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

CODE B23_0r1_9: November 2024

Includes ORPG Build 23_0r1_9

Volume 2. ORPG Application Software Development Guide

- Compile Software - Configure Algorithms/Products - Data Structures -

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.

Volume 2. ORPG Application Software Development Guide

This guide is intended for programmers with software development experience in a Unix environment and an appropriate background in Radar Meteorology.

CODE is produced in two versions:

1. **National Weather Service Edition** - This is the complete version of CODE. Distribution is limited to within the National Weather Service and other U.S. Government Agencies.

2. **Public Edition** - This version of CODE is intended for public release. Certain proprietary software components have been removed to permit release outside the U.S. Government.

Differences between the two CODE editions are described in Appendix F.

Introduction

These documents provide guidance for WSR-88D algorithm developers including:

a. a limited overview of the ORPG software architecture
b. a description and instructions for using the WSR-88D development environment
c. documentation of internal ORPG data formats for algorithm developers

The information presented here is independent of writing algorithm source code but does contain some references to the Application Programming Interface (API). CODE Guide Volume 3 - *WSR-88D Algorithm Programming Guide* contains the tutorial, reference, and sample algorithms for the *WSR-88D Algorithm API*, and guidance for the structure of algorithms.

Documentation of the ORPG specific development and analysis utilities is provided with CODE Guide Volume 4 - *CODE Utility Guide*

Procedures for starting and stopping the ORPG along with troubleshooting hints are included with CODE Guide Volume 1 Document 1 Section IV. A quick reference to starting the ORPG is provided in Appendix G of this Volume.

Document 1. The ORPG Architecture

This document provides a brief overview of the ORPG architecture and defines some terms used throughout CODE Guide Volume 2 - *ORPG Application Development Guide* and CODE Guide Volume 3 - *WSR-88D Algorithm Programming Guide*.

Document 2. The ORPG Development Environment

This is a basic description of the ORPG software development environment intended to provide sufficient information to compile software and to configure development algorithms for running in the ORPG. The ORPG development team has created a flexible environment to allow many programmers to contribute to a development effort. Not all aspects of this environment are described here.

Section I Integrating Development Software with ORPG Source Code
Section II Compiling Software in the ORPG Environment
Section III ORPG Configuration for Application Developers
Section IV Configuring Site Specific Adaptation Data

Document 3. WSR-88D Final Product Format

The structure of the products distributed to users is described along with the content of the product header. Guidance on the use of the various traditional data packets and the new generic product components is provided.

Section I Product Block Structure
Section II Traditional Product Data Packets
Section III Generic Product Components
Section IV ORPG Application Dependent Parameters

Document 4. ORPG Internal Data for Algorithm Developers

This document contains helpful technical information concerning ORPG internals including the format of the radar base data input to the algorithms and configuration and use of adaptation data for algorithms.

Section I Base Data Format
Section II Algorithm Adaptation Data - Configuration & Use
Section III Other Data Inputs

Appendices

Provides documentation of the base data header fields, the generic moment structure and other topics.

Volume 2. ORPG Application Software Development Guide

Document 1. The ORPG Architecture

This brief introduction is intended to cover the basic terms and operating concepts of the ORPG for WSR-88D algorithm developers without exposure to unnecessary detail. The ORPG Software Design Description (SDD), which is part of the formal ORPG documentation, includes a more complete description of the ORPG design.

Tasks and Linear Buffers

The Open Radar Product Generator (ORPG) consists of more than 100 loosely coupled *tasks* (processes). These tasks are launched when the ORPG is started. Related tasks are grouped into functional areas called computer program components (CPC) such as: user interface functions, radar data acquisition functions, product distribution, monitor and control functions, communications management, and the meteorological and product formatting algorithms. ORPG tasks communicate via two mechanisms. Some tasks respond to ORPG registered events which are posted by other tasks. However, the primary concept of operation of the ORPG is a data flow paradigm. While the legacy RPG used centrally managed buffers to exchange data, the ORPG tasks pass data via *linear buffers*.

A *linear buffer* is a data storage facility that provides the primary mechanism for inter-task communication in the WSR-88D ORPG. Virtually all persistent internal data storage is accomplished via linear buffers. Most linear buffers are of type 'file' where the stored data exist in files which are persistent. For performance purposes, a few linear buffers are configured as 'shared memory' buffers which only contain data while the ORPG is running. All data is stored in a linear buffer in the form of "messages". ORPG service libraries hide the details of reading and writing linear buffer messages. However, the reader of a linear buffer message must know the structure of the data stored in the message.

Linear buffers are configured when created and can be configured to behave in a variety of ways. The default behavior is a sequential queue of messages -- a *message queue* type. The buffer is configured for a specific size and a maximum number of messages. When filled, the oldest message is retired. A buffer can also be configured as "replaceable" or a *message database* type, where messages are overwritten by new messages (for example, adaptation data storage and the ORPG product database).

Algorithms and Product Storage

The basic data flow from base data to product storage is illustrated in Figure 1. A task called 'Process Base Data' takes the incoming messages from the Radar Data Acquisition (RDA) subsystem, strips off

the data messages and formats them for use by the ORPG algorithms. ORPG algorithm tasks write each product type to a unique linear buffer. Simple algorithms input base data and output a product for distribution (a *final product*). More complex algorithms are implemented via a series of tasks producing *intermediate products*.

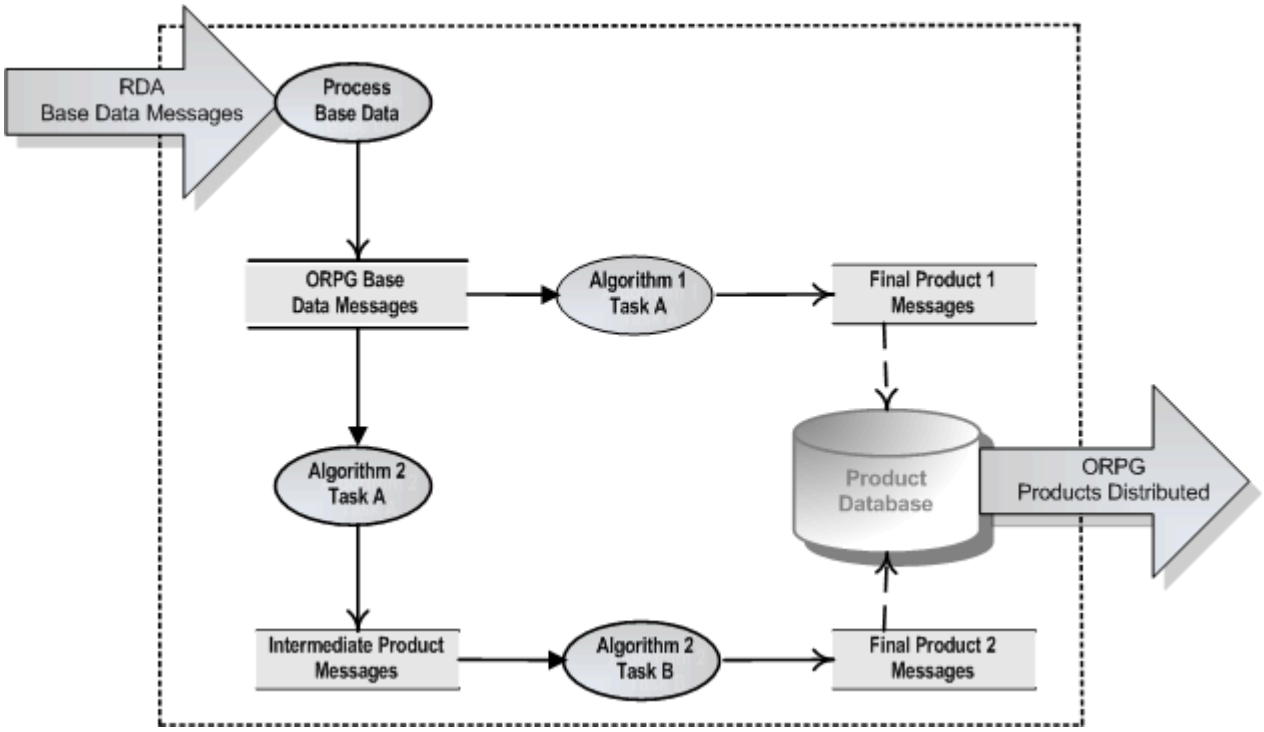


Figure 1. Basic ORPG Product Data Flow

Final products are formatted according to the Interface Control Document (ICD) for the RPG to Class 1 User, document 2620001. The *Algorithm API* provides some support for correctly formatting *final products*. Currently, there is no specified format for *intermediate products* which are not distributed to users. When a product is written to a linear buffer, the ORPG inserts an internal 96-byte header to each product message for storage in linear buffers. This "Pre-ICD" product header is used by the ORPG product distribution infrastructure.

Each product is stored individually in a separate linear buffer message. Generally, *intermediate products* are stored (along with the 96-byte internal header) in the corresponding product-specific linear buffer. *Final products* are stored (along with the 96-byte internal header) in the main product database linear buffer. In this case, the product-specific linear buffer message contains only the 96-byte internal header that includes a reference to the product database linear buffer message containing the product. All of this is transparent to the algorithm. The *WSR-88D Algorithm API* handles all of the details.

The product-specific linear buffers are configured to retain the several of the most recent product messages (typically 10 for volume final products and 40 for elevation final products). The retention of product messages in the product database linear buffer is centrally managed and can be configured by the user.

ORPG Product Data Flow

If an algorithm is implemented as a series of tasks, each task must be connected by at least one intermediate product to the next task in order to satisfy the data flow scheduling mechanism. Any task can have multiple product data inputs. A task can have more that one product output, either intermediate or final. In the example provided in Figure 2, the task "**hailalg**" has two product inputs (**CENTATTR** & **TRFRCATR**) and two product outputs (**TRENDATR** & **HAILATTR**).

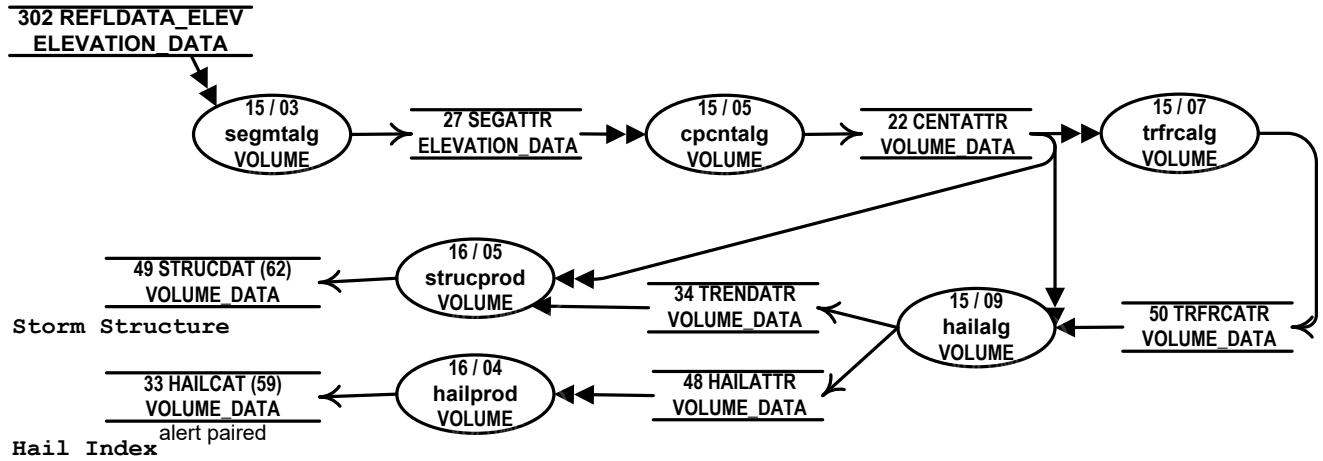


Figure 2. Data Flow for the Hail Algorithm

The data flow diagram contains many configuration parameters (see Figure 3). Generally parameters are associated with either the program task or the product data store.

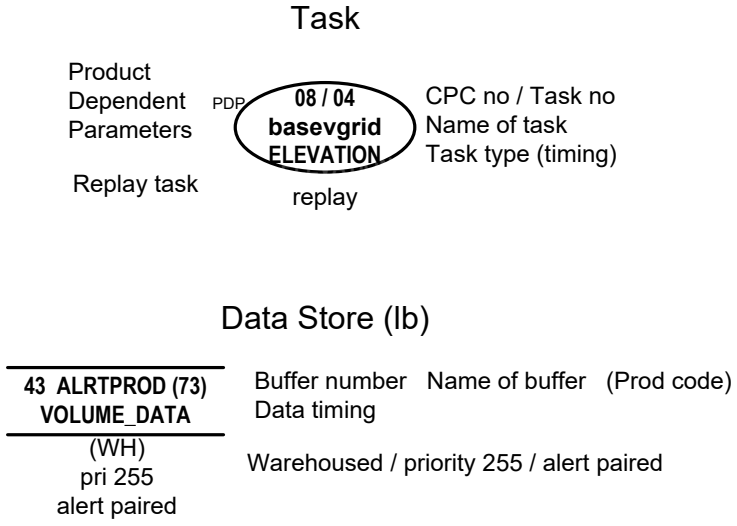


Figure 3. Task & Data Store Legend

The parameters include the name of the actual executable (Name of task), a reference to the location in the ORPG source code (CPC number / Task number), buffer number, and buffer name (internal name of the product). The product code is the external numerical reference for the product. Data timing

corresponds to the frequency that the product is generated, generally once per elevation or once per volume. These configuration parameters and others are described in detail in Document 2 of this Volume of the CODE Guide.

A data flow diagram showing tasks and products for all existing WSR-88D algorithms is contained in the **`algorithm_data_flow_bNN.pdf`** file (located in the **`--/pdf_doc/`** directory on the CODE CD).

NOTE: Not all of the products in the diagram are present with the Public Edition of CODE. A list of products not included with the Public Edition is provided in Appendix F.

In order to completely understand this diagram, the reader must be familiar with the Algorithm API documented in CODE Guide Volume 3 - *WSR-88D Algorithm Programming Guide*. Note that not all data used and shared by algorithms are shown in these diagrams; only the product data are shown. Algorithms also read *adaptation data* in order to obtain site specific information (radar location, elevation, etc.) and to customize algorithm performance via parameters contained in configuration files rather than through recompilation of the software. Many legacy FORTRAN algorithms use an additional mechanism for sharing data called *Inter-Task Common Blocks* (ITC Blocks). More recent algorithms use *non-product data stores* for additional persistent data.

Data Driven Algorithm Tasks

All algorithm tasks are data driven processes. Each task in the chain of processing for a final product will not enter it's processing loop until the required dependent product input is available. The two types of data driven tasks are illustrated in Figure 4. Task 1 is the most common type. The processing loop is entered when registered "driving" product input is available. Subsequent reads of other inputs will block until the input is available. Task 2 has multiple product inputs but enters its processing loop when any one of the registered product inputs are available. Only that single available input is used.

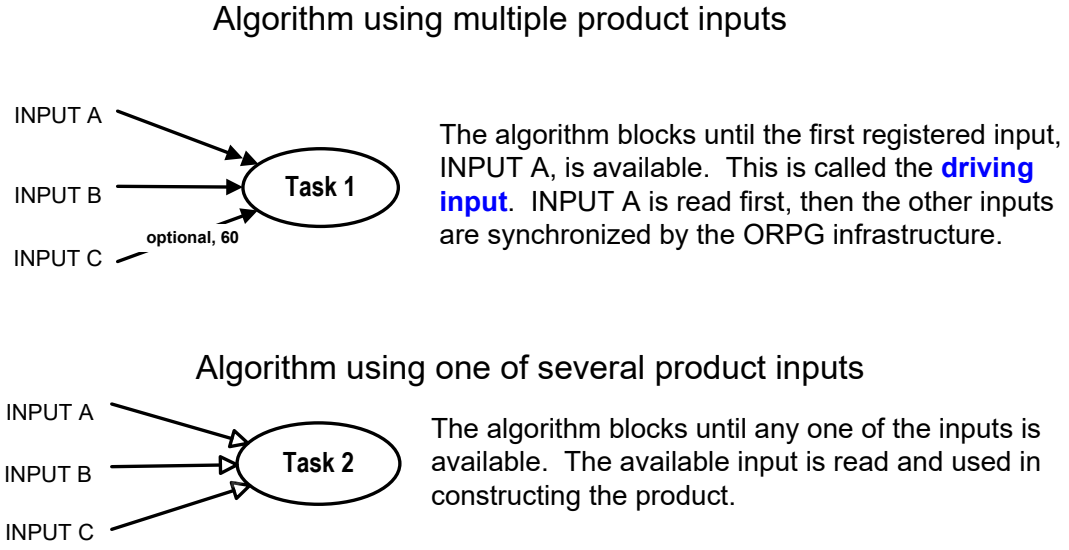


Figure 4. Data Driven Algorithm Tasks

Location of Linear Buffer Files

The location of linear buffer files is determined by the value of the **`ORPGDIR`** environmental variable. The linear buffer files are arranged in subdirectories based upon the type of information contained. For example, the adaptation data linear buffers are contained in the **`adapt`** subdirectory, the product-specific linear buffer files related to base data are in the subdirectory **`base`** and the main product database linear buffer is in the **`pdist`** subdirectory which contains all storage buffers related to product distribution. The specific subdirectory and linear buffer filenames are determined by ORPG configuration files which are discussed in CODE Guide Volume 2 Document 2 - *The ORPG Development Environment*.

Volume 2. ORPG Application Software Development Guide

Document 2. The ORPG Development Environment

This is a basic description of the ORPG software development environment intended to provide sufficient information to compile software and to configure development algorithms for running in the ORPG. The ORPG development team has created a flexible environment to allow many programmers to contribute to a development effort. Not all aspects of this environment are described here.

The following discussion assumes that the ORPG has been installed and configured in accordance with the instructions provided with the ORPG Software Distribution.

Section I Integrating Development Software with ORPG Source Code

A description of the organization of the ORPG source code and guidance on the placement of development software within that structure.

Section II Compiling Software in the ORPG Environment

An explanation on the use of the ORPG software makefile system and instructions on compiling development software.

Section III ORPG Configuration for Application Developers

Instructions covering configuration procedures for adding executable tasks (specifically new algorithms), ORPG data stores (i.e., linear buffer files), and new products to the ORPG.

Section IV Configuring Site Specific Adaptation Data

This is a brief guide in providing the correct site adaptation data for the RDA that produced the radar data being ingested into the ORPG in the development environment.

Vol 2. Document 2 -
The ORPG Development Environment

Section I Integrating Development Software with ORPG Source Code

This document covers the following topics.

- The organization and contents of the ORPG source code directory structure.
- Guidance on locating new development software (algorithm source code) in the ORPG directory structure.
- Instruction on moving existing development software to a new ORPG source code tree.

ORPG Source Code Directory Structure

Below is a partial summary of the ORPG directory structure. Some of the directories are created when the ORPG is compiled.

The following are the top level directories with respect to the directory into which the ORPG is installed. The variable **`$HOME`** is set to this installation directory.

`src`
Contains the makefiles and the source code for the ORPG. This directory is subdivided into major computer program components (CPC) contained in individual directories (e.g., **`cpc001`**, **`cpc101`**), which include sub directories containing individual libraries and executable tasks (**`lib003`**, **`tsk001`**).

`include`
Contains the include files associated primarily with ORPG tasks. Algorithm related header files containing definitions shared outside the algorithm task (for example algorithm adaptation data structure definition) are placed here.

`lib`
`lib/include`
contains the include files associated with the ORPG libraries.
`lib/lnux_x86`
contains the ORPG shared libraries for the Linux PC platform (after software is compiled and installed).

`man`
Contains the ORPG man pages.
`man/cat1`
The man pages for algorithm tasks are placed here when the source code is checked into the operational system..
`man/cat4`

The man pages for the algorithm data stores (product and non-product) are placed here when the source code is checked into the operational system.

conf

Make description files which are included into appropriate makefiles. Some description files are architecture specific.

cfg

ORPG configuration files (after software is compiled and installed). These files involve all aspects of ORPG configuration including the hardware configuration (number of computing nodes, number of external interfaces, etc.), executable tasks, product and data storage. Many of these files installed in the CODE algorithm development environment are not installed in the operational ORPG. During development several files in this directory are customized for task and product configuration.

cfg/dea

contains configuration files for algorithm adaptation data (after software is compiled and installed). During the development stage, it is recommended that the algorithm adaptation data files (**.alg**) be manually copied from the **src/cpc104/lib006** directory rather than modifying the makefiles.

cfg/extensions

is a directory used during development to contain the 'snippet' files that configure the tasks and product data stores for the development algorithm

cfg/pgt

this directory contains the product_generation_table that lists products to be generated by the RPG, even without a specific narrowband user request for the product.

cfg/vcp

this directory contains the definitions of the volume coverage patterns (VCP) for the radar.

data

Beginning with Build 10, this is the standard location for the ORPG persistent data files. By default, the variable **ORPGDIR** points to this directory.

bin

ORPG executable files (after software is compiled and installed). General scripts are at the top level while compiled executables are in the appropriate architecture specific directory, i.e., **bin/lnux_x86** or **bin/slrs_spk**.

tools

If ORPG tools are installed via compiling source code, they are installed here. If installed from a binary distribution, they may be installed in a different location.

tools/bin

contains executable files (after software is compiled and installed) for ORPG utilities. General scripts are at the top level while compiled executables are in the appropriate architecture specific directory, i.e., **tools/bin/lnux_x86** or **tools/bin/slrs_spk**.

tools/cfg

contains copies of ASCII configuration and adaptation data files (after software is compiled and installed).

tools/data

contains the background map files and other tool data files (after software is compiled and installed).

tools/cvgN.N

contains default configuration files for the CODE CVG utility (after software is compiled and installed).

`tools/cvg_map`

is the default location of the CVG background map files (after software is compiled and installed).

`ar2data`

this directory is created by the CODE installation and contains 3 sample archive II volume files.

Adding Algorithm Software to the ORPG

Introduction

This section only provides guidance for the location of development source code in the ORPG software directory structure. An introduction to the ORPG Makefile system in the following section, *Compiling Software in the ORPG Environment*, provides the necessary information to ensure that the software is correctly compiled and binaries are properly installed into the ORPG.

The top level directory structure of the source code tree divides the ORPG software into units called computer program components or CPCs. These are not components in the strict architectural sense but represent a partitioning of the software into related functionality. For example, software in the **`cpc001`** directory contains source code for tasks and libraries related to the human computer interface (HCI) portion of the ORPG and software in **`cpc007`** is related to algorithms producing reflectivity, velocity, and spectrum width products. A description of the contents of source code directories is provided in the man page **`cpcmap`**. ORPG algorithms are contained in the following **`cpc`** directories:

`src/cpc007`
Base Data Products
`src/cpc008`
Message Processing
`src/cpc010`
MIGFA (NWS CODE only)
`src/cpc013`
Precipitation Algorithms
`src/cpc014`
Precipitation Products
`src/cpc015`
Storm Series Algorithms
`src/cpc016`
Storm Products
`src/cpc017`
Kinematic Algorithms
`src/cpc018`
Kinematic Products
`src/cpc022`
OTHER (NWS CODE only)
`src/cpc104/lib006`
Algorithm adaptation data files (**`.alg`**)

The second level directory structure of the source code tree divides each **`cpc`** directory into source directories for individual algorithm tasks or libraries. The task directories are named **`tsk001`**, **`tsk002`**, etc. and the library directories are named **`lib001`**, **`lib002`**, etc.

Guidance for Integration of Algorithms into the ORPG Baseline

There is currently no formal directive concerning where to place the source code when developing new algorithms. Ultimately, the NWS Radar Operations Center (ROC) will decide into which CPC a new algorithm will be placed when it is integrated into the operational baseline. This location is normally determined during one of the design / integration reviews with the ROC.

Guidance for Algorithm Developers

The following guidance will help minimize the effort required to integrate development source code with the baseline ORPG source code.

Filename conventions for development source code

The configuration management system used by the ROC requires unique filename regardless of the directory in which the file is located. The following suggestions 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.

Location of development source code in ORPG source tree

- Consider creating a new **`tsk`** subdirectory within an existing **`cpc`** directory when
  - Modifying an existing algorithm
  - Creating an algorithm closely related to existing algorithms

- If creating a new **`cpc`** directory, consider the following in choosing an unused **`cpc`** number.
  - Using **`cpc099`** and below is recommended. This is the area where new operational algorithms may be located.
  - In addition, any unused directory in the **`cpc300`** series could be used (**`cpc305`** is used for CODE sample algorithms).
  - The following **`cpc`** directories should **not** be used.
    - **`cpc100`** series contains system-level software and should not be used.

- If a series of related algorithms are under development
  - It would be appropriate to place them in the same **`cpc`** directory
  - If these algorithms use a common shared library that is not of interest to other algorithms or the ORPG in general, the source code for that shared library should be located in a **`lib`** subdirectory within that CPC.

- Source code for developer added shared libraries should not be placed within a **`cpc`** directory containing existing system libraries (the **`cpc100`** series). Designating a development library as a "system level" resource requires formal approval.

- Additional restrictions on the organization of source code are based upon using the ORPG makefile system (see Section II of this document, *Compiling Software in the ORPG Environment*). These policies must be followed to take advantage of the ORPG makefile system:

- Source code for binary executables should be located in a subdirectory named **`tsk001 - tsk999`**. For operational tasks (including algorithms), there is normally only one binary executable produced from the source code within each subdirectory.
- Source code for shared libraries should be located in a subdirectory named **`lib001 - lib999`**. There is normally one shared library produced from the source code within each subdirectory.

Location of other algorithm related files in ORPG source tree.

Algorithm adaptation data definition files

The algorithm adaptation data files should be placed in **`cpc104/lib006`**. These files are installed into **`cfg/dea`** directory for run-time use. **It is recommended they be manually copied into the `cfg/dea` directory during development rather than modifying the makefile.** See Vol 2 Doc 4, Section II.

Algorithm adaptation data include files

The algorithm adaptation data include files should be placed into the **`include`** directory prior to handoff to the ROC for integration into the operational system. If only needed by a single task they can remain in the individual task directory during development

Algorithm task man pages

The algorithm task man pages (**`*.1`**) and output data store man pages (**`*.4`**) should be in the task's source directory (**`src/cpcXXX/tskXXX`**). When the code is officially checked into the operational system they are placed into the appropriate directories (**`man/cat1 man/cat4)`.**

Algorithm task and output data configuration

The algorithm task and output product data store configuration is supplied to the ROC in the form of 'snippet' files which are placed into the **`cfg/extensions`** directory. See Vol 2, Doc 2, Section III Part C item 5 for organization of data and naming conventions for the snippet files.

Moving Development software to a New ORPG

When upgrading your development environment to a new version of the ORPG source code, the new ORPG should be installed and compiled in a new account.

- Development source code is then copied into the new directory structure.
- The new ORPG will have to be configured for the algorithm tasks that are copied over into the new account. The development products and tasks should be configured using 'snippet' files which are placed into the **`cfg/extensions`** directory rather than by modification of the **`task_attr_table`**, **`task_tables`**, **`product_attr_table`**, and **`data_attr_table`** configuration files, and by modification of the **`pgt/product_generation_tables`** configuration file (see section III of this guide titled, *ORPG Configuration for Application Developers*, for more information). If the new ORPG includes new tasks or products, some changes to your previous modifications might be required in order to avoid conflicts. The following attributes must be unique within the ORPG (see section III of this guide titled, *ORPG Configuration for Application Developers*, for more information).
  - Product Codes for final products
  - Linear Buffer Numbers and Product Names
  - Linear Buffer filenames
  - Task Names (executables and logical name)
  - Data Store Numbers and Data Store Names

  If the structure or semantic content of any of the configuration files has been changed in the new ORPG, the configuration of the development algorithms will have to be re accomplished following the new guidance provided in this guide. If the structure has not changed the configuration snippet files can be used without modification and the and the default generation table entries in the **`pgt/product_generation_tables`** configuration file can be copied from the old of the file..
- On occasion the contents of the individual task makefiles (named **`*.mak`**) change with a new ORPG Build. If this occurs a new makefile will have to be created following the examples provided in Section II of this document.
- If the directories containing your development code are used by this new ORPG, you will have to relocate your code.
  - Moving source code to a different directory requires some editing of the makefiles. With proper use of non hard coded directory references these changes are minimal, if any.
  - Beginning with Build 9, the task number (which was based upon the source code directory) is no longer an attribute in either the **`task_attr_table`** or the **`product_attr_table`** configuration files.

Vol 2. Document 2 -
The ORPG Development Environment

Section II Compiling Software in the ORPG Environment

This document covers the following topics.

- Part A. describes the ORPG makefile system and provides guidance on modifying template makefiles for new algorithms.
- Part B. provides instructions for compiling individual algorithms.
- Part C. explains how to use the makefile system to install non source code components such as scripts.

Part A. ORPG Makefile Guide for ORPG Source Code

This introduction to the ORPG Makefile system provides the necessary information to compile new applications in the ORPG environment. Guidance concerning the placement of development source code in the ORPG directory structure is provided in the previous section, *Overview of ORPG Software*.

Environmental Variables

The following environmental variables must be defined for the ORPG makefile system to function properly.

MAKETOP
The top level directory of the source code tree.

LOCALTOP
An "alternate" top level directory. If defined, this represents the top level of the "install" tree. If not defined, the install tree is identical to the source code tree defined by **MAKETOP**.

MAKEINC
The top level directory under which the makefile configuration directory **conf** is located.

ARCH
Only a Linux PC platform is currently supported. The value is **lnux_x86**.

Basic instructions for defining these variables are provided in CODE Guide Volume 1 Document 1 - *CODE Specific ORPG Installation Instructions*. The following instructions assume a basic configuration that has the source code and installed binaries in the same directory tree.

Global Make Description Files

Two global make description files contain the details for the makefiles used in the ORPG environment. They are referenced at the beginning of all makefiles.

```
include $(MAKEINC)/make.common
include $(MAKEINC)/make.$(ARCH)
```

File (**make.common**) is generic and configured for the local development environment as part of the ORPG SW installation procedures.

Other global makefiles are used in specific situations. For example, **make.cbin** is used with the single target binary make file for an ANSI-C binary and **make.subdirs** is used in the CPC level makefiles. Their use should be noted in the following examples.

CPC Level Makefiles

The makefile in the CPC level directory allows the compilation of all libraries and or tasks in the subdirectories with one command (and is also required to compile this CPC with the rest of the ORPG). The only modification to the template that is required is the listing of the subdirectories in the specific **cpc**. This makefile includes **make.subdirs** at the end. The following example is taken from **cpc007**.

```
# RCS info
# $Author: steves $
# $Locker:  $
# $Date: 2008/01/07 23:23:22 $
# $Id: cpc007.make,v 1.9 2008/01/07 23:23:22 steves Exp $
# $Revision: 1.9 $
# $State: Exp $

include $(MAKEINC)/make.common
include $(MAKEINC)/make.$(ARCH)

SUBDIRS = tsk001 \
        tsk002 \
        tsk003 \
        tsk004 \
        tsk006 \
        tsk008 \
        tsk009 \
        tsk011 \
        tsk012 \
        tsk013 \
        tsk014 \
        tsk015

CURRENT_DIR = .
include $(MAKEINC)/make.subdirs
```

Note: Using a CPC level makefile is not required for development work. If not used the sub tasks and libraries must be compiled individually.

Task / Library Level Makefiles

The makefile in the task / library level directory allows the compilation of all of the required objects for that task / library with one command. This file (called the "parent" makefile) references one or more single-target "children" makefiles which are named with a **.mak** extension. If generating a task, this makefile includes **make.parent_bin** after listing all of the children **BINMAKEFILES**. If generating a library, the makefile includes **make.parent_lib** after listing all of the children **LIBMAKEFILES**. The following example is from **cpc017/tsk010**.

```
# RCS info
# $Author: ccalvert $
# $Locker:  $
# $Date: 2003/07/03 20:46:03 $
# $Id: cpc017_tsk010.make,v 1.1 2003/07/03 20:46:03 ccalvert Exp $
# $Revision: 1.1 $
# $State: Exp $


include $(MAKEINC)/make.common
include $(MAKEINC)/make.$(ARCH)

BINMAKEFILES = mda3d.mak

include $(MAKEINC)/make.parent_bin
```

Single Target Binary Makefiles

These are the "children" makefiles which are named with a **.mak** extension. These makefiles provide the instructions to produce only a single binary executable (task) or single shared library, which is the convention followed in the ORPG. Typically only a few items from a template need modification. Examples for a FORTRAN binary, ANSI-C binary, FORTRAN library, and an ANSI-C library are provided.

IMPORTANT NOTE: Algorithms should no longer be written in FORTRAN. Though there is no firm time table, all current Legacy FORTRAN algorithms are being ported to ANSI-C during the next several build cycles. A list of recently ported algorithm tasks is provided in Appendix F of Volume 3 of the CODE guide.

- **FORTRAN Binary**
  For a single FORTRAN binary file the critical make variables include:
  - **FC_LOCAL_LIBRARIES** - The standard libraries needed by FORTRAN algorithms are defined by **F_ALGORITHM_LIBS** in **make.common**. Append any algorithm specific libraries if required.
  - **FSRCS** - list of FORTRAN source files.
  - **TARGET** - the name of the binary file.
  -

```
# RCS info
# $Author: ccalvert $
# $Locker:  $
# $Date: 2004/02/05 22:48:47 $
# $Id: cmprfape.mak,v 1.7 2004/02/05 22:48:47 ccalvert Exp $
# $Revision: 1.7 $
# $State: Exp $

# Template make description file for describing a Fortran binary file

include $(MAKEINC)/make.common
include $(MAKEINC)/make.$(ARCH)

# In case if there are any includes that are not in standard places.
# Normally this should always be blank.
FC_LOCAL_INCLUDES =
FPP_LOCAL_INCLUDES =
# In case if there are any local fortran defines.
FC_LOCAL_DEFINES =
FPP_LOCAL_DEFINES =

# This list has all the includes that are needed for the compile.
# Local copies of a file are given preference over the system
# location.
FC_ALL_INCLUDES = $(FC_STD_INCLUDES) $(FC_LOCALTOP_INCLUDES) \
                                $(FC_TOP_INCLUDES) $(FC_LOCAL_INCLUDES)
FPP_ALL_INCLUDES = $(FPP_STD_INCLUDES) $(FPP_LOCALTOP_INCLUDES) \
                                $(FPP_TOP_INCLUDES) $(FPP_LOCAL_INCLUDES)

# This is a list of all fortran defines.
FC_ALL_DEFINES = $(FC_STD_DEFINES) $(FC_OS_DEFINES) $(FC_LOCAL_DEFINES)
FPP_ALL_DEFINES = $(FPP_STD_DEFINES) $(FPP_OS_DEFINES)
$(FPP_LOCAL_DEFINES)

# If extra library paths are needed for this specific module.
FC_LIBPATH =

# These libraries are named the same on all architerctures at this time.
# Location of each of libraries depends on the architecture. e.g. ORPG HP
# libraries are located in $(TOP)/lib/hpux_rsk. If there are separate
# library names for different architectures, the below portion of the
makefile
# will have to be moved to $(MAKEINC)/make.$(ARCH).
# libraries specific to orpg
FC_LOCAL_LIBRARIES =  -laprcom $(F_ALGORITHM_LIBS)
FC_PURE_LOCALLIBS = $(FC_LOCAL_LIBRARIES) -lbzip2
FC_DEBUG_LOCALLIBS = $(FC_LOCAL_LIBRARIES) -lbzip2
FC_EXTRA_LIBRARIES = -lV3

# architecture specific system libraries and load flags.
FC_slrs_spk_LIBS = $(F_ALGORITHM_SYS_LIBS)
FC_slrs_x86_LIBS = $(F_ALGORITHM_SYS_LIBS)
FC_hpux_rsk_LIBS = -lV3
FC_slrs_spk_LDFLAGS =
FC_slrs_x86_LDFLAGS =
FC_hpux_rsk_LDFLAGS = +U77

# some system libraries, if needed.
FC_SYS_LIBS =
```

```
FCFLAGS = $(COMMON_FCFLAGS) $(FC_ALL_INCLUDES) $(FC_ALL_DEFINES)
PUREFCFLAGS = $(COMMON_PUREFCFLAGS) $(FC_ALL_INCLUDES) $(FC_ALL_DEFINES)
DEBUGFCFLAGS = $(COMMON_DEBUGFCFLAGS) $(FC_ALL_INCLUDES) $(FC_ALL_DEFINES)
FPPFLAGS = $(COMMON_FPPFLAGS) $(FPP_ALL_INCLUDES) $(FPP_ALL_DEFINES)
PUREFPPFLAGS = $(COMMON_PUREFPPFLAGS) $(FPP_ALL_INCLUDES)
$(FPP_ALL_DEFINES)
DEBUGFPPFLAGS = $(COMMON_DEBUGFPPFLAGS) $(FPP_ALL_INCLUDES)
$(FPP_ALL_DEFINES)

# We cannot "makedepend" Fortran source files ... it is important to
define
# DEPENDFILE to be "empty" (protects the depend command lines)
#DEPENDFLAGS =
DEPENDFILE =

FSRCS =   cmprfape.ftn \
     a307b1.ftn

# Following is for specifying any non-local object files (e.g., cpc-level
# object files)
ADDITIONAL_OBJS =

TARGET = cmprfape

include $(MAKEINC)/make.fbin
```

- **ANSI-C Binary**
  For a single C binary file the critical make variables include:
  - **`LOCAL_LIBRARIES`** - The standard libraries needed by C algorithms are defined by **`C_ALGORITHM_LIBS`**. These are: **`-ladaptstruct -lrpgc -lorpg -linfr -lbzip2 -lm`**. The library **`-lrpgc`** is specific to algorithms written in C and **`-ladaptstruct`** is for the new adaptation data mechanism. Algorithm unique libraries (if any) are included here after the standard libraries.
  - **`SRCS`** - list C source files.
  - **`TARGET`** - the name of the binary file. **NOTE:** having a 'space' after the target name causes multiple warnings, double compilation, and a link failure.

```
# RCS info
# $Author: ccalvert $
# $Locker:  $
# $Date: 2004/02/05 22:41:11 $
# $Id: recclalg.mak,v 1.7 2004/02/05 22:41:11 ccalvert Exp $
# $Revision: 1.7 $
# $State: Exp $


include $(MAKEINC)/make.common
include $(MAKEINC)/make.$(ARCH)


LOCAL_INCLUDES =
```

```
# You can also include architecture specific includes, if needed, by
# defining $(ARCH)_INC and then adding it to the list of ALL_INCLUDES.

LOCAL_DEFINES =

# You can also include architecture specific defines, if needed, by
# defining $(ARCH)_DEF and then adding it to the list of ALL_DEFINES.

ALL_INCLUDES = $(STD_INCLUDES) $(LOCALTOP_INCLUDES) $(TOP_INCLUDES) \
                        $(LOCAL_INCLUDES)

# This is a list of all defines.
ALL_DEFINES = $(STD_DEFINES) $(OS_DEFINES) $(LOCAL_DEFINES)

# If extra library paths are needed for this specific module.
# Specify $(X_LIBPATH) when appropriate ...
LIBPATH =

# These libraries are named the same on all architectures at this time.
# Location of each of libraries depends on the architecture. e.g. ORPG HP
# libraries are located in $(TOP)/lib/hpux_rsk. If there are separate
# library names for different architectures, the below portion of the
# makefile will have to be moved to $(MAKEINC)/make.$(ARCH).
# libraries specific to orpg
LOCAL_LIBRARIES = $(C_ALGORITHM_LIBS)

# Different order/set of libraries needed to build with debug information
DEBUG_LOCALLIBS = $(LOCAL_LIBRARIES)
GPROF_LOCALLIBS = $(DEBUG_LOCALLIBS)

PURE_LOCALLIBS = $(DEBUG_LOCALLIBS)
QUAN_LOCALLIBS = $(DEBUG_LOCALLIBS)
PRCOV_LOCALLIBS = $(DEBUG_LOCALLIBS)

EXTRA_LIBRARIES =

# Architecture and debug or profiling tool dependent linker options for
# slrs_spk
slrs_spk_LD_OPTS =

slrs_spk_DEBUG_LD_OPTS = -lnsl -lsocket -lelf -lrt
slrs_spk_GPROF_LD_OPTS = $(slrs_spk_DEBUG_LD_OPTS)

slrs_spk_PURE_LD_OPTS = $(slrs_spk_DEBUG_LD_OPTS)
slrs_spk_QUAN_LD_OPTS = $(slrs_spk_DEBUG_LD_OPTS)
slrs_spk_PRCOV_LD_OPTS = $(slrs_spk_DEBUG_LD_OPTS)

# Architecture and debug or profiling tool dependent linker options for
# lnux_x86
lnux_x86_LD_OPTS =

lnux_x86_DEBUG_LD_OPTS =
lnux_x86_GPROF_LD_OPTS = $(lnux_x86_DEBUG_LD_OPTS)

lnux_x86_PURE_LD_OPTS = $(lnux_x86_DEBUG_LD_OPTS)
lnux_x86_QUAN_LD_OPTS = $(lnux_x86_DEBUG_LD_OPTS)
lnux_x86_PRCOV_LD_OPTS = $(lnux_x86_DEBUG_LD_OPTS)

# Flags to be passed to compiler
```

```
CCFLAGS = $(COMMON_CCFLAGS) $(ALL_INCLUDES) $(ALL_DEFINES)

DEBUGCCFLAGS = $(COMMON_DEBUGCCFLAGS) $(ALL_INCLUDES) $(ALL_DEFINES)
GPROFCCFLAGS = $(COMMON_GPROFCCFLAGS) $(ALL_INCLUDES) $(ALL_DEFINES)

PURECCFLAGS = $(COMMON_PURECCFLAGS) $(ALL_INCLUDES) $(ALL_DEFINES)
QUANCCFLAGS = $(COMMON_QUANCCFLAGS) $(ALL_INCLUDES) $(ALL_DEFINES)
PRCOVCCFLAGS = $(COMMON_PRCOVCCFLAGS) $(ALL_INCLUDES) $(ALL_DEFINES)

# use following for makefile-specific makedepend flags
# (re: SYS_DEPENDFLAGS in make.$(ARCH)
DEPENDFLAGS =

SRCS = recclalg_main.c \
        recclalg_classifyEcho.c \
        recclalg_computeProbs.c

TARGET = recclalg

DEPENDFILE = ./depend.$(TARGET).$(ARCH)

include $(MAKEINC)/make.cbin

-include $(DEPENDFILE)
```

- **FORTRAN Library**
  For a single Fortran library the critical make variables include:
  - **LIB_FSRCS** - list of Fortran source files (.ftn).
  - **LIB_TARGET** - the name of the binary file.

```
# $Date: 1999/07/30 15:57:19 $
# $Id: libaprcom.mak,v 1.1 1999/07/30 15:57:19 steves Exp $
# $Revision: 1.1 $
# $State: Exp $

include $(MAKEINC)/make.common
include $(MAKEINC)/make.$(ARCH)

# In case if there are any includes that are not in standard places.
# Normally this should always be blank.
FC_LOCAL_INCLUDES =
FPP_LOCAL_INCLUDES =

# In case if there are any local fortran defines.
FPP_LOCAL_DEFINES =

# This list has all the includes that are needed for the compile.
# Local copies of a file are given preference over the system
# location.
FC_ALL_INCLUDES = $(FC_STD_INCLUDES) $(FC_LOCALTOP_INCLUDES) \
      $(FC_TOP_INCLUDES) $(FC_LOCAL_INCLUDES)
FPP_ALL_INCLUDES = $(FPP_STD_INCLUDES) $(FPP_LOCALTOP_INCLUDES) \
      $(FPP_TOP_INCLUDES) $(FPP_LOCAL_INCLUDES)

# This is a list of all fortran defines.
```

```
FC_ALL_DEFINES = $(FC_STD_DEFINES) $(FC_OS_DEFINES) $(FC_LOCAL_DEFINES)
FPP_ALL_DEFINES = $(FPP_STD_DEFINES) $(FPP_OS_DEFINES)
$(FPP_LOCAL_DEFINES)

FC_slrs_spk_LDFLAGS =
FC_slrs_x86_LDFLAGS =
FC_hpux_rsk_LDFLAGS = +U77

FCFLAGS = $(COMMON_FCFLAGS) $(FC_ALL_INCLUDES) $(FC_ALL_DEFINES)
PUREFCFLAGS = $(COMMON_PUREFCFLAGS) $(FC_ALL_INCLUDES) $(FC_ALL_DEFINES)
DEBUGFCFLAGS = $(COMMON_DEBUGFCFLAGS) $(FC_ALL_INCLUDES) $(FC_ALL_DEFINES)
FPPFLAGS = $(COMMON_FPPFLAGS) $(FPP_ALL_INCLUDES) $(FPP_ALL_DEFINES)
PUREFPPFLAGS = $(COMMON_PUREFPPFLAGS) $(FPP_ALL_INCLUDES)
$(FPP_ALL_DEFINES)
DEBUGFPPFLAGS = $(COMMON_DEBUGFPPFLAGS) $(FPP_ALL_INCLUDES) \
                $(FPP_ALL_DEFINES)

# We cannot "makedepend" Fortran source files ...
DEPENDFILE =

LIB_FSRCS = a30740.ftn \
        a30744.ftn \
        a30745.ftn \
        a30746.ftn \
        a30748.ftn \
        a30749.ftn \
        a3074a.ftn \
        a31483.ftn \
        a31484.ftn \
        a31485.ftn \
        a31486.ftn \
        a31487.ftn \
        a31488.ftn \
        a3148a.ftn \
        a3148b.ftn \
        a3148c.ftn \
        a3148e.ftn \
        a3148f.ftn \
        a3148h.ftn


LIB_TARGET = aprcom

clean::
 $(RM) $(ARCH)/*.f

# Following required only to support "getfiles" target ...
# Add header files, for example, to the list as required
GROUP = cpc101
getfiles::
 for file in $(LIB_FSRCS) ;\
 do \
       $(RAZOR) -c get -f $$file -g $(GROUP) -o ;\
 done


include $(MAKEINC)/make.cflib
```

- **ANSI-C Library**
  For a single C library the critical make variables include:
  - **LIB_CSRCS** - list of ANSI-C source files.
  - **LIB_TARGET** - the name of the binary file.

```
# RCS info
# $Author: dzittel $
# $Locker:  $
# $Date: 2005/02/17 16:13:47 $
# $Id: libsaa.mak,v 1.3 2005/02/17 16:13:47 dzittel Exp $
# $Revision: 1.3 $
# $State: Exp $

include $(MAKEINC)/make.common
include $(MAKEINC)/make.$(ARCH)

LOCAL_INCLUDES =

# You can also include architecture specific includes, if needed, by
# defining $(ARCH)_INC and then adding it to the list of ALL_INCLUDES.

LOCAL_DEFINES =
# You can also include architecture specific defines, if needed, by
# defining $(ARCH)_DEF and then adding it to the list of ALL_DEFINES.

# Append X_INCLUDES to the list of includes if you want to
# use include files for X and motif.
ALL_INCLUDES = $(LOCAL_INCLUDES) $(STD_INCLUDES) $(LOCALTOP_INCLUDES) \
         $(TOP_INCLUDES)

# This is a list of all defines.
ALL_DEFINES = $(STD_DEFINES) $(OS_DEFINES) $(XOPEN_DEFINES)
$(LOCAL_DEFINES)

CCFLAGS = $(COMMON_CCFLAGS) $(ALL_INCLUDES) $(ALL_DEFINES)
PURECCFLAGS = $(COMMON_PURECCFLAGS) $(ALL_INCLUDES) $(ALL_DEFINES)

DEBUGCCFLAGS = $(COMMON_DEBUGCCFLAGS) $(ALL_INCLUDES) $(ALL_DEFINES)

SHRDLIBLD_SEARCHLIBS = -lbzip2 -lz

LIB_CSRCS = build_saa_color_tables.c \
        padback.c \
        padfront.c \
        radial_run_length_encode.c \
        saa_max_value.c \
        short_isbyte.c \
        compute_area.c

LIB_TARGET = saa

DEPENDFILE = ./depend.lib$(LIB_TARGET).$(ARCH)
DEPENDFLAGS = -f $(DEPENDFILE)

include $(MAKEINC)/make.cflib

-include ./makedepend.$(ARCH)
```

Whether working in FORTRAN or ANSI-C, the 'local includes' and 'local defines' portions of the makefiles are normally blank for software that is being integrated into the ORPG. However, there are situations where local definitions are appropriate. These sections of the *binary makefiles* can be used for definitions that are not of general interest to the ORPG as a whole, as in the example provided for an ANSI-C library.

Use of C++

The ORPG contains some C++ source code however, use of C++ requires a waiver approved by the Radar Operations Center and is generally discouraged.

ORPG Components That Are Not Compiled

The ORPG contains non source code components such as executable scripts and configuration files. Part C. of this document discusses how to use the ORPG makefiles for these items.

Part B. Build Procedures

Normally recompiling the whole ORPG is not necessary and should be avoided.

Compiling Selected Libraries / Tasks

Using the makefiles described in the previous section, building software requires the following commands be issued from the appropriate directory.

Individual Libraries

For libraries execute these commands from the **~/src/cpcNNN/libNNN** directory.

```
chmod 0777 $HOME/lib/$ARCH/*.s?
make clean
make libinstall
chmod 0555 $HOME/lib/$ARCH/*.s?
```

The first and last commands are required to install shared libraries. For performance reasons, ORPG shared libraries are installed read only.

Individual Tasks

For executable tasks execute these commands from the **~/src/cpcNNN/tskNNN** directory.

```
make clean
make all
make install
```

All Tasks and Libraries in a CPC.

To build the entire contents of a CPC containing both libraries and executable tasks execute the following commands from the **~/src/cpcNNN** directory.

```
chmod 0777 $HOME/lib/$ARCH/*.s?
make clean
make libinstall
make all
make install
chmod 0555 $HOME/lib/$ARCH/*.s?
```

Additional Notes:

If new source code or include files are added, a **make depend** command could be used prior to the three commands listed to rebuild the dependency list at the end of the makefile. However, the **make depend**

command may not always handle dependencies correctly and the ORPG development team does not rely on it. Therefore the following guidance should be followed.

- Executing all three commands (- **`clean`**, - **`all`**, - **`install`**) in sequence is the norm.
- The - **`clean`** command can be omitted if the only change is to source code files.
- If only include files have changed or if linked static libraries have been modified, all three commands must be executed.

Recompiling the Entire ORPG

When Compiling the ORPG is needed

Normally recompiling the whole ORPG is not necessary and should be avoided. There are three situations where recompiling the whole ORPG is needed.

1. An include file used by more than one algorithm is modified. It may be easier to recompile the ORPG than all of the algorithms affected.
    a. One example is a modification of **`orpgdat.h`** when configuring a new non-product data store. Here the entire ORPG must be recompiled.
    b. Another example is a modification of an adaptation data include file used by more than one algorithm.

2. A system library is modified. This is not normally accomplished by an algorithm developer. There are two examples.
    a. A patch to a system library has been received to solve a specific issue.
    b. The developer has added a newly developed DEA access function to the adaptation data library **`libadaptstruct`**. This is not necessary for development purposes and is normally accomplished by the ROC when delivering a new algorithm for integration into the operational system.

3. Compiler options may have been changed for debugging purposes.

Procedures for recompiling the ORPG

Before compiling the ORPG steps must be taken to avoid losing any modified configuration files. All but one of these files are in the **`~/cfg`** directory tree. If guidance within this document and Volume 1 has been followed, all files specifically modified for CODE and the particular CODE account have backup copies saved. A list of relevant files is included in OPTION 2 below.

OPTION 1

This option works as long as the reason for recompiling the ORPG is not a patch that affects the contents of the **`~/cfg`** directory.
Rather than following the guidance for recompiling the ORPG in Volume 1, it is easier to make a backup of the entire contents of the **`~/cfg`** directory and subdirectories. The following steps safely accomplish a recompile of the entire ORPG.

1. Ensure there is a backup copy of the existing **.rssd.conf** file in the account home directory.

2. Rename the **~/cfg** directory.

3. Compile the ORPG by executing the following command from the account home directory.

```
make_rpg $HOME >& <your output filename>
```

4. Restore the previous **.rssd.conf** file in the account home directory.

5. Erase the entire contents of the new **~/cfg** directory and restore the saved **~/cfg** directory.

OPTION 2

If the patch was to **cpc104**, the contents of the **~/cfg** directory may be affected. In this case saving and restoring the existing contents of the **~/cfg** directory will not work. The compile procedures contained in Volume 1 may be more appropriate. Begin with the steps under "Compiling the ORPG Source Code" in Volume 1, Document 1 Section II and continue to the end of the section. The following is a list of files that are customized for the account and must be restored after compiling the ORPG.

The contents of the **.rssd.conf** file in the account home directory are customized for the account.

The files within the **~/cfg** directory that may have been modified during algorithm configuration are:

data_attr_table (if a snippet was not used)
product_attr_table (if a snippet was not used)
pgt/product_generation_tables
task_attr_table (if a snippet was not used)
task_tables (make sure you are using the appropriate version - NWS / Public)

The files within the **~/cfg** directory that may have been modified if using the **nbtcp** tool are:

comms_link.conf
tcp.conf
service_class_table (if the number of requests for a class was modified)

A new adaptation data definition file (DEA file) may have been added to the **~/cfg/dea** directory.

Configuration 'snippet' files may have been added to the **~/cfg/extensions** directory.

Summary of Additional Makefile Targets

Here is a summary of additional Makefile targets supported by the ORPG Makefile system (assuming the proper environment has been configured).

WARNING: Use caution using these additional targets. For example, the ORPG development team no longer uses the **depend** target because they found it unreliable.

Targets for Compiling and Installing Source Code	
clean	remove compiled binaries from source tree
depend	generate dependency list (not reliable)
all	generate a defined list of targets (executable binaries)
install	install binaries in appropriate location
liball	generate a defined list of libraries
libinstall	install libraries in appropriate location
Targets for Debugging and Profiling Utilities	
debugall	generate binaries for gdb
debuginstall	install binaries for gdb
gprofall	generate binaries for GNU profiler
gprofinstall	install binaries for GNU profiler
pureall	generate binaries for Purify
pureinstall	install binaries for Purify
quantifyall	generate binaries for Quantify
quantifyinstall	install binaries for Quantify

Part C. Installing Components Containing No Source Code

The ORPG makefile system is also used to install components that are not compiled: executable scripts and configuration files. There are two methods this is accomplished.

Using a Single Target Binary Makefile

These are the "children" makefiles that are named with a **.mak** extension. A modified ANSI-C binary makefile is used.

- **Script File or Configuration File**
  For a script or a configuration file the only make variable that matters is:
  - **TARGET** - the name of the script or configuration file to install.

  The single target makefile is modified as follows:

  - For an executable script, **make.script** is included near the end rather than **make.cbin**
  - For a configuration file, **make.cfg** is included rather than **make.cbin**

  The following excerpt is taken from the library that installs the basic text configuration files.

```
# RCS info
# $Author: ccalvert $
# $Locker:  $
# $Date: 2004/06/30 16:54:32 $
# $Id: alert_table.mak,v 1.2 2004/06/30 16:54:32 ccalvert Exp $
# $Revision: 1.2 $
# $State: Exp $

# Template make description file for describing a C binary file

include $(MAKEINC)/make.common
include $(MAKEINC)/make.$(ARCH)


LOCAL_INCLUDES =
          .                    .                    .
          .                    .                    .
          .                    .                    .
          .                    .                    .

ADDITIONAL_OBJS =

TARGET = alert_table

#DEPENDFILE = ./depend.$(TARGET).$(ARCH)

include $(MAKEINC)/make.cfg

#-include $(DEPENDFILE)
```

Omitting the Single Target Binary Makefile

A short cut method can be used to eliminate the need for the single target binary makefiles for scripts and configuration files. In this case additional information is included with the Task / Library level makefiles. This added information is equivalent to the contents of **make.script** and **make.cfg**.

This example Task / Library level makefile installs a script into the **~/tools/bin** directory (**$TOOLSCRIPTDIR**).

```
# RCS info
# $Author: ccalvert $
# $Locker:  $
# $Date: 2005/06/02 14:54:56 $
# $Id: cpc102_tsk001.make,v 1.1 2005/06/02 14:54:56 ccalvert Exp $
# $Revision: 1.1 $
# $State: Exp $

# This is the parent make description file for cpc_grep tool

include $(MAKEINC)/make.common
include $(MAKEINC)/make.$(ARCH)

install::
     @if [ -d $(TOOLSCRIPTDIR) ]; then set +x; \
     else (set -x; $(MKDIR) $(TOOLSCRIPTDIR)); fi
     $(INSTALL) $(INSTBINFLAGS) cpc_grep.script $(TOOLSCRIPTDIR)/cpc_grep
```

In this example, files in cpc102/tsk001 named "xyz.script" are renamed to "xyz" when installed. Note that with no **BINMAKEFILES** listed there is no need to include **make.parent_bin**.

This example Task / Library level makefile installs communications related configuration files into the **~/tools/data** directory (**$TOOLSDATADIR**) and the **~/tools/cfg** directory (**$TOOLSCFGDIR**).

```
# RCS info
# $Author: ccalvert $
# $Locker:  $
# $Date: 2005/02/24 23:20:03 $
# $Id: cpc104_lib002.make,v 1.13 2005/02/24 23:20:03 ccalvert Exp $
# $Revision: 1.13 $
# $State: Exp $

# This is the parent make description file for Task Data Files
# The data directory needs to be made here for the build_install_files script

include $(MAKEINC)/make.common
include $(MAKEINC)/make.$(ARCH)

install::
     @if [ -d $(TOOLSDATADIR) ]; then set +x; \
     else (set -x; $(MKDIR) $(TOOLSDATADIR)); fi
     @if [ -d $(DATADIR) ]; then set +x; \
     else (set -x; $(MKDIR) $(DATADIR)); fi
     @if [ -d $(TOOLSCFGDIR) ]; then set +x; \
```

```
     else (set -x; $(MKDIR) $(TOOLSCFGDIR)); fi
     $(INSTALL) $(INSTDATFLAGS) ktlx.map $(TOOLSDATADIR)/ktlx.map
     $(INSTALL) $(INSTDATFLAGS) klwx.map $(TOOLSDATADIR)/klwx.map
     $(INSTALL) $(INSTDATFLAGS) change_radar.dat
$(TOOLSCFGDIR)/change_radar.dat
```

Vol 2. Document 2 -
The ORPG Development Environment

Section III ORPG Configuration for Application Developers

This document covers the following topics

- Part A. Is an Introduction to Adding a New Algorithm
- Part B. Describes some Configuration and Naming Issues
- Part C. Covers the procedures for adding Tasks and Product Data Stores
- Part D. Describes the configuration of Non-product Data Stores

Special Instructions if upgrading from a previous version of the ORPG

Warning: Do not attempt to reuse the configuration from an installation of an earlier version of the ORPG. This includes the **`product_attr_table`**, **`task_attr_table`**, **`task_tables`**, **`data_attr_table`**, and the **`pgt/product_generation_tables`** configuration files.

Part A. Introduction to Adding a New Algorithm

This section describes configuration procedures for adding executable tasks (specifically, new algorithms), ORPG data stores (i.e., linear buffer files), and new products to the ORPG. These procedures will ensure that development algorithms will run in the ORPG environment. Section I of this document contains guidance concerning the organization of source code for new algorithms.

Before proceeding, please note the following:

1. Three (possibly four) configuration files have to be modified in order to add an algorithm to the ORPG. When using the *WSR-88D Algorithm API*, **an algorithm will not run if certain configuration information is not entered correctly**.
2. When editing configuration files, care must be taken to preserve the pairing of opening and closing braces in the tables. **This and other syntax errors in the configuration tables can prevent the ORPG from starting**.
3. After changing configuration files, certain data files in the ORPG data directory (**`$ORPGDIR`**) must be replenished in order for the changes to be recognized. This can be accomplished by using the **-p** option when starting the ORPG which deletes the contents of the product database, message logs, adaptation data, etc.

Summary of 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 CODE Guide Volume 3, Document 3, Section I, *Guidance for the Structure of Algorithms*, 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 this document.

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.

- **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 step 2 in Part C of this document).

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 Volume 3, Document 3, Section I, Part C - *Algorithm Initialization and Control Loop*.

Summary of Steps Required to Configure the ORPG for a new Algorithm Task

- **Determine the intermediate product data to be used as input.**
- **Determine the attributes needed to describe the algorithm task and product(s).**
- **Configure the task by modifying the `task_attr_table` and the `task_tables` configuration files.**
- **Configure the new product(s) by modifying the `product_attr_table` configuration file. For the product to be generated by default, the `pgt/product_generation_tables` file must also be modified.**
- **Configure any desired non-product data stores by modifying the `data_attr_table` configuration file.**
- **Using the Algorithm API, implement the algorithm.**

Part B. Configuration and Naming Issues

The current procedures for adding algorithms to the ORPG involve modification of 3 or 4 configuration files. Some of this information is redundant and must be consistent in all of the files. The critical information, which must be consistent and not conflict with other tasks / data stores, is listed in the following table.

SPECIAL NOTE: A **<Prod_Registration_Name>(<Prod_Buffer_Number>)** pair in the **task_attr_table** entry for each task is used to determine the data inputs and outputs for the algorithm. This mechanism makes it easy for a developer to change data inputs for testing or to configure different task names to use different inputs (see CODE sample algorithm 1). However even though it is not required, when an algorithm is formally integrated into the ORPG, the **<Prod_Registration_Name>** should be the same as the **<Prod_Buffer_Name>** in the **product_attr_table** file. Using a consistent name makes the data flow of multiple algorithms easier to understand.

Task Attributes		
ITEM	VALUE	DESCRIPTION
<Executable_Name>	A valid filename (typically lower case) of the executable file, for example **rad_refl** **Note 1**	Filename of the executable binary.
<Task_Name>	A unique name (typically lower case) that is used by the ORPG infrastructure to identify the task, for example **rad_refl** **Note 2**	This name is normally identical to the **<Executable_Name>**. A special use for a different name is the 'replay' instance of a task. **Note 2**

Note 1: The maximum length for an Executable Name is not documented. It is recommended that this name be kept to under 25 characters and ensure that the length of the complete pathname for the installed executable does not exceed 59. This limit also permits the **<Task_Name>** of the replay instance to not exceed 32 characters (see step 2 - *Configuring a New Task* in Part C of this document).

Note 2: Normally the **<Task_Name>** and **<Executable_Name>** should be the same. This would mean limiting their length to 25 characters. (The absolute maximum length for the **<Task_Name>** is 32 characters defined by **ORPG_TASKNAME_LEN 32** in **orpgtat.h**). There are specific cases where these names differ and this involves having two different tasks (**<Task_Name>**) having the same **<Executable_Name>**. Using multiple instances of a task is discussed in step 2 - *Configuring a New Task* in Part C of this document.

Product Data Attributes

ITEM	VALUE	DESCRIPTION
<Prod_Buffer_Name>	An alphanumeric (upper case by custom), for example **FAA_DIGVEL** **Note 1**	Prior to Build 9, this name was used to reference the product within the algorithm. It is still useful as a handle to refer to the data in documentation and conversationally so should remain unique.
<Prod_Registration_Name>	An alphanumeric (upper case by custom), for example **FAA_DIGVEL** **Note 1**	Beginning with Build 9, this name is used by the algorithm (via API calls and the contents of the **task_attr_table**) to refer to the input / output product data. There is no requirement for this name to be unique within the ORPG. ***Though not required, it is highly recommended that the unique <Prod_Buffer_Name> be used as the registration name.***
<Prod_Buffer_Number>	An integer ranging from **0** to **1999** **Notes 2 & 3**	The linear buffer number (also called "Product ID" or "Buffer ID")
<Product_Code>	An integer ranging from **0** to **1999** **Notes 2 & 3**	The code used by the legacy system to request products (also referred to as "pcode" or "message code") **Note 4**
<LB_filename>	A valid filename for a linear buffer in the form of ***.lb** **Note 5**	The linear buffer filename corresponding to the **<Prod_Buffer_Number>**

Note 1: The maximum length for a Product Name and a Product Registration Name is not documented. A maximum length of 25 characters should be used to avoid a breakdown of parsing the **task_attr_table**.

Note 2: Legacy products are assigned buffer numbers / product codes between 0 and 130. The buffer numbers and product codes are normally not the same value for legacy final products.

Note 3: New products are assigned butter numbers / product codes between 131 and 1999. The buffer numbers and product codes are the same for new final products. **Though any unused number between 131 and 1999 can be used, the block 1940-1989 has been reserved for development and can be used without the risk of being stepped on by newly integrated algorithms.** Buffer numbers 1990-1999 are reserved for CODE sample algorithms.

Note 4: Only final products (i.e., products distributed to users) are assigned a unique positive **<Product_Code>**. All intermediate products have a **<Product_Code>** of 0.

Note 5 The filename should be kept relatively short. The specific limit on the linear buffer filename appears to be 199 characters for the complete pathname of the file (**#define MAX_LBNAME_SIZE 200** in **orpgda.c**).

Intermediate products have a **<Product_Code>** of **0** while *final products* (those distributed outside the ORPG) have a unique positive integer value assigned.

Using base data (from the RDA) as an input is a special case. For example, two types of radial base data input buffers that are frequently used are.

- **<Prod_Buffer_Number> 79**, **REFLDATA**, provides reflectivity data only.
- **<Prod_Buffer_Number>** of **96**, **COMBBASE** provides Doppler data (velocity and spectrum width) in addition to reflectivity data. See Volume 2, Document 4, Section I, *Base Data Format*, for additional information.

There are many types of base data that can be used as input but all contain the basic moments. See Volume 2, Document 4, Section I, *Base Data Format*, for additional information.

Non-Product Data Attributes

ITEM	VALUE	DESCRIPTION
<Data_Buffer_Name>	An alphanumeric (upper case by custom), for example **SAAUSERSEL** **Note 1**	The name used by the ORPG to reference a particular data buffer (by referencing the buffer number)
<Data_Buffer_Number>	An integer ranging from **3000000** to **4000000** **Note 2**	The linear buffer number corresponding to the **<Data_Buffer_Name>** (also called "Buffer ID")
<Buffer_filename>	A valid upper case filename for a linear buffer in the form of ***.DAT** **Note 3**	The linear buffer filename corresponding to the **<Data_Buffer_Number>**

Note 1: The maximum length for a Data Name is not documented. It is recommended that it be kept to a maximum of 25 characters in length.

Note 2: A block of buffer numbers that can be used for non-product data stores has not been reserved for development use. The developer must ensure that the chosen buffer number is not previously used in either the **data_attr_table** configuration file or associated snippet files.

Note 3: The filename should be kept relatively short. The specific limit on the linear buffer filename appears to be 199 characters for the complete pathname of the file (**#define MAX_LBNAME_SIZE 200** in **orpgda.c**).

Part C. Adding Tasks and Product Data Stores

In addition to writing an application that correctly uses the appropriate interface to the ORPG services (using the API), the ORPG must be configured for the executable task, the persistent data store (linear buffer) for the product, and the output product itself. Currently, this is accomplished by editing three configuration files. The configuration of public non-product data stores is covered in Part D.
IMPORTANT: Use caution when editing these tables to retain the matching of open and close braces. Syntax errors in the configuration tables will result in incorrectly configured algorithms and can also prevent the ORPG from launching.

The example provided here is a simple task that inputs base data and outputs a final product. Algorithms made up of multiple tasks connected through intermediate products require corresponding configuration file entries for each task and product in the chain.

1. Preparation - Defining Configuration Parameters

The necessary parameters should be determined in advance.

In this example there are two input products for the task (**RECCLDIGREF** and **RECCLDIGDOP**) and two output products (**RECCLREF** and **RECCLDOP**) for the task. However **RECCLDIGREF** is the only input used for the final product (**RECCLREF**) in our example. See the contents of the **product_attr_table** file for this example.

The parameters used in this example are:

ITEM		VALUE
task parameters for the task to be added	**<Executable_Name>** Note 1	recclprods
	<Task_Name> Note 1	recclprods
input data parameters of an existing product	**<Prod_Registration_Name>** Note 2, 6	RECCLDIGREF
	<Prod_Buffer_Number> Note 3, 5	298
output data parameters for the product being added	**<Prod_Buffer_Name>** Note 2, 6	RECCLREF
	<Prod_Registration_Name> Note 2, 6	RECCLREF
	<Prod_Buffer_Number> Note 3, 5	132
	<Product_Code> (final product) Note 4, 5	132
	<LB_filename>	recclprodsref.lb

Note 1: The **<Executable_Name>** and the **<Task_Name>** are typically the same. There are specific cases where they differ and this involves having two different tasks (each with a unique **<Task_Name>**) having the same **<Executable_Name>**. Using multiple instances of a task is discussed in step 2 - *Configuring a New Task*.

Note 2: **Though not required, it is recommended that the unique <Prod_Buffer_Name> be used for the <Prod_Registration_Name>. Using this standard name for the product ID makes it easier to view and understand the data flow of multiple algorithms.**

Note 3: Safe linear buffer numbers *for new products* are any unused number from 131-1999. See the **product_attr_table** configuration file (including any snippet files in the **extensions** directory) to determine appropriate unused buffer numbers.

Note 4: Safe product codes *for new final products* are any unused code from 131 - 1999 and are identical to the linear buffer number. Intermediate products (not distributed outside the ORPG) are assigned a product code of 0.

Note 5: **Though any unused buffer number between 131 and 1999 can be used, the block 1940-1989 has been reserved for development and can be used without the risk of being stepped on by newly integrated algorithms.** Buffer numbers 1990-1999 are reserved for CODE sample algorithms.

Note 6: Maximum length for **<Prod_Buffer_Name>**, **<Prod_Registration_Name>**, **<Executable_Name>**, **<Task_Name>** and **<LB_filename>** is covered above in Part B. *Configuration and Naming Issues*

2. Configuring a New Task

Task Attribute Table in the $HOME/cfg/task_attr_table file

This file contains the task attribute table. In the operational system there is an entry in this table for every ORPG task. **For development tasks, the configuration should always be made in a 'snippet' file rather than directly in the task_attr_table configuration file.** The following paragraphs provide a brief discussion of these parameters with respect to this configuration example. A basic (however incomplete) description of these attributes is provided at the beginning of the **task_attr_table** configuration file. **If any changes are made to the task_attr_table file, ensure a backup copy is saved.**

The following figure is copied from the **task_attr_table** configuration file.

```
Task recclprods {
    filename        recclprods
    input_data      RECCLDIGREF(298) RECCLDIGDOP(299)
    output_data     RECCLREF(132) RECCLDOP(133)
    desc            "Radar Echo Classifier Clutter Likelihood Products"
    args
                    0 ""
}
```

Attribute Descriptions

Task name

The **<Task_Name>** is entered before the opening bracket of the table entry of the task, immediately after the keyword **Task**. If the **<Task_Name>** is not specified, the value of the **filename** attribute is used. This is a logical task name used by the ORPG infrastructure. The prefix **'replay_'** is reserved for a special purpose.

filename

The attribute **filename** refers to **<Executable_Name>** which is the name of the executable binary file. Though the maximum length of the **<Executable_Name>** is not documented, it is recommended not to exceed 25 characters.

input_data

A list of short product names (**<Prod_Registration_Name>**) followed by numbers in parentheses which are the corresponding (**<Prod_Buffer_Number>**)s. There is one pair of names and id's for each of the input data. For a data driven algorithm having more than one input, the "driving input" must be first in the list.

output_data

A list of short product names (**<Prod_Registration_Name>**) followed by numbers in parentheses which are the corresponding (**<Prod_Buffer_Number>**)s. There is one pair of names and id's for each of the output data.

desc

The **desc** attribute is a brief description of what the task does - recommended 64 characters maximum.

data_stream (normally not used)

This is a special purpose attribute. The value **1** specifies the input data stream is real-time and the value **2** specifies that the replay data stream is used. If the attribute is omitted, real-time is assumed. The normal method of specifying a replay instance of a task is to have a separate entry with a **<Task_Name>** identical to the real-time task with a prefix **replay_**. This attribute must be used to specify the replay data stream for tasks having no real-time stream counterpart.

args

The **args** attribute lists the command line arguments used when the ORPG task is launched following an ordinal instance number. An entry of **0** for the instance number represents the first (and in this example, the only) instance of a task. For algorithm tasks the arguments should normally be `0 ""`.

Using Multiple Instances of a Task

A mechanism is provided to configure multiple instances of an algorithm task and involves having multiple entries in the Task Attribute Table (**task_attr_table** file) using a different logical task name for each entry.

1. The primary purpose of this technique is to configure a 'replay' instance of a task. In this case a second entry for the task is used with a special task name which has the prefix '**replay_**' added to the first task name. All other attributes of the replay task configuration are identical to the first task. Tasks having a replay instance must either ingest the original (recombined) base data or warehoused intermediate products whit 255 priority. See *Responsiveness to One-Time Requests* in Part A of this document.

2. This technique can also be used to facilitate algorithm development / debugging by providing a means of having multiple instances of a task with different inputs / outputs using one set of source code. This technique is normally not needed (or useful) for operational algorithms, though it is used for those legacy tasks reading the replay data stream. This technique does have an impact on resource overhead. Using a different **<Task_Name>** and **<Executable_Name>** to create multiple instances of a task is also demonstrated in CODE sample algorithm 1.

Recommendation:

If multiple instances are considered for operational use, approval must be granted based on a cost/benefit analysis presented during the Design Approach Review.

Operational Process List in the `$HOME/cfg/task_tables` file

The only portion of this file that concerns an algorithm developer is the section titled **`Operational_processes`**. **For development tasks, the configuration should always be made in a 'snippet' file rather than directly in the `task_tables` configuration file.** Here the task's **<Task_Name>** must be included in the list in order for the task to launch when the ORPG is started. If the **<Task_Name>** is different than the **<Executable_Name>**, the task is launched with a **`-T`** option. For an example, look at the execution commands for the replay tasks in the output of the **`mrpg -v startup`** command. **If any changes are made to the `task_tables` file, ensure a backup copy is saved.**

When multiple instances of a task are configured in the **`task_attr_table`** configuration file, they must be listed individually in the **`task_tables`** configuration file. See the replay tasks in the following extract from **`task_tables`**.

```
Operational_processes {
# DE-ACTIVATED FOR CODE
#   cm_ping
    .
    .
   basrflct
    .
    .
   recclprods
   user_sel_LRM
    .
    .

# Replay input data stream tasks follow
   replay_basrflct
    .
    .
   replay_user_sel_LRM
}
```

If the algorithm task is not launched when the ORPG is started it can be launched from the command line with a simple command such as: **recclprods**. It should be noted however that because of the integrated nature of the ORPG services, an algorithm task must be 'installed' into the appropriate directory to be launched. Executing the compiled binary in the source code directory tree will not work.

3. Configuring a New Product

Product Attribute Table in the $HOME/cfg/product_attr_table file

The product attribute table contains parameters associated with both final products and intermediate products. In the operational system there is an entry in this table for every ORPG product. **For development products, the configuration should always be made in a 'snippet' file rather than directly in the product_attr_table configuration file.** The following paragraphs provide a brief discussion of some of these parameters with respect to this configuration example. A basic (however incomplete) description of these attributes is provided at the beginning of the **product_attr_table** configuration file. **If any changes are made to the product_attr_table file, ensure a backup copy is saved.**

The following figure is copied from the **product_attr_table** configuration file.

```
    Product {
    prod_id          132 RECCLREF
    prod_code        132
    gen_task         recclprods
    wx_modes         7
    disabled         0
    n_priority       4
    priority_list        70 70 70 70 70
    n_dep_prods      1
#   dependent products: RECCLDIGREF
    dep_prods_list       298
    desc         "CLR Clutter Likelihood Reflectivity: 11 level/0.54 nm"
    type             1
    alert            0
    warehoused       0
    elev_index       2
    path             base/recclprodsref.lb
    lb_n_msgs        10
# NOTE: Final products are links to product data base
    max_size         96
    params
                 2 -20 3599 0 10 "Elevation" "Degrees"
    }
```

Attribute Descriptions

We do not provide detailed guidance for all attributes listed. Some attributes (e.g., **priority_list**) need only to have representative values assigned prior to the final integration onto the operational system. Others (like **alert**) only apply to unique circumstances. Some attributes are always set to a

specific value for normal products (e.g., **disabled**). Others are specific to base data or radial data and are not described here (e.g., **class_id**, **class_mask**, **warehouse_id**, **warehouse_acct_id**).

The following attributes must be defined carefully in order for the algorithm to run in the ORPG environment.

prod_id

The **prod_id** attribute indicates the **<Prod_Buffer_Number>** of the product. In addition, a **<Prod_Buffer_Name>** is listed. Though no longer used by the ORPG infrastructure, it is recommended the attribute **<Prod_Buffer_Name>** be maintained as a convenient handle for referring to the product.

prod_code

A **prod_code** of **0** is used for intermediate products. For new final products this value is the same as the **<Prod_Buffer_Number>**

gen_task

The **gen_task** entry includes the logical name **<Task_Name>** (not the **<Executable_Name>**) corresponding to the task generating the product.

wx_modes

Set this value to 7 which indicates the product can be produced in all weather modes. This value may be changed when the product is integrated into an operational system. Precipitation(=4), Clear Air(=2), and Maintenance(=1) modes.

disabled

Set this value to 0 (=no).

n_priority

Set n_priority to 4 (one for each weather mode and a default)

priority_list <N1> <N2> <N3> <N4>

Enter a representative value (e.g., 75) for default value, maintenance mode, precip mode, and clear air mode in that order. This value may be changed when the product is integrated into an operational system.

NOTE: Entering the maximum value [255 255 255 255] results in the product always being scheduled for generation. A priority of 255 is used with the **warehoused** attribute to support replay tasks. There is one warehoused product that does not have a priority of 255 (this particular product does not require always being generated because it is either an optional input or an input to an algorithm using the **WAIT_ANY** form of the control loop).

compression

The **compression** attribute determines whether the product is compressed. If this attribute is missing or set to **0**, the product is not compressed. If set to **1**, bzip2 compression is applied. If set to **2**, zlib compression is applied. Intermediate products can be compressed in addition to final products. Only bzip2 compression, **1**, should be used for final products.

n_dep_prods

The product attribute table entry also includes a reference to other products used as inputs to the algorithm. The **n_dep_prods** attribute indicates the number of required product data inputs, in

this case **1**.

dep_prods_list

The buffer numbers (**<Prod_Buffer_Number>**) for required product data input(s) are listed after the **n_dep_prods** attribute. Optional product data should be included here (*there is one existing case where this is not done which should be ignored*). Another configuration that should not be used as an example is product 156, **NTDA_EDR**, which configures the other output of the task as a dependent input for this product.

 # dependent_products:

This is actually a comment line rather than an attribute. The short product names (**<Prod_Buffer_Name>**) corresponding to the numbers in the **dep_prods_list** only serve to make the configuration more readable.

n_opt_prods

(*Recently added attribute*) It is possible to designate non-driving dependent inputs as optional inputs. The **n_opt_prods** attribute indicates the number of optional inputs, if any. *Configuration of optional inputs is discussed further below.*

opt_prods_list

(*Recently added attribute*) The **<Prod_Buffer_Number>** for the optional input(s) are listed after the **opt_prods_list** attribute. *Configuration of optional inputs is discussed further below.*

desc

The **desc** attribute is a brief description of the product - recommended 64 characters maximum. There are two forms of this attribute.

- **For final products** the first character after the opening quote is not white space, the initial characters (all upper case) up to the first white space represent a product mnemonic, which is used by **hci** display screens. Currently the mnemonic is limited to 3 characters. The remainder of the text string is the product description.
- **For intermediate products** the first character after the opening quote is white space (and is discarded), the remainder of the string represents the product description without a product mnemonic.

type

The type attribute refers to the frequency that the product is generated.

- Base data are **type 5** for "Radial".
- Most meteorological products are either **type 0** for "Volume" or **type 1** for "Elevation".
- Products that are not produced on a regular schedule are **type 3** for "On Demand" and this type is normally not used for meteorological products based upon volumetric data (base data). Algorithms can output "On Demand" products even if not requested.
  SPECIAL CASE: If you have a product that is based upon volumetric data that is NOT produced every elevation or volume, consider using "Volume" type for the following reason. If this task reads an "Elevation" intermediate product rather than base data, defining the output product as "Volume" type rather than "On Demand" will ensure all elevations of the intermediate product are scheduled for generation.

The following types are either not supported or recommended for general use.

- There are no "Time" products or "External" products in the legacy system. "Time" products are not yet supported by the ORPG.
- There is a special case of "On Request" products that is not recommended for general use.

- A detailed discussion of product data types used for algorithm input and output and how they are related to each other is provided in *Volume 3 , Document 3, Section I Part C*.

The **type** attribute determines the actual data timing for product. The input/output data type definitions in **rpg_port.h** are used by the infrastructure and some deprecated API registration functions. The values for the **type** attribute are related to (but not the same as) the input/output data type definitions in **rpg_port.h**. See the topic "Input / Output Registration" in Volume 3, Document 2, Section I, Part A.

alert

Set this value to 0. Products can be paired with defined alert conditions and generated when the alert threshold is exceeded. The topic of alerting is beyond the scope of this document.

warehoused

One purpose of warehoused products is to support one-time requests using historical data (tasks having replay instances). In this case the priority in **priority_list** is set to 255. The **warehoused** attribute is only set for intermediate products (and some base data types). **A normal value of 0** stipulates that the intermediate product is not warehoused. Any value greater than 0 means the intermediate product is warehoused with a retention time (in seconds) specified. Warehoused intermediate products are stored in the product database linear buffer rather than the individual linear buffer. When warehoused, the configuration of the **lb_n_msgs** and the **max_size** attributes are the same as final products.

elev_index 2

The **elev_index** attribute should be used with all elevation final products and should always be set to **2**. *See the special note below concerning configuration of elevation products.* This attribute should be absent for all other products.

path

The **path** attribute specifies the existing subdirectory under the ORPG data directory (**$ORPGDIR**) in which the linear buffer is created and the name of that file (**<LB_filename>**).

no_create **DO NOT USE**

This attribute is NOT used with normal product configuration.

lb_n_msgs and
max_size

The **lb_n_msgs** and the **max_size** attributes determine the size of the product specific linear buffer. The maximum size of the linear buffer is based upon two limitations. There can be no more that 32K messages in a buffer and the file system limit of 2 GB on the size of a file.

- **For final products** (and warehoused intermediate products) typical values for **lb_n_msgs** are 10 for volume products and 40 for elevation products, and **max_size** is set to 96. This indicates that 10 / 40 messages of 96 bytes are in the product specific linear buffer. The **max_size** of 96 bytes represents an internal header that is created for all product messages. This header includes a reference into the product database where the final product is actually stored.
- **For intermediate products**, the **max_size** attribute must be larger than the maximum possible size of the product (including the 96 byte header). Recommend an initial configuration of 5 messages (**lb_n_msgs**) for volume intermediate products. For elevation intermediate products, the minimum number of messages (**lb_n_msgs**) should be no less than

the maximum number of elevations in a volume (currently 20). If fewer messages are configured, it can cause the algorithm task to be shutdown by the infrastructure during a volume restart.

- If **lb_n_msgs** is not specified a default value of 10 is used.
- A **max_size** of 0 is a special case and is not used with normal products.

params

The **params** attribute provides a description of the product dependent parameters used in the product request message. Except for the elevation angle parameter, the ***params*** attributes are **defined only for final products**.

- All parameters used in the request message for a product (including all elevation based products) must be defined here, otherwise the product will not be distributed. Product dependent parameters 1 - 6 correspond to **params** 0 - 5 in the product attribute table.
- Configuration of elevation intermediate products is a special case (see the special note below).

Comprehensive guidance concerning product dependent parameters is included in CODE Guide Volume 2 Document 3 Section IV - *Application Dependent Parameters*. A few examples are provided here.

Additional description of the params attribute

In the following example, the first two parameters (like most parameters) have a straight forward interpretation. **params 0** (product dependent parameter 1) represents Azimuth from 0 to 359.9 degrees in tenths of a degree (0 degrees is the default). **params 1** (product dependent parameter 2) represents Range from 0 to 124.0 NM in tenths of a NM (0 NM is the default). The third parameter representing elevation (the most common parameter in the system) is a special case.

```
params
        0     0   3599      0      10   "Azimuth"    "Degrees"
        1     0   1240      0      10   "Range"      "Nmiles"
        2   -20   3599      0      10   "Elevation"  "Degrees"
      index  min   max   default  scale   name         units
```

The elevation parameter must be used with all elevation final products. It's use is optional but recommended for elevation intermediate products. The elevation parameter is always **params 2** and is interpreted as follows.

The elevation is scaled in units*10 and can range from -2.0 units (**-20**) to plus 359.9 units (**3599**). The default value is 0 units. The actual interpretation is not that simple. Negative numbers actually represent slices rather than angles and the scale is not applied. This means -4 (unscaled) represents the first 4 elevations in a volume and -20 (unscaled) represents the first 20 elevations in a volume. Scaling is applied to positive numbers. Small positive numbers represent positive elevation angles (34 represents + 3.4 degrees) and a very large positive number represents negative elevations (3595 represents - 0.5 degrees). Note: It is not clear where the transition between representing positive angles and negative angles occurs. According to the ICD for RPG to Class 1 User, the maximum negative angle is -1.0 degrees and the maximum positive angle is 45.0 degrees. Currently the radar does not scan at negative elevation angles.

Special Note concerning configuration of Elevation Products

Guidance in this area has evolved. The latest was provided with Build 9.

A final product that is elevation based must be configured using both the **`elev_index 2`** attribute and the **`params 2`** definition of the elevation parameter.

Elevation based intermediate products can be configured in two ways: using both the **`elev_index`** attribute and the **`params 2`** definition of the elevation parameter and not defining either of these attributes.

Using these attributes for the configuration of elevation intermediate products is OPTIONAL but is recommended because in certain situations this saves resources.

In either case, when a down-stream volume product is requested all elevations of the intermediate product are scheduled for generation. If all down-stream consumer tasks create elevation products, only those elevations requested are scheduled for generation.

Comprehensive guidance concerning product dependent parameters in the request message and their relationship to the product dependent parameters in the final product message is included in CODE Guide Volume 2 Document 3 Section IV - *Application Dependent Parameters*.

Detailed Instructions for the Configuration of Optional Inputs

There are two mechanisms for configuring an input as optional.

1. New Method:
   Including the product id (**`<Prod_Buffer_Number>`**) in the **`opt_prods_list`** of the **`product_attr_table`** file and setting **`n_opt_prods`** accordingly. A default block time of 5 seconds is used.

2. Original Method:
   Using the **`RPGC_in_opt_by_name()`** registration command in the algorithm. This method allows a non-default block time to be specified.

Factors in Selecting the Method.
Current Recommendation:

The new method of listing the optional product IDs in the **`opt_prods_list`** attribute is recommended. The default block time of 5 seconds typically suffices. If there is a need to increase block time, the function **`RPGC_in_opt_by_name()`** can be called in the algorithm.

4. Configuring Product Generation

Default Product Generation List in the `$HOME/cfg/pgt/product_generation_tables`

The following description of the default generation table is taken from the beginning of the **pgt/product_generation_tables** configuration file.

```
prod_id: product ID - the product buffer number

wx_modes: 2 - precip mode; 4 - clear air mode; 6 - precip & clear air

gen_pr: generation period in number of volume scans

stor_retention: storage retention time in minutes

p1 - p6: product dependent parameters.

NOTE: If product is elevation-based, the elevation parameter can specify
      either a single elevation or multiple elevations. The format must
      be one of the following:

      "xx-yy" or "yy"

      where "xx" is either

           10 - all elevation cuts of the VCP are requested.  The
                parameter entry should read "10-0" in this case.

           01 - All elevations at and below the specified angle (defined
by
                "yy") are requested.  The format of the angle is defined
                below.

           11 - The lowest number of cuts (define by "yy") are requested.

           00 - A single elevation (define by "yy") are requested.  The
                format of the angle is defined below.

      The format of "yy" if "yy" denotes a single elevation angle, is
      (degree * 10) for positive angles or (3600 + degree * 10) for
      negative angles.

NOTE: If using "xx-yy" format, there can not be any white space within the
      specification unless the specification is quoted (e.g., xx - yy is
      incorrect, while xx-yy and "xx - yy" are correct).

When entering the elevation slice parameter for a product which is to
generated for different weather modes, unless the product is to be
generated for the exact elevation slices, separate entries for each
weather
mode should be made.  Furthermore, if the elevation slice parameter
specifies a range of elevations, then the same product can not be entered
for a specific elevation slice unless the product is for a different
weather mode.
```

IMPORTANT NOTE: Fully integrated algorithms do not produce products unless their output product is requested. One way of insuring this is to add an entry for the final product to the default product generation table in the **pgt/product_generation_tables** file. **If any changes are made to the pgt/product_generation_tables file, ensure a backup copy is saved.**

An entry in the table below (for our example **`product id 132`**) must be added because this product is not generated by default in the operational system. The entry includes the value 11-4 for the **`p3`** product dependent parameter. For elevation based products this means the 4 lowest elevation cuts are always generated.

```
Default_prod_gen {
#prod_id wx_modes gen_pr stor_reten  p1    p2    p3     p4    p5    p6
     2       6       1     180      UNU   UNU   01-45  UNU   UNU   UNU
     4       2       1     180      UNU   UNU   11-3   UNU   UNU   UNU
     4       4       1     180      UNU   UNU   11-3   UNU   UNU   UNU
     7       4       1     180      UNU   UNU   11-1   UNU   UNU   UNU
     8       6       1     180      UNU   UNU   11-1   UNU   UNU   UNU
     9       4       1     180      UNU   UNU   11-3   UNU   UNU   UNU
    10       6       1     180      UNU   UNU   11-4   UNU   UNU   UNU
     .                               .                              .
   132       6       1     180      UNU   UNU   11-4   UNU   UNU   UNU
     .                               .                              .
   151       6       1     180      UNU   UNU    UNU   UNU   UNU   UNU
   152       7       1     360      UNU   UNU    UNU   UNU   UNU   UNU
   298       6       1     180      UNU   UNU   10-0   UNU   UNU   UNU
   301       6       1     180      UNU   UNU   10-0   UNU   UNU   UNU
    }
```

Note: Product dependent parameter **p3** in the default generation table corresponds to **`params 2`** in the product attribute table.

5. The task_attr_table, task_tables, and product_attr_table snippets

New products and tasks should always be configured without editing the `product_attr_table`, `task_attr_table`, and `task_tables` configuration files directly. To use this method, special "snippet" files are placed into a subdirectory named **`extensions`** under the **`~/cfg`** directory. The naming conventions for the snippet files are **`product_attr_table.NNNNN`**, **`task_attr_table.NNNNN`**, and **`task_tables.NNNNN`** where the suffix "**`NNNNN`**" is any meaningful string.

For the CODE sample algorithms, all configuration information is collected into one set of snippet files (**`task_attr_table.sample_snippet`**, **`task_tables.sample_snippet`**, and **`product_attr_table.sample_snippet`**). However, in several situations the proper approach is to have a set of snippet files for each task and the products which are outputs from that task. In this case the suffix could be the **`<Task_Name>`**. This approach should be used when providing source code to another organization (including the ROC for integration into the ORPG) and when using a configuration management system requiring code check in and checkout.

The following examples contain the configuration entries for the CODE sample algorithm 1.

The **task_attr_table** snippet contains all of the information that would be placed into the task attribute list:

```
## Added for CODE
    Task sample1_base {
        filename        sample1_dig
        input_data      SR_REFLDATA(78)
        output_data     SR_DIGREFLBASE(1990)
        desc            "Create Sample 1 Product - 256-level Base
Reflectivity"
        args
                        0
    }

    Task sample1_raw {
        filename        sample1_dig
        input_data      REFL_RAWDATA(66)
        output_data     SR_DIGREFLRAW(1995)
        desc            "Create Sample 1 Product - 256-level Raw Reflectivity"
        args
                        0
    }
```

Notice that for sample algorithm 1 there are two entries for the executable task **sample1_dig** with the unique task names **sample1_raw** and **sample1_base**. The algorithm determines under which name it was launched and uses different input and output registrations.

The **task_tables** snippet contains all of the information that would be placed into the operational process list:

```
Operational_processes {

    sample1_base
    sample1_raw

}
```

Sample algorithm executable **sample1_dig** is started twice using the configured task names **sample1_base** and **sample1_raw**.

The **product_attr_table** snippet contains all of the information that would be placed into the product attribute list:

```
## Added for CODE
    Product {
        prod_id                 1990     SR_DIGREFLBASE
        prod_code               1990
        gen_task                sample1_base
        wx_modes                7
        disabled                0
        n_priority              4
        compression             1
```

```
        priority_list           89 89 89 89
        n_dep_prods             1
#       dependent products:     SR_REFLDATA
        dep_prods_list          78
        desc            "S1 Sample 1 - SR Base Reflectivity: 256 level/0.13 nm"
        type                    1
        alert                   2
        warehoused              0
        elev_index              2
        path                    base/sample1_base_refl.lb
        lb_n_msgs               10
# Note: Final products are links to product database.
        max_size                96
        params
                                2 -20 3599 0 10 "Elevation" "Degrees"
    }

## Added for CODE

    Product {
        prod_id                 1995     SR_DIGREFLRAW
        prod_code               1995
        gen_task                sample1_raw
        wx_modes                7
        disabled                0
        compression             1
        n_priority              4
        priority_list           89 89 89 89
        n_dep_prods             1
#       dependent products:     REFL_RAWDATA
        dep_prods_list          66
        desc            "S1 Sample 1 - SR Raw Reflectivity: 256 level/0.13 nm"
        type                    1
        alert                   2
        warehoused              0
        elev_index              2
        path                    base/sample1_raw_refl.lb
        lb_n_msgs               10
# Note: Final products are links to product database.
        max_size                96
        params
                                2 -20 3599 0 10 "Elevation" "Degrees"
    }
```

Note that the **gen_task** attribute contains the logical task name, not the executable name, of the task creating the product.

The default product generation list contained in the **pgt/product_generation_tables** configuration file is not supported by snippets.

6. The include files - a309.inc and a309.h

There is no longer a reason to modify these files. The new "by_name" product buffer access API functions (which should be used with all new algorithm development) eliminate the need to modify these files.

With new algorithm development, do not rely on the definitions in **`a309.h`** that map buffer numbers (or product ID) with product names as these are no longer maintained by the ROC.

7. Make backup copies of all modified configuration files

Note: These files are overwritten every time the complete ORPG is compiled.

8. Erase appropriate files in the ORPG data directory

For configuration changes to be recognized, certain data files in **`$ORPGDIR`** must be rebuilt during the next ORPG launch. This can be accomplished by using the **-p** option when starting the ORPG which deletes the contents of the product database, message logs, adaptation data, etc.

Part D. Non-product Data Stores

Configuration of Public Non-Product Data Stores

1. Preparation

The necessary parameters should be determined in advance.

ITEM		VALUE
data parameters for the buffer being added	**<Data_Buffer_Name>**	**SAAUSERSEL**
	<Data_Buffer_Number>	**300000**
	<Buffer_filename>	**SAAUSERSEL.DAT**

- Maximum length for **<Data_Buffer_Name>** is assumed to be 15 characters.
- There is no reserved block of data buffer numbers that can be used for development. The developer must ensure that the number and name have not been previously used in the **data_attr_table** configuration file or associated snippet file.

2. The $HOME/cfg/data_attr_table configuration file

Data stores configured in the **data_attr_table** configuration file are considered *Public* ORPG data stores. They are created automatically upon startup and managed like product data stores. *Private* non-product data stores are not configured in **data_attr_table** and are standard disk files controlled by the algorithm.

Public non-product data stores are configured in a similar manner to product data stores by using the data store attribute table in the **data_attr_table** configuration file. **For development data stores, the configuration should always be made in a 'snippet' file rather than directly in the data_attr_table configuration file.** A more complete reference for non-product data attributes remains to be developed. **If any changes are made to the data_attr_table file, ensure a backup copy is saved.**

The following figure is copied from the **data_attr_table** configuration file.

```
Datastore {
    data_id         300000 SAAUSERSEL
    path            snow/SAAUSERSEL.DAT
    persistent
    Lb_attr {
        remark      "SAAUSERSEL.DAT"
        msg_size    0
        maxn_msgs   31
        types       "LB_REPLACE"
        tag_size    32
    }
}
```

We do not provide detailed guidance for all attributes used in the **data_attr_table** file. Only those attributes and attribute values that are meaningful for use in configuration of an algorithm non-product data store are described.

data_id

The **data_id** attribute indicates the **<Data_Buffer_Number>** and the internal name, **<Data_Buffer_Name>**, of the non-product data. Though the limit is not documented, keep the **<Data_Buffer_Name>** a maximum of 25 characters.

path

The **path** attribute specifies the existing subdirectory under the ORPG data directory (**$ORPGDIR**) in which the linear buffer is created and the filename of that file (**<Buffer_filename>**).

persistent

If this attribute is specified, the data store is not erased during a ORPG start using **mrpg -p startup**.

Lb_attr

The following attributes actually configure the linear buffer file behavior.

remark

For the purposes of algorithm data stores, the remark usually is a string containing the filename. The maximum length of the string is 63 characters (not including the enclosing quotes). **Note 1**

msg_size

Using 0 as the message size configures the buffer for varying length messages. This typically is used for *message database* type linear buffers. It permits the buffer to be a variable size while containing messages of different sizes. If a value other than 0 is specified, it represents the average size of messages in bytes. This is typically used for *message queue* type buffers. **Note 2**

maxn_msgs

The maximum number of messages that a buffer may contain. If the msg_size is not 0, then this attribute along with **msg_size** determines the size of the buffer file (typically with a *message queue* type buffer). If not specified, a default value of 40 is used. **Note 2**

types

Their are several values that can be OR'd together to determine the buffer type. For the purposes of an algorithm non-product data store most of the defaults are appropriate. The type **LB_DB** (replacing the **LB_REPLACE** type) configures the buffer as a *message database* type buffer. **LB_DB** can exhibit either the behavior of the **LB_REPLACE** or **LB_MSG_POOL** type. For details see Part I *Non-Product Data Access* in Volume 3, Document 2, Section II.

The default is a *message queue* type linear buffer (**LB_QUEUE**)

tag_size

This attribute can be omitted.

Note 1: The maximum length is defined by the value of **LB_REMARK_LENGTH 64** in **lb.h**.

Note 2:	The maximum size of the linear buffer is based upon two limitations. There can be no more that 32K messages in a buffer and the file system limit of 2 GB on the size of a file.

3. The data_attr_table snippet file.

New non-product data stores should always be configured without editing the `data_attr_table` configuration file directly. To use this method, special "snippet" file as described under item 5 in Part C. *Adding Tasks and Product Data Stores*.

The following example contains the configuration entries for a future CODE sample algorithm.

```
Datastore {
    data_id     399999 SAMPLE5_ENVIRON_DATA
path        sample_alg/environ_data.lb
persistent
    Lb_attr {
    remark      "SAMPLE5_ENVIRON_DATA"
        msg_size    0
        maxn_msgs   2
        types   "LB_REPLACE"
    }
    write_permission {
        2   sample5
        *   *
    }
}
```

The **`write_permission`** attribute means that only task **`sample5`** can write message ID 2 in the data store. Any task can write all other message IDs.

4. The include files - orpgdat.h and orpgdat.inc

The include files, **`orpgdat.h`** and **`orpgdat.inc`**, do not have to be modified in order for the new algorithm task to run in the ORPG environment. However, when the algorithm is formally integrated into the operational system these changes are made for non-product data stores. **If any changes are made to the `orpgdat.h` file, ensure a backup copy is saved.**

The only known consequence of not modifying these files is that the corresponding product buffers must be referenced by their number rather than their name when using the API calls. In order to use buffer names with the API calls (that is, use a globally defined constant rather than a number for the input parameter), the following changes must be made before the algorithm source code **and the ORPG code** is compiled.

When the algorithm is formally integrated into the operational system, these include files are modified to include the following changes.

- For algorithms written in FORTRAN, using our configuration example, a parameter **SAAUSERSEL** with value **300000** would be defined in **$HOME/include/orpgdat.inc**

```
C* data store for Snow Accumulation Algorithm
      INTEGER SAAUSERSEL
      PARAMETER( SAAUSERSEL = 300000 )
```

- For algorithms written in ANSI-C, using our configuration example, a constant **SAAUSERSEL** with the value **300000** would be defined in **$HOME/include/orpgdat.h**

```
/* Data store to support Snow Accumulation Algorithm */
#define SAAUSERSEL                300000
```

Algorithm API Support for Non-Product Data Stores

The algorithm API has supports access of product data stores, non-product data stores, and adaptation data. These functions are documented in Volume 3.

Public Non-product Data Stores

Public data stores are implemented as linear buffers. The algorithm API provides functions to open and close the buffers and to read and write messages in the buffers.

Private Non-product Data Stores

Private data stores are implemented as standard disk files. The only API support provided is to assist in providing the complete path name to the data store.

Name and location of non-product data stores

Public Non-product Data Stores

The name and location (a subdirectory under **$ORPGDIR**) are specified in the **data_attr_table** configuration entry. The name is usually all upper case letters with a **.DAT** suffix.

Private Non-product Data Stores

The location of the file is provided by the API function and is the current ORPG working directory (**$HOME/tmp**).

Part E. Adaptation Data

Instructions for changing site specific adaptation data are included in Section IV of this guide.

Instructions for creating and installing algorithm specific adaptation data are in CODE Guide Vol 2, Document 4, Section II - *Algorithm Adaptation Data - Configuration & Use*.

Vol 2. Document 2 -
The ORPG Development Environment

Section IV Configuring Site Specific Adaptation Data

Introduction

The ORPG can produce valid products using Archive II data as its source of radar data. However, site-related adaptation data from the original site is not passed on the ORPG. Instead, the site adaptation data configured on the development ORPG will be used in algorithm processing. This brief guide will describe how to modify the adaptation data to make it correspond to the input data source.

This data should be changed because:

- Antenna location and elevation are part of a valid WSR-88D product header information. This information will be incorrect if the site data is not set.
- If the algorithm depends upon the location and height of the radar, the product itself will contain errors.
- County background maps will not display if the radar location is incorrect.

The site adaptation data are contained in a text configuration file named **`site_info.dea`**. A blockage data file provides the beam blockage information caused by nearby terrain and significant man made structures. CODE only provides a generic blockage file. If accomplishing precipitation analysis requiring this data, it can be obtained from the ROC.

If you are interested, a tabular layout of the site adaptation data for all WSR-88D sites is provided in Appendix A.

Instructions

1. The ORPG **`change_radar`** utility is used to set the site adaptation data. For **`change_radar`** to work as intended with the CODE algorithm development environment, the environmental variable **`ORPG_NONOPERATIONAL`** must be defined (this is accomplished during CODE setup in Volume 1 of the CODE guide).

2. In order to change site adaptation data, execute the **`change_radar`** script from the command line while logged in to the account into which the ORPG is installed. The four letter ICAO identifier is the primary parameter. Two additional parameters are useful in the development environment.

   `-s` This optional flag prevents an automatic shutdown of the RPG software (if running).

   `-R` This flag prevents an automatic restart of the RPG software when the script is finished. **This flag should always be used in the development environment because the restart command used does not include the `-p` flag.**

For example, if the input data (Archive II tape, Archive II disk files, or BDDS on a LAN) are from Melbourne, FL, the command would be:

```
change_radar -r kmlb -S -R
```

The output of the script looks like this:

```
SITE ADAPTATION DATA HAS BEEN CHANGED TO THE FOLLOWING:

RPG ICAO:       kmlb
RPG ID:         302
RDA LATITUDE:   28113
RDA LONGITUDE:  -80654
RDA ELEVATION:  116
```

This corresponds to the following fields in **site_info.dea**

```
site_info.rpg_name = KMLB
site_info.rda_lat = 28113
site_info.rda_lon = -80654
site_info.rda_elev = 116
site_info.rpg_id = 302
```

3. In order to replace the existing binary adaptation data files, they must be erased before the next ORPG start. This can be accomplished by using the **-p** option with the ORPG start command: **mrpg -p startup**.

The **change_radar** utility has other options including the capability to list all supported radars and to interactively enter the desired site data. See the man page or execute **change_radar -h**.

Volume 2. ORPG Application Software Development Guide

Document 3. WSR-88D Final Product Format

This document contains helpful technical information concerning ORPG internals and also provides guidance in certain areas. The information presented here is independent of writing algorithm source code but does contain some references to the Application Programming Interface (API). CODE Guide Volume 3 - *WSR-88D Algorithm Programming Guide* contains the tutorial, reference, and sample algorithms for the *WSR-88D Algorithm API* and guidance for the structure of algorithms.

Section I Product Block Structure

An introduction to the WSR-88D product ICD format. The high level block structure and the contents of the product header information are covered. This section consolidates some of the information contained in the Interface Control Document (ICD) for the RPG to Class 1 User, document 2620001.

Section II Traditional Product Data Packets

The original WSR-88D used over 30 data packets in various ways to construct weather products. There are several packets which contain text information and several data packets for drawing vectors or lines on the graphic display. Several packets are used to represent special symbols. There is a packet used to represent radial data (polar coordinate) intended for display and another to represent raster data (rectangular coordinate) intended for display. There is a data packet used to represent 8-bit radial data that was not originally intended for display.

Section III Generic Product Components

This generic data packet (packet 28) can be relatively self-descriptive if correctly used. There are several types of *grid* components that can be used to represent two dimensional binary data. The *radial* component can be used to contain radial data in various formats. The *area* component can be used to represent single geographic points, a geographic line or an enclosed geographic area. The *table* component is used to represent text information in an organized tabular format. The *text* component is used to represent simple text.

Section IV ORPG Application Dependent Parameters

Application dependent parameters can be used to provide customizing parameters via the product request message that can be used to change the nature of the product for that specific request. They can also be used to provide additional fields of information in the formatted final product. This section

explains the relationship between the product specific parameters contained in the request message and the product dependent parameters in the product and provides some rules to maintain consistency in use.

Vol 2. Document 3 - WSR-88D Final Product Format

Section I Product Block Structure

Part A. Introduction

The ORPG transmits products to users in the Graphic Product Message Format described in this document. Recently a new concept of structuring the data content of WSR-88D products has been introduced resulting in 2 kinds of products.

1. **Traditional Products**
   The original WSR-88D used over 30 data packets in various ways to construct weather products. There are several packets which contain text information to be used in a very specific manner in different parts of the product. There are also several data packets for drawing vectors or lines on the graphic display, some for mono color lines and some variable color. Several packets are used to represent special symbols and some of these are nested within other packets. There is a packet used to represent radial data (polar coordinate) intended for display and another to represent raster data (rectangular coordinate) intended for display. There is a data packet used to represent 8-bit radial data that was not originally intended for display.

   One common thread in many of these traditional data packets is that the packet data includes display information to some extent. The display content ranges from a specific encoding of display labels, using coordinates that assume a specific display screen resolution, and in some cases contain pixel size / coordinate information used to draw the specific symbols.

   Guidance for the use of data packets in within the *Symbology Block* and use of multiple layers within the *Symbology Block* is provided in Section II - *Traditional Product Data Packets* of this document.

2. **Generic Products**
   A new type of data packet has been developed that contains no display information. This generic data packet (packet 28) can be relatively self-descriptive if correctly used. The generic data packet is a collection of generic components each with a different purpose. There are several types of *grid* components that can be used to represent two dimensional binary data in rectangular or polar coordinates. The *radial* component can be used to contain radial data in various formats. The *area* component can be used to represent single geographic points, a geographic line or an enclosed geographic area. The *table* component is used to represent text information in an organized tabular format. The *text* component is used to represent simple text.

   A primary advantage of using generic components is the flexibility of the information that can be contained in a product without defining new structures or packet types. Another advantage is separation of the look and feel of the display from the product content. One disadvantage is that in some cases the display system may require more modification for a new generic product.

   Guidance for the use of data packet 28 within the *Symbology Block* is provided in Section III - *Generic Product Components* of this document.

Part B. Content of Final Products

This recent WSR-88D Product Specification document  **2620003t_prod_spec.pdf** (located in the **--/pdf_doc/** directory on the CODE CD) contains a detailed description of all final products.

With traditional products this specification can be difficult to interpret if not familiar with the WSR-88D documentation. This is due to the document containing information beyond a specification of the contents of the products. For example:

- The specification also includes information reflecting PUP functionality (that is, how the PUP displays the products to include screen layouts and actual colors).
- The term "Product Interactions" describes which products the PUP (or AWIPS) operator can overlay on top of the product being described. It would have been better to just define the product as either a geographic product (underlay), a geographic overlay product, or a non geographic product.
- Appendix A of the ICD lists standard annotations that are displayed; defines special symbols and characters; and for various products, lists what additional annotations are to be displayed on the PUP / AWIPS. Much of the material in Appendix B of the ICD is a description of the layout of the PUP / AWIPS display screen. Appendix C of the ICD addresses the format of the Tabular Alphanumeric Block (TAB) for current products.

Part C. Structure of Final Products

The ORPG transmits products to users in the Graphic Product Message formats shown below. The format for the Graphic Product is documented in the *Interface Control Document (ICD) for the RPG to Class 1 User, Document number 2620001*. Care should be used when referring to this document because the document contains some inconsistencies in the numbering of figures. A recent version is provided in the  **2620001u_rpg_class1.pdf** file located in the **--/pdf_doc/** directory on the CODE CD.

A description of the contents of the header fields and the format of the major product blocks is provided here. Guidance for the use of the data packets within the *Symbology Block* and use of multiple layers within the *Symbology Block* is provided in Section II - *Traditional Product Data Packets* and Section III - *Generic Product Components* of this document.

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 shown.

4. Though not explicitly stated in the *ICD for the RPG Class 1 User* (mandatory requirements):
    a. All 2-byte and 4-byte data fields must begin at an even numbered byte offset from the beginning of the product message.
    b. 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.
    c. 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.

5. Rules for the structure of the layers and data packets within the *Symbology Block* are provided in Section II of this document.

The products are stored in the product database in this format and the ORPG completes the contents of the *Message Header Block* when the products are transmitted.

Documentation Note: Some terms used in the ORPG software have a different (and somewhat confusing) meaning when compared with existing documentation. For example, an ANSI-C structure named **Graphic_Product** represents only the MHB and the PDB, not the complete Graphic Product. There is another structure named **Prod_header** that does not refer to the MHB and PDB, but rather to another header used by the internal ORPG infrastructure.

Traditional Product Block Structure

Block Name	Halfword Number (2 bytes)
Message Header Block (MHB)	1 - 9
Product Description Block (PDB)	10 - 60
Product Symbology Block (variable length)	5 halfword header plus Data Layers containing traditional data packets.
Graphic Alphanumeric Block (GAB) (variable length)	5 halfword header plus Text Packet Pages
Tabular Alphanumeric Block (TAB) (variable length)	6 halfword header plus additional MHB and PDB, plus Character Data Pages

Generic Product Block Structure

Block Name	Halfword Number (2 bytes)
Message Header Block (MHB)	1 - 9
Product Description Block (PDB)	10 - 60
Product Symbology Block (variable length)	5 halfword header plus a data layer containing data packet 28 (generic product data packet)

The generic product format is relatively new. Formal guidelines for the structure and use of this product are evolving. Though not explicitly prohibited, it is recommended that generic products do not contain a GAB or a TAB and that the symbology block does not contain any traditional data packets.

Message Header Block

NOTE: WSR-88D products contain 4-byte, 2-byte, and 1-byte data fields. Since the ORPG is supported on little Endian architectures, the data must be written into the product in a very specific manner. Properly accounting for the byte-swapping infrastructure and the alignment of data fields is described in CODE Guide Volume 3, Document 3, Section III - *Writing Product Data Fields*.

The Message Header Block is included in all radar messages. It is of fixed length and consists of 9 halfwords (18 bytes). It contains important details about decoding the rest of the message. A listing of WSR-88D message codes (or product codes) is contained in Table III of the RPG to Class 1 User ICD.

The Message Header Block Format

Contents	Halfword (2 bytes)
MESSAGE CODE	01
DATE OF MESSAGE	02
TIME OF MESSAGE (MSW)	03
TIME OF MESSAGE (LSW)	04
LENGTH OF MESSAGE (MSW)	05
LENGTH OF MESSAGE (LSW)	06
SOURCE ID	07
DESTINATION ID	08
NUMBER OF BLOCKS	09

Message Header Block Field Descriptions

The field descriptions are documented in Figure 3-3 of the ICD for the RPG to Class 1 User. Note that the type notations are based on the original FORTRAN documentation. **`INT*4`** represents a 4 byte (32 bit) integer and **`INT*2`** represents a 2 byte (16 bit) integer.

All date-times are GMT.

HALFWORD	FIELDNAME	TYPE	UNITS	RANGE	PRECISION / ACCURACY	REMARKS
01	Message Code	INT*2	N/A	*[-300]* to -16, 0 to *+[1999]*	N/A	NEXRAD Message Code (or product code) defined in Table III of the RPG to Class 1 User ICD. *Note: The original upper limit for product codes was 300. This has been increased to 1999.*

02	Date of Message	INT*2	Julian Date	1 to 32,767	1	Modified Julian Date at time of transmission (number of days since 1 January 1970, where 1=1 January 1970). To obtain actual Julian Date, add 2,440,586.5 to the modified date
03-04	Time of Message	INT*4	Seconds	0 to 86,399	1	Number of seconds after midnight, Greenwich Mean Time (GMT).
05-06	Length of Message	INT*4	N/A	18 to 409856	1	Number of bytes in message including header
07	Source ID	INT*2	N/A	0 to 999	1	Source (originator's) ID of the sender
08	Destination ID	INT*2	N/A	0 to 999	1	Destination ID (receiver's) for message transmission
09	Number Blocks	INT*2	N/A	1 to 51	1	Header Block plus the Product Description Blocks in message

NOTE: Several fields are modified when the product is distributed. Prior to distribution the 'Date of Message' and 'Time of Message' are product generation time. After distribution these fields are set to the product distribution time. The Destination ID is '0' prior to distribution.

Product Description Block

NOTE: WSR-88D products contain 4-byte, 2-byte, and 1-byte data fields. Since the ORPG is supported on little Endian architectures, the data must be written into the product in a very specific manner. Properly accounting for the byte-swapping infrastructure and the alignment of data fields is described in CODE Guide Volume 3, Document 3, Section III - *Writing Product Data Fields*.

The Product Description Block contains identifying information about the product including the site that produced the product and the date and time of production. The contents of many fields within this block are product dependent. The use of the 10 product dependent parameters and the 16 data level thresholds is discussed in Section IV of this document, *ORPG Application Dependent Parameters*. The product dependent parameters contain the input parameters (if any) in the product request message and also return other data. The data level threshold fields are used to define the color tables for many traditional products. For products with no defined color table, these parameters have different uses.

The Product Description Block Format

Contents	Halfword (2 bytes)
BLOCK DIVIDER (-1)	10
LATITUDE OF RADAR (MSW)	11
LATITUDE OF RADAR (LSW)	12
LONGITUDE OF RADAR (MSW)	13
LONGITUDE OF RADAR (LSW)	14
HEIGHT OF RADAR	15
PRODUCT CODE	16
OPERATIONAL MODE	17
VOLUME COVERAGE PATTERN	18
SEQUENCE NUMBER	19
VOLUME SCAN NUMBER	20
VOLUME SCAN DATE	21
VOLUME SCAN START TIME (MSW)	22
VOLUME SCAN START TIME (LSW)	23
PRODUCT GENERATION DATE	24
PRODUCT GENERATION TIME (MSW)	25
PRODUCT GENERATION TIME (LSW)	26
PRODUCT DEPENDENT (P1)	27
PRODUCT DEPENDENT (P2)	28
ELEVATION NUMBER	29
PRODUCT DEPENDENT (P3)	30
DATA LEVEL 1 THRESHOLD	31

DATA LEVEL 2 THRESHOLD	32
DATA LEVEL 3 THRESHOLD	33
DATA LEVEL 4 THRESHOLD	34
DATA LEVEL 5 THRESHOLD	35
DATA LEVEL 6 THRESHOLD	36
DATA LEVEL 7 THRESHOLD	37
DATA LEVEL 8 THRESHOLD	38
DATA LEVEL 9 THRESHOLD	39
DATA LEVEL 10 THRESHOLD	40
DATA LEVEL 11 THRESHOLD	41
DATA LEVEL 12 THRESHOLD	42
DATA LEVEL 13 THRESHOLD	43
DATA LEVEL 14 THRESHOLD	44
DATA LEVEL 15 THRESHOLD	45
DATA LEVEL 16 THRESHOLD	46
PRODUCT DEPENDENT (P4)	47
PRODUCT DEPENDENT (P5)	48
PRODUCT DEPENDENT (P6)	49
PRODUCT DEPENDENT (P7)	50
PRODUCT DEPENDENT (P8)	51
PRODUCT DEPENDENT (P9)	52
PRODUCT DEPENDENT (P10)	53
VERSION (MS BYTE) SPOT BLANK (LS BYTE)	54
OFFSET TO SYMBOLOGY (MSW)	55
OFFSET TO SYMBOLOGY (LSW)	56
OFFSET TO GRAPHIC (MSW)	57
OFFSET TO GRAPHIC (LSW)	58
OFFSET TO TABULAR (MSW)	59
OFFSET TO TABULAR (LSW)	60

Product Description Block Field Descriptions

The field descriptions are documented in Figure 3-6 (sheet 6) of the ICD for the RPG to Class 1 User. Note that the type notations are based on the original FORTRAN documentation. **`INT*4`** represents a 4 byte (32 bit) integer and **`INT*2`** represents a 2 byte (16 bit) integer and **`INT*1`** represents a 1 byte integer.

All date-times are GMT.

Note regarding ingest of historical radar data: When ingesting historical radar data into the ORPG, the Generation Data and Time are the current time and the Volume Data and Time are based upon when the data was recorded. The radar data input tool '**`play_a2`**' has a switch to force the Volume Date Time to the current date time.

HALF WORD	FIELDNAME	TYPE	UNITS	RANGE	PRECISION / ACCURACY	REMARKS
10	Block Divider	INT*2	N/A	-1	N/A	Integer value of -1 used to delineate the header from the Product Description Block
11 - 12	Latitude of Radar	INT*4	Degrees	-90 to +90	0.001	North (+) or South (-) of the Equator, in thousandths of a degree
13 - 14	Longitude of Radar	INT*4	Degrees	-180 to +180	0.001	East (+) or West (-) of the Prime Meridian, in thousandths of a degree
15	Height of Radar	INT*2	Feet	-100 to +11000	1	Feet above mean sea level
16	Product Code	INT*2	N/A	16 to 299 ***[1999]***, -16 to -299	N/A	Internal NEXRAD product code of weather product being transmitted (Refer to Table III of the RPG to Class 1 User ICD). ***Note: The original upper limit for product codes was 299. This has been increased to 1999.***
17	Operational Mode	INT*2	N/A	0 to 2	N/A	0 = Maintenance 1 = Clean Air 2 = Precipitation/Severe Weather
18	Volume Coverage Pattern	INT*2	N/A	1 to 767	1	RDA volume coverage pattern for the scan strategy being used
19	Sequence Number	INT*2	N/A	-13, 0 to 32767	1	Sequence number of the request that generated the product (Refer to Figure 3-4). For products generated by an Alert Condition, sequence number = -13
20	Volume Scan Number	INT*2	N/A	1 to 80	1	Counter, recycles to one (1) every 80 volume scans
21	Volume Scan Date	INT*2	Julian Date	1 to 32767	1	Modified Julian Date; integer number of days since 1 Jan 1970
22 - 23	Volume Scan Start Time	INT*4	Seconds GMT	0 to 86399	1	Number of seconds after midnight, Greenwich Mean Time (GMT)
24	Generation Date of Product	INT*2	Julian Date	1 to 32767	1	Modified Julian Date as above
25 - 26	Generation Time of Product	INT*4	Seconds GMT	0 to 86399	1	Number of seconds after midnight, Greenwich Mean Time (GMT)
27 - 28	----------PRODUCT DEPENDENT PARAMETERS 1 AND 2 (SEE TABLE V)----------					

29	Elevation Number	INT*2	N/A	1 to 20*	1	Elevation number within volume scan. ***Can be up to 25 for TDWR radars.***
30	----------PRODUCT DEPENDENT PARAMETