location of the beginning of this packet radial header (number of bytes, start angle, and delta angle) followed by the data array. This structure can be represented by the structure **`Packet_16_data_t`** defined in **`packet_16.h`**.

`start_angle`

Radial start angle in units (deg*10). For example: 1262 represents 126.2 degrees. Can be obtained from **`Base_data_header.start_angle`**.

`angle_delta`

Radial angle delta in units (deg*10). For example: 10 represents 1.0 degrees. Can be obtained from **`Base_data_header.delta_angle`**.

`data`

A pointer to the beginning of the radial data array, the first data value in the radial. This must be an array of 8-bit data (**`unsigned char`**).

`num_values`

The number of items (data values) in radial data array.

Notes / Rules for use

1. Must include **`rpgp.h`** to use the **`RPGP_set_packet_16_radial`** function.
2. The input and output data arrays must always start and end at the same bin number and the bin step is assumed to be 1.

```
int RPGC_digital_radial_data_array( void *input, int input_size,
                                    int start_data_idx, int end_data_idx,
                                    int start_idx, int num_rad_bins,
                                    int binstep, int start_angle,
                                    int delta_angle, void *output )
```

PARAMETER:

`input`	pointer to the input radial data array values
`input_size`	number of bits representing an individual input data value
`start_data_idx`	starting index in the **`input`** array
`end_data_idx`	ending index in the **`input`** array
`start_idx`	starting index in the data array portion of packet 16

num_rad_bins	number of data values in the radial array
binstep	bin step size
start_angle	radial start angle (deg*10)
delta_angle	radial angle delta (deg*10)
output	pointer to the radial portion of packet 16 (**OUTPUT**)

RETURN VALUE: number of bytes in this radial (this includes the header portion of the radial). Includes a padded byte at the end if required.

Constructs a single radial portion of data packet 16. The radial header fields: number of bytes in the radial (including pad), radial start angle, and radial delta angle are filled along with the radial data array values. As required for the ORPG byte-swapping infrastructure, the function writes all data as shorts, including the 1-byte data fields. The function also pads a 0 if there is an odd number of data fields.

EXAMPLE: cpc007/tsk013/bref8bit.c
cpc007/tsk014/bvel8bit.c
cpc014/tsk003/hybrprod.c

Parameter Descriptions

input

A pointer to the input radial data array values. The type of input data array should correspond to the value set for **input_size**.

input_size

The number of bits representing an individual input data value. Either **RPGC_BYTE_DATA** (8), **RPGC_SHORT_DATA** (16), and **RPGC_INT_DATA** (32) are defined in **rpgc.h**..

start_data_idx

The starting index in the **input** array. If the first item at the address provided by **input** is the first data to be processed, this is "**0**".

end_data_idx

The index in the **input** array containing the last data value to place in the output array.

start_idx

The starting index in the data array portion of packet 16 For normal radial products this is always "**0**". If a larger number is used, then output array values before the **start_idx** are set to **0**.

num_rad_bins

The number of data values in the output radial array. If this exceeds the last input data value to be processed, **input[end_data_idx]**, then the corresponding output array value is set to **0**.

binstep

The bin step size. Currently this must always be "**1**".

start_angle

Radial start angle in units (deg*10). For example: 1262 represents 126.2 degrees. Can be obtained from **`Base_data_header.start_angle`**.

`delta_angle`

Radial angle delta in units (deg*10). For example: 10 represents 1.0 degrees. Can be obtained from **`Base_data_header.delta_angle`**.

`output` (output)

A pointer to the location in which a radial portion of packet 16 is to be built. This is that portion repeated for each radial that contains a radial header (number of bytes, start angle, and delta angle) followed by the data array. This structure can be represented by the structure **`Packet_16_data_t`** defined in **`packet_16.h`**.

Notes / Rules for use

1. This function is preferred over the previous RPGP_set_packet_16_radial.
2. Using a value for **`binstep`** other than **1** has not been tested.
3. Must include packet_16.h.

Supporting Packet 17 Header and Data Array

The following functions are going to be used to port the DPA product from Fortran to C.

Applicability limited to a specific product.
Will be documented in a future edition of this guide.

```
int RPGC_digital_precipitation_data_hdr( int lfm_boxes_in_row, int num_rows,
                                         void *output )
```

PARAMETER:	**`lfm_boxes_in_row`**	short_desc
	`num_rows`	short_desc
	`output`	short_desc
RETURN VALUE:		retval_descript

Fills the header portion of data packet 17 in the DPA product.

EXAMPLE:

Parameter Descriptions

`lfm_boxes_in_row`

long_desccription

num_rows
long_desccription

output
long_desccription

Notes / Rules for use

1. *This function is not yet used in any algorithm.*

Applicability limited to a specific product.
Will be documented in a future edition of this guide

```
int RPGC_digital_precipitation_data_array( void *input, int input_size,
                                           int lfm_boxes_in_row, int num_rows,
                                           void *output )
```

PARAMETER:	**input**	short_desc
	input_size	short_desc
	lfm_boxes_in_row	short_desc
	num_rows	short_desc
	output	short_desc
RETURN VALUE:		retval_descript

Enters data values into the data array portion of data packet 17 in the DPA product.

EXAMPLE:

Parameter Descriptions

input
long_desccription

input_size
long_desccription

lfm_boxes_in_row
long_desccription

num_rows
long_desccription

output
long_desccription

Notes / Rules for use

1. *This function is not yet used in any algorithm.*

Supporting AF1F and BA07 RLE Headers and Data Arrays

Currently there are no helper functions for the header portion of data packets **AF1F** and **BA07**.

AF1F: The guidance provided for setting parameters for **RPGC_digital_radial_data_hdr** can be used in entering values in the header portion of packets **AF1F**.

BA07: Guidance to be provided in a future version of this document.

There are two functions which provide a standard means to run-length data for the **AF1F** radial data packet. The only difference between the functions is the type of input data array to be encoded.

RPGC_run_length_encode
uses an input array of 2-byte integers (as found in the radial basedata messages).

RPGC_run_length_encode_byte
uses an input array of 1-byte integers (as found in the compact radial in the elevation basedata messages).

```
int RPGC_run_length_encode( int start, int delta, short *inbuf,
                            int startix, int endix, int max_num_bins,
                            int buffstep, short *cltab, int *nrleb,
                            int buffind, short *outbuf )
```

PARAMETER:	**start**	radial start angle (deg*10)
	delta	radial delta angle (deg*10)
	inbuf	input data to be run-length encoded
	startix	index into **inbuf** to start of data to run-length encode
	endix	index into **inbuf** to end of data to run-length encode
	max_num_bins	maximum number of data bins
	buffstep	number of 2-byte words per entry in **inbuf**
	cltab	pointer to color table (for data level translation)
	***nrleb**	number of run-length encoded bytes (**OUTPUT**)
	buffind	index into **outbuf** for beginning of rle data
	outbuf	pointer to output buffer
RETURN VALUE:		Currently always returns 0

From an input data array of 256-level data, creates an output data array of 16-level data (or 8 or 4 level) that has been run-length encoded in a manner consistent with the AF1F data packet.

EXAMPLE: See CODE sample algorithm 2.
cpc007/tsk002/basvlcty.c
cpc014/tsk003/hybrprod.c

```
int RPGC_run_length_encode_byte( int start, int delta, unsigned char *inbuf,
                                 int startix, int endix, int max_num_bins,
                                 int buffstep, short *cltab, int *nrleb,
                                 int buffind, short *outbuf )
```

PARAMETER:	**start**	radial start angle (deg*10)
	delta	radial delta angle (deg*10)
	inbuf	input data to be run-length encoded
	startix	index into **inbuf** to start of data to run-length encode
	endix	index into **inbuf** to end of data to run-length encode
	max_num_bins	maximum number of data bins
	buffstep	number of 2-byte words per entry in **inbuf**
	cltab	pointer to color table (for data level translation)
	***nrleb**	number of run-length encoded bytes (**OUTPUT**)
	buffind	index into **outbuf** for beginning of rle data
	outbuf	pointer to output buffer
RETURN VALUE:		retval_descript

From an input data array of 256-level data, creates an output data array of 16-level data (or 8 or 4 level) that has been run-length encoded in a manner consistent with the AF1F data packet.

EXAMPLE: *This function is not yet used in any algorithm*

Parameter Descriptions

start

Radial start angle (deg*10) For example: 1262 represents 126.2 degrees. Can be obtained from **Base_data_header.start_angle** if reading base data.

delta

Radial delta angle (deg*10) For example: 10 represents 1.0 degrees. Can be obtained from **Base_data_header.delta_angle** if reading base data.

inbuf

Pointer to the input data to be run-length encoded. This is the beginning of the data array. This is an array of 2-byte integers for the **RPGC_run_length_encode** function and an array of 1-byte integers for the **RPGC_run_length_encode_byte** function.

startix

The index into **inbuf** to start of data to run-length encode. The first "good" data bin. This is a 0-based index, that is the first position in the array is 0. Set to **(Base_data_header.surv_range - 1)** if reading directly from base data.

endix

The index into **inbuf** to end of data to run-length encode. The last "good" data bin. This index is based upon 0 being the first position in the array. For input base data this never exceeds 70,000 ft MSL. Set to **(Base_data_header.n_surv_bins - 1)** if reading directly from base data. **WARNING:** At lower elevations the following correction must be made if the range of the product is less than the range of the base data. The value of this parameter must be reduced as required so that (**endix - startix +1**) does not exceed the **max_num_bins** parameter. *This correction may be placed inside future versions of this function.*

max_num_bins or

The maximum number of data bins. This corresponds to the maximum range of the product (at lower elevation angles). For example, many products using reflectivity data only produce the product to 230 km even though data to 460 is available.

buffstep

The size of the input buffer data elements in short integers (2-byte). This is usually "1". If set to "2", every other data value in the input data array is read. If set to "4", every fourth entry in the input data array is read.

cltab

The pointer to color table (for data level translation). This table converts the 256-level input data into one of 16 (or 8 or 4) data levels to be run-length encoded. For legacy products this conversion is defined in the Product Specification ICD.

***nrleb** (output)

The number of run-length encoded bytes (function **OUTPUT**). The size in bytes of the output run-length encoded data array. This includes the 6-byte radial header in addition to the number of bytes in the run-length encoded array. Always an even number.

buffind

The index into **outbuf** for beginning of the run-length encoded data. This index is in 2-byte integers (halfwords).

- For the first radial: If **outbuf** is a pointer to the beginning of a geographic final product and the RLE data is in an AF1F data packet in the first layer, **buffind** = 75. If **outbuf** only contains the RLE data array, **buffind** = 0.
- For subsequent radials. The value of **buffind** must be increased by the size of the previous radial data which is the output: **buffind = buffind + (*nrleb / 2)**

outbuf

The pointer to buffer to contain the run-length encoded data. The previous parameter is used to place the run-length encoded data within this buffer.

Notes / Rules for use

1. **RPGC_run_length_encode** can be used with data from the radial basedata message and the function **RPGC_run_length_encode_byte** can be used with data from the elevation basedata message.
2. At higher elevation angles, where the maximum number of bins is adjusted for 70,000 ft MSL, the size of the resulting product may vary from previous implementations of the run-length encoding function.

The following function provides a standard means to run-length data for the **BA07** raster data packet.

This function is not completely documented.

```
int RPGC_raster_run_length( int nrows, int ncols, short *inbuf, short *cltab,
                            int maxind, short *outbuf, int obuffind,
                            int *istar2s )
```

PARAMETER:	**nrows**	Number of rows in data grid
	ncols	Number of columns in data grid
	inbuf	Input buffer address - used to reference input data grid
	cltab	Product data levels adaptation data for the product
	maxind	Maximum output buffer index that this module can store into
	outbuf	output buffer address used to specify output buffer location to store a byte
	obuffind	output buffer starting index
	***istar2s**	number of 2-byte words stored in the product output buffer by this module (**OUTPUT**)
RETURN VALUE:		0 - complete, 1 incomplete

From an input data array of 256-level data, creates an output data array of 16-level data (or 8 or 4 level) that has been run-length encoded in a manner consistent with the BA07 data packet.

EXAMPLE: cpc008/tsk022/radcdmsg.c

Parameter Descriptions

nrows
Number of rows in data grid

ncols
Number of columns in data grid

inbuf
Input buffer address - used to reference input data grid

cltab
Product data levels adaptation data for the product

maxind
Maximum output buffer index that this module can store into

outbuf
output buffer address used to specify output buffer location to store a byte

obuffind
output buffer starting index

***istar2s** (output)
The number of 2-byte words stored in the product output buffer by this module (function **OUTPUT**).

Notes / Rules for use

1. This function cannot be used for the elevation base data message, only the radial message.
2. *Has not been tested in an algorithm.*

Functions used to Assemble Header Blocks

The following subroutines construct the two blocks always present in any ICD graphic product: **RPGC_prod_desc_block** supports the *Product Description Block* and **RPGC_prod_hdr** supports the *Message Header Block.*

There are two functions supporting several data fields in the *Product Description Block* that are not filled in by **RPGC_prod_desc_block**.

RPGC_set_dep_params
RPGC_set_prod_block_offsets

There is no support for the construction of the *Symbology Block* layers.

The function **RPGC_stand_alone_prod** is used only for a special stand alone alphanumeric product and is **not recommended for new algorithms**.

```
int RPGC_prod_desc_block( void *ptr, int prod_id, int vol_num )
```

PARAMETER:	**ptr**	pointer to the beginning of the final product
	prod_id	Linear buffer id of the output product.
	vol_num	A sequential volume number.
RETURN VALUE:		Not Used

Places appropriate data into the *Product Description Block* portion of the ICD graphic product. Those portions of the block representing the 16 threshold levels, the 10 product dependent parameters, and block offsets are not set by this function, and must be set manually at a later time. The block offsets must be set prior to calling **RPGC_prod_hdr**.

EXAMPLE:

Parameter Descriptions

ptr

A pointer to the beginning of the final product. Normally this is the product buffer that is returned by the corresponding **RPGC_get_outbuf_by_name** call.

prod_id

Linear buffer id of the corresponding output product. This is the same value of the output product registered.

vol_num

A sequential volume number. The current volume sequence number is obtained via the **RPGC_get_buffer_vol_num** function.

```
int RPGC_set_dep_params( void *ptr, short *params )
```

PARAMETER: **ptr** pointer to the beginning of the final product

params Pointer to an array of product dependent parameters.

RETURN VALUE: Not Used

Sets the 10 product dependent parameters within the *Product Description Block* by direct mapping from an array of 10 elements (short integers).

EXAMPLE: CODE Sample Algorithm 2
cpc007/tsk001/basrflct.c
cpc018/tsk001/mdaprodMain.c

Parameter Descriptions

ptr

A pointer to the beginning of the final product. Normally this is the product buffer that is returned by the corresponding **RPGC_get_outbuf_by_name** call.

params

Pointer to an array of product dependent parameters (10 short integers).

```
int RPGC_set_prod_block_offsets( void *ptr, int sym_offset,
                                 int gra_offset, int tab_offset )
```

PARAMETER: **ptr** pointer to the beginning of the final product

sym_offset offset to the symbology block (in 2-byte integers)

gra_offset offset to the GAB (in 2-byte integers)

tab_offset offset to the TAB (in 2-byte integers)

RETURN VALUE: retval_descript
The block offsets must be set prior to calling **RPGC_prod_hdr**.

Description paragraph

EXAMPLE: cpc007/tsk014/bvel8bit.c
cpc014/tsk003/hybrprod.c

Parameter Descriptions

ptr

A pointer to the beginning of the final product. Normally this is the product buffer that is returned by the corresponding **RPGC_get_outbuf_by_name** call.

sym_offset

The offset from the beginning of the product to the symbology block (in 2-byte integers).

gra_offset

The offset from the beginning of the product to the GAB (in 2-byte integers).

tab_offset

The offset from the beginning of the product to the TAB (in 2-byte integers).

```
int RPGC_prod_hdr( void *ptr, int prod_id, int *length )
```

PARAMETER: **ptr** pointer to the beginning of the final product

prod_id Linear buffer id of the output product.

length Pointer to the current length of the product, in bytes.

RETURN VALUE: Returns **0** upon success.

Places appropriate data into the *Message Header Block* portion of the ICD graphic product. The block offsets must be entered in the *Product Description Block* before calling this function.

EXAMPLE:

Parameter Descriptions

ptr

A pointer to the beginning of the final product. Normally this is the product buffer that is returned by the corresponding **RPGC_get_outbuf_by_name** call.

prod_id

Linear buffer id of the corresponding output product. This is the same value of the output product registered.

length

The length of the product (*length) used as an input parameter to **RPGC_prod_hdr** is the total length of the assembled product (in bytes) minus the size of the *Message Header Block* and the *Product Description Block* (minus 120 bytes). In other words, the length passed as input is the total length of all optional parts of the product (the *Symbology Block*, the *GAB*, and the *TAB*). When **RPGC_prod_hdr** returns, the product length has been increased by 120 bytes to reflect the total product size.

Notes / Rules for use

1. The programmer is responsible for assembling the *symbology block* (and any other blocks), calculating their length and setting the block offsets in the *Product Description Block* prior to calling **RPGC_prod_hdr**
2. The following product construction functions must be used in the following order: **RPGC_prod_desc_block**, **RPGC_set_dep_params**, **RPGC_prod_hdr**.

Part B. Support for the Generic Product Data Packet

Initial support for the construction of the generic product data packet (packet 28) was added in Build 8. **`rpgp.h`** must be included to use these functions. All generic product structures are defined in **`orpg_product.h`** which is included with **`rpgp.h`**. **`orpg_product.h`** also contains enumerations for product types, component types, grid types, and area types.

The DMD product in cpc018/tsk001 creates a generic product using area components. The DMD product uses numerous component parameters to convey most of the information.

A detailed description of the block structure of a final product is provided in CODE Guide Volume 2, Document 3, Section I.

Structure of Symbology Block in Generic Products

The following guidelines are not formal 'rules' contained in the Class 1 User ICD or the Product Specification ICD. However they describe how to use generic product components within the symbology block with the goal of minimizing unexpected impacts on users and simplifying the decoding logic required to interpret the products while not overly constraining the use of these components.

1. It is recommended that a symbology block not be empty.

2. The symbology block must contain only one data packet 28 in the first layer.

3. It is recommended that the other optional blocks, the Graphical Alphanumeric Block (GAB) and Tabular Alphanumeric Block (TAB) NOT be used.

4. For the two dimensional data arrays (the generic radial component and the generic grid component) intended for graphical display

    - The 2-D data array component should be the first component in the product.
    - Only one 2-D array component should be included in any generic product.
    - Generic components for any additional data should follow.

5. For the header portion of data packet 28, all 2-byte and 4-byte data fields must begin at an even numbered byte offset from the beginning of the product message.

NOTE: The restrictions on alignment of 2-byte and 4-byte data fields and the requirement to use 1-byte data fields in pairs does NOT apply to the data portion of packet 28. Very specific C structures are used to assemble this data. Because the data is serialized by the API function provided, the structure of the data portion of the actual message cannot be diagramed graphically as the other portions of the final product.

An introduction to the generic data packet and component types used in a generic product is provided in CODE Guide Volume 2, Document 3, Section III.

Functions Used For Product Construction

The functions **RPGP_build_RPGP_product_t** and **RPGP_finish_RPGP_product_t** are used to set the header fields in the generic product structure **RPGP_product_t**. The functions are called in that order. The list of product parameters and the array of components must be constructed before calling **RPGP_finish_RPGP_product_t**. **Using these functions ensures that contents of the generic product structure header fields are correct.**

```
int RPGP_build_RPGP_product_t( int prod_id, int vol_num, char *name,
                               char *description, RPGP_product_t *prod )
```

PARAMETER:	**prod_id**	Product ID
	vol_num	Volume Scan Number (1-80)
	name	Product mnemonic
	description	Product Description
	prod	Pointer to the Generic Product Structure
RETURN VALUE:		Negative value on error, 0 on success.

Sets the value of most of the generic product structure, **RPGP_product_t**, header fields. Much of this information is redundant with contents of the product description block.

EXAMPLE:

```
RPGP_build_RPGP_product_t( STATPROD, vol_num, "ASP",
"Archive III Status Product", xdrbuf );
```

where **STATPROD** is defined in **a309.h**, **vol_num** is the return value of **RPGC_get_buffer_vol_num**, and **xdrbuf** is a pointer to the product structure.

Parameter Descriptions

prod_id

Product ID as configured in the **product_attr_table**. configuration file. If **a309.h**/**a309.inc** have been modified to define to define the Buffer ID's, the **<Prod_Buffer_Name>** can be used rather than the **<Prod_Buffer_Number>**.

vol_num

Volume Scan Number (1-80). The return value of **RPGC_get_buffer_vol_num**.

name

Product mnemonic. By convention, the same as (for final products) the initial 1-3 characters prior to the first space in the **desc** attribute in the **product_attr_table** configuration file.

description

Product Description. By convention, the same as the text in the **desc** attribute in the **product_attr_table** configuration file after the 1-3 character mnemonic.

prod

Pointer to the preallocated Generic Product Structure **RPGP_product_t**.

Notes / Rules for use

1. This function is called before **RPGP_finish_RPGP_product_t**.
2. HINT: Though not required, it is suggested that the list of product parameters and the array of product components be completed prior to calling this function. This function sets the generation time field. Calling this function as late as possible will result in a generation time as close as possible to the time recorded in the product description block.

```
int RPGP_finish_RPGP_product_t( RPGP_product_t *prod, int numof_prod_params,
                                RPGP_parameter_t *prod_params,
                                int numof_components, void **components )
```

PARAMETER:	**prod**	Pointer to the Generic Product Structure
	numof_prod_params	Number of product parameters in the product structure.
	prod_params	Pointer to the parameter list.
	numof_components	Number of components in the product.
	components	Address of the component array.
RETURN VALUE:		Negative value on error, 0 on success.

Finishes populating the generic product structure with the number of parameters, the product parameters, the number of components and the array of components.

EXAMPLE: **RPGP_finish_RPGP_product_t(xdrbuf, 0, NULL, num_messages, (void **) text_packet);**

where **xdrbuf** is a pointer to the product structure, 0 is the number of product parameters, **NULL** is value of the pointer to parameters, **num_messages** is the number of components, and **text_packet** is the address of an array of component pointers (**RPGP_text_t **text_packet** in this example).

Parameter Descriptions

prod

Pointer to the preallocated Generic Product Structure **RPGP_product_t**.

numof_prod_params

Number of product parameters in the product structure.

prod_params

Pointer to the parameter list. This is an array of **RPGP_parameter_t** structures.

numof_components

Number of components in the product. This is the size of the **void **components** array.

components

Address of the component array. This is actually a pointer to an array of **void *** or type **void **.** This can support an array of component structures of different types. Currently all algorithm examples use components of the same type.

Notes / Rules for use

1. This function is called after **RPGP_build_RPGP_product_t** and after creating the product parameter list and the array of product components.

Both the generic product structure and each component structure can have a list of parameters. Each parameter is represented by the structure **RPGP_parameter_t** which consists of an ID and a list of attributes (both type **char ***). Attributes include "name", "type", "unit", "range", "value", "default", and "description". The following functions assist in population of the parameter structure.

RPGP_set_int_param
: Used for parameters having one of the integer types for "type".

RPGP_set_float_param
: Used for parameters having one of the floating point types for "type".

RPGP_set_string_param
: Used for parameters having "type" of string.

```
int RPGP_set_int_param( RPGP_parameter_t *param, char *id, char *name,
                        int type, void *value, int size, int scale, ... )
```

PARAMETER:

param	Pointer to the parameter.
id	Parameter ID
name	Parameter Name
type	Type of integer
value	Value of parameter.
size	Number of values (size of **value** array).
scale	Integer value scale factor.
...	Optional argument; **0** indicates last (ignored) argument.

RETURN VALUE: Not Used

Creates an integer parameter with attributes based upon function arguments.

EXAMPLE:

```
RPGP_set_int_param( param, id, name, RPGP_TYPE_INT, value,
size, scale, RPGP_ATTR_UNITS, units, 0 );
```

where **value** is type **int ***, setting the value of the integer array element [**size**], the scale factor is **scale**, and an additional attribute "units" is being set.

---

cpc018/tsk001/buildDMD_PSB.c

```
int RPGP_set_float_param( RPGP_parameter_t *param, char *id, char *name,
                          int type, void *value, int size, const int fld_width,
                          const int precision, const double scale, ... )
```

PARAMETER:

param	Pointer to the parameter.
id	Parameter ID

name	Parameter Name
type	Type of floating point
value	Value of parameter.
size	Number of values (size of **value** array).
fld_width	Maximum number of characters to represent the **value**.
precision	Number of characters to the right of the decimal point.
scale	Floating point value scale factor.
...	Optional argument; **0** indicates last (ignored) argument.

RETURN VALUE: Not Used

Creates a floating point parameter with attributes based upon function arguments.

EXAMPLE:

```
RPGP_set_float_param( param, id, name, RPGP_TYPE_FLOAT,
(void *) &value, 1, fld_width, precision, 1.0, RPGP_ATTR_UNITS,
units, 0 );
```

where **value** is type **const float**, this is a single value parameter (**size = 1**), the scale factor is **1.0**, and an additional attribute "units" is being set.

---

cpc018/tsk001/buildDMD_PSB.c

```
int RPGP_set_string_param( RPGP_parameter_t *param, char *id, char *name,
                           void *value, int size, ... )
```

PARAMETER:	**param**	Pointer to the parameter.
	id	Parameter ID
	name	Parameter Name
	value	Value of parameter.
	size	Number of values (size of **value** array).
	...	Optional argument; **0** indicates last (ignored) argument.

RETURN VALUE: Not Used

Creates a string parameter with attributes based upon function arguments.

EXAMPLE:

```
RPGP_set_string_param(
     text_packet[ num_nodes ]->comp_params, "Msg Type",
     "", "RDA ALARM CLEARED", 1, 0 );
```

where we are setting a parameter in the text component **text_packet[num_nodes]**, with an **id** of **"Type"**, a blank **name**, and a single **value** of **"RDA ALARM CLEARED"**.

---

cpc018/tsk001/buildDMD_PSB.c

Parameter Descriptions

param

Pointer to the parameter structure **RPGP_parameter_t**.

id

Parameter ID. A unique single token identifier. The ID is intended for machine processing rather than display to a human user. Each parameter ID in a product should have a single consistent meaning throughout the product (think of an ID as a type or class). In addition for the parameter ID's to be useful: (1) the product parameter ID's should be unique, and (2) a component's parameter ID's should be unique. If multiple components within a product represent the same thing then these components use the same component parameter ID's with their purpose and meaning remaining consistent. Components within a product that are either (a) different types of components or are (b) the same component type but represent different kinds of objects, should not share the same component parameter ID's.

name

Parameter Name. A short description (several words) for the parameter which could be displayed as a title or label. The optional "description" attribute on the other hand is intended to provide additional information to the user.

type

For integer parameters: **RPGP_TYPE_INT**, **RPGP_TYPE_UINT**, **RPGP_TYPE_SHORT**, **RPGP_TYPE_USHORT**, **RPGP_TYPE_BYTE**, **RPGP_TYPE_UBYTE**, and **RPGP_TYPE_BIT** are defined in **rpgp.h**. For floating point parameters: **RPGP_TYPE_FLOAT** and **RPGP_TYPE_DOUBLE** are defined in **rpgp.h**.

value

Value of the parameter. This is an integer / floating point / string value as appropriate.

size

Number of values (size of the **value** array). Normally **1** for a single value parameter.

fld_width

Maximum number of characters to represent the **value** for floating point parameters. Use **0** for unlimited. This is analogous to the formatting of **printf** statements to determine how many digits to use in the character representation.

precision

Number of characters to the right of the decimal point for floating point parameters. This is analogous to the formatting of **printf** statements to determine the number of decimal places of precision.

scale

Scale factor used for integer and floating point parameter values. Determines location of decimal point. **1.0** means the decimal is positioned correctly in **value**.

...

Optional Argument. The last argument is always represented by **0**. Additional arguments are used to specify additional attributes for the parameter. Additional arguments are specified using tuples <**attr**, **string_val**> where the supported **attr** are defined in **rpgp_prod_support.h** and **string_val** are strings. The defined **attr** include: **RPGP_ATTR_UNITS** for "units", **RPGP_ATTR_DESCRIPTION** for "description", **RPGP_ATTR_RANGE** for "range", and **RPGP_ATTR_DEFAULT** for "default". The format of the **string_val** depends on the item.

Notes / Rules for use

1. Parameter construction is accomplished before **RPGP_finish_RPGP_product_t** is called.

The function **RPGP_product_serialize** is used to serialize the serial data portion of data packet 28 prior to releasing the product buffer (writing to the product database). Once the product output is complete, generic product memory allocation is released using **RPGP_product_free**.

```
int RPGP_product_serialize( RPGP_product_t *prod, char **serial_data )
```

PARAMETER:	**prod**	Pointer to the Generic Product Structure
	serial_data	Pointer to the resulting serialized data.
RETURN VALUE:		The number of bytes of the serialized data on success or **-1** on failure.

Takes the fully populated generic product and creates a serialized data block. The returned pointer must be freed if not NULL.

EXAMPLE: cpc018/tsk001/buildDMD_PSB.c

Parameter Descriptions

prod

Pointer to the fully Generic Product Structure **RPGP_product_t**.

serial_data

Pointer to the resulting serialized data.

Notes / Rules for use

1. If not NULL, after copying the serialized data into data packet 28 the returned pointer must be freed.

```
int RPGP_product_free( RPGP_product_t *prod )
```

PARAMETER:	**prod**	Pointer to the Generic Product Structure
RETURN VALUE:		Returns **0** on success or **-1** on failure.

Frees the generic product structure.

EXAMPLE: cpc018/tsk001/mdaprodMain.c

Parameter Descriptions

prod

Pointer to the fully Generic Product Structure **RPGP_product_t**.

Notes / Rules for use

1. The generic product data structure must be freed after product construction.

Functions Used For Debugging Generic Products

The following functions can be used in debugging a populated generic product structure. If the product data is already serialized, **RPGP_product_deserialize** is called to create the populated generic product structure. The remainder of the functions are used to create a text output of the product.

RPGP_print_prod prints the entire product to which the argument is pointing. The output is to standard output, **stdout**. If the function is called by a running algorithm which has been launched during ORPG startup, the output is captured in the tasks log file named <**task_name>.output** in the **logs** directory under **$ORPGDIR**.

> LIMITATION: Even though all components can be deserialized, only area components and event components are supported by the debugging print functions listed below.

The rest of the print functions could be used to selectively print portions of the product.

After the product has been completed (and in the product database or in a single product binary file) the CODE utilities can be used to view the product. **CVG** provides basic display of the generic product (currently area and radial components are supported). **CVT** provides a text output generic product (all components supported). **CVT** supports all data types. Both provide the ability to decode unsigned integer arrays used by the radial component.

The following print functions could be used within an algorithm to analyze the contents of a generic product but they have not been expanded to include all data types (only short, unsigned short, and unsigned char are supported), the format of the output is not as easy to examine in some cases as the **CVT** output, and there is no capability to decode unsigned integer arrays.

These functions are used by **void RPGP_print_prod (RPGP_product_t *prod)** to print the entire generic product to **stdout**.

```
void RPGP_print_components( int n_comps, char **comps )

void RPGP_print_area( RPGP_area_t *area )

void RPGP_print_points( int n_points, RPGP_location_t *points )

void RPGP_print_params( int n_params, RPGP_parameter_t *params )

void RPGP_print_event( RPGP_event_t *event )
```

```
int RPGP_product_deserialize( char *serial_data, int size,
                              RPGP_product_t **prod )
```

PARAMETER: **serial_data** Pointer to the data to be deserialized.

size Size in bytes of the serialized data.

prod Pointer to the resulting deserialized data.

RETURN VALUE: Returns **0** on success and **-1** on failure.

Deserializes the serialized product data.

EXAMPLE: *This function is not yet used in any algorithm*

Parameter Descriptions

serial_data
Pointer to the data to be deserialized.

size
Size in bytes of the serialized data.

prod
Pointer to the resulting deserialized data.

Notes / Rules for use

1.

```
void RPGP_print_prod( RPGP_product_t *prod )
```

PARAMETER: **prod** Pointer to the Generic Product Structure

RETURN VALUE: Not Used

Prints the entire generic product to standard output.

EXAMPLE:

Parameter Descriptions

prod
Pointer to the Generic Product Structure **RPGP_product_t**

Notes / Rules for use

1.

Functions Specific to Using External Model Data

Five functions were added in Build 9 and will be documented in a subsequent edition of this guide.

```
int RPGCS_get_model_data( int model, char **data )
```

```
void* RPGCS_get_model_attrs( int model, char *data )

void* RPGCS_get_model_field( int model, char *data, char *field )

double RPGCS_get_data_value( RPGCS_model_grid_data_t *data, int level,
                             int i_ind, int j_ind, int *units )

void RPGCS_free_model_field( int model, char *data )
```

Vol 3. Document 2 -
The WSR-88D Algorithm API Reference

Section IV API Convenience Functions

NOTE: Preliminary support for Dual Pol data was provided in Build 11 in order to support implementation. Actual Dual Polarization implementation is provided in Build 12.

NOTE

Not all of the API functions have been documented. The functions not documented are not critical to algorithm development and many of the recently added functions were created to support the porting of legacy FORTRAN algorithms to ANSI-C.

- Some of the new functions are redundant with existing functions.
- Some are not generally useful for algorithm development.
- Some of the new functions have not yet been used in algorithms.

NOTE: Some previously deprecated functions are being used to port FORTRAN algorithms.

The contents of this section is organized as follows:

- Part A. Data Conversion Functions
- Part B. Date / Time Conversion Functions
- Part C. Coordinate Conversion Functions
- Part D. Lambert Projection / Grid Conversion Functions

Part A. Data Conversion Functions

This is a collection of functions that assist in encoding and decoding the WSR-88D standard representation of velocity, reflectivity, and spectrum width data using integer variables.

Velocity Resolution Functions

Unlike reflectivity and spectrum width moments, the WSR-88D has two operating modes for the velocity data moment. Velocity mode 1 (high resolution) is the normal mode providing velocity data from -63.5 **m/s** to 63 **m/s** in 0.5 **m/s** intervals. Velocity mode 2 (low resolution) provides velocity data from -127 **m/s** to 126 **m/s** in 1.0 **m/s** intervals.

The following functions establish the correct velocity resolution for the conversion functions. **RPGCS_set_velocity_reso** must be called before using the function **RPGCS_velocity_to_ms**. The return value of either function is used as a parameter to provide the resolution to **RPGCS_ms_to_velocity**. The current velocity resolution must be obtained at least once each volume scan before using the velocity conversion functions.

RESO RPGCS_get_velocity_reso(int dop_res)

PARAMETER: **dop_res** Base data radial velocity resolution

RETURN VALUE: One of the two possible values of the enumerated type **RESO** (either **HI_RES** or **LOW_RES**). See **rpgcs_data_conversion.h**.

Returns the **RESO** value corresponding to the Doppler resolution input via the **reso** parameter but does not change / set the variable used by the conversion functions.

EXAMPLE: cpc008/tsk001/alerting_grid_processing.c

Notes / Rules for use

1. **RPGCS_get_velocity_reso** is not actually needed for use with the velocity conversion functions because the return value from **RPGCS_set_velocity_reso** can be used as well.

RESO RPGCS_set_velocity_reso(int dop_res)

PARAMETER: **dop_res** Base data radial velocity resolution

RETURN VALUE: One of the two possible values of the enumerated type **RESO** (either **HI_RES** or **LOW_RES**). See **rpgcs_data_conversion.h**.

Returns the **RESO** value corresponding to the Doppler resolution input via the **reso** parameter and sets an internal infrastructure variable used by the velocity conversion functions.

EXAMPLE: cpc007/tsk002/basvlcty.c
cpc007/tsk014/bvel8bit.c

Parameter Descriptions

dop_res

The velocity resolution obtained from the **dop_resolution** field of the base data header. The structure **Base_data_header** is defined in **basedata.h**.

Notes / Rules for use

1. **RPGCS_set_velocity_reso** must be called before using **RPGCS_velocity_to_ms**. The return value of this function (or from **RPGCS_get_velocity_reso**) is an input to **RPGCS_ms_to_velocity**.

Base Data Value Decoding

These functions decode the scaled data values for the three radar moments (reflectivity, velocity, and spectrum width) which are contained in the base data radial messages and base data elevation messages. This encoded data is also contained in products using the digital data array (packet code 16). The scaled data (values 2-255) are decoded into the real physical parameters having the following units of measure: **dBZ** for reflectivity, **meters/second** for velocity, and **meters/second** for spectrum width. The special value -999.0 is returned to represent the binary encoded flag value 0 (Below Threshold) and special value -888.0 is returned to represent the binary encoded flag value 1 (Range Folded).

float RPGCS_reflectivity_to_dBZ(int value)

PARAMETER: **value** Encoded (scaled) data value for reflectivity

RETURN VALUE: Returns **INVALID_DATA** (-777.0) if input not 0-255. Otherwise returns the real reflectivity value in **dBZ** or -999.0 for below threshold and -888.0 for missing data.

Converts the scaled reflectivity value to the real data value in units of **dBZ**. The value -999 is returned for the below threshold flag (binary 0) and the value -888 is returned for the missing data flag (binary 1).

EXAMPLE: cpc007/tsk001/basrflct.c
cpc007/tsk013/bref8bit.c

float RPGCS_velocity_to_ms(int value)

PARAMETER: **value** Encoded (scaled) data value for velocity

RETURN VALUE: Returns **INVALID_DATA** (-777.0) if input not 0-255. Otherwise returns the real velocity value in **meters/second** or -999.0 for below threshold and -888.0 for range folded.

Converts the scaled velocity value to the real data value in units of **meters/second**. The value -999.0 is returned for the below threshold flag (binary 0) and the value -888.0 is returned for the range folded data flag (binary 1).

EXAMPLE: cpc007/tsk002/basvlcty.c
cpc007/tsk014/bvel8bit.c

float RPGCS_spectrum_width_to_ms(int value)

PARAMETER: **value** Encoded (scaled) data value for spectrum width

RETURN VALUE: Returns **INVALID_DATA** (-777.0) if input not 0-255. Otherwise returns the real spectrum width value in **meters/second** or -999.0 for below threshold and -888 for range folded.

Converts the scaled spectrum width value to the real data value in units of **meters/second**. The value -999.0 is returned for the below threshold flag (binary 0) and the value -888.0 is returned for the range folded data flag (binary 1).

EXAMPLE: cpc007/tsk003/basspect.c
cpc007/tsk015/superes8bit.c

Parameter Descriptions

value

The encoded (scaled) data value for one of three radar moments (reflectivity, velocity, and spectrum width). These values are contained in base data radial messages, base data elevation messages, and in products using the digital data array packet (packet code 16).

Notes / Rules for use

1. **RPGCS_set_velocity_reso** must be called at least once each volume scan before using **RPGCS_velocity_to_ms**.

Base Data Value Encoding

These functions encode real data for the three radar moments (reflectivity, velocity, and spectrum width) into the proper scaled values (2-255) that can be represented by a single byte integer. The encoding of flag values for binary data values 0 and 1 (which are flag values) must be handled separately. This encoding is often used in the creating of products using the digital data array (packet code 16). The input data must be in the following units of measure: **dBZ** for reflectivity, **meters/second** for velocity, and **meters/second** for spectrum width.

```
int RPGCS_dBZ_to_reflectivity( float value )
```

PARAMETER: **value** reflectivity in **dBZ**

RETURN VALUE: Returns **DATA_OUT_OF_RANGE** if input **value** exceeds WSR-88D limits (-32.0 to 94.5 **dBZ**). Otherwise returns reflectivity encoded appropriately (values 2-255) for use in a digital data array (packet code 16). Flag values 0 and 1 must be handled separately.

Converts real value of reflectivity (in **dBZ**) to the appropriate scaled reflectivity value for use in a digital data array (packet code 16). Flag values 0 and 1 must be handled separately.

EXAMPLE: cpc008/tsk001/alerting_grid_processing.c

```
int RPGCS_ms_to_velocity( RESO reso, float value )
```

PARAMETER: **reso** the API velocity resolution code

value velocity in **meters/second**

RETURN VALUE: Returns **DATA_OUT_OF_RANGE** if input **value** exceeds WSR-88D limits (-63.5 to 63 **m/s HIGH_RES** and -127 to 126 **m/s LOW_RES**). Returns **INVALID_DATA** if input **reso** not valid. Otherwise returns velocity encoded appropriately (values 0-255) for use in a digital data array (packet code 16). Flag values 0 and 1 must be handled separately.

Converts real value of velocity (in **meters/second**) to the appropriate scaled velocity value for use in a digital data array (packet code 16). Flag values 0 and 1 must be handled separately.

EXAMPLE: cpc007/tsk014/bvel8bit.c

int RPGCS_ms_to_spectrum_width(float value)

PARAMETER: **value** spectrum width in **meters/second**

RETURN VALUE: Returns **DATA_OUT_OF_RANGE** if input **value** exceeds WSR-88D limits (0.0 to 10.0 **m/s**). Otherwise returns spectrum width encoded appropriately (values 0-255) for use in a digital data array (packet code 16). Flag values 0 and 1 must be handled separately.

Converts real value of spectrum width (in **meters/second**) to the appropriate scaled spectrum width value for use in a digital data array (packet code 16). Flag values 0 and 1 must be handled separately.

EXAMPLE:

Parameter Descriptions

value

The real data value of the corresponding moment (reflectivity in dBZ, radial velocity in meters/second, and spectrum width in meters/second).

reso

The API value for representing the velocity resolution or the RDA. **HIGH_RES** corresponds to velocity mode 1 and **LOW_RES** corresponds to velocity mode 2. This value (an enumerated type **RESO** defined in **rpgcs_data_conversion.h**) is returned by the functions **RPGCS_set_velocity_reso** and **RPGCS_get_velocity_reso**.

Notes / Rules for use

1. The current resolution must be obtained at least once each volume scan before calling **RPGCS_ms_to_velocity**.

Decoding Generic Base Data Field (Future Build 12 Dual Polarization Data)

This function decodes radar data contained in the Generic moment structure. This structure contains advanced data fields including Dual Polarization data fields implemented Build 12. This function can

also decode the basic moments (R, V, SW), but only if these moments were read with **RPGC_get_radar_data**.

The design of this function assumes that the data never uses the lowest two data levels (0 and 1) for encoded numerical values. That is, either data levels 0 and 1 are flag values or they are not used. This is the case for the base data basic moments and all Dual Polarization data fields in the generic radial structure.

- For typical base data, the first flag (encoded as integer value `0`) represents data that are below threshold and the second flag (encoded as integer value `1`) represents data that are range folded. The parameters **below_threshold** and **range_folded** are used to specify the decoded values of these flags.
- **The Dual Polarization data fields currently use data level 0 for below threshold or no data. It is unknown whether data level 1 represents a flag value or is just not used.**

It is possible to use this function even with data that have encoded numerical values in data levels 0 and 1 as with some final product data. If not a flag value, the decoded value for data level 0 is entered for below_threshold and the decoded value for data level 1 is entered for range_folded.

```
int RPGCS_radar_data_conversion( void* data, Generic_moment_t *data_block,
                                 float below_threshold, float range_folded,
                                 float **converted_data )
```

PARAMETER:	**data**	Pointer to the input data array
	data_block	Pointer to the Generic Moment Structure
	below_threshold	Value to assign to below threshold data (the first flag)
	range_folded	Value to assign to range folded data (the second flag)
	***converted_data**	Pointer to the output data array
RETURN VALUE:		-1 on error, 1 on success, 0 no conversion accomplished because input array was not integer

Decodes data in the input data array (**data**). Input value `0` is decoded to the value of the **below_threshold** parameter and input value `1` is decoded to the value of the **range_folded** parameter. All other values in the input data array (**data**) are decoded according to the **scale** and **offset** field values contained in the input generic moment structure (**data_block**). The decoded array is in **converted_data**.

EXAMPLE: *This function is not yet used in any algorithm*

Parameter Descriptions

data

A pointer, cast to **void ***, of the input data array. This is normally the return value of the function **RPGC_get_radar_data()**.

data_block

A pointer to a generic moment structure associated with the input data array. This is normally the ***mom** parameter of the function **RPGC_get_radar_data()**. This structure contains the **scale** and **offset** fields used to decode the data array **data**.

below_threshold

Value to assign to input data level `0` (which is below threshold or no data). Input data level `0` is assigned this value in the output data. The value `-999.0` should be used for the basic data moments (R, V, and SW). This parameter can be used to assign a decoded value for data level `0` regardless of the meaning.

`range_folded`

Value to assign to input data level `1` (which is range folded in basic Doppler data). Input data value `1` is assigned this value in the output data. The value `-888.0` should be used for the basic data moments (R, V, and SW). This parameter can be used to assign a decoded value for data level `1` regardless of the meaning.

`*converted_data`

The output array of decoded data values (type `float`). Must be freed by caller.

Notes / Rules for use

1. Though not required, this function is conveniently used in conjunction with **`RPGC_get_radar_data()`**.
2. The allocated memory **`*converted_data`** must be freed by the algorithm.
3. This function and decode input data in bytes and shorts. The **`unsigned int`** option in **`Generic_moment_t`** is not supported.
4. This function assumes that there are two leading flag values encoded as integer value `0` and integer value `1`. That is it assumes that data levels `0` and `1` do not represent encoded numerical values. However it is possible to use this function even with data that have only 1 or have no flag values.
    a. The correct decoded value for integer `0` must be used for the **`below_threshold`** parameter and the correct decoded value for integer `1` must be used for the **`range_folded`** parameter, whether they represent flag values or numerical values.
    b. Comments should be included to clarify when these are not flag values.

Part B. Date / Time Conversion Functions

This collection of functions provides conversions for modified Julian date, UNIX time, and WSR-88D radial time. A time span function using Julian date and radial time is also included.

System Date / Time

```
int RPGCS_get_time_zone( )
```

PARAMETER: NONE

RETURN VALUE: Time zone in hours West of GMT.

Returns the time zone using the POSIX **`tzset`** function.

EXAMPLE: *This function is not yet used in any algorithm*

Notes / Rules for use

1. This function will not work unless the variable **`TZ`** is present in the ORPG account environment.

```
int RPGCS_get_date_time( int *ctime, int *cdate )
```

PARAMETER: **`ctime`** calculated current time (seconds after midnight)

`cdate` calculated current Modified Julian date

RETURN VALUE: always returns 0

From the system time calculates the seconds after midnight and the modified Julian date.

EXAMPLE: cpc008/tsk001/alerting_product_generation.c

Parameter Descriptions

`ctime` (output)
The calculated current time (in seconds after midnight)

`cdate` (output)
The calculated current Modified Julian date.

Notes / Rules for use (for a data driven task):

1.

Julian Date Conversion

These functions provide a two way conversion between modified Julian date and a date represented by year, month, and day.

```
int RPGCS_julian_to_date( int julian_date, int *year, int *month, int *day )
```

PARAMETER:	**julian_date**	modified Julian date (day 1 is Jan 1, 1970)
	year	calculated whole year
	month	calculated month number
	day	calculated day number
RETURN VALUE:		Currently always returns 0

Calculates the whole year, month number, and day number from the modified Julian date.
The minimum date is 1: 1970/01/01, the maximum is 534361755: 1465002/10/17

EXAMPLE: cpc018/tsk003/tdaru.c

```
int RPGCS_date_to_julian( int year, int month, int day, int *julian_date )
```

PARAMETER:	**year**	whole year
	month	month number
	day	day number
	julian_date	calculated modified Julian date (day 1 is Jan 1, 1970)
RETURN VALUE:		Currently always returns 0

Calculates the modified Julian date from the whole year, month number, and day number.
The minimum date is 1: 1970/01/01, the maximum recommended date is 65535: 2149/06/05

EXAMPLE: cpc013/tsk012/dp_dua_accum_func.c (Build 12)
cpc014/lib002/dp_time_conversions.c (Build 12)

Parameter Descriptions

julian_date
The modified Julian date where day 1 is January 1, 1970.

year
The whole year (e.g., 1972)

month
The month number (1-12)

day
The day number (1-31)

Notes / Rules for use

1. The earliest modified Julian date of 1 corresponds to the earliest valid date of 1970/01/01. Though not the actual limit of the function, the latest modified Julian date that is recommended is 65,535 which corresponds to the latest recommended date of 2149/06/05..
2. For **RPGCS_date_to_julian,** it is the user's responsibility to ensure that the earliest date and the latest date are not exceeded and that the range limits of the individual parameters **year**, **month**, and **day** are not exceeded.
3. For **RPGCS_julian_to_date**, it is the user's responsibility to ensure the input Julian date is between 1 and 65,535 inclusive.

This function calculates a time span in seconds between two modified Julian dates and times.

```
time_t RPGCS_time_span( int start_time, int start_date, int end_time,
                        int end_date )
```

PARAMETER:	**start_time**	seconds after midnight
	start_date	Modified Julian
	end_time	seconds after midnight
	end_date	Modified Julian
RETURN VALUE:		Time span in seconds.

Calculates the time span in seconds between two date / time inputs.

EXAMPLE: *This function is not yet used in any algorithm*

Parameter Descriptions

start_time
The seconds after midnight on the starting date.

start_date
The modified Julian date where day 1 is January 1, 1970.

end_time
The seconds after midnight on the end date.

end_date
The modified Julian date where day 1 is January 1, 1970.

Notes / Rules for use

1.

Unix Time Conversion

These functions provide a two way conversion between UNIX time and a time represented by year, month, day, hours, minutes, and seconds.

```
int RPGCS_unix_time_to_ymdhms( time_t time, int *year, int *month, int *day,
                               int *hour, int *minute, int *second )
```

PARAMETER:	**time**	UNIX time
	year	calculated year
	month	calculated month number
	day	calculated day number
	hour	calculated hours
	minute	calculated minutes
	second	calculated seconds
RETURN VALUE:		Returns -1 with an input UNIX time out of range; returns 0 otherwise.

Calculates the year, month, day, hours, minutes, and seconds from UNIX time.
The earliest time is 1970/01/01 00:00:00, the latest time is 2106/02/07 06:28:15

EXAMPLE: cpc013/tsk012/dp_dua_accum_func.c (Build 12)
cpc014/lib002/dp_time_conversions.c (Build 12)

```
int RPGCS_ymdhms_to_unix_time( time_t *time, int year, int month, int day,
                               int hour, int minute, int second )
```

PARAMETER:	**time**	calculated UNIX time
	year	year
	month	month number
	day	day number
	hour	hours
	minute	minutes
	second	seconds
RETURN VALUE:		Returns -1 with the input total time out of range; returns 0 otherwise. **NOTE: This does not reliably warn about an individual parameter being out of range.**

Calculates UNIX time from whole year, month number, day number, hours, minutes, and seconds.

The earliest time is 1970/01/01 00:00:00, the latest time is 2106/02/07 06:28:15

EXAMPLE: cpc013/tsk012/dp_dua_accum_func.c (Build 12)

Parameter Descriptions

time
: UNIX time. (1 - 4294967295)

year
: The whole year (1970 - 2106)

month
: The month number (1-12)

day
: The day number (1-31)

hour
: The hour using a 24 hour clock (e.g., 0-23)

minute
: Minutes (0-59)

second
: Seconds (0-59)

Notes / Rules for use

1. The earliest UNIX time of 1 corresponds to the earliest valid date of 1970/01/01 and time of 00:00:00. The latest UNIX time of 4294967295 corresponds to the latest valid date of 2106/02/07 and time of 06:28:15.
2. For **RPGCS_ymdhms_to_unix_time**, it is the user's responsibility to ensure that the earliest date-time and the latest date-time are not exceeded and that the range limits of the individual parameters **year**, **month**, **day**, **hour**, **minute**, and **second** are not exceeded.

Radial Time Conversion

This function converts WSR-88D radial time to a time represented by hours, minutes, seconds, and milliseconds.

```
int RPGCS_convert_radial_time( unsigned int time, int *hour, int *minute,
                               int *second, int *mills )
```

PARAMETER:

time	radial time
hour	calculated hours
minute	calculated minutes
second	calculated seconds
mills	calculated milliseconds

RETURN VALUE: Returns -1 on error and 0 on success.

Calculates hours, minutes, seconds, and milliseconds from radial time.

EXAMPLE: cpc018/tsk003/tdaru.c

Parameter Descriptions

`time`

The radial time obtained from the ORPG radial header (the number of milliseconds past midnight).

`hour`

The number of hours using a 24 hour clock (0-23).

`minute`

The number of minutes

`second`

The number of seconds.

`mills`

The number of milliseconds.

Notes / Rules for use

1.

Part C. Coordinate Conversion Functions

This is a collection of functions that assist in

- converting between polar coordinates (azimuth and range or **azran**) and rectangular coordinates (**x** and **y**). The rectangular or Cartesian grid is centered upon the same point as the polar coordinate reference.
- converting between Latitude and Longitude and either polar coordinates (**azran**) or rectangular coordinates (**x** and **y**).

> In order to use these coordinate conversion functions, the file **rpgcs_coordinates.h** must be included in addition to the standard algorithm include files. See CODE Guide Volume 3, Document 3, Section I, *Guidance for the Structure of Algorithms* for a complete discussion or required header files.

The WSR-88D base data and much of the internal data are in polar coordinates with the antenna location at the point of origin. These conversion functions assume a Cartesian grid also centered about the antenna location (positive values of **y** are North of the antenna, negative **y** is South, positive **x** is East, and negative values of **x** are West of the antenna.

Three sets of functions are provided. The first can be used to change the default units of measurement for distance. The second involve basic transformations in a single plane. The third set involves a projection from polar coordinates in conical sections (that is a WSR-88D elevation scan) to a Cartesian grid (flat plane at the antenna).

All functions except the Latitude / Longitude conversion functions accept either type **float** or **double** for the parameters representing polar coordinate range (**range**) and azimuth (**azm**); Cartesian North-South (**y**) and East-West (**x**); elevation angle (**elev**); and height above the surface (**height**). The Latitude / Longitude conversion functions use type **float** for all parameters.

The type **REAL** is used to represent either type **float** or **double** in the function parameters. **REAL** is defined as follows.

Including the definition of **RPGCS_FLOAT** and **REAL** prior to including the header file will permit the use of parameters of type **float**. Including the definition of **RPGCS_DOUBLE** and **REAL** prior to including the header file will permit the use of parameters of type **double**.

`#define RPGCS_FLOAT`
`#define REAL float`
`#include <rpgcs_coordinates.h>`
 	
`int retval;`
 	
`float range_p;`
`float azm_p;`
	`#define RPGCS_DOUBLE`
	`#define REAL double`
	`#include <rpgcs_coordinates.h>`
	
	`int retval;`
	
	`double range_p;`
	`double azm_p;`

`float elev_p;`
`float x_op=0.0;`
`float y_op=0.0;`
 	
`float x_p;`
`float y_p;`
`float range_op=0.0;`
`float azm_op=0.0;`
	`double elev_p;`
	`double x_op=0.0;`
	`double y_op=0.0;`
	
	`double x_p;`
	`double y_p;`
	`double range_op=0.0;`
	`double azm_op=0.0;`

```
retval = RPGCS_azranelev_to_xy(range_p,azm_p,elev_p,&x_op,&y_op);

retval = RPGCS_xy_to_azran_u(x_p,y_p,METERS,&range_op,NMILES,&azm_op);
```

Setting Units of Measure

DISTANCE

The Latitude / Longitude conversion functions use kilometers for all distances both Cartesian **x y** and polar coordinate **range**. The remaining functions permit a selection of the unit of measure used for distance.

Except for the Latitude / Longitude conversion functions, the default unit of measure for distance is **meters**. This applies to **range** in polar coordinates, **x** and **y** values in Cartesian coordinates and to **height** as used as input and output for the transformation functions in this section.

The following functions are provided to change this default value. Currently *meters* (**m**), *thousands of feet* (**KFt**), and *Nautical miles* (**NM**) are supported. If *meters* is satisfactory for all distance measures used in an algorithm, nothing has to be done.

The unit of measure can be set independently for function inputs and outputs. The units can be changed (set) at any time. If all inputs in an algorithm are in meters (the original default) and all outputs are in **NM**, then the action required is to set the output to **NM** once prior to calling the first transformation function.

ANGLE

The default unit of measure for angles is **degrees**. This applies to azimuth angle (**azm**) and elevation angle (**elev**) values in polar coordinates. There are no functions provided to change this default.

```
void RPGCS_set_input_units( int units )
```

PARAMETER: **units** input unit of measure for distance

RETURN VALUE: Not Used

Sets the input unit of measure for distance for subsequent invocations of the coordinate transformation functions.

EXAMPLE: *This function is not yet used in any algorithm*

`void RPGCS_set_output_units( int units )`

PARAMETER: **`units`** output unit of measure for distance

RETURN VALUE: Not Used

Sets the output unit of measure for distance for subsequent invocations of the coordinate transformation functions.

EXAMPLE: *This function is not yet used in any algorithm*

`int RPGCS_get_input_units( )`

PARAMETER: None

RETURN VALUE: The input unit of measure for distance.

Returns the current setting of the input unit of measure for distance being used by the coordinate transformation functions.

EXAMPLE: *This function is not yet used in any algorithm*

`int RPGCS_get_output_units( )`

PARAMETER: None

RETURN VALUE: The output unit of measure for distance.

Returns the current setting of the output unit of measure for distance being used by the coordinate transformation functions.

EXAMPLE: *This function is not yet used in any algorithm*

Parameter Descriptions

`units`

The unit of measure for distance. This value is defined separately for inputs and outputs for the coordinate transformation functions. It applies to the **`range`**, **`x`**, **`y`**, and **`height`** parameters. The current permissible values for this parameter are: **`METERS`** (1), **`KFEET`** (2), and **`NMILES`** (3) as defined in **`rpgcs_coordinates.h`**.

Notes / Rules for use

1. If units are not set, the default value for both input and output is *meters* (a parameter value of **`METERS`**)
2. If the default input / output unit of measure is changed, it retains this new value until either reset or the task terminates and is restarted.

Single-Plane Cartesian x y Conversions

This set of functions accomplish two-way conversion between polar coordinates (**azran**) and rectangular coordinates (x and y). Some of the functions use the current default setting for input and output range units while other functions use the input and output units as specified via function parameters.

WARNING: Ignore duplicate implementations of theses functions in cpc018/tsk006.

```
int RPGCS_xy_to_azran( REAL x, REAL y, REAL *range, REAL *azm )
```

PARAMETER:

x	Cartesian x distance (East-West) meters
y	Cartesian y distance (North-South) meters
range	calculated range (polar coord) meters
azm	calculated azimuth (polar coord) degrees

RETURN VALUE: Currently, always returns 0.

Given Cartesian distance inputs x and y, calculates polar coordinate range and azimuth using the set units of measure.

EXAMPLE: See Part C. Introduction
---
cpc018/tsk003/tdaru.c

```
int RPGCS_xy_to_azran_u( REAL x, REAL y, int xy_units, REAL *range,
                         int range_units, REAL *azm )
```

PARAMETER:

x	Cartesian x distance (East-West)
y	Cartesian y distance (North-South)
xy_units	unit of measure for **x** and **y** parameters
range	calculated range (polar coord)
range_units	unit of measure for **range** parameter
azm	calculated azimuth (polar coord)

RETURN VALUE: Returns -1 if any unit of measure parameter (**xy_units** and **range_units**) has an invalid value, otherwise returns 0.

Given Cartesian distance inputs x and y, calculates polar coordinate range and azimuth using the input parameters **xy_units** and **range_units**.

EXAMPLE: See Part C. Introduction

Parameter Descriptions

x

The Cartesian East-West distance (positive is East).

y

The Cartesian North-South distance (positive is North).

range

The polar coordinate range value calculated by the function.

azm

The polar coordinate azimuth value calculated by the function.

xy_units

The unit of measure for the input Cartesian coordinates. Permitted values are **METERS**, **KFEET**, and **NMILES** as defined in **rpgcs_coordinates.h**.

range_units

The unit of measure for the calculated polar coordinate range. Permitted values are **METERS**, **KFEET**, and **NMILES** as defined in **rpgcs_coordinates.h**.

Notes / Rules for use

1. **WARNING:** Ignore duplicate implementations of theses functions in cpc018/tsk006.

```
int RPGCS_azran_to_xy( REAL range, REAL azm, REAL *x, REAL *y )
```

PARAMETER:

range	range (polar coord) meters
azm	azimuth (polar coord) degrees
x	calculated Cartesian x distance meters
y	calculated Cartesian y distance meters

RETURN VALUE: Currently, always returns 0.

Given the polar coordinate range and azimuth, calculates the Cartesian coordinates x and y using the set units of measure.

EXAMPLE: See Part C. Introduction

---

cpc017/tsk011/makeTVSAssociation.c

```
int RPGCS_azran_to_xy_u( REAL range, int range_units, REAL azm,
                         int azm_units, REAL *x, REAL *y, int xy_units )
```

PARAMETER:

range	range (polar coord)
range_units	unit of measure for **range** parameter
azm	azimuth (polar coord)
azm_units	unit of measure for **azm** parameter
x	calculated Cartesian x distance
y	calculated Cartesian y distance
xy_units	unit of measure for **x** and **y** parameters

RETURN VALUE: Returns -1 if any unit of measure parameter (**range_units**, **azm_units**, and **xy_units**) has an invalid value; otherwise returns 0.

Given the polar coordinate range and azimuth, calculates the Cartesian coordinates x and y using the input parameters **range_units** and **xy_units**.

EXAMPLE: See Part C. Introduction

This function is not yet used in any algorithm

Parameter Descriptions

range
The polar coordinate range value.

azm
The polar coordinate azimuth value.

x
The Cartesian East-West value calculated by the function.

y
The Cartesian North-South value calculated by the function.

range_units
The unit of measure for the input polar coordinate range. Permitted values are **METERS**, **KFEET**, and **NMILES** as defined in **rpgcs_coordinates.h**.

azm_units
The unit of measure for the input polar coordinate azimuth. Permitted values are **DEG** (degrees) and **DEG10** (degrees*10) as defined in **rpgcs_coordinates.h**. The **DEG10** unit is used by the ORPG in order to represent tenths of a degree with integer variables.

xy_units
The unit of measure for the calculated Cartesian coordinates. Permitted values are **METERS**, **KFEET**, and **NMILES** as defined in **rpgcs_coordinates.h**.

Notes / Rules for use

1.

Single-Plane Latitude / Longitude Conversions

This set of functions accomplish two-way conversion between Latitude and Longitude and either polar coordinates (**azran**) or rectangular coordinates (x and y). These functions do not use the multiple units of measure for distance that are used by the other functions. All distances are in kilometers.

These functions use the antenna location contained in the Site Information adaptation data. In a development environment, the Site Information adaptation data must be set prior to the creation of the products.

WARNING: Ignore duplicate implementations of theses functions in cpc018/tsk001. Some of these duplicates have parameters in a different order

```
int RPGCS_xy_to_latlon( float x, float y, float *lat, float *lon )
```

PARAMETER: **x** Cartesian x distance (East-West) kilometers

y Cartesian y distance (North-South) kilometers

lat calculated Latitude

lon calculated Longitude

RETURN VALUE: Currently, always returns 0.

Given Cartesian coordinate inputs x and y, calculates the Latitude and Longitude.

EXAMPLE:

```
int RPGCS_latlon_to_xy( float lat, float lon, float *x, float *y )
```

PARAMETER: **lat** Latitude

lon Longitude

x calculated Cartesian x distance (kilometers)

y calculated Cartesian y distance (kilometers)

RETURN VALUE: Currently, always returns 0.

Given the Latitude and Longitude, calculates the Cartesian coordinates x and y.

EXAMPLE: *This function is not yet used in any algorithm*

```
int RPGCS_azran_to_latlon( float rng, float azm, float *lat, float *lon )
```

PARAMETER: **rng** range (polar coord) kilometers

azm azimuth (polar coord) degrees

lat calculated Latitude

lon calculated Longitude

RETURN VALUE: Currently, always returns 0.

Given the polar coordinate range and azimuth, calculates the Latitude and Longitude.

EXAMPLE:

```
int RPGCS_latlon_to_azran( float lat, float lon, float *rng, float *azm )
```

PARAMETER: **lat** Latitude

lon Longitude

rng calculated range (polar coord) kilometers

	azm	calculated azimuth (polar coord) degrees
RETURN VALUE:		Currently, always returns 0.

Given Cartesian distance inputs x and y, calculates the polar coordinate range and azimuth.

EXAMPLE:

Parameter Descriptions

x
The Cartesian East-West distance in kilometers (positive is East).

y
The Cartesian North-South distance in kilometers (positive is North).

rng
The polar coordinate range value (kilometers).

azm
The polar coordinate azimuth value (degrees).

lat
The Latitude of the point.

lon
The Longitude of the point.

Notes / Rules for use

1. **WARNING:** Ignore duplicate implementations of theses functions in cpc018/tsk001. Some of these duplicates have parameters in a different order.

Elevation Scan to Ground Cartesian Coordinates

These functions provide a projection from a conical section in polar coordinates (**azran**) to a Cartesian grid (x and y in a plane at the antenna) and provide a height calculation from the conical section to the earth's surface. Some functions use the current default setting for input and output range units while other functions use the input and output units as specified via function parameters.

WARNING: Ignore duplicate implementations of theses functions in cpc018/tsk006.

```
int RPGCS_azranelev_to_xy( REAL range, REAL azm, REAL elev, REAL *x, REAL *y )
```

PARAMETER:	**range**	slant range (conical section) meters
	azm	azimuth angle (conical section) degrees
	elev	elevation angle (conical section) degrees
	x	calculated Cartesian x distance meters
	y	calculated Cartesian y distance meters

RETURN VALUE: Returns -1 if the elevation angle parameter is out of range, otherwise returns 0.

Given the conical section slant range, azimuth, and elevation angle, calculates the flat-plane Cartesian coordinates x and y using the set units of measure.

EXAMPLE: See Part C. Introduction
---
cpc018/tsk003/tdaru.c

```
int RPGCS_azranelev_to_xy_u( REAL range, int range_units, REAL azm,
                             int azm_units, REAL elev, int elev_units,
                             REAL *x, REAL *y, int xy_units )
```

PARAMETER:

range	slant range (conical section)
range_units	unit of measure for the **range** parameter
azm	azimuth angle (conical section)
azm_units	unit of measure for the **azm** parameter
elev	elevation angle (conical section)
elev_units	unit of measure for the **elev** parameter
x	calculated Cartesian x distance
y	calculated Cartesian y distance
xy_units	unit of measure for the **x** and **y** parameters

RETURN VALUE: Returns -1 if the elevation angle parameter is out of range; returns -1 if any unit of measure parameter (**range_units**, **azm_units**, **elev_units**, and **xy_units**) has an invalid value; otherwise returns 0.

Given the conical section slant range, azimuth, and elevation angle, calculates the flat-plane Cartesian coordinates x and y using the input parameters **range_units**, **azm_units**, **elev_units**, and **xy_units**.

EXAMPLE: See Part C. Introduction

This function is not yet used in any algorithm

Parameter Descriptions

range
The conical section slant range value.

azm
The conical section azimuth angle value.

elev
The conical section elevation angle value.

x (output)
The Cartesian East-West value calculated by the function.

y (output)
The Cartesian North-South value calculated by the function.

range_units
The unit of measure for the input slant range. Permitted values are **METERS**, **KFEET**, and **NMILES** as defined in **rpgcs_coordinates.h**.

azm_units
The unit of measure for the input azimuth angle. Permitted values are **DEG** (degrees) and **DEG10** (degrees*10) as defined in **rpgcs_coordinates.h**. The **DEG10** unit is used by the ORPG in order to represent tenths of a degree with integer variables.

elev_units
The unit of measure for the input elevation angle. Permitted values are **DEG** (degrees) and **DEG10** (degrees*10) as defined in **rpgcs_coordinates.h**. The **DEG10** unit is used by the ORPG in order to represent tenths of a degree with integer variables.

xy_units
The unit of measure for the calculated Cartesian coordinates. Permitted values are **METERS**, **KFEET**, and **NMILES** as defined in **rpgcs_coordinates.h**.

Notes / Rules for use

1.

```
int RPGCS_height( REAL range, REAL elev, REAL *height )
```

PARAMETER: **range** slant range (conical section) meters

elev elevation angle (conical section) degrees

height calculated height above the earth's surface (meters)

RETURN VALUE: Returns -1 if the elevation angle parameter is out of range, otherwise returns 0.

Given the conical section slant range and elevation angle, calculates the height above the earth's surface using the preset units of measure.

EXAMPLE: cpc022/tsk005/hreet_compute_eet.c

```
int RPGCS_height_u( REAL range, int range_units, REAL elev, int elev_units,
                    REAL *height, int height_units )
```

PARAMETER: **range** slant range (conical section)

range_units unit of measure for **range** parameter

elev elevation angle (conical section)

elev_units unit of measure for **elev** parameter

height calculated height above the earth's surface

height_units	unit of measure for **height** parameter

RETURN VALUE: Returns -1 if the elevation angle parameter is out of range; returns -1 if any unit of measure parameter (**range_units**, **elev_units**, and **height_units**) has an invalid value; otherwise returns 0.

Given the conical section slant range and elevation angle, calculates the height above the earth's surface using the input parameters **range_units**, **elev_units**, and **height_units**.

EXAMPLE: cpc024/tsk003/mlprod.c (Build 12)

Parameter Descriptions

range
The conical section slant range.

elev
The conical section elevation angle.

height (output)
The height above the earth's surface calculated by the function.

range_units
The unit of measure for the input slant range. Permitted values are **METERS**, **KFEET**, and **NMILES** as defined in **rpgcs_coordinates.h**.

elev_units
The unit of measure for the input elevation angle. Permitted values are **DEG** (degrees) and **DEG10** (degrees*10) as defined in **rpgcs_coordinates.h**. The **DEG10** unit is used by the ORPG in order to represent tenths of a degree with integer variables.

height_units
The unit of measure for the calculated height. Permitted values are **METERS**, **KFEET**, and **NMILES** as defined in **rpgcs_coordinates.h**.

Notes / Rules for use

1.

```
int RPGCS_range( REAL height, REAL elev, REAL *range )
```

PARAMETER:	**height**	height above the earth's surface
	elev	elevation angle (conical section)
	range	calculated slant range (conical section)

RETURN VALUE: Returns -1 if the elevation angle parameter is out of range, otherwise returns 0.

Given the height above the earth's surface and elevation angle, calculates the conical section slant range using the preset units of measure.

EXAMPLE: *This function is not yet used in any algorithm*

```
int RPGCS_range_u( REAL height, int height_units, REAL elev, int elev_units,
                   REAL *range, int range_units )
```

PARAMETER:	**height**	height above the earth's surface
	height_units	unit of measure for **height** parameter
	elev	elevation angle (conical section)
	elev_units	unit of measure for **elev** parameter
	range	calculated slant range (conical section)
	range_units	unit of measure for **range** parameter
RETURN VALUE:		Returns -1 if the elevation angle parameter is out of range; returns -1 if any unit of measure parameter (**range_units**, **elev_units**, and **height_units**) has an invalid value; otherwise returns 0.

Given the height above the earth's surface and elevation angle, calculates the conical section slant range using the input parameters **range_units**, **elev_units**, and **height_units**.

EXAMPLE: *This function is not yet used in any algorithm*

Parameter Descriptions

height
: The height above the earth's surface.

elev
: The conical section elevation angle.

range (output)
: The conical section slant range calculated by the function.

height_units
: The unit of measure for the calculated height. Permitted values are **METERS**, **KFEET**, and **NMILES** as defined in **rpgcs_coordinates.h**.

elev_units
: The unit of measure for the input elevation angle. Permitted values are **DEG** (degrees) and **DEG10** (degrees*10) as defined in **rpgcs_coordinates.h**. The **DEG10** unit is used by the ORPG in order to represent tenths of a degree with integer variables.

range_units
: The unit of measure for the input slant range. Permitted values are **METERS**, **KFEET**, and **NMILES** as defined in **rpgcs_coordinates.h**.

Notes / Rules for use

1.

Window Product Helper

This function supports construction of a very specific type of product referred to as a "window" product. This is a polar array of data where only a subset of radial contain data. Within those radials the data are only valid between a minimum and maximum range.

Documentation of This Function is Not Complete

```
void RPGCS_window_extraction( float radius_center, float azimuth_center,
                              float length, float *max_rad, float *min_rad,
                              float *max_theta, float *min_theta )
```

PARAMETER:	**radius_center**	short_desc
	azimuth_center	short_desc
	length	short_desc
	max_rad	short_desc
	min_rad	short_desc
	max_theta	short_desc
	min_theta	short_desc
RETURN VALUE:		retval_descript

Description paragraph

EXAMPLE:

Parameter Descriptions

radius_center
long_desccription

azimuth_center
long_desccription

length
long_desccription

max_rad
long_desccription

min_rad
long_desccription

`max_theta`

long_desccription

`min_theta`

long_desccription

Notes / Rules for use

1.

Part D. Lambert Projection / Grid Conversion Functions

Nine functions were added in Build 9 and will be documented in a subsequent edition of this guide. These functions support conversions between a specific Lambert projection coordinates and various grid coordinates. They are useful only for reading external model data that are in that specific projection.

```
int RPGCS_lambert_grid_point_xy( int i_ind, int j_ind, double *x, double *y )


int RPGCS_lambert_grid_point_latlon( int i_ind, int j_ind, double *lat,
                                     double *lon )


int RPGCS_lambert_grid_point_azran( int i_ind, int j_ind, double *azm,
                                    double *ran )


int RPGCS_lambert_latlon_to_grid_point( double lat, double lon, int *i_ind,
                                        int *j_ind )


int RPGCS_lambert_xy_to_grid_point( double x, double y, int *i_ind, int *j_ind )


int RPGCS_lambert_latlon_to_xy( double lat, double lon, double *x, double *y )


int RPGCS_lambert_xy_to_latlon( double x, double y, double *lat, double *lon )


void RPGCS_lambert_init( RPGCS_model_attr_t *model_attr )


int RPGCS_lambertuv_to_uv( double ur, double vr, double lon, double *u, double *v )
```

Volume 3. WSR-88D Algorithm Programming Guide

Document 3. WSR-88D Algorithm Structure and Sample Algorithms

For an algorithm to function correctly in an operational WSR-88D ORPG, the API must be used correctly which includes following a basic logical structure. CODE Guide Vol 3 Document 2 covered rules for using the individual API services. This document places this guidance in the context of a logical structure of an algorithm and provides sample algorithms that follow that structure. In addition, procedures for correctly writing product data fields are covered.

New algorithm development in FORTRAN is no longer supported by the National Weather Service. The WSR-88D algorithms written in FORTRAN are being ported to ANSI-C during the next few development cycles. Development activities must be coordinated with the Radar Operations Center Engineering Branch if modifying an existing FORTRAN algorithm. A list of recently ported algorithm tasks is provided in Appendix B.

Section I Guidance for the Structure of Algorithms

Section II Sample Algorithms

Section III Writing Product Data Fields

Vol 3. Document 3 -
WSR-88D Algorithm Structure and Sample Algorithms

Section I Guidance for the Structure of Algorithms

The following rules for the basic logical structure of an algorithm must be followed to function correctly in an operational WSR-88D ORPG. Many of these rules are not otherwise published and are derived from the structure of legacy algorithms, from the documentation for the API libraries (librpg and librpgc), and via consultation with the ROC Engineering Branch.

The typical reason an algorithm is aborted is an inability to read input data or input data that are out of sequence. All algorithms must do one of the following:

1. **Successfully output a product, or**
2. **Call the appropriate abort function.**

BUILD 12:

- Emphasized that algorithms producing a product must either successfully produce that product or call the appropriate abort service.
- Added that **`PGM_DISABLED_MOMENT`** is also used when a required advanced data field (Dual Pol data) is not in the radial message.
- Added guidance for algorithms producing **`RADIAL_DATA`**. Part C provides limits on input data and output data registration for radial producing algorithms and guidance concerning when radial outputs should be used and when they cannot be used. Part D provides limits on when radial producing algorithms can abort and additional guidance for when there are data problems and the radial is passed along without aborting.
- Updated discussion of high level algorithm design issues**.** Emphasized that normal meteorological algorithms usually have no need to register for events but listed two situations where an event driven algorithm might be appropriate.

BUILD 11:

- Documented the purpose and behavior of replay instances of data driven and event driven tasks. Improves responsiveness to one-time requests.
- Documented unique algorithm logic contained in existing algorithms that are either special cases or non-standard. This list will be expanded in the future.
    - The logic in the task **`ntda_fp`** is non-standard and should not be used as an example. The task has two outputs and both products are generated without checking for a product request.
    - Two tasks have a task timing less than the timing of an output. The **`superob`** and **`mdaprod`** tasks are **`ELEVATION_BASED`** and have a **`VOLUME_DATA`** output.
    - Several tasks have **`VOLUME_DATA`** inputs and an **`ELEVATION_DATA`** output.
- Documented the relationship between input product timing and output product timing

and the two unique exceptions to the rule that output timing must be greater than or equal to the largest input timing.

- Expanded the description of mixing input timing types.
- Clarified instructions on the handling of multiple product outputs. The function RPGC_check_data_by_name is recommended if the task produces more than one output. However if not used the result of attempting to obtain the output buffer will reflect whether the product is requested.
- Clarified some aspects of the use of event driven algorithms.

Part A. Introduction

It should be noted that a task may run and produce a valid product even though some of the rules in this document are not followed. However, the algorithm would not be correctly integrated into the ORPG. Not properly structuring an algorithm (following these guidelines) could result in unexplained behavior (for example, an algorithm task that crashed upon a commanded volume restart).

Additional rules for using the API services are described in CODE Guide Vol 3, Document 2 - *The WSR-88D Algorithm API Reference*, which should be read before studying the sample algorithms.

Part B discusses high level design issues for multiple task algorithms and the classification of persistent data. Part C provides guidance for initialization of data driven algorithms and event driven algorithms and describes inter dependencies on the type of algorithm loop control. Part D covers the logic after passing the control loop in data driven algorithms (or prior to returning from event handlers in event driven algorithms) for basic algorithms producing one output product. Part E provides additional guidance for algorithms producing more than one product and algorithms that use input parameters contained in a request message.

NOTE: This document makes multiple references to two forms of a data driven algorithm control loop. This is the continuous main loop of an algorithm that is entered after all registrations and initializations are complete.

1. The WAIT_ALL form of the control loop is the most common and is used in algorithms that have only one data input or need more than one data input to create a product. The loop control function used is **`RPGC_wait_act(WAIT_DRIVING_INPUT)`**
2. The WAIT_ANY form of the control loop is used in algorithms having multiple registered data inputs but use only one input to produce a product. There are two functions that can be used: **`RPGC_wait_act(WAIT_ANY_INPUT)`** and **`RPGC_wait_for_any_data(WAIT_ANY_INPUT)`**.

File Naming Conventions

The configuration management system used by the ROC requires unique filename regardless of the directory in which the file is located. The following suggestion will minimize potential filename conflicts.

- All files for an individual task (files within a **`tsk`** subdirectory) should begin with a prefix related to the tasks purpose or the product produced by the task.

Required Header Files

There are four header files to be included in main source code file for every algorithm: **`rpg_globals.h`**, **`gen_stat_msg.h`**, **`rpgc.h`**, and **`rpgcs.h`**. Note that **`rpg_globals.h`** includes several standard libraries (**`stdio.h`**, **`stdlib.h`**, **`time.h`**, **`ctype.h`**, and **`string.h`**) along with numerous ORPG header files (for example **`basedata.h`** and **`a309.h`**). Some algorithms include individual header files instead of rpg_globals.h.

If using the coordinate conversion functions, the file **`rpgcs_coordinates.h`** must be included. If using any **`RPGP`** helper function, including support functions for the generic data packet, the file **`rpgp.h`** must be included. The file containing the structures for the generic data packet, **`orpg_product.h`**, is included by **`rpgp.h`**. In addition, if the algorithm uses adaptation data, the header file defining that data structure must be included.

SPECIAL NOTE: Algorithms should use the macros defined in the include files mentioned above rather than defined new macros within the algorithm. For example, an algorithm should use **`END_VOL`** defined in **`a309.h`** rather than creating a new macro **`#define END_OF_VOL 4`** in the algorithm. It would be acceptable to define a macro **`#define END_OF_VOL END_VOL`** while including **`a309.h`**.

Required Linked Libraries

See the makefile discussion in CODE Guide Volume 2, Document 2, Section II, *Compiling Software in the ORPG Environment* for the list of system and ORPG libraries that must be linked.

Inappropriate Use of non-API Services

Some existing algorithms do not completely follow the guidance and use non-API functions from the main ORPG library of services. These services should not be used in development code and the ROC should not accept algorithms using non-API services. Many of these algorithms are referenced as examples (**`cpc007/tsk003/basspect.c, cpc007/tsk013/bref8bit.c, cpc007/tsk014/bvel8bit.c, cpc007/tsk015/superes8bit.c,`** etc.) and the use of the following functions in these algorithms should not be considered correct. Most of the ORPG library functions incorrectly used have a direct counterpart in the algorithm API.

Inappropriately Used Service Function	Corresponding Algorithm API Function

LE_send_msg	**RPGC_log_msg**
ORPGPAT_get_elevation_index	**RPGC_get_buffer_elev_index**
ORPGPAT_get_prod_id_from_code	
ORPGPAT_get_num_dep_prods	
ORPGDA_read	**RPGC_data_access_read**
ORPGDA_write	**RPGC_data_access_write**
ORPGDA_seek	**RPGC_data_access_seek**
ORPGDA_clear	**RPGC_data_access_clear**
ORPGDA_open	**RPGC_data_access_open**
ORPGDA_info	**RPGC_data_access_msg_info**
ORPGDA_list	**RPGC_data_access_list**
ORPGDA_lbfd	
ORPGRDA_get_rda_config	
ORPGVCP_BAMS_to_deg	**RPG_elev_angle_BAMS_to_deg (Fortran API)**
ORPGVST_get_volume_time	

Summary of Deprecated API Functions

The algorithm API is continuously evolving. Often newly added functions do not add a new capability but rather a modified / improved capability. You will still see the deprecated functions in existing algorithms (including those recently ported from Fortran). However, they should not be used in new algorithms. The following charts include the old deprecated functions highlighted in red. A list of deprecated functions is provided as a convenient reference.

Deprecated API Function	Replaced by new API Function
RPGC_abort_datatype_because	**RPGC_abort_dataname_because**
RPGC_check_data	**RPGC_check_data_by_name**
RPGC_get_current_elev_index	**RPGC_get_buffer_elev_index**
RPGC_get_current_vol_num	**RPGC_get_buffer_vol_num**
RPGC_get_inbuf	**RPGC_get_inbuf_by_name**
RPGC_get_outbuf_for_req	**RPGC_get_outbuf_by_name_for_req**
RPGC_get_outbuf	**RPGC_get_outbuf_by_name**
RPGC_in_data	**RPGC_reg_inputs** **RPGC_reg_io**
RPGC_in_opt	**RPGC_in_opt_by_name**
RPGC_out_data	**RPGC_reg_outputs** **RPGC_reg_io**

RPGC_out_data_wevent	**RPGC_out_data_by_name_wevent**
RPGCS_get_last_elev_index	**RPGC_is_buffer_from_last_elev**

Non-Standard or Special Case Program Logic / Structure

Some algorithms do not completely follow the guidelines presented in this document. Though this list will be expanded in the future, there will be no-attempt to make a complete list of algorithm tasks not following standard guidance. Some departures from this guidance are minor and some significant.

Non-Standard Task Logic Resulting in Unusual Configuration

One task that is significant is the ntda_fp task (included only with NWS CODE).

Task **`ntda_fp`**

- The **`task_attr_table`** configuration of the output **`NTDA_EDR`** (156) is non-standard in that the other output of the task, **`NTDA_CONF`** (157), is listed as a dependent product. This permits normal operation if only product 156 is requested.
- The output final products use data packet 16. The threshold level fields contain the Scale Offset parameters but are not encoded in the recommended method, which was defined after these products were developed.

Elevation Task Timing with Volume Output Product - Special Case

There are two tasks having a task timing less than the timing of an output product. This is unusual but the result of this is the volume product can be produced even if some of the elevations of the intermediate product are not available.

Task **`superob_vel`** is an **`ELEVATION_BASED`** task and it's product **`SUPEROBVEL`** (136) is **`VOLUME_DATA`**.

Task **`mdaprod`** is an **`ELEVATION_BASED`** task and one of the output products, **`MDAPROD`** (141), is **`VOLUME_DATA`**.

Volume Task Timing with Elevation Output Product - Special Case

The several tasks has a task timing of **`VOLUME_BASED`** and the output is **`ELEVATION_DATA`**. Most of these are of the Kinematic algorithm type. This configuration is used if the output products for later elevations in the volume should not be produced if there is any problem with the lower elevations of input data. In other words the downstream tasks cannot handle a missing elevation (e.g., they produce **`VOLUME_DATA`** products).

Volume Input with an Elevation Output

Normally an **EVELVATION_DATA** product cannot have a **VOLUME_DATA** input. This is a limitation of the WAIT_ALL form of control loop. There are two tasks where this occurs but the task uses the WAIT_ANY form of control loop.

> The **tdaruprod** task and the **mesoruprod** task have one **VOLUME_DATA** input for their **ELEVATION_DATA** output. However they also have **ELEVATION_DATA** inputs that provide data for the early elevations. In both cases it is an **ELEVATION_DATA** input that controls the output of the **ELEVATION_DATA** product. The **VOLUME_DATA** input is used to obtain a data set that is saved and used while producing the next volume scan of elevation data.

Not Using Predefined System Macros

The include files such as a309.h and basedata.h include system macros which should be used when applicable. Some algorithm tasks redefine these locally. For example **mda1d** defines **END_OF_VOL** as **4** and **PSU_END_OF_VOL** as **9** instead of using **END_VOL** and **PSEND_VOL** from a309.h. However, it would have been acceptable to define **END_OF_VOL** as **END_VOL** and define **PSU_END_OF_VOL** as **PSEND_VOL**.

Part B. High Level Algorithm Design Issues

There are many issues affecting high-level algorithm design many of which are beyond the scope of this document. Some of the factors to be considered:

Data driven or Event driven

Virtually all algorithms should be data driven. That is the algorithm begins processing with the availability of input product data. Very few meteorological algorithms have a need to register for events. Most existing algorithms using events are involved in system monitoring and control, for example pcipdalg, cltutprod).

- The main reason for using an event driven algorithm would be if the task had non-product data inputs and no product inputs. In this case an alternative would be to have a driving product data input of the desired timing (elevation or volume) to act as a trigger to activate the algorithm. This alternative should be used if the task also has a replay version.
- Another reason for using an event driven algorithm is an algorithm that does not function with the data flow timing of the ORPG. In this case any output product produced is not **RADIAL_DATA**, **ELEVATION_DATA**, or **VOLUME_DATA**. The input product (if any) would be **DEMAND_DATA** and the output data also **DEMAND_DATA**.

Input data and number of tasks

- Can the algorithm be implemented in one task or should the processing be divided among a series of tasks producing intermediate products?
    - One factor in this decision is that customizing parameters in the request message (other than elevation for elevation products) are only passed to the task producing the final product.
    - Another factor is that with multiple tasks, intermediate products can be produced that could be used by other downstream tasks.
    - Finally, responsiveness to one-time requests for products could be a factor in dividing the processing into multiple tasks.

- Should the algorithm input base data or are there existing intermediate products that can form the basis of input data? Obviously use caution in depending upon intermediate product data from an algorithm stream controlled by another organization.

- LIMITATION: If an algorithm task has multiple product data inputs, the data timing (VOLUME, ELEVATION, RADIAL) of the product inputs are typically the same. There are restrictions on using multiple product data inputs having different data timings. See Part C of this document for more

information.

- Are there any existing *Public* non-product data stores that would be useful?

Types of Persistent Algorithm Data

There are several classifications of persistent data within the ORPG. From an algorithm design / configuration perspective there are three classes of interest: "product data", "non-product data", and "adaptation data". These classes are configured in a different manner.

- Product data stores are implemented as linear buffers and configured using the **`product_attr_table`** configuration file.
- Public non-product data stores are implemented as linear buffers and configured with the **`data_attr_table`** file.
- Private non-product data stores are standard disk files and require no configuration because they are not managed by the ORPG.
- Adaptation data is covered in CODE Guide Volume 2, Document 2, Section IV and Volume 2, Document 4, Section II.

Several Legacy Fortran algorithms use another mechanism of sharing persistent data called 'Inter-Task Communication (ITC) Blocks'. Support for this communication mechanism was implemented in the ORPG infrastructure. Even though the C Algorithm API includes functions to use ITC blocks, they are not recommended for use in new algorithms. In place of 'ITC Blocks', special data access functions are provided to support the non-product data stores described in CODE Guide Volume 2, Document 2 Section III.

Basic Definition:

- Any data distributed to external users via product distribution interfaces are product data. These are called "final products".
- Base data messages from the RDA are product data.
- LIMITATION: In an algorithm where processing is divided among two or more tasks, the tasks must be connected by at least one "intermediate product" (product data). Within the ORPG architecture, intermediate products are a process triggering mechanism for downstream tasks.

Beyond the basic definition, how are algorithm persistent data classified as product or non-product?

Data classification Factors

- Generally data should be classified as **product data** when:
    - The data is derived from or associated with the radar scan. If requested or scheduled for production, this data is modified either every elevation or every volume.
    - The data must be synchronized. The ORPG API product reading functions automatically ensure the data are of the same elevation and/or the same volume as appropriate.

- Generally data should be classified as **non-product data** when:

- The data is some kind of algorithm state data. It may change frequently or infrequently but is not necessarily associated with the radar scan. One example is accumulation data that spans time periods not associated with volume scans.
- Synchronization is not needed. The non-product data access API functions provide no synchronization of data. Depending upon the type of data this may be desirable.

Storage Implementation Factors

- **Product linear buffers** are always configured as a *message queue* type buffer and the algorithm API product reading functions read the messages sequentially until the appropriate elevation / volume data is found. All messages must contain the same type (and structure) of data.

- **Public non-product linear buffers** can be configured in two ways.
  - A *message database* type buffer. The purpose of this type of buffer is to provide a non-sequential message set. The algorithm is responsible for making room for more messages when the buffer is full. With this type of linear buffer messages are written and read with a specified message ID. The user has more control because any specific message can be read or updated (replaced). Beginning with Build 10, the API provides a database style access for use with very large data sets. Each message may contain different types of data.
  - A *message queue* type buffer. The purpose of this type of buffer is to write message sequentially and read message sequentially. When full, the older messages are automatically deleted. This might be advantageous for use even with product data (associated with the radar scan) if there is a need to read previous messages. With the product API, once a message in a message queue buffer is read the message pointer automatically points to the next message. All messages should contain the same type (and structure) of data.

- **Private non-product disk files** are not managed by the ORPG and are accessed via the standard C file input/output library. Private data stores are useful if the data do not need to be shared with other algorithm tasks. They are also useful for data that must be preserved even if the ORPG is shutdown. If data are used by more than one task, the public non-product data store should be considered.

Responsiveness to One-Time Requests

The response time for one-time requests is a concept of operations issue. Normally this consideration only applies to those products that use the 6 product dependent parameters in the request message to customize a product in some fashion. For data driven algorithms reading the original base data (recombined base data) or reading intermediate product data, this is a configuration issue not an algorithm design issue.

Types of Product Requests

There are two basic types of product requests.

- **Routine Requests.** These products are produced every volume. The request is a result of being part of the default product generation list or listed on a Routine Product Set (RPS) list of a Class 1 user.
- **One-Time Requests.** These products are produced once in response to each received one-time request, unless already produced as a result of routine requests.

It does not make sense to have routine requests for some products having content customized with the request parameters. One example are the vertical cross section products. For other customized products, such as precipitation accumulation products, having the product automatically produced could be useful.

Replay Tasks

The purpose of a replay task is to respond more quickly to a one-time request than the normal real-time instance of a task can. A replay task is a second instance of an algorithm task which handles the one-time requests for the output products while the original instance of the task handles the routine requests for the product. For data driven tasks this is handled seamlessly by the infrastructure. For event driven tasks, the algorithm must use an API function to determine which instance it is running as. Replay tasks are configured using the **`task_attr_table`** and **`task_tables`** configuration files (see Volume 2, Document 2, Section III, Part C).

For a replay task to function as intended (that is to respond to the one-time request as soon as possible), the input product data must be immediately available. The current and previous volumes of the original base data (**`BASEDATA`**, **`REFLDATA`**, and **`COMBBASE`**) are available in special buffer for use by replay tasks. Any intermediate product data used by replay tasks must be configured as warehoused with generation assured with a priority of 255.

For tasks using one of the new non-recombined base data streams or a raw data stream (which are not stored in a replay buffer), the algorithm could be divided into multiple tasks with needed intermediate product data warehoused.

There is a tradeoff for obtaining this responsiveness. When satisfying a one-time request, a replay task will use data from the current volume scan if available. If not available, data from the previous volume is used to immediately satisfy the request.

The behavior of a replay task is covered in Document 3, Section I, Part C - *Algorithm Initialization and Control Loop*.

Part C. Algorithm Initialization and Control Loop

This portion of the document is a summary of basic rules for using the algorithm registration and initialization routines and determining which form of the algorithm control loop function to use. Complete guidance for using individual API service calls is contained in Document 2 of this volume.

Control Loop for Data Driven Algorithms

Virtually all of the existing WSR-88D algorithms are dependent on the flow of internal WSR-88D data and generally produce elevation or volume based output. There are a few types of internal message products that are produced on demand but their input data is either elevation based or volume based.

Behavior of Data Driven Control Loop

There are two types of data driven algorithms using either the WAIT_ALL or the WAIT_ANY form of the control loop function.

1. Algorithms with a driving input. These algorithms block until a specific input is available (the first input if more than one is registered) and use **`RPGC_wait_act(WAIT_DRIVING_INPUT)`** for blocking. Each data input is accomplished with an individual call to **`RPGC_get_inbuf_by_name`**.
2. Algorithms with no driving input. These algorithms block until any one of several inputs is available and use **`RPGC_wait_act(WAIT_ANY_INPUT)`** or **`RPGC_wait_for_any_data`** for blocking. The available input is read with a single call to **`RPGC_get_inbuf_any`**.

The release conditions of these two types of algorithms differ significantly.

1. The WAIT_ALL form of the control loop releases when the registered driving input is available and there is a request for at least one of the registered output products.
2. The WAIT_ANY form of the control loop releases when any one of the registered inputs is available. This requires an algorithm to check that the output product is requested. See sample algorithm 4.

Replay Tasks - a special case

The purpose of a replay task is to respond more quickly to a one-time request than the normal real-time instance of a task can. A replay task is a second instance of an algorithm task which handles the one-time requests for the output products while the original instance of the task handles the routine requests for the product.

- Replay tasks can only be used with the WAIT_ALL form of the control loop.
- A normal real-time task will release at most once each elevation with an **`ELEVATION_BASED`** driving input and once each volume with a **`VOLUME_BASED`** driving input.
- A replay instance of a task will release every time a one-time request is received for an output product.

For a replay task to function as intended (that is to respond to the one-time request as soon as possible), the input product data must be immediately available. The current and previous volumes of the original base data (**BASEDATA**, **REFLDATA**, and **COMBBASE**) are available in special buffer for use by replay tasks. Any intermediate product data used by replay tasks must be configured as warehoused with generation assured with a priority of 255.

If the replay task fails or is unable to handle the one-time request from the saved data, the real-time instance can handle the request with the next volume data. For data driven tasks this is handled seamlessly by the infrastructure.

Rules for input / output data registration

Input Data Limits

- Data driven algorithms must be registered for at least one product input and one output product. There are a couple of algorithms where this is not followed, but they should be considered special cases. Data driven algorithms may also register for "external" events.
- There can be only one **RADIAL_DATA** input to an algorithm.
- For the WAIT_ANY form of algorithm loop control, input(s) must not be **RADIAL_DATA**.
- With multiple data inputs, they are generally all **ELEVATION_DATA** or **VOLUME_DATA**. However, a mix of input timing types is allowed with restrictions. For algorithms having the WAIT_ALL form of control loop:
    a. **ELEVATION_DATA** and **VOLUME_DATA** can be mixed if the driving input is **ELEVATION_DATA**. In addition, the programmer must assure that the **VOLUME_DATA** being input is from the same volume as the driving input. One way to accomplish this is to read the volume input after reading all elevation inputs needed.
    b. It is possible to mix **RADIAL_DATA** and **ELEVATION_DATA** inputs in the same manner (used in Sample Algorithm 4). The driving input must be **RADIAL_DATA**. When obtaining the **ELEVATION_DATA** input, the programmer must be sure it is asked for after the acquisition of the first radial of the expected elevation/volume and before the acquisition of the first radial of the next elevation/volume.
- **Normally meteorological algorithms do not produce RADIAL_DATA. However the existing algorithms that do produce radial data have only one input: RADIAL_DATA. It is recommended that algorithms that produce RADIAL_DATA have only one input and the timing of that input is RADIAL_DATA.**
- In the case where the algorithm blocks until any one of several inputs are available (the WAIT_ANY form of the algorithm control loop), the input timing types can be any combination but cannot include **RADIAL_DATA**.

Output Data Limits

- Most algorithms with multiple outputs have either all **ELEVATION_DATA** or all **VOLUME_DATA** outputs. **DEMAND_DATA** is used for data that is not directly related to the volumetric radar data. An algorithm can output **DEMAND_DATA** even if there is no request for it.
- **VOLUME_DATA** and **ELEVATION_DATA** have been mixed and **VOLUME_DATA** and **DEMAND_DATA** have been mixed. In both cases the task timing is **VOLUME_BASED**.

- **Normally meteorological algorithms do not produce `RADIAL_DATA`. However the existing algorithms that do produce radial data have no other outputs. Theoretically there is nothing that prevent mixing `RADIAL_DATA` output with `ELEVATION_DATA` or `VOLUME_DATA`.**

Input Timing Related To Output Timing

- Normally the timing of the input data must be equal to or less than the timing of the output data. For example, a **`VOLUME_DATA`** output could have any input timing with the restrictions noted above. But an **`ELEVATION_DATA`** output should not have a **`VOLUME_DATA`** input. There are two unique exceptions to this.
    a. **SPECIAL CASE:** The **`tdaruprod`** task and the **`mesoruprod`** task. The only reason this makes sense is that these tasks have other **`ELEVATION_DATA`** inputs and also have a **`WAIT_ANY`** type of control loop. In these algorithms the **`ELEVATION_DATA`** input actually drives the production of the **`ELEVATION_DATA`** output. The **`VOLUME_DATA`** input only serves to store this data for use in the processing of the elevation data in the next volume scan.
    b. The **`alerting`** task which is event driven and the **`combattr`** task also mix **`ELEVATION_DATA`** and **`VOLUME_DATA`** inputs but are not exceptions because the output is **`VOLUME_DATA`**.

The task initialization function, **`RPGC_task_init`**, must follow all registrations for input / output data and adaptation data. The task timing type registered by **`RPGC_task_init`** is normally **`ELEVATION_BASED`** if output products are **`ELEVATION_DATA`** and **`VOLUME_BASED`** if output products are **`VOLUME_DATA`**. Task timing cannot be **`EVENT_BASED`** for data driven algorithms.

Control Loop for Event Driven Algorithms

The nature of the events registered affect how the algorithm responds. The following guidance does not cover all possible situations

Normally an event driven algorithm is not used to create products based upon WSR-88D data. However, event driven algorithms are used for special applications that either

1. are not synchronized with the internal data flow of WSR-88D data, or
2. do not have product data as an input.

Basic requirements for an event driven algorithm are:

- Event driven algorithms use the **`RPGC_wait_for_event`** form of the control loop function.
- Event driven algorithms must be registered for at least one event.

There are two classes of event driven algorithms

1. Event driven algorithms that also register for product data inputs must use a task timing compatible with the input data, for example **`VOLUME_BASED`**.
2. Event driven algorithms that are not registered for product data input must use a task timing of **`EVENT_BASED`**.

Behavior of the Event Driven Control Loop

Regardless of where the algorithm is, when a registered event is posted, the logic in the callback function registered for that event is processed immediately. If there is more than one registered event posted to the event queue, the callbacks are executed sequentially.

If the algorithm is holding in the loop control function (**RPGC_wait_for_event**), the callback is executed and the control function releases for the algorithm to begin processing the logic in the main loop. If the algorithm is already processing the main loop, when a registered event is posted, the loop is temporarily interrupted, the callback function is executed, and the main loop continues when the callback returns.

With event driven algorithms, normally all algorithm logic is contained within the event handler callback routine rather than within the while loop containing the control function. Note: In addition to the events explicitly registered, the loop control function (**RPGC_wait_for_event**) releases when the **ORPGEVT_END_ELEV** event is posted. Therefore any logic in the main loop of the algorithm is executed at least every volume regardless of whether a registered event is posted.

Replay Tasks - a special case

The purpose of a replay task is to respond more quickly to a one-time request than the normal real-time instance of a task can. A replay task is a second instance of an algorithm task which handles the one-time requests for the output products while the original instance of the task handles the routine requests for the product. This is handled seamlessly by data driven task but must be handled in the algorithm code for data driven tasks.

An event driven task is used when there are no product data inputs. The even driven algorithm must determine which instance of the task it is to register for the appropriate event. The function **RPGC_get_input_stream** returns the type stream.

- The replay instance of the task registers for the **EVT_REPLAY_PRODUCT_REQUEST** event. The callback is executed every time a one-time request is received for a product. The product request list obtained will only include the one-time requests. The task could also register for other events (for example **ORPGEVT_START_OF_VOLUME** to accomplish processing at beginning of volume).
- The normal real-time instance of the task typically registers for LB notification or **ORPGEVT_START_OF_VOLUME.** The product request list obtained has the routine requests and any one-time requests not serviced by the replay instance.

Rules for input / output data registration

- Input data to event driven algorithms not derived from WSR-88D base data should be configured as **DEMAND_DATA**.
- If input product data is **DEMAND_DATA** or there is no input product data, then the output data must be configured as **DEMAND_DATA**.
- Event driven algorithms must use **RPGC_get_inbuf_any** to read the available data input. More than one data input can be registered but the algorithm can read only one available input using the API.

The task initialization function, **RPGC_task_init**, must follow all registrations for input / output data and adaptation data. If the task type is **EVENT_BASED**, at least one of the event registrations must precede the task initialization.

Task Timing vs. Data Timing for Data Driven Algorithms

Determining a Product's Data Timing

Input and Output data timing is reasonably straight forward. Data are either **RADIAL_DATA**, **ELEVATION_DATA**, or **VOLUME_DATA** based upon how often a product is produced. Usually, but not always, this corresponds to how much data are needed to produce the product..

- Many volume products do not require a full volume scan of base data but use the first several elevations (the 4 lowest elevations is common). Since the product is generated once each volume it is configured as **VOLUME_DATA**.
- There are a few elevation products that use input data from multiple elevations. They accumulate information and provide a product updated each elevation (an example is the MESO Rapid Update product). Since the product is generated once each elevation it is configured as **ELEVATION_DATA**.
- **Normally meteorological algorithms do not produce RADIAL_DATA. Radial intermediate products should only be used in unusual situations, for example a downstream task would be more efficient if its input data were RADIAL_DATA. RADIAL_DATA should never be used for final products.**

The configuration of the output data timing is (almost) always the same as or greater than the input data timing. For example, it makes no sense to have an elevation product produced entirely from volume input data.

Determining a Task's Timing

Selecting the appropriate task timing is almost as straight forward. Often the task timing is the same as the output data timing (a **RADIAL_BASED** task producing a **RADIAL_DATA** product and an **ELEVATION_BASED** task producing an **ELEVATION_DATA** product). However that is not always the case.

The task timing setting only determines what happens if an algorithm must abort product generation and return to the beginning of the algorithm control loop. If a **VOLUME_BASED** task abort product generation, the control loop function will not release until the beginning of the next volume and an **ELEVATION_BASED** task will release at the beginning of the next elevation.

- **Normally the task timing is either the same as or greater than the largest input timing and the same as the output timing.** There are several algorithms where this is not the case.
    a. **SPECIAL CASE:** The **superob_vel** task and the **mdaprod** task have a task timing of **ELEVATION_BASED** and the output product is **VOLUME_DATA**. In other words, the volume product is produced even if there are missing elevations. These are not typical algorithms.

b. **SPECIAL CASE:** The several tasks has a task timing of **VOLUME_BASED** and the output is **ELEVATION_DATA**. Most of these are of the Kinematic algorithm type. This configuration is used if the output products for later elevations in the volume should not be produced if there is any problem with the lower elevations of input data. In other words the downstream tasks cannot handle a missing elevation (e.g., they produce **VOLUME_DATA** products).

- **Though not required by the infrastructure, if a task is producing an intermediate product, the task timing should be set to the greatest output timing of downstream tasks.** Typically this means that if any downstream task produces **VOLUME_DATA** products, it is usually a good idea to register upstream tasks as **VOLUME_BASED** even if only producing **ELEVATION_DATA**. The preferred method is to not return to the beginning of the control loop until all elevations in the volume have been processed (and products output) or an abort has occurred. See the topic *Read All Input Data (If not Aborting)* in Part D of this document.

Handling Multiple Product Inputs

In the case where the algorithm blocks until a specific input is available (the WAIT_ALL form of the algorithm control loop), inputs in addition to the "driving" are typically of the same timing (**ELEVATION_DATA, VOLUME_DATA, DEMAND_DATA**) or must be specified as a time-based input. Time based inputs are not yet supported by the Algorithm API. A mix of timing types is allowed with restrictions:

- **ELEVATION_DATA** and **VOLUME_DATA** can be mixed if the driving input is **ELEVATION_DATA**. In addition, the programmer must assure that the **VOLUME_DATA** being input is from the same volume as the driving input. One way to accomplish this is to read the volume input after reading all elevation inputs needed.
- It is possible to mix **RADIAL_DATA** and **ELEVATION_DATA** inputs in the same manner (has been tested in Sample Algorithm 4). The driving input must be **RADIAL_DATA**. When obtaining the **ELEVATION_DATA** input, the programmer must be sure it is asked for after the acquisition of the first radial of the expected elevation/volume and before the acquisition of the first radial of the next elevation/volume.

In the case where the algorithm blocks until any one of several inputs are available (the WAIT_ANY form of the algorithm control loop), the timing types do not have to be the same.

- Care must be used with the WAIT_ANY form. While the algorithm may produce more than one output product, any product produced can only require one of the registered inputs. Attempting to synchronize multiple inputs will not work

Handling Multiple Product Outputs

With data driven tasks producing multiple outputs, the algorithm must determine which of the outputs has been requested. With data driven tasks using the WAIT_ANY form of the control loop, the algorithm must determine whether a product is requested even if only one product is produced.

For tasks producing products not using customizing parameters in the product request message, the function **RPGC_check_data_by_name** can be used to determine whether a product has been requested. If this function is not used, the function that obtains the output buffer (**RPGC_get_outbuf_by_name**) will return an 'opstatus' other than RPGC_NORMAL if there is no request for the product.

For tasks producing final products using the customizing parameters in the request message the requests must always be obtained. The function **RPGC_get_request_by_name** determines how many requests (if any) there are for the specified product and returns the customizing parameters for each request. There is no need to call **RPGC_check_data_by_name**.

The above can also apply to event driven algorithms.

Task Timing vs. Data Timing for Event Driven Algorithms

The guidance for event driven algorithms is very general. The nature of the events registered affect how the algorithm responds. The following guidance does not cover all possible situations.

Determining a Task's Timing

The task timing for event driven algorithms is **EVENT_BASED** for tasks with no product inputs. For tasks with product data inputs, the timing is appropriate to the type of data input, typically **VOLUME_BASED**.

Determining a Product's Data Timing

If an event driven algorithm has product input data, it should be either **ELEVATION_DATA** or **VOLUME_DATA**, not **RADIAL_DATA**. Currently the outputs of event driven algorithms are **DEMAND_DATA**.

Since event driven algorithms must use **RPGC_get_inbuf_any** to read input buffers, the output product can only use one input at a time, unless the data are stored locally.

Handling Multiple Product Outputs

If the output product is **DEMAND_DATA**, which is typically the case, there is no need to determine whether their is a request for that product.

One case where the output may not be **DEMAND_DATA** could be an algorithm producing timed output data having no product inputs. In this case the task must activate on an event. The output products could be either **EVENT_DATA** or **VOLUME_DATA**.

Part D. Basic Algorithm Structure

The typical reason an algorithm is aborted is an inability to read input data or input data that are out of sequence. All algorithms must do one of the following:

1. **Successfully output a product, or**
2. **Call the appropriate abort function.**

Data driven algorithms must always return to the beginning of the control loop (both WAIT_ALL and WAIT_ANY) upon successful product output and when output is not successful (product abort).

With event driven algorithms, the same principles apply. Event driven algorithms typically have all of the algorithm logic in the event handler function. In this case the event handler function must return upon successful product output and when the output is not successful (product abort) and the algorithm control loop function **RPGC_wait_for_event** is placed in a continuous loop.

The guidance provided for basic algorithm structure apply to both data driven and event driven algorithms that produce product data.

Read All Input Data (If not Aborting)

When successfully completing a product (no sequence error or other problems):

- An algorithm reading base data (or any **RADIAL_DATA** input) and producing **ELEVATION_DATA** output must read until the *actual* end-of-elevation even if data only up to *pseudo* end-of-elevation are needed. **Do not read beyond the *actual* end-of-elevation.**
- An algorithm reading base data (or any **RADIAL_DATA** input) and producing **VOLUME_DATA** output must read until the *actual* end-of-volume even if data only up to *pseudo* end-of-volume are needed. **Do not read beyond the *actual* end-of-volume.**
- An algorithm reading **ELEVATION_DATA** input and producing a **VOLUME_DATA** output must read all elevations produced for that volume by the upstream task, even if fewer elevations are actually used in constructing the product.

The need to read all elevations / radials produced can be avoided only in algorithms that produce a **VOLUME_DATA** output and have a task timing of **VOLUME_BASED**. The function **RPGC_abort_remaining_volscan** can be used to accomplish this. This must be used with the WAIT_ALL form of the loop control rather than with WAIT_ANY.

SPECIAL CASE: There are tasks (most related to tornado detection and mesocyclone detection) that have a task timing less than the output. These tasks are **VOLUME_BASED** but output **ELEVATION_DATA** products. Currently some of these tasks read all input data for a volume scan before returning to the control loop. This is the preferred method and requires reading inputs for each elevation and outputting the elevation product for each elevation before returning to the control loop (unless an abort occurs).

Release All Acquired Buffers

- Before returning to the control loop (or returning from the event handler), all input buffers successfully obtained must be released with **RPGC_rel_inbuf** (or **RPGC_rel_all_inbufs**) and all output buffers successfully obtained must be released with **RPGC_rel_outbuf** (or **RPGC_rel_all_outbufs**). **Note**: Attempting to release an output buffer that was not obtained, while using a disposition of FORWARD, will cause the algorithm task to be terminated (in other words, attempting to output a product with no output buffer will crash the algorithm).
- With successful product construction **RPGC_rel_outbuf** is called with the **FORWARD** flag and when product construction is not complete the **DESTROY** flag is used. In addition, in the case of incomplete product construction one of the abort functions must be called.
- The function **RPGC_cleanup_and_abort** is a convenient method of releasing all input and output buffers before returning to the algorithm control loop. Any output buffer that is open will be released with a disposition of DESTROY.

Call Required Abort Services

This guidance is based upon changes in the ORPG algorithm abort services introduced after the ORPG was initially deployed. Some algorithms have a different pattern of usage based upon the legacy FORTRAN algorithms (which were not completely consistent).

The typical reason an algorithm is aborted is an inability to read input data or input data that are out of sequence. All algorithms must do one of the following:

1. **Successfully output a product, or**
2. **Call the appropriate abort function.**

If all products cannot be successfully completed, at least one abort service must be called. The appropriate abort function is called after releasing the appropriate open output buffer (if any) and prior to returning to the loop control function.

The current guidance is to use **RPGC_abort** when possible. This basic function can be used if the failure is due to the inability to obtain either an input or output buffer and the task only produces one product. In the following situations, the only function available requires an abort code parameter.

- With algorithms producing more than one product type, **RPGC_abort_dataname_because** is used when unable to complete the specified product type but able to continue the generation of other products.
- With algorithms producing products that use customizing data from the product request message, **RPGC_abort_request** is used when unable to complete generation of a product for a specific request message but able to continue the generation of that product for other requests.
- The convenience function **RPGC_cleanup_and_abort** can be used in situations where no remaining requested products can be completed. .

WARNING: The function **RPGC_cleanup_and_abort** has caused task failures in some cases. Until the cause has been determined and resolved, it is recommended that other abort functions be used along with functions releasing appropriate input and output buffers.

A reason code must be passed when the applicable abort function requires a code and when aborting for reasons other than the failure to obtain an input or output buffer. The correct reason code to be passed is:

- Typical Algorithms (those NOT producing `RADIAL_DATA`)
  - If the abort is because of an inability to get a buffer with *get_inbuf* or *get_outbuf*, the **`opstatus`** returned is used as the reason code.
  - If the abort is because not all required base data moments are enabled (in the case where the **`RPGC_what_moments`** test was used), **`PGM_DISABLED_MOMENT`** is the reason code passed. Note: The above test is not required if the required moments are registered (see **`RPGC_reg_moments`**).
  - Beginning with Build 12, **`PGM_DISABLED_MOMENT`** is also used if a needed advanced data field (Dual Pol in the generic moment structure) is not available.
  - If the abort is because of the failure to obtain memory via **`malloc`** or **`calloc`**, **`PGM_MEM_LOADSHED`** is passed.
  - If there is some algorithm unique reason for being unable to complete product generation, then
    - Use the **`PGM_INPUT_DATA_ERROR`** code if problem is generally related to the nature of the input data.
    - Use the **`PGM_INVALID_REQUEST`** code if the problem is related to the nature of the product request parameters.
  - DEFAULT: If the reason for aborting the product generation does not clearly fall into one of the above cases, **`PGM_INPUT_DATA_ERROR`** is a safe default reason code that can be used.
- Algorithms producing **`RADIAL_DATA`**
  - The only time an algorithm producing **`RADIAL_DATA`** aborts is for the failure to obtain an input buffer or output buffer. The status returned by the API function is used as the abort reason so **`RPGC_abort`** is used if the radial output is the only output.
  - For other conditions (disabled basic moment, an advanced Dual Pol data missing, etc.) the radial is passed along. See Special Instructions for Radial Producers below.

Some abort functions permit continued processing while others do not

- After either **`RPGC_abort`**, **`RPGC_abort_because`** or **`RPGC_cleanup_and_abort`** are called, processing (reading input, writing output, etc.) must not continue. The algorithm must accomplish any necessary cleanup and return to the loop control function.
- After **`RPGC_abort_dataname_because`** is called, processing of other registered output products (if any are requested) may continue.
- After **`RPGC_abort_request`** is called, processing of additional requests for that product (if any) may continue.

Special Instructions for Radial Producers

As discussed above, the only time an algorithm producing **`RADIAL_DATA`** aborts is for the failure to obtain an input buffer or output buffer and the reason code is the status of the *get_inbuf* / *get_outbuf* function. With other problems the radial is passed along either with or without processing. The downstream tasks must be aware of this possibility.

When passing along a radial having a data problem (missing needed data moment for example) the producing task has two options.

1. Check the value of the basedata header field **`pbd_alg_control`** field and make sure it is not 0. Here is sample code:

   ```
   bdh = (Base_data_header *) obuf_ptr;
   if( bdh->pbd_alg_control == 0 )
      bdh->pbd_alg_control = PBD_ABORT_INPUT_DATA_ERROR | PBD_ABORT_FOR_NEW_VV;
   RPGC_rel_outbuf( obuf_ptr, FORWARD | RPGC_EXTEND_ARGS, size );
   ```

   This is accomplished when the next downstream task is not a radial producer and is not expected to handle special situations. The downstream task will get a non-normal status from the *get_inbuf* function and therefore abort.

2. Pass the radial and rely on all downstream tasks to be able to make the appropriate decision whether to produce their output. This is accomplished when the next downstream task is a radial producer or when the non-radial producer is expected to handle the special situation.

Resource Cleanup

- All resources / allocated memory must be freed prior to returning to the beginning of the control loop. This requirement also applies to the memory allocated by **`RPGC_get_customizing_data`**.
- Failure to obtain resources (e.g., memory with **`malloc`**) must be tested for and handled gracefully following all rules for returning to the beginning of the control loop.

> NOTE: Not all CODE sample algorithms comply with this item in that not all previously allocated memory is freed before calling abort and returning to the loop control function.

Templates for Basic Algorithms

The flow charts contained in Appendix A illustrate the guidance currently provided in this guide. Many legacy algorithms have not been updated to reflect the changes in abort services. The following flow charts are provided:

1. An algorithm reading base data (radial based) and outputting an elevation based product. This chart provides a framework for Sample Algorithm 1 - Digital Reflectivity algorithm, Sample Algorithm 2 - Radial Reflectivity algorithm, and the first task in Sample Algorithm 3 - Multiple Task algorithm.
2. An algorithm reading an elevation based product and outputting a volume based product. This chart provides a framework for the second task in Sample Algorithm 3 - Multiple Task algorithm
3. An algorithm reading base data (radial based) and outputting a volume based product. (No sample algorithm at this time).

Part E. Additional Topics

Algorithms Producing More Than One Product

With more than one registered output all algorithms must determine which of the registered outputs are requested.

- For tasks producing products not using customizing parameters in the product request message, the function **RPGC_check_data_by_name** can be used to determine whether a product has been requested. If this function is not used, the function that obtains the output buffer (**RPGC_get_outbuf_by_name**) will return an 'opstatus' other than RPGC_NORMAL if there is no request for the product. Using **RPGC_check_data_by_name** is recommended in two situations:
    - For data driven algorithms producing more than one output having a driving input (the WAIT_ALL control loop), the function is used to determine which outputs have been requested.
    - For data driven algorithms that can proceed with only one of the several registered inputs, the WAIT_ANY control loop releases when any one of the inputs is available. In this case, it is possible for none of the registered outputs to be requested.

- For tasks producing final products using the customizing parameters in the request message, **RPGC_get_request_by_name** determines how many requests (if any) there are for the specified product and returns the customizing parameters for each request. There is no need to call **RPGC_check_data_by_name**.

With more than one registered output, multiple output buffers may be opened as long as there is only one buffer of each type open at any given time.

Algorithms with Customized Products

For algorithms whose product depends upon the contents of the product dependent parameters in the request message (customizing information), up to 10 unique requests can be made for that product type. **RPGC_get_customizing_data** obtains this information along with the number of requests, but is limited to algorithms having only one product output. For algorithms producing more than one product, **RPGC_get_request_by_name** (added in Build 10) must be used to obtain request information.

RPGC_get_outbuf_by_name_for_req must be used to obtain the output buffer (one of the input parameters contains the request information for that individual product request).

The output buffers must be obtained and released sequentially since only one output buffer for a given product type can be open at any given time.

Vol 3. Document 3 -
WSR-88D Algorithm Structure and Sample Algorithms

Section II Sample Algorithms

NEW FOR Sample Algorithms 1.22a (Build 12 r1.3)

- Added a sample algorithm default generation file that modifies the product_generation_tables from the Build 12 ORPG.

NEW FOR Sample Algorithms 1.21 (Build 11 Final)

- For Sample 1 and Sample 3, added additional print diagnostics for the generic moment structure header.
- For Sample 2, provided a capability to change the input data from **REFLDATA** to **DUALPOL_REFLDATA**.

NEW FOR Sample Algorithms 1.20 (Build 11)

- Modified Sample 4 Task 1 to properly demonstrate a task with driving input (**WAIT_ALL** control loop) producing more than one product.. Previously the algorithm assumed that both intermediate outputs were scheduled (i.e., both final products requested). The task now handles any combination of outputs requested.
- Modified Sample 3 Task 1
  - to use the new and approved function to determine the last elevation being processed rather than use a Legacy method of reading a flag in the intermediate product. Also eliminated and artificial limitation of only considering the first 5 elevations of intermediate products.
  - to fix a problem where elevations after the first had some blank radials.
- Modified Sample 3 Task 2 to write the new Scale Offset decoding parameters to the threshold level fields in the product.
- Modified Sample 3 tasks 1 and 2 to use a new structure for the intermediate product and task 2 makes a distinction between the intermediate product read and the internal data array.
- For all tasks reading radial base data, added a test to ensure that the algorithm did not attempt to read or copy more bins than are actually in each base data radial.
- For Sample 2 and Sample 3, eliminated overlapping radials by finishing processing at pseudo end of elev / volume and then read to the actual end of elev / volume. Sample 1 was not modified but a note was added referencing the alternative processing. Currently sample 1 uses super res data which is not available before the radials became aligned.
- Eliminated locally defined structures for packets af1f and 16.
- Corrected the value for scale factor in the packet 16 header in Sample 1 and in Sample 3 Task 2.
- Changed intermediate product IDs.

NEW FOR Sample Algorithms 1.19 (Build 11):

- Eliminated compile warnings with Red Hat 5 and Build 11.
- Added ROC RCS header templates to all source files.
- Modified for change to RPGC_is_buffer_from_last_elev.

The Radar Operations Center is enforcing all ANSI C standards beginning with Build 11. The most common impact on development code is that C++ style comments using '`//`' are not permitted.

These algorithms provide an example of algorithm structure and demonstrate use of the API. In addition, they can be used as templates upon which to build more complex algorithms. If the sample algorithms are not installed, existing operational algorithms can be used as examples.

Logic Demonstrated	Sample Algorithm	Operational Algorithm
Producing a 256 level elevation product from radial basedata.	Sample 1	cpc007/tsk013/ bref8bit
		cpc007/tsk014/ bvel8bit
		cpc007/tsk015/ superes8bit
Producing a16-level RLE elevation product from radial basedata.	Sample 2	cpc007/tsk001/ basrflct
		cpc007/tsk002/ basvcty
Producing a volume product from elevation basedata.	None	cpc013/tsk003/ epre
A multiple task algorithm with elevation intermediate products and volume final product. The second task demonstrates a task producing a volume output having to dynamically test for last elevation in a volume.	Sample 3	cpc017/tsk009/ tda2d3d cpc018/tsk005/ tdaprod
A multiple task algorithm with first task having multiple outputs using the **`WAIT_ALL`** form of control loop and the final task having multiple outputs using the **`WAIT_ANY`** form of control loop.	Sample 4	cpc004/tsk006/ recclalg cpc004/tsk007/ recclprods
Using a super resolution basedata input.	Sample 1	cpc007/tsk015/ superes8bit
An event driven task.	None	future build

NOTE: This document makes multiple references to two forms of a data driven algorithm control loop. This is the continuous main loop of an algorithm that is entered after all registrations and initializations are complete.

- The WAIT_ALL form of the control loop is the most common and is used in

algorithms that have only one data input or need more than one data input to create a product. The loop control function used is **RPGC_wait_act(WAIT_DRIVING_INPUT)**

- The WAIT_ANY form of the control loop is used in algorithms having multiple registered data inputs but use only one input to produce a product. There are two functions that can be used: **RPGC_wait_act(WAIT_ANY_INPUT)** and **RPGC_wait_for_any_data(WAIT_ANY_INPUT).**

File Naming Conventions

The configuration management system used by the ROC requires unique filename regardless of the directory in which the file is located. The following suggestion will minimize potential filename conflicts.

- All files for an individual task (files within a **tsk** subdirectory) should begin with a prefix related to the tasks purpose or the product produced by the task.

Required Header Files

There are four header files to be included in main source code file for every algorithm: **rpg_globals.h**, **gen_stat_msg.h**, **rpgc.h**, and **rpgcs.h**. Note that **rpg_globals.h** includes several standard libraries (**stdio.h**, **stdlib.h**, **time.h**, **ctype.h**, and **string.h**) along with numerous ORPG header files (for example **basedata.h** and **a309.h**). Some algorithms include individual header files instead of rpg_globals.h.

If using the coordinate conversion functions, the file **rpgcs_coordinates.h** must be included. If using any **RPGP** helper function, including support functions for the generic data packet, the file **rpgp.h** must be included. The file containing the structures for the generic data packet, **orpg_product.h**, is included by **rpgp.h**. In addition, if the algorithm uses adaptation data, the header file defining that data structure must be included.

Required Linked Libraries

See the makefile discussion in CODE Guide Volume 2, Document 2, Section II, *Compiling Software in the ORPG Environment* for the list of system and ORPG libraries that must be linked.

Overview of sample algorithms

1. Sample algorithm 1 creates a digital geographic product containing the data packet most commonly used for new products - packet 16 the digital radial data array. Using the new super resolution data is demonstrated. An elevation product is created from the input radial base data. Using a registered callback function to read adaptation data is demonstrated. Un related test code is included that was used to test API helper functions. This algorithm has been modified to demonstrate one use of configuring a task name that is different than the executable name.

2. Sample algorithm 2 creates a 16-level geographic product containing the data packet most commonly used in the original products intended for graphic display - packet AF1F the radial data packet. The data in this packet has a maximum of 16 levels and written with a special run-length encoding (RLE) algorithm. The algorithm uses a standard run-length encoding helper functions. An elevation product is created from the input radial base data.
3. Sample algorithm 3 demonstrates a two-task algorithm creating an elevation intermediate product and a volume final product. Currently the intermediate product is a special prototype format. The final product is identical to one of the elevation products produced by sample algorithm 1. The elevation used is determined by adaptation data. In the future this algorithm will demonstrate the "generic product format" data for the intermediate product and the generic data packet 28 (which contains a "generic product format" data) for the final product.
4. Sample algorithm 4 demonstrates a two-task algorithm where the first task creates two elevation intermediate products and the second task creates two elevation final products. Each final product uses one of the intermediate products. The first task demonstrates combining a radial driving input and an elevation optional input with the WAIT_ALL form of the control loop. The second task demonstrates the "WAIT_ANY" form of the control loop.

Part A. Algorithms Producing One Product

SAMPLE 1. Digital Reflectivity Algorithm - Elevation Product from Base Data

The CODE software includes a sample digital reflectivity algorithm. This is a basic algorithm that ingests base data and produces an elevation-based final product. No science is accomplished; the algorithm reads base reflectivity data and creates a graphic product using digital data array packet type. The base reflectivity data are super resolution. This sample algorithm demonstrates the assembly of an ICD product with one layer consisting of a digital data array packet type.

Though not required for this product, the algorithm demonstrates registration for and access of DEA adaptation data. This algorithm also tests multiple API helper functions which can be used as an example of using these functions.

Beginning with Build 9, a distinction is made between a task name and a task executable name. Normally these names are identical. This algorithm demonstrates using two different task names with one executable name to produce two algorithm tasks from one source code.

After installation, the source code for this algorithm is located in **`$HOME/src/cpc305/tsk001`**. The top level source file and primary header file are provided in Part B of the Volume 3 Annex in the file **`annex_v3_sample_alg_code.pdf`** (located in the **`--/pdf_doc/`** directory on the CODE CD) for convenience.

SAMPLE 2. Radial Reflectivity Algorithm - Elevation Product from Base Data, 16 level RLE

The CODE software includes a sample radial reflectivity algorithm. This is a basic algorithm that ingests base data and produces an elevation-based final product. With exception of the product code (message code), this product is similar to the 16-level base reflectivity product (BREF19). This sample algorithm

demonstrates the assembly of an ICD product with one layer consisting of a radial run length encoded packets.

A standard run-length encoding function is used in this algorithm. The ORPG added support for the color table index for the ANSI-C version of the adaptation data services. This sample algorithm does not use this feature.

After Installation, the source code for this algorithm is located in **`$HOME/src/cpc305/tsk002`**. The top level source file and primary header file are provided in Part C of the Volume 3 Annex in the file **`annex_v3_sample_alg_code.pdf`** (located in the **`--/pdf_doc/`** directory on the CODE CD) for convenience. This algorithm does not completely handle the failure of **`malloc`** in that not all previously allocated memory is freed before calling abort and returning to the loop control function.

SAMPLE 3. Two Task Algorithm, with Elevation Intermediate Product & Volume Final Product

The purpose of this sample algorithm is to

1. Demonstrate a two task algorithm with the first task producing an elevation based intermediate product and the second task producing a volume based final product. Please note that NO USEFUL SCIENCE IS ACCOMPLISHED BY THE SECOND TASK.
2. ***A future version of this algorithm will demonstrate using the generic data structure in an intermediate product.*** The selected elevation is determined by adaptation data.

An algorithm reading multiple elevation data messages does not require any sequence checks for missing elevations because this is accomplished by the **`RPGC_get_inbuf_by_name`** call. However, some method is required to assure that the first message read is the first elevation of a volume and that the algorithm does not try to read past last elevation of a volume.

Method 1. When elevation intermediate products are used as inputs to a task producing volume based outputs, the legacy elevation intermediate product included a flag to indicate whether the product was based upon the first elevation in a volume or the last elevation in a volume. Though not recommended, this scheme can be used for new algorithms. Sample algorithm 3 uses this method and the data structure used for the intermediate product includes a "last elevation" flag.

Method 2. One could choose another method by placing responsibility on the second task (the reader of the elevation data rather than the producer of the elevation data). This works if the producer task always produced the same number of elevations in which case the reader simply tracks the number read. This also works if the producer task always produced a product for each elevation in the volume coverage pattern (VCP). The API provides functions that can be used to determine the number of elevations in the VCP currently being used. **However, the future RDA may not always process all elevations in a volume scan so any algorithm reading elevation data and producing volume must use the function `RPGC_is_buffer_from_last_elev`. Because the RDA will determine the last elevation dynamically, this function must be called once each elevation.**

Any algorithm producing an elevation intermediate product for a reading task which produces a volume product may require special configuration of that task. In this example, the data produced by the first task is elevation based (see the **`product_attr_table`** entry for **`SAMPLE3_IP`**) but the task itself is

volume based (**RPGC_task_init(VOLUME_BASED,argc,argv);**). This would be used when the design of the algorithm / down-stream task is based upon the requirement that all elevations are needed to produce valid data for processing (no missing elevations permitted). This configuration is generally used when a down-stream task produces a volume-based product.

NOTE: Prior to Sample Algorithm version 1.20:

- The second task in the sample algorithm was parameterized to only consider the first 5 elevation cuts. This artificial limitation has been removed.
- The second task used the "last elevation flag" (Method 1 above). This has been replaced by using the approved API function **RPGC_is_buffer_from_last_elev**. (Method 2 above).

Adaptation data provides a value that represents the elevation which the final product is based. The second task stops reading input and produces the final product at this elevation. The requirement for the second task to read all elevations produced by the first task is avoided through the use of **RPGC_abort_remaining_volscan**.

After installation, the source code for this algorithm is located in **$HOME/src/cpc305/tsk003** and **$HOME/src/cpc305/tsk004**. The top level source file and primary header file for task 1 and the top level source file and primary header file for task 2 are provided in Part D of the Volume 3 Annex in the file **annex_v3_sample_alg_code.pdf** (located in the **--/pdf_doc/** directory on the CODE CD) for convenience. This algorithm does not completely handle the failure of **malloc** in that not all previously allocated memory is freed before calling abort and returning to the loop control function.

Sample Algorithm - Volume Product from Elevation Base Data

FUTURE EDITION OF CODE

Part B. Algorithms Producing More Than One Product

SAMPLE 4. A two task Algorithm, the second task producing two products, each dependent upon a different input

NOTE: Prior to Sample Algorithm version 1.20, the first task always assumed that both intermediate product outputs were scheduled (requested). This required that the both final products produced by the second task were always requested. This was not a realistic assumption. Task 1 was rewritten to demonstrate the requirement to determine which of the intermediate products were requested before acquiring the output buffer and before accomplishing product construction.

The first task produces two elevation based intermediate products and the second task produces two elevation based final products. This sample algorithm demonstrates several special cases.

1. The second task produces a final product output with either one of two registered inputs. This requires the second task use the WAIT_ANY form of the control loop and use **RPGC_get_inbuf_any** to read the data. The WAIT_ANY loop control function will release when any input is available regardless of whether any of the outputs have been requested.
2. The second task has two registered output products. Output 1 depends upon input 1 and output 2 depends upon input 2.
3. The first task demonstrates combining radial driving input with an elevation input and demonstrates configuration of an optional input (the elevation input).
4. The first task also demonstrates producing more than one output with a driving input (the WAIT_ALL form of the control loop. This requires determining which of the registered output intermediate products has been requested. (Sample Algorithm version 1.20 and later)

The optional input to the first task is **SAMPLE3_IP(1989)** produced by sample 3. The **opt_prods_list** attribute in the **product_attr_table** is used to configure that input as optional. Beginning with Build 10 the default delay is 5 seconds. The function **RPGC_in_opt_by_name** is used to set the delay to 8 seconds.

As stated earlier, when reading input using the WAIT_ANY form of the control loop, one can never assume any of the registered output products is requested. Other algorithms reading this input data could generate requests for elevations not needed by the second task in this algorithm.

This sample algorithm creates no useful products. The intermediate products contain simple text data and the final products are simple graphic products with a single text data packet in the first layer of the symbology block.

After installation, the source code for this algorithm is located in **$HOME/src/cpc305/tsk005** and **$HOME/src/cpc305/tsk006**. The top level source file and primary header file for task 1 and the top level source file and primary header file for task 2 are provided in Part E of the Volume 3 Annex in the file **annex_v3_sample_alg_code.pdf** (located in the **--/pdf_doc/** directory on the CODE CD) for convenience.

Part C. Algorithms with Customized Products (Using Product Dependent Parameters)

Sample Algorithm -

FUTURE EDITION OF CODE

Vol 3. Document 3 -
WSR-88D Algorithm Structure and Sample Algorithms

Section III Writing Product Data Fields

BUILD 11:

Updated the discussion of alignment of C structures in Part C. Corrected the details concerning when the compiler pads the end of the structure.

Part A. Introduction

Issue 1: ORPG Infrastructure Byte-Swapping

The standard format of WSR-88D product messages is Big Endian format. This is a result of the Legacy RPG and the initial deployment of the ORPG using a Big Endian hardware architecture. An infrastructure byte-swapping mechanism was implemented when the ORPG was ported to a Little Endian platform in part to avoid modification of all of the algorithms. The ORPG infrastructure accomplishes necessary byte-swapping of the product before it is written to the product data base. However, the data must be written into the final product in a specific manner in order for the byte swapping to be successful.

WSR-88D final products consist of four types of data fields:

- 4-byte integer data
- 4-byte floating point data
- 2-byte integer data
- 1-byte integer data

NOTE: Currently, there is only one floating point data field used by several base data products which is being phased out. Future algorithms can define floating point data (in the header) if placed into two consecutive halfwords within the product dependent parameters portion of the product description block.

The nature of the byte-swapping of data fields is primarily a result of the implementation of the original product algorithms in which the method of writing data fields was not accomplished in an entirely consistent manner. The infrastructure and all algorithms were modified to provide a consistent method of reading and writing 4-byte integer and floating point data fields. The swapping of 2-byte data fields in the header portions of the product (not the data packets) is accomplished in a consistent manner.

There are approximately 30 data packet types defined for use in WSR-88D products. The byte-swapping accomplished on the 2-byte and 1-byte data fields in these packets requires specific guidance for writing each data packet type.

Issue 2: User-Defined Structure Alignment

Interface Control Documents (ICDs) specify the structure of WSR-88D messages down to the position of individual bytes. The messages are in Big Endian, or 'network', format. The original algorithms written in FORTRAN used offsets into a buffer to write individual data fields. With C, it is convenient to use structures to access the individual data fields by name. This works if the structures are "aligned".

Part B. Final Product Byte-Swapping

Writing Product Data Fields

As a result of ORPG Byte-Swapping of Final Product Data Fields: For an algorithm to work on both a Big Endian and a Little Endian Machine:

1. The functions **`RPGC_set_product_int`** and **`RPGC_set_product_float`** must be used to correctly write 4-byte data fields to the final products. Subsequently, the functions **`RPGC_get_product_int`** and **`RPGC_get_product_float`** can be used to read these 4-byte fields.
2. The algorithm API helper functions provided for construction of the message header block (MHB) and the product description block (PDB) portions of the product should be used. These functions utilize the "set_product_int" function to correctly write 4-byte fields.
3. In general, product data fields represented by a 2-byte halfword are written as a short rather than individual bytes. This is true for all header fields (including the MHB, PDB, and the headers for the symbology block, symbology layers, GAB, GAB pages, and the TAB).
4. However, there is no general rule that can be stated for writing 2-byte or 1-byte data fields at the data packet level. The individual data packets used within the symbology block, the GAB, and the TAB are not written in a completely straight forward manner and the guidance provided in the next paragraph must be followed.

Instructions for Writing WSR-88D Data Packets

The Legacy algorithms wrote data packets in a consistent fashion, that is each packet type was always written in the same manner. Most data are written as shorts (halfwords) but there is no simple rule for describing the exceptions. It would be convenient if all 2-byte data were written as a short (a halfword) and all 1-byte data written in individual bytes. This is not the case. In some cases 1-byte data are written as a halfword (a short) and there are even cases where 2-byte data are written as individual bytes.

The structure of all data packets is provided in the Interface Control Document (ICD) for the RPG to Class 1 User. A copy of this document is provided with Volume 2 of the CODE Guide.

The table below contains guidance for writing WSR-88D data packets and can be summarized as follows.

1. The majority of WSR-88D data packets require that all of their data fields be written as shorts (halfwords). In some cases this includes 1-byte data.
2. The following data packets include some data fields that must be written as individual bytes: 1, 8, 12, 15, 21, 26
3. Data packet 28 is unique in that helper functions are used for all but the packet ID. Functions are provided to aid in assembling the packet and for writing integer, float, and string parameters.
4. A helper function is provided to write the radial portion of data packet 16 (this permits the radial data values to be processed as individual bytes). Helper functions for the run-length encoded

(RLE) data in packets AF1F, BA0F, and BA07 are provided beginning with Build 9. CODE sample algorithm 1 & 2 use these functions.

Packet Number	Description	Instructions for writing data fields
1	Text (No Value)	Character data are written as individual bytes. All other data written as shorts (halfwords).
2	Special Symbol (No Value)	All shorts (halfwords), including the character / symbol data.
3	Mesocyclone	All shorts (halfwords).
4	Wind Barb	All shorts (halfwords).
5	Vector Arrow	All shorts (halfwords).
6	Linked Vector (No Value)	All shorts (halfwords).
7	Unlinked Vector (No Value)	All shorts (halfwords).
8	Text (Uniform Value)	Character data are written as individual bytes. All other data written as shorts (halfwords).
9	Linked Vector (Uniform Value)	All shorts (halfwords).
10	Unlinked Vector (Uniform Value)	All shorts (halfwords).
11	3D Correlated Shear	All shorts (halfwords).
12	TVS Symbol	The header portion is written in shorts (halfwords). This looks strange but, the 2-byte (halfword) I-position and J-position data fields are written as individual bytes.
13	Hail Positive	*Packet no longer used.*
14	Hail Probable	*Packet no longer used.*
15	Storm ID	The header portion is written in shorts (halfwords). This looks strange but, the 2-byte (halfword) I-position and J-position data fields are written as individual bytes. The two-character Storm Id is written as individual bytes.
16	Digital Radial Data Array	All shorts (halfwords), including the 8-bit data fields. **`RPGP_set_packet_16_radial`** should be used to write the radial portions of this data packet. This permits using a byte array (char*) while processing the data and then avoiding a byte swap on a Linux platform.
17	Digital Precipitation Data Array	All shorts (halfwords), including the run-length encoded data.
18	Digital Rate Array	All shorts (halfwords), including the run-length encoded data.
19	HDA Hail Data	All shorts (halfwords).
20	MDA Data	All shorts (halfwords).

21	Cell Trend Data	The two-character cell ID is written as individual bytes. The remainder of the product is written as shorts (halfwords). Note that the trend data contains two-character Storm IDs that must be written as a short.
22	Cell Trend Volume Scan Times	All shorts (halfwords).
23	SCIT Past Data	All shorts (halfwords).
24	SCIT Forecast Data	All shorts (halfwords).
25	STI Circle	All shorts (halfwords).
26	ETVS Symbol	The header portion is written in shorts (halfwords). This looks strange but, the 2-byte (halfword) I-position and J-position data fields are written as individual bytes.
27	Superob Data	All shorts (halfwords).
28	Generic Product Format Data	The packet ID is written as a short (halfword). **`RPGC_set_product_int`** must be used for the 4-byte length field. Several helper functions were added in Build 8 that should be used to construct this data packet. Functions are provided to aid in assembling the packet and for writing integer, float, and string parameters. These functions are listed in CODE Guide Volume 3, Document 2, Section II.
AF1F	Radial Run-Length Encoded Data	All shorts (halfwords), including the run-length encoded data. *Currently product 137 (ULR) does not follow this pattern and should not be used as an example.* **`RPGC_run_length_encode`** should be used to pack the data array into the data packet.
BA0F/ BA07	Raster Run-Length Encoded Data	All shorts (halfwords), including the run-length encoded data. **`RPGC_raster_run_length`** should be used to pack the data array into the data packet.

Part C. User-Defined Structures

The CODE documentation generally avoids covering basic C language programming topics. However, structure alignment is covered because it is related to the successful construction of final products.

What is 'alignment'?

Defining C structure alignment is not straight forward. The short answer is that data structure alignment is implementation dependent (compiler / operating system / cpu combination). With the CODE ORPG environment the description of alignment can be fairly well defined.

User Defined C Structures

With C, it is convenient to use structures to access individual data fields by name. C language structures must be used carefully if used to write data into (or read from) final product messages. It is imperative that alignment of 1-byte, 2-byte, and 4-byte data fields be followed.

1. The starting position of 8-byte fields (**`int`** or **`float`**) must be offset from the beginning of the structure by a number of bytes evenly divisible by 8. If not, the structure will be padded to accomplish this.

2. The starting position of 4-byte fields (**`int`** or **`float`**) must be offset from the beginning of the structure by a number of bytes evenly divisible by 4. If not, the structure will be padded to accomplish this.

3. The starting position of 2-byte fields (**`short`**) must be offset from the beginning of the structure by a number of bytes evenly divisible by 2. If not, the structure will be padded to accomplish this.

4. 1-byte fields (**`char`**) must be used in pairs. The first 1-byte field in a series of 1-byte fields must be offset from the beginning of the structure by a number of bytes evenly divisible by 2. If not, the structure will be padded to accomplish this.

5. Padding is sometimes added at the end of the structure even if all of the fields are aligned according to items 1 - 4 above. This sometimes (not always) occurs if the total size of the data fields is not divisible by 4. The only way to reliably determine this is to use the **`sizeof`** operator and look at the results.

If items 1 - 4 are not true, data fields will not be written in the expected location in the product buffer. The memory block containing the C structure will be padded wherever needed to place the 1-byte, 2-byte, and 4-byte data at the aligned offset from the beginning of the memory block.

If item 5 is not true, the result of the **`sizeof`** operator may or may not be correct. The memory block containing the C structure could be padded at the end force the size of the total structure to be evenly divisible by 4. The only way to reliably determine this is to use the **`sizeof`** operator and look at the results. See *End of Structure Padding* at the end of this section.

Using Structures with the Final Product Message

It should be noted that the structure of the final product message, described in Volume 2, Document 3, is only partially aligned (from the perspective of 4-byte integers). This results in having logical or repeating portions of a product that cannot be represented by a C structure.

a. There are no 8-byte data fields in the final product message.

b. The product message follows items 3 and 4, which results in 2-byte and 4-byte data beginning at byte offsets evenly divisible by 2.

c. Item 2 is not followed. The 4-byte fields may not begin with a byte offset evenly divisible by 4. This is partially a result of the Legacy product definition dividing 4-byte integers into two 2-byte integers called halfwords. However, the individual C structures defined for various portions of the product comply with this alignment requirement.

d. Item 5 is not followed (currently not a problem in the CODE ORPG environment). The total size of the product is not required to be evenly divisible by 4. .

e. There is no alignment required for the serialized data portion of data packet 28, the generic product data packet. Specified C structures must be used when creating or reading the serialized data.

ORPG Pre-defined C Structures

Most of the C structures related to product message content that are defined in the basic ORPG include files are fully aligned. A few are aligned with the exception of item 5, the sum of the size of the data fields is not evenly divisible by 4. However, only one of these (**`Graphic_alpha_block`**) is padded at the end causing an invalid result from the **`sizeof`** operator.

Here is a brief summary of predefined structures that can be used in construction of the final product. Only one structure representing the GAB header requires special handling as a result of the value returned by the **`sizeof`** operator.

Base Data Headers

The structure **`Base_data_header`** (defined in **`basedata.h`**) representing the header portion of the internal radial base data message is partially aligned. The structure must be used to read the header fields. The '**`sizeof`**' operator must be used if skipping over the header because this structure is subject to change.

The structures supporting the elevation base data message are defined in **basedata_elev.h** . Both structures, **Compact_radial** and **Compact_basedata_elev**, are completely aligned.

Internal Header

The structures representing the internal 96-byte header are completely aligned. These are **Prod_gen_msg** and **Prod_header** which are defined in **prod_gen_msg.h**.

Product Headers MHB & PDB

A structure **Graphic_product** (defined in **product.h**) representing the combined MHB an PDB is completely aligned.

Symbology Block Header

A structure **Symbology_block** (defined in **product.h**) representing the combined symbology block header and first layer header is completely aligned.

The GAB Header

A structure **Graphic_alpha_block** (defined in **product.h**) representing the combined GAB header and first page header does not comply with item 5. Currently, the **sizeof** operator returns a value that is 2-bytes larger than the sum of the data fields.

The TAB Header

A structure **Tabular_alpha_block** (defined in **product.h**) representing the first 8 bytes of the TAB is completely aligned.

Text Packet 1

Structures defined in **packet_1.h** supporting the construction of text packet 1 are completely aligned. These are: **packet_1_hdr_t**, **packet_1_data_t**, and **packet_1_t**.

Text Packet 8

Structures supporting the construction of text packet 8 are defined in **packet_8.h**. Structure **packet_8_hdr_t** is completely aligned and structure **packet_8_data_t** and **packet_8_t** do not comply with item 5 but this currently is not an issue.

Vector Packet 10

Structures supporting the construction of vector packet 10 are defined in **`packet_10.h`**. The structure **`packet_10_hdr_t`** does not comply with item 5 but this currently is not an issue. The structure **`packet_10_data_t`** is completely aligned.

Storm ID Packet 15

Structures supporting the construction of data packet 15 are defined in **`packet_15.h`**. Structure **`packet_15_hdr_t`** is completely aligned and structure **`packet_15_data_t`** and **`packet_15_t`** do not comply with item 5 (currently this is not an issue).

Packet 16 - Digital Radial Data Array

Structures supporting the construction of radial data packet 16 are defined in **`packet_16.h`**. Both structures, **`Packet_16_hdr_t`** and **`Packet_16_data_t`** do not comply with item 5 (currently this is not an issue).

Point Feature Packet 20

Structures supporting the construction of point feature packet 20 are defined in **`packet_20.h`**. All structures, **`packet_20_hdr_t`**, **`packet_20_point_t`**, and **`packet_20_t`**, are completely aligned.

Packet AF1F - RLE Radial Data

Structures supporting the construction of radial data packet AF1F are defined in **`packet_af1f.h`**. The structure **`packet_af1f_hdr_t`** does not comply with item 5 which is currently not an issue. The structure **`packet_af1f_data_t`** is completely aligned.

End of structure Padding

It is difficult to predict when the sizeof operator will return an incorrect value for a structure that completely aligned except the sum of size of all fields is not evenly divisible by 4 (that is, the compiler padded the end of the structure). For example:

The expression **`sizeof(Graphic_alpha_block)`** returns 12 instead of 10.

```
typedef struct {
    short divider;
    short block_id;
    int   block_len;
    short n_pages;
} Graphic_alpha_block;
```

But the expression **sizeof(packet_15_data_t)** correctly returns 6.

```
typedef struct
{
   short  pos_i;
   short  pos_j;
   char   char1;
   char   char2;
} packet_15_data_t;
```

The answer may be that if the structure is otherwise completely aligned (items 1-4 followed), if the structure contains a 4-byte data element, the total length of the structure must be evenly divisible by 4 (item 5). Otherwise it is sufficient for the total length to be evenly divisible by 2.

If Item 5 was a factor as with the struct **Graphic_alpha_block**, the easiest approach is to subtract the appropriate number of bytes from the result of **sizeof**.

```
A_struct_t  *my_struct;
```

a. If allocating space for a structure in order to use in writing temporary data, the results of the **sizeof** function must be used, for example. **my_struct = malloc(sizeof(A_struct_t));**
b. The result of **sizeof(A_struct_t)** cannot be used as an offset into a large memory block representing a final product in order to skip over the data contained in the data fields represented by that structure.
c. If copying data from an internal data structure, **my_struct**, into a block of memory representing the final product, the result of **sizeof(A_struct_t)** should not be used to represent the number of bytes copied. If **sizeof(A_struct_t)** is used, the additional padded data is also copied.

Volume 3. WSR-88D Algorithm Programming Guide

Document 4. Special Topics

This document contains miscellaneous topics that have not been placed in other documents. As this type of information increases, a reorganization of documentation will be considered.

Section I Topics Related to Using the Development Environment

Section II Topics Related to Reading Radial Base Data

Section III Topics Related to Writing Algorithms

Vol 3. Document 4 -
Special Topics

Section I Topics Related to Using the Development Environment

BUILD 12 CHANGES:

- Added guidance for measuring algorithm CPU usage. Part B.
- Added guidance for measuring algorithm Memory usage. Part C.

Part A. Algorithm Error Messages

Writing Error Messages to the Log File

First, it should be pointed out that standard output and standard error should be used only for debugging purposes. After an algorithm is implemented, if there is a desire to capture error or information messages, the **`RPGC_log_msg`** service should be used. **If using standard print statements, the unbuffered standard error** (**`stderr`**) **via `fprintf` must be used.** Standard output (**`stdout`**) will not appear in the configured output file.

Location of Error Messages

When running algorithms that are integrated into the ORPG, neither standard output (**`stdout`**) nor standard error (**`stderr`**) appear at the console. This is the case if the algorithm task is started by the ORPG launch process. There are special error log files for every ORPG task located in the log directory. This directory is defined by the environmental variable **`LE_DIR_EVENT`** and by default is set to a directory **`logs`** under the configured ORPG data directory (**`$ORPGDIR`**). Each task has two files named according to the **`task_name`**. The file **`<task_name>.output`** contains **`stderr`** (it does not contain **`stdout`**). The file **`<task_name>.log`** contains the output of the ORPG Log Error (LE) library message calls used in the ORPG infrastructure software and the output of **`RPGC_log_msg`** used by algorithms.

If the algorithm is started from the command line rather than with the ORPG start command, the standard output and standard error messages appear at the console and the LE messages (including those resulting from **`RPGC_log_msg`**) are not written to the **`.log`** file. Note, even when starting algorithms from the command line, the **`product_attr_table`** and **`task_attr_table`** configuration files must be modified for the algorithm task to launch.

Reading the Log File

The contents of the log file **`<task_name>.log`** cannot be read my most text editors. A utility is provided. While the ORPG is running, at the command line in a terminal window, execute

```
lelb_mon <task_name>
```

The contents of the log file is continuously updated until **`Ctrl-C`** is pressed or the terminal window is closed.

Part B. Algorithm CPU Usage

The following Guide for measuring an algorithm's CPU usage is provided by the ROC in the file: **`~/doc/mon_cpu_use.doc`**.

Measuring Process CPU Utilization On The RPG

1. Overview:

To measure the CPU utilization of a task in the RPG environment, use the *mon_cpu_use* tool. This tool provides CPU time and utilization on an elevation by elevation scan basis. Output from this tool will be used by the ROC to estimate RPG resource impacts well before the algorithm becomes available operationally.

2. How to use *mon_cpu_use:*

To use the *mon_cpu_use* tool, one needs to know the Process ID (*pid*) of the process you want to monitor. There are several ways to get the *pid*. The easiest method is to use *rpg_ps*:

rpg_ps | grep taskname

where *taskname* is the task in question. For example,

rpg_ps | grep recomb

recomb	6931	2680m	28M	19h	recomb -l 1000

The pid is the number in the second column and appears just after the task name. In the above example, the *pid* for *recomb* is *6931*.

The *mon_cpu_use* command line is of the following form:

mon_cpu_use [options] pid

mon_cpu_use supports the following options:

-V Specifies the number of volume scans to montior. By default, the tools runs until the user kills the tool (i.e, Ctrl-C, Ctrl-C).

-C Specifies the CPU capability number. This number is the value output from *use_resource -t*.

A complete listing of command line options and information related to these options can be obtained by running the tool using the *-h* option, i.e., **mon_cpu_use -h**.

mon_cpu_use produces output similar to the following example:

>>>> mon_cpu_use: CPU Use Monitoring Tool for RPG Applications <<<<

CMD LINE: recomb -l 1000

 # CPUs: 2

UTILIZATION SCALE FACTOR: 1.013

Volume: # 1, VCP: 121, Date: 10/24/08, Time: 15:19:11

Date/Time	El #	Elapsed Time (s)	Utilization (%)	Est (%)
10/24/08 15:19:11	1	20	0.00	0.00
10/24/08 15:19:31	2	19	0.00	0.00
10/24/08 15:19:50	3	14	0.04	0.04
10/24/08 15:20:04	4	17	0.06	0.06
10/24/08 15:20:21	5	19	0.03	0.03
10/24/08 15:20:40	6	19	0.03	0.03
10/24/08 12:20:59	7	14	0.07	0.08
1024/08 15:21:13	8	18	0.00	0.00
10/24/08 15:21:31	9	19	0.03	0.03
10/24/08 15:21:50	10	14	0.00	0.00
10/24/08 15:22:04	11	18	0.00	0.00
10/24/08 15:22:22	12	17	0.03	0.03
10/24/08 15:22:39	13	14	0.04	0.04
10/24/08 15:22:53	14	17	0.00	0.00
10/24/08 15:23:10	15	23	0.00	0.00
10/24/08 15:23:33	16	13	0.00	0.00
10/24/08 15:23:46	17	20	0.00	0.00
10/24/08 15:24:06	18	13	0.08	0.08
10/24/08 15:24:19	19	13	0.04	0.04
10/24/08 15:24:32	20	22	0.04	0.04
Totals:	20	343	0.02	0.02

The ***CMD LINE*** line of the output specifies the command line used to invoke the process being monitored. In the example, the ***CMD_LINE*** shows the process being monitored was ***use_resource –c 10***. The next line, ***# CPUs***, specifies the number of CPUs detected on the machine where the process is being monitored. In this example, the machine used to monitor the ***use_resource*** process had an Intel Dual Core processor (2 CPUs). The third line, **UTILIZATION SCALE FACTOR**, is a scaling factor used by *mon_cpu_use* to estimate what the processor utilization would be on baseline

RPG equipment.

For each volume scan monitored, the volume start date and time, VCP number and volume scan number are specified. Following this header, information about each elevation cut in the VCP is listed: start of elevation time, elevation number, elevation scan duration (secs), CPU time (secs) and CPU utilization (%).

At the end of a volume scan, volume scan totals are listed: accumulated elapsed time for the volume scan (sec), accumulated CPU time for the volume scan (sec) and average utilization for the volume scan (%).

3. Measuring CPU Utilization:

To provide CPU utilization for a process, one first needs to run the *use_resource* tool. This tool will provide a estimate of the CPU capabilities of the machine being used. The output of this tool will be used by the *mon_cpu_use* tool to estimate the processing load of the process in question on baseline RPG equipment.

Step 1:

Run ***use_resource -t*** from the command line. This command should be run when the RPG is not busy. Ideally it should be run when the RPG is not processing radar data.

The output of *use_resource* will be a 5 minute average CPU capability value. It is a measure of the number of times a set of complex code can be executed per second. The higher the value, the higher the capability.

This number will be written to the screen and also to a hidden file in the $HOME/cfg directory, *.CPU_capability*. (For more information on the use of *use_resource*, invoke the command using the *-h* switch.)

Step 2:

Start the RPG using ***mrpg -p startup***. In a terminal window, start **script mon_cpu_use.out**

Step 3:

Start the *mon_cpu_use* tool, specifying the number of volume scans to monitor (10) and optionally, the CPU capability number (Note: If the ***$HOME/cfg/.CPU_capability*** file exists, the CPU capability number will be read from this file by *mon_cpu_use*. Otherwise, this number will need to be specified on the command line using the -C option. If the file does not exist and a value is not specified on the command line, the ***UTILIZATION SCALE FACTOR*** defaults to 1.0).

A typical command line for invoking this tool is:

mon_cpu_use -V 10 <pid>

where <pid> is the process ID of the process you are monitoring.

Step 4:

Start playback the full load data set (this data set will be specified elsewhere).

Step 5:

After ten (10) volume scans, the *mon_cpu_use* tool will stop. End the terminal logging by ***script*** using CTRL-D. Note: Ten (10) volume scans is generally sufficient to get a clear picture of CPU utilization. However some algorithms may require longer monitoring periods. For example an algorithm that produces output based on clock time, the monitoring period needs to encompass the time when the output is produced.)

4. What The Numbers Mean and How Are They Used:

Many factors determine the CPU utilization of a process. One cannot simply compare the clock speeds between processors unless the PC architecture between different machines are identical in all respects expect the clock speed. Instruction architecture, cache size, bus speed, compiler optimization (there are many others) affect how fast a process executes on a particular machine. For this analysis, we use a simple technique to provide an estimate of the CPU needs for the process being monitored as if this process were being measured on the actual baseline hardware. We assume a good estimate of process utilization on baseline RPG hardware can be obtained by scaling the process utilization on the development hardware by:

*(CPU Capability(hardware) / CPU Capability(RPG)) * (# CPUs (development) / # CPUs (RPG))*

The output from *mon_cpu_use* provides the CPU utilization of a process on an elevation by elevation basis as well as an estimate of the CPU cost as if the process were running on an actual fielded RPG. This information will be used ROC to determine the impacts of this algorithm on processing load before the algorithm is integrated into the baseline software. The ROC uses this information to determine if the processing reserve of an RPG can support the new algorithm. The ROC can also provide feedback to the developer/implementer concerning the need for possible efficiency improvements to make better use of the RPG shared resource.

Part C. Algorithm Memory Usage

The following Guide for measuring an algorithm's Memory usage is provided by the ROC in the file: **`~/doc/mon_mem_use.doc`**.

Measuring Process Memory Utilization On The RPG

1. Overview:

To measure a the memory footprint of a task in the RPG environment, use the *mon_mem_use* tool. This tool provides private and shared memory utilization used by a process. Output from this tool will be used by the ROC to estimate RPG resource impacts well before the algorithm becomes available operationally.

2. How to use *mon_mem_use*:

To use the tool, one needs to know the Process ID (*pid*) of the process in question. There are several ways to get the *pid*. The easiest method is to use *rpg_ps*:

rpg_ps | grep *taskname*

where *taskname* is the task in question. For example,

rpg_ps | grep tvsprod

tvsprod	8159	0m	4488K	144s	tvsprod

The pid is the number in the second column and appears just after the task name. In the above example, the *pid* for *tvsprod* is *8159*.

The *mon_mem_use* command line is of the following form:

mon_mem_use [options] <pid>

where *mon_mem_use* supports the following options:

-l produces memory utilization summary and detailed listing of utilization

-r specifies a periodic update rate: the default rate is 0 or “one and done”

-m specifies a monitoring period: the default period is 0 or “one and done”.

Note: The -r and -m command line options are intended to be used together.

A complete listing of command line options and information related to these options can be obtained by running the tool using the *-h* option, i.e., **mon_mem_use -h**.

mom_mem_use <pid> produces out similar to the following example:

CMD LINE: tvsprod

VMSIZE: 6432 kB

RSS: 2668 kB total

SHARED: 2068 kB total

PRIVATE: 600 kB total

The first line of the output specifies the command line used to invoke the process being monitored. In this example the command line was *tvsprod*.

The second line of the ***mon_mem_use*** output is the ***V****irtual* ***M****emory* ***SIZE***. (Note: 1 kB = 1024 bytes). The virtual memory size is reported by many tools including *top* and *ps*. This number is mostly meaningless because it does not represent the actual memory needs of a process. It is included here for completeness. For those interested in seeing a map of the virtual address space of a process, use the -l command line option (see Section 5).

The third line of the ***mon_mem_use*** output is the ***R****esident* ***S****et* ***S****ize*. This important measure gives the size of process address space physically in RAM. This size however includes address space private to the process and address space shared by other processes (e.g., shared library text segment ... i.e., code). It does not give a complete picture of the memory footprint of a process. This will be discussed more in detail later.

In the absence of memory paging activity owing to a lack of physical memory to support all application needs, the ***RSS*** represents the process's working set. The working set is the address space referenced by processes during processes execution. If the working set size is smaller than the resident set size, page faults occur. The process at this point can do no useful work until the referenced pages are in physical memory. For systems short on physical RAM, loading pages into RAM requires the replacement of (or paging out) pages used by some other process. Frequent page replacement is referred to as thrashing.

If the working set size matches the ***RSS***, the ***RSS*** value should remain nearly constant. If the RSS changes over time, paging activity may be occurring along with performance degradation.

The fourth line (***SHARED***) of the ***mon_mem_use*** output is the amount of the ***RSS*** shared by other processes. Memory shared by other processes is generally shared library code (executable instructions) but may also include data within shared memory segments.

Shared libraries can be viewed as consisting of two (2) parts: text (code) and data. The text part

consists of the executable instructions. The data part consists of the data on which the code operates. The code can be shared among processes since the code is not writable (can only be read). The data portion of shared libraries (e.g., calculations performed on the processes behalf when calling a shared library routine) cannot be shared.

The last and final line (***PRIVATE***) of the ***mon_mem_use*** output is the amount of the ***RSS*** that is private to the process. This includes process code and data as well as the data portion of shared libraries.

The ***PRIVATE*** size is the best measure of the memory footprint of a task running in the RPG environment.

3. Measuring Memory Utilization:

To provide memory utilization for a process, one needs to run the *mon_mem_use* for a period of time. The amount of time will depend on when and how often the process produces output. For elevation-based and volume based product generators, running for a period of 20 minutes (-m 1200) and a periodic update rate of 1 minute (-r 60) is sufficient.

The following steps describe how to measure memory utilization.

Step 1:
Start the RPG using ***mrpg -p startup***. In a terminal window, start **script mon_mem_use.out**

Step 2:
In a different terminal window, start playback of the full load data set (this data set will be specified elsewhere).

Step 3:
Start the *mon_mem_use* tool in the terminal window specified in Step 1, specifying the monitoring period (1200) and the update rate (60). A typical command line for invoking this tools is:

mon_mem_use -m 1200 -r 60 <pid>

where <pid> is the process ID of the process your are monitoring.

Step 4:
After the mon_mem_use tool has stopped monitoring, end the terminal logging by script using CTRL-D.

4. What The Numbers Mean and How Are They Used:

As specified in Section 2, the information most important to the ROC for sizing information are the Resident Set Size (RSS) and Private sizes. The mon_mem_use tool will be used by the ROC to determine the memory footprint of the process. This information will also been used to access whether optimizations should be investigated in the footprint is too large.

5. Additional Information:

For a more detailed listing of the memory mapping, use the long-listing (-l) option. The following shows example output.

The long-listing shows both the private and shared mappings of the task. This information is derived from file */proc/<pid>smaps*. The *rss clean* and *rss dirty* columns specify the amount of mapped memory that is actually memory resident broken down to the amount that is clean (hasn't been modified) and the amount that is dirty (has been modified). Should swapping out of process memory become necessary, the dirty resident pages are candidates for writing to the swap device. Clean resident pages are simply discarded.

You might notice some special file names. Files marked as *anon* are memory pages allocated by malloc, and *vdso* is a Virtual Dynamic Shared Object – a single page created by kernel that contains the actual implementation of the system call entry/exit mechanism. Every user process will contain a *vdso* mapped entry. Although the following example does not show this, files beginning with *SYSV* are Interprocess Control (IPC) shared memory segments. These will correspond to shared memory LBs. Algorithms that access base data will likely show a *SYSV* entry.

PRIVATE MAPPINGS:			
vmsize	rss clean	rss dirty	file
4 kB	0 kB	4 kB	/export/home/orpg1/lib/lnux_x86/libadaptcomblk.so
56 kB	0 kB	36 kB	/export/home/orpg1/lib/lnux_x86/librpgcm.so
8 kB	0 kB	8 kB	/export/home/orpg1/lib/lnux_x86/libinfr.so
8 kB	0 kB	8 kB	/export/home/orpg1/lib/lnux_x86/liborpg.so
76 kB	0 kB	28 kB	[anon]
124 kB	0 kB	8 kB	/export/home/orpg1/lib/lnux_x86/librpg.so
608 kB	0 kB	12 kB	[anon]
....	....	....	....
....	....	....	....
24 kB	24 kB	0 kB	/export/home/orpg1/bin/lnux_x86/tvsprod
264 kB	0 kB	180 kB	[anon]
4 kB	4 kB	0 kB	/export/home/orpg1/data/kinematic/tvsprod.lb
20 kB	16 kB	0 kB	/export/home/orpg1/data/logs/tvsprod.log
16 kB	0 kB	16 kB	[anon]
84 kB	0 kB	20 kB	[stack]
SHARED MAPPINGS:			
vmsize	rss clean	rss dirty	file
76 kB	8 kB	0 kB	/export/home/orpg1/lib/lnux_x86/libadaptcomblk.so
56 kB	16 kB	0 kB	/export/home/orpg1/lib/lnux_x86/librpgcm.so
4 kB	4 kB	0 kB	[vdso]
272 kB	216 kB	0 kB	/export/home/orpg1/lib/lnux_x86/libinfr.so
388 kB	160 kB	0 kB	/export/home/orpg1/lib/lnux_x86/liborpg.so

124 kB	116 kB	0 kB	/export/home/orpg1/lib/lnux_x86/librpg.so
....	....	....	....
....	....	....	....
508 kB	128 kB	0 kB	/usr/lib/libgfortran.so.1.0.0
44 kB	12 kB	0 kB	/lib/libgcc_s-4.1.2-20080102.so.1
896 kB	356 kB	0 kB	/usr/lib/libstdc++.so.6.0.8
4 kB	4 kB	0 kB	/export/home/orpg1/data/kinematic/tvsattr.lb
4 kB	4 kB	0 kB	/export/home/orpg1/data/pdist/prod_request.lb
16 kB	16 kB	0 kB	/export/home/orpg1/data/msgs/prgenmsg.lb
4 kB	4 kB	0 kB	/export/home/orpg1/data/storm/trfrcatr.lb
4 kB	4 kB	0 kB	/export/home/orpg1/data/msgs/scan_summary.lb
4 kB	4 kB	0 kB	/export/home/orpg1/data/msgs/gen_stat_msg.lb
4 kB	4 kB	0 kB	/export/home/orpg1/data/adapt/adapt_data.lb
380 kB	108 kB	0 kB	/export/home/orpg1/data/pdist/product_data_base.lb
16 kB	4 kB	0 kB	/export/home/orpg1/data/mngrpg/tat.lb
4 kB	4 kB	0 kB	/export/home/orpg1/data/mngrpg/pat.lb

Vol 3. Document 4 -
Special Topics

Section II Topics Related to Reading Radial Base Data

BUILD 12 CHANGES:

- Modified as required to update the procedure for determining maximum radial array size. Changes were made to improve clarity and safety.
- Updated the description of determining the number of radials to reflect the final definition of the **`msg_type`** bit **`SUPERRES_TYPE`**.
- Updated the description of determining the maximum number of bins to reflect the new **`msg_type`** bit **`HIGHRES_REFL_TYPE`**.
- Updated Testing for End of Elevation / Volume for clarity.
- Added discussion on how to safely handle elevation restarts by the RDA.

BUILD 11 CHANGES:

- Renamed and completely reorganized this section.
- Completed the topic of reading and using the Dual Pol fields in the generic moment structure.
- Improved the topic of reading and using the Basic Moment fields in the base data radial.
- Minor corrections / clarification to the description of determining the size and number of radials.
- Modified the guidance for testing for end of elevation and end of volume. The future RDA (with the AVSET mod) may not always produce the number of elevations in the defined VCP. This guidance affects algorithms reading radial data and producing volume data and algorithms reading elevation data and producing volume data.
- Documented a Build 11 issue with the output of the recombination algorithm.

Part A. Testing for End of Elevation / Volume

Algorithms Reading Radial Data

Actual End of Elevation / Volume

The last radial in an elevation is flagged with end-of-elevation except for the last elevation in a volume where it is flagged end-of-volume. Therefore, to test for end of elevation, either an end-of-elevation flag or an end-of-volume flag must be present. To test for end of volume, the end-of-volume flag must be present.

Pseudo-End of Elevation / Volume

The WSR-88D RDA can produce radials in a scan that overlap the first radial(s). In the original WSR-88D there were normally one or two extra radials in each elevation. Beginning with Build 10, the recombination algorithm (which reduces the increased radial resolution back to the original 1 degree spacing) can produce an overlapping radial. Because of this overlap, the last one or two radials were not needed to get 360 degrees coverage. The last radial with no overlap is flagged either pseudo end of elevation or pseudo end of volume (last elevation in the volume).

Most algorithms would normally eliminate this overlap by not using radials after the pseudo-end of elevation / volume.

For proper operation of an algorithm:

1. All radials in an elevation or volume must be read even if not using radials after pseudo end.
2. If the algorithm needs to eliminate overlapping radials, it must test for pseudo end to eliminate the overlap.

The flag that carries this information is the **`status`** field (the 16th short) in the base data header. The flag values of the **`status`** field are defined in **`a309.h`**.

```
#define GOODBEL     0x00     (good) beginning of elevation
#define GOODBVOL    0x03     (good) beginning of volume
#define GENDEL      0x02     (good) end of elevation
#define GENDVOL     0x04     (good) end of volume
#define GOODINT     0x01     (good) intermediate radial
#define PGENDEL     0x08     (good) pseudo end of elevation
#define PGENDVOL    0x09     (good) pseudo end of volume
```

One example of a test for the end of elevation is:

The following gets the base data radial:

```
basedataPtr = (Base_data_radial*)RPGC_get_inbuf_by_name("BASEDATA",
                                                        &opstatus);
```

The following reads the radial status flag in the header at the beginning of the radial:

```
radial_status = basedataPtr->hdr.status & 0xF;
```

The following tests for Pseudo end of elevation:

```
if(radial_status==PGENDEL || radial_status==PGENDVOL)
```

The following tests for actual end of elevation:

```
if(radial_status==GENDEL || radial_status==GENDVOL)
```

The future RDA may not always process all elevations in a volume scan so any algorithm

reading radial data and producing a volume product must always test for end of volume.

The following tests for actual end of volume:

```
if(radial_status==GENDVOL)
```

When assigning the value to **`radial_status`** a mask of **`0xF`** is used because the high order bit can be set to indicate a "bad" radial. Look at the contents of **`a309.h`** and you will find that there is a "bad" beginning of elevation and a "bad" intermediate radial, etc. After masking, we only need to test for "good" radials. Note: Don't ask what "bad" radials are; just make the tests as indicated.

Handling an Elevation Restart from the RDA

Many of the existing algorithms do not look for or handle elevation restarts. The legacy algorithms in FORTRAN) do. This results in using the new radial messages for that elevation rather than aborting. NOTE: Currently the sample algorithms do not handle elevation restarts.

This technique only applies to algorithms that read base data radial messages and produce elevation data or volume data. Since the opstat of the get_inbuf function does not report this condition, the radial status must be used to detect this condition.

To test for an elevation restart:

1. If the radial status reports a beginning of an elevation before the pseudo-end or actual end of elevation / volume (last elevation in the volume), an elevation restart has probably occurred.
2. To be safe, also test the **`rpg_elev_ind`** field of the basedata header. If it is the same as the partial elevation read an elevation restart has occurred.

Once a restart is detected, the algorithm discards the existing radial data collected for that elevation and reaccomplishes any data initialization required.

Algorithms Reading Elevation Data

Algorithms reading elevation data and producing volume data need to determine the last elevation of a volume. In the past the number of elevations defined in a volume scan could be used. **However, the future RDA may not always process all elevations in a volume scan so any algorithm reading elevation data and producing volume must use the function `RPGC_is_buffer_from_last_elev`. Because the RDA will determine the last elevation dynamically, this function must be called once each elevation.** When this RDA modification is deployed, existing algorithms will be modified to comply with this guidance.

Part B. Determining the Number of Elevations & Radials

NOTE: This explanation takes into account all possible cases. In some existing algorithms where only Legacy resolution data is used and specific constraints have been placed on the product size, the logic observed will differ from that presented here.

Initially, an algorithm may allocate internal data arrays based upon the maximum size of the polar array of the input basedata. This is accomplished by determining the maximum number of radials that can be produced by the RDA (Part B.) and the maximum number of data elements in each radial (Part C.).

Determining the number of Elevations

The number of elevations in a VCP can be obtained with the **`rpg_elev_cuts`** field in the **`Scan_Summary`** structure or the **`num_elev_cuts`** field in the **`Vol_stat_gsm_t`** structure. This could be used for sizing. **However, the future RDA may not always process all elevations defined in a VCP.** When this RDA modification is deployed, existing algorithms will be modified to comply with the following guidance.

> Algorithms reading elevation data inputs and producing volume data outputs need to know the real number of elevation cuts being produced. Since the RDA dynamically determines when to terminate processing elevation cuts in a volume scan, the function **`RPGC_is_buffer_from_last_elev`** must be called once each elevation when reading elevation input data.
>
> For algorithms reading radial data and producing volume data outputs, the radial status flag must be tested for end of volume.

Determining Maximum number of Radials

When reading one of the Legacy resolution data types the maximum number of radials is **`BASEDATA_MAX_RADIALS`** (currently 400).

Determining the maximum number of radials when reading one of the Super Resolution data types is straight forward. Rather than always using **`BASEDATA_MAX_SR_RADIALS`** (currently 800), resources could be reduced by determining the maximum value for each elevation. If the value of the basedata header field **`azm_reso`** is 2, the maximum number of radials is **`BASEDATA_MAX_RADIALS`** (currently 400). If **`azm_reso`** is 1, the maximum is **`BASEDATA_MAX_SR_RADIALS`** (currently 800). This field should be tested on the first radial of each elevation.

> With Build 12, the function **`RPGC_check_radial_type`** can be used to test the radial type bits of the **`msg_type`** field for **`SUPERRES_TYPE`** to determine the maximum number of radials (400 or 600).

Part C. Determining Radial Array Sizes

NOTE: This explanation takes into account all possible cases. In some existing algorithms where only Legacy resolution data is used and specific constraints have been placed on the product size, the logic observed will differ from that presented here.

Initially, an algorithm may allocate internal data arrays based upon the maximum size of the polar array of the input basedata. This is accomplished by determining the maximum number of radials that can be produced by the RDA (Part B.) and the maximum number of data elements in each radial (Part C.).

Algorithm Restraints on Range

The algorithm may constrain the data by having a product range that is less than the range of the input data. For example, some existing products use a 230 KM range even though the reflectivity data is provided to 460 KM. So at this stage, an algorithm could have a variable (the following example uses **max_num_bins**) that has been set based upon the data type being read and any desired constriction on product range.

Altitude Constraints

The current radar produces no useful data above 70,000 feet MSL. A helper function is provided that can be used to further reduce the size of the internal data structure allocated.

```
num_bins_70k = RPGC_bins_to_ceiling(radial_pointer, rad_hdr_ptr->dop_bin_size);

if(bins_to_70k < max_num_bins)
    bins_to_process = bins_to_70k;
else
    bins_to_process = max_num_bins;
```

It should be noted that even if using **RPGC_bins_to_ceiling**, the basedata header field representing the number of good bins still must be checked when using the data in the radial. See next paragraph.

C-1. Basic Moment Data (reflectivity, velocity, spectrum width)

In addition to determining the size and number of radials, it is critical that the algorithm does not read and use data that exceed the limits of the first and last "good" bin. See Part D. If the algorithm internal arrays or product arrays are larger than this, they should be padded with '0'.

Determining Maximum Array Size

When reading one of the Legacy resolution data types the maximum size of the reflectivity array is **BASEDATA_REF_SIZE** (currently 460) and the maximum size of the velocity and spectrum width arrays is **BASEDATA_VEL_SIZE** (currently 920).

When reading one of the Super Resolution data types (**SR_COMBBASE**, etc.) or dual polarization types (**DUALPOL_COMBBASE**, etc.), it is safe use **MAX_BASEDATA_REF_SIZE** (currently 1840) as the maximum size for reflectivity arrays and **BASEDATA_DOP_SIZE** (currently 1200) as the maximum size for the velocity and spectrum width arrays.

> Due to the design of the VCPs, the maximum Doppler array size is different at higher and lower elevations. The simplest solution would be to assume the extended 300 km range (1200 bins). With the current VCP definitions, at some elevation for each VCP the array size is reduced to the original Doppler range of 230 km (920 bins).
>
> With Build 12, the function **RPGC_check_radial_type** can be used to test the radial type bits of the **msg_type** field for **HIGHRES_REFL_TYPE** to determine the horizontal resolution of the surveillance data. 250 m resolution would correspond to 460 bins. 1000 m resolution would correspond to 1840 bins.

If statically allocating memory at the beginning of each elevation, resources could be reduced by testing the number of data bins in the first radial of the elevation. **However the following method fails if the first radial is spot blanked and the number of bins set to 0, and therefore is not recommended.**

Determining Array Type

The Basic Moment arrays in the base data radial message are of type unsigned short..

First Data Element

When processing basic moment radial base data the first "good" data element in the radial array should be determined. The first good or usable data element is determined directly from the contents of the basedata header.

Please note that the index calculated for the first good bin (in the following examples this is **first_bin_index**) is a C array index. That is the first data element is at index **0**.

The basedata header field **surv_range** states the index for reflectivity and the field **dop_range** states the index for radial velocity and spectrum width. This index is valid for an index system beginning with 1. Therefore 1 must be subtracted from this number to provide an index for a C array (an index with 0 as the first element).

```
rad_hdr_ptr = (Base_data_header *) radial_pointer;

first_bin_index = rad_hdr_ptr->dop_range - 1;
```

Note: The value of **surv_range** and **dop_range** appear to always be **1**. Some current algorithms assume this to always be the case, which would simplify some of the remaining code examples. However, until ROC Engineering confirms this is always the case, these fields should be tested to determine the first good bin.

Last Data Element

When processing basic moment radial base data the last "good" data element in the radial array must be determined. The last good data element is determined by several factors. Please note that the indexes calculated for the first good bin and last good bin (in the following example these are **first_bin_index** and **last_bin_index**) are C array indexes. That is the first data element is at index **0**.

If reading an array from a basedata radial, the basedata header field **n_surv_bins** states the number of reflectivity data elements and the field **n_dop_bins** states the number of velocity and spectrum width data elements. This is considered the number of "good" data elements. **These fields must be read and evaluated for each radial.** The C array index for the last data element is calculated as follows.

```
rad_hdr_ptr = (Base_data_header *) radial_pointer;

last_bin_index = first_bin_index + rad_hdr_ptr->n_dop_bins - 1;
```

Even though the number of bins to process may have been constrained by using the **RPGC_bins_to_ceiling** function, the **last_bin_index** as determined by the fields **n_surv_bins** and **n_dop_bins** must be evaluated for each radial. There are several reasons for this.

1. The function **RPGC_bins_to_ceiling** does not consider the antenna height when calculating the number of bins to 70,000 ft.
2. The RDA for some reason may not provide the expected number of bins.
3. This radial may be spot blanked with **n_surv_bins** set to 0.

Range of Data Elements

The range (actually slant range) of the data elements are determined by the size of the range bin represented by each data element. The basedata header field **surv_bin_size** specifies the size of each reflectivity bin and the field **dop_bin_size** the size of the velocity and spectrum width bins. Both are in meters. The range (in meters) to the beginning of the first good data element is given by the base data fields **range_beg_surv** and **range_beg_dop**.

C-2. Generic Moment Data Fields (future Dual Polarization Data)

In addition to determining the size and number of radials, it is critical that the algorithm does not read and use data beyond the last bin. If the algorithm internal arrays or product arrays are larger than this, they should be padded with '0'.

Determining Maximum Array Size

The Dual Pol data fields available to algorithms are either 1200 or 1840. The maximum number of data bins in each type of array are:

`1200 for ZDR, RHO, PHI, KDP, SDP`
`1840 for SNR, SMZ, SDZ` (related to number of Z gates)
`1200 for SMV` (related to number of V gates)

Currently it is unclear whether in Build 12 the number of gates could be 900 instead of 1200 and 460 instead of 1840. This could occur at higher elevations or if 'Super Resolution' is turned off at the RDA. If preallocating arrays use 1200 or 1840 as appropriate.
Even though the following are defined in **`basedata.h`**, as you can see this list is not complete.

BASEDATA_RHO_SIZE	1200	Size of the DRHO Correlation Coefficient array (shorts)
BASEDATA_PHI_SIZE	1200	Size of the DPHI Differential Phase array (shorts)
BASEDATA_SNR_SIZE	1840	Size of the DSNR Signal to Noise Ratio array (bytes)
BASEDATA_ZDR_SIZE	1200	Size of the DZDR Differential Reflectivity array (bytes)
BASEDATA_RFR_SIZE	240	Ignore for now.

Determining Array Type

The Dual Pol fields in the generic moment structure can be of several types (unsigned char, unsigned short, unsigned int, float). The generic moment field **`data_word_size`** is used to determine the type of data array (currently all are either **`8`** for unsigned char or **`16`** for unsigned int).

First Data Element

For the Dual Pol data fields in the generic moment structure, the first element in the array is always good.

Last Data Element

When processing generic moment radial Dual Pol data the last data element in the radial array must be determined. The last data element is determined by several factors. Please note that the index calculated for the last bin (in the following example this is the **`last_bin_index`**) is a C array index. That is the first data element is at index **0**.

The generic moment structure field **no_of_gates** states the number of the valid data array elements

If reading an array from a basedata radial, the generic moment structure field **no_of_gates** states the number of the valid data array elements. **This field must be read and evaluated for each radial.** The C array index for the last data element is calculated as follows.

```
s_data = (short *) RPGC_get_radar_data( (void *)radial, RPGC_DRHO, &gen_moment );

    last_bin_index = gen_moment.no_of_gates - 1;
```

Even though the number of bins to process may have been constrained by using the **RPGC_bins_to_ceiling** function, the **last_bin_index** as determined by the field **no_of_gates** must be evaluated for each radial. There are several reasons for this.

1. The function **RPGC_bins_to_ceiling** does not consider the antenna height when calculating the number of bins to 70,000 ft.
2. The RDA for some reason may not provide the expected number of bins.
3. This radial may be spot blanked. In this case either the generic moment does not exist or the field **no_of_gates** is 0.

Range of Data Elements

The range (actually slant range) of the data elements are determined by the size of the range bin represented by each data element. The generic moment structure field **bin_size** specifies the size of each bin in meters. The range (in meters) to the center of the first data element is given by the generic moment structure field **first_gate_range**.

Part D. Radial Processing Example

Please note that the indexes calculated for the first good bin and last good bin (in the following examples these are **`first_bin_index`**, **`last_bin_index`**, and **`last_bin_product_index`**) are all C array indexes. That is the first data element is at index **0**.

Case 1: Limited by the number of bins (or gates) in the radial message avoid using "bad bins" (`n_dop_bins` or `n_surv_bins` for basic moment data or `no_of_gates` for generic moment data):

It is the algorithm's responsibility to not use data before or after the "good" data bins.

Case 1a: Bins before the first good bin. (Basic Moment Data only)

Since the first bin is usually the first "good" data bin (some algorithms assume this), you normally do not see this compensated for. The algorithm would either set the data elements before the first good bin to **0** or not use an array index before the first good bin.

Case 1b: Bins after the last good bin.

The algorithm should not read or process any data beyond the last "good" data element. The internal array size exceeding the number of good bins, can be handled in two ways.

METHOD 1: is to set the values of the internal array data elements after the last "good" bin to **0** and process the entire array. The advantage of this method is all algorithm internal radial arrays are the same size. The final product radial data arrays should be the same size.

METHOD 2: is reducing the number of bins in the internal data array to not exceed the location of the last good bin in the input data. The final product radial data arrays should be the same size. The following two examples are logically equivalent.

```
if( (last_bin_index - first_bin_index + 1) < bins_to_process )
    num_bins_in_product = last_bin_index - first_bin_index + 1;
else
    num_bins_in_product = bins_to_process;
```

```
if( bins_to_process > (last_bin_index - first_bin_index + 1) )
    num_bins_in_product = last_bin_index - first_bin_index + 1;
else
    num_bins_in_product = bins_to_process;
```

The input parameter representing the number of bins for the original packet 16 helper function, **`RPGP_set_packet_16_radial()`**, must reflect the reduction above if METHOD 2 is used.

Case 2: Limited by internal array size - reduce the last bin index:

If the algorithm's internal data array never uses all of the available "good" data bins in the radial, the originally calculated index for the last bin must be corrected before being used as an input parameter for several helper functions. The following two examples are logically equivalent.

```
if( bins_to_process < (last_bin_index - first_bin_index + 1) )
    last_bin_product_index = bins_to_process - first_bin_index - 1;
else
    last_bin_product_index = last_bin_index;
```

```
if( last_bin_index > (bins_to_process - first_bin_index - 1) )
    last_bin_product_index = bins_to_process - first_bin_index - 1;
else
    last_bin_product_index = last_bin_index;
```

This correction must be made for the end index parameter for both run-length encoding functions: **`RPGC_run_length_encode()`** and **`RPGC_run_length_encode_byte()`** and the new packet 16 helper function: **`RPGC_digital_radial_data_array()`**.

NOTE: There is a minor error in applying this correction in several existing algorithms: **`cpc007/tsk013/bref8bit.c, cpc007/tsk014/bvel8bit.c, cpc007/tsk015/superes8bit.c`**

API NOTE: If using one of the three helper functions to read a basic moment array (for example: **`RPGC_get_vel_data`**), the second parameter returns the first good bin and the third parameter returns the last good bin. However, these values have already been corrected for a 0 based C array.

Vol 3. Document 4 -
Special Topics

Section III Topics Related to Writing Algorithms

This guide is a work in progress. Suggestions for improvement are welcome.

Volume 3. WSR-88D Algorithm Programming Guide

Appendices

Appendix A. Algorithm Structure Templates

Appendix B. Algorithm Tasks Recently Ported to C

Vol 3. Appendices

Appendix A. Algorithm Structure Templates

These flow charts illustrate the guidance currently provided in this guide. Many legacy algorithms have not been updated to reflect the changes in abort services. NOTE: The logical structures contained in these charts represent just one method of complying with all required structure rules. They are not the only way an algorithm can be structured.

The following flow charts are provided:

1. An algorithm reading base data (radial based) and outputting an elevation based product. This chart provides a framework for Sample Algorithm 1 - Digital Reflectivity algorithm, Sample Algorithm 2 - Radial Reflectivity algorithm, and the first task in Sample Algorithm 3 - Multiple Task algorithm.
2. An algorithm reading an elevation based product and outputting a volume based product. This chart provides a framework for the second task in Sample Algorithm 3 - Multiple Task algorithm
3. An algorithm reading base data (radial based) and outputting a volume based product. (No sample algorithm at this time).

Many of the named activities in these flow charts correspond to specific API services. For example, the flow chart for the elevation product can be compared to the source code of the digital reflectivity sample algorithm. Comment labels have been placed in the source code as an aid in relating the source code to the flow chart.

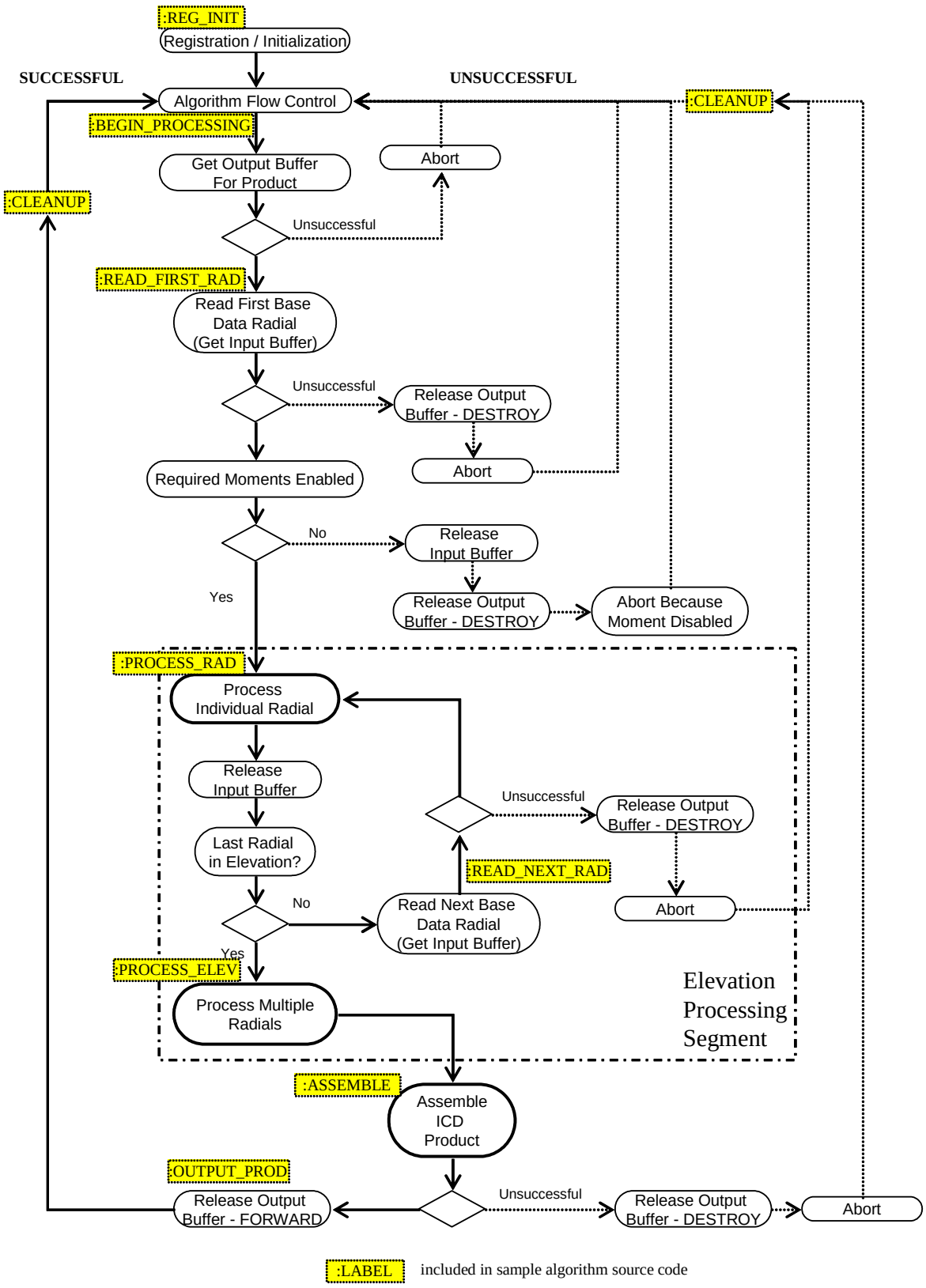
If the sample algorithms are not installed, existing operational algorithms can be used as examples.

Logic Demonstrated	Sample Algorithm	Operational Algorithm
Producing a 256 level elevation product from radial basedata.	Sample 1	cpc007/tsk013/ bref8bit
		cpc007/tsk014/ bvel8bit
		cpc007/tsk015/ superes8bit
Producing a16-level RLE elevation product from radial basedata.	Sample 2	cpc007/tsk001/ basrflct
		cpc007/tsk002/ basvcty
Producing a volume product from elevation basedata.	None	cpc013/tsk003/ epre
A multiple task algorithm with elevation intermediate products and volume final product.	Sample 3	cpc017/tsk009/ tda2d3d cpc018/tsk005/ tdaprod
A multiple task algorithm with final task using the WAIT_ANY form of control loop	Sample 4	cpc004/tsk006/ recclalg cpc004/tsk007/ recclprods
An event driven task.	None	future build

Updated April 2006

One Product Algorithm

Radial_Based Input Elevation_Based Output

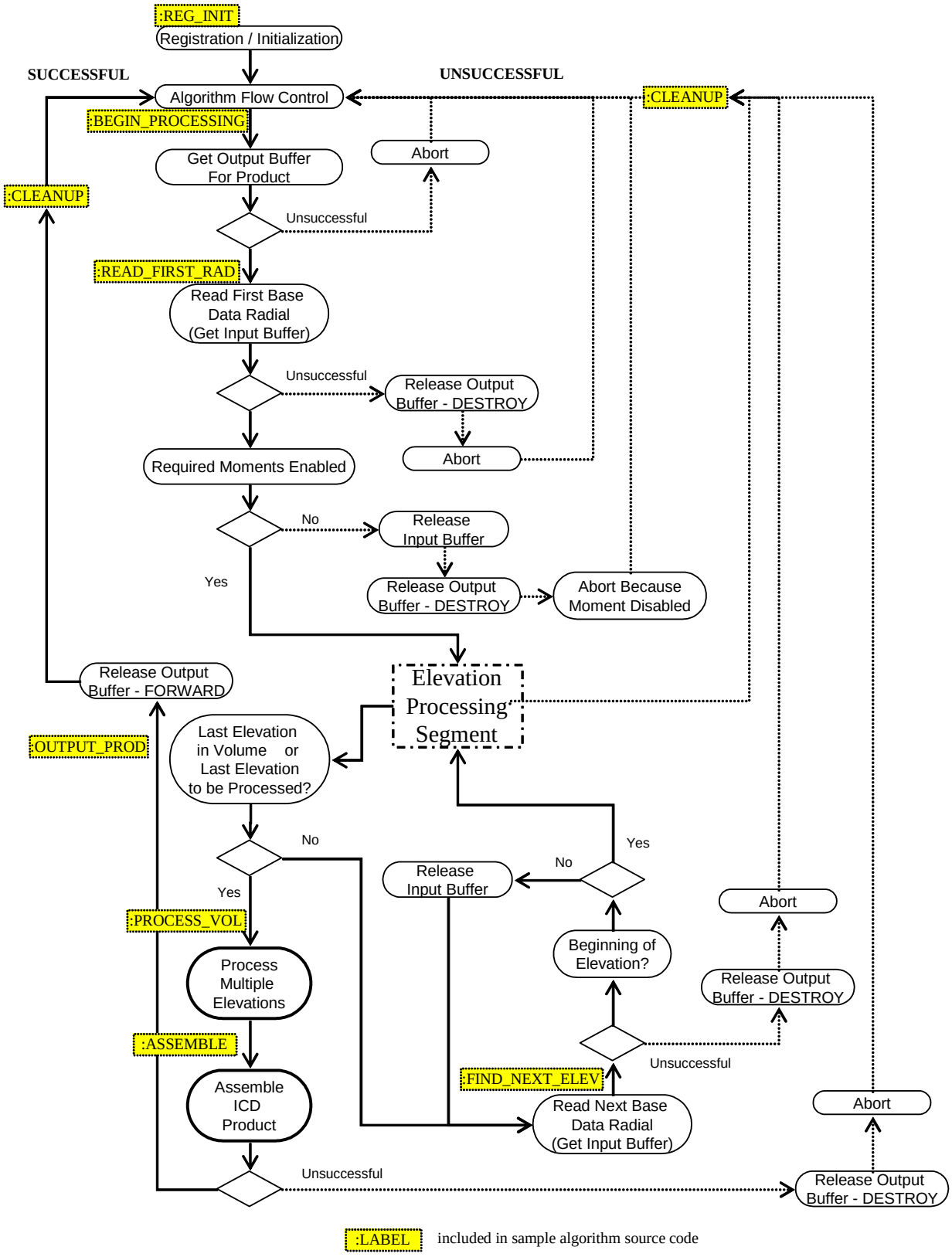


:LABEL included in sample algorithm source code

Updated April 2006

One Product Algorithm

Radial_Based Input Volume_Based Output

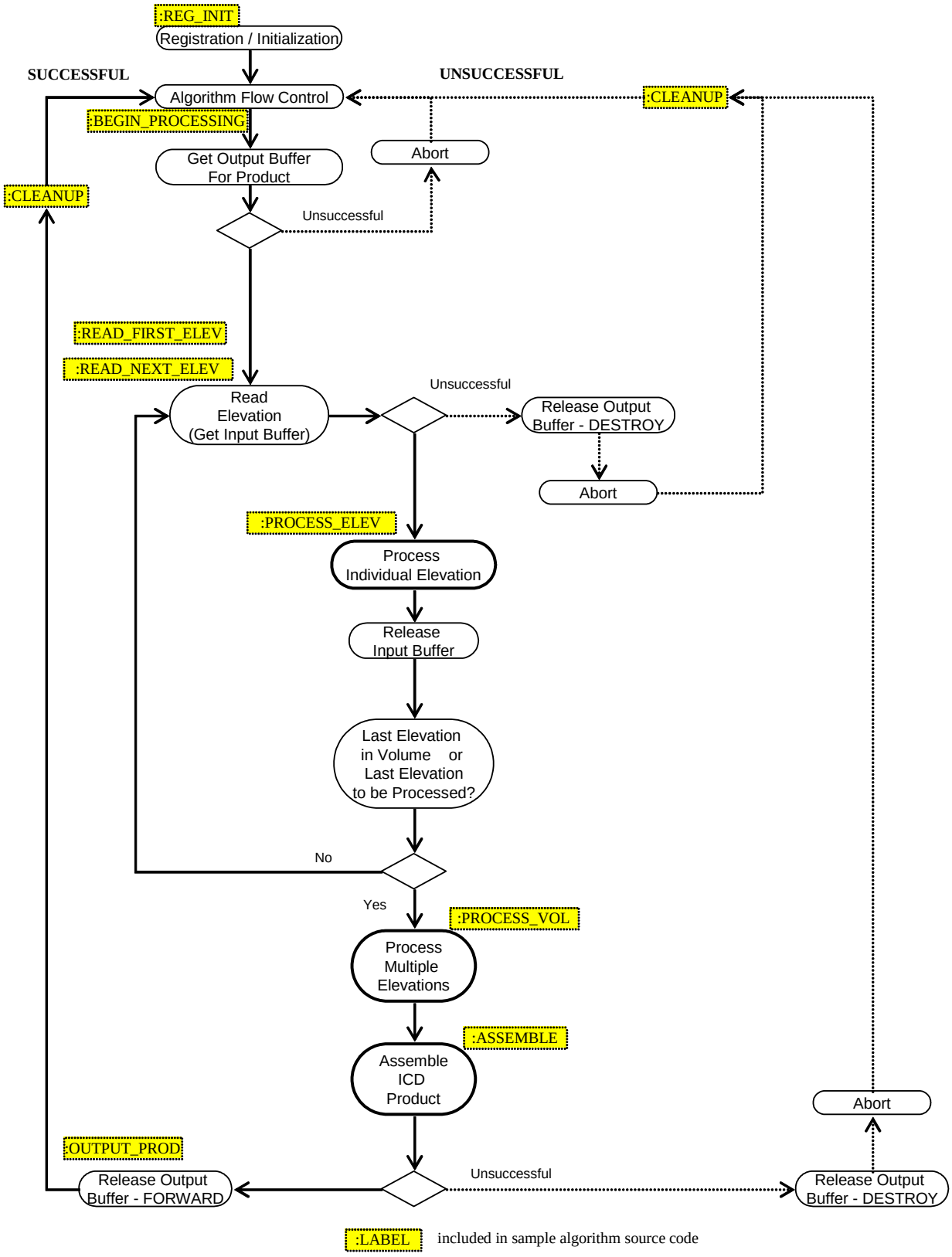


:LABEL included in sample algorithm source code

Updated April 2006

One Product Algorithm

Elevation_Based Input Volume_Based Output



Appendix B. Algorithm Tasks Recently Ported to C

The ROC is currently porting all Legacy algorithms from FORTRAN to ANSI-C. All new algorithm development should be accomplished in C. There is no firm time table in accomplishing this. If modifying an existing FORTRAN algorithm, the ROC Engineering Branch should be contacted to coordinate development activities.

	Location (cpc/tsk)	task executable name	Notes
	cpc007/tsk001	basrflct	
	cpc007/tsk002	basvlcty	
	cpc007/tsk003	basspect	
	cpc007/tsk004	cmprflct	
	cpc007/tsk006	itwsdbv	
	cpc007/tsk013	bref8bit	
	cpc007/tsk014	bvel8bit	
	cpc008/tsk001	alerting	
	cpc008/tsk003	combattr	
	cpc008/tsk004	basvgrid	
	cpc014/tsk003	hybrprod	
	cpc016/tsk002	srmrmrv	

During the ported process the ORPG source code will contain unused code. This will typically occur at large task numbers (e.g., cpc007/tsk044, cpc008/tsk022, cpc016.tsk22). This temporary 'dummy' code can be confirmed by looking at the cpc level makefile to see if this task subdirectory is included.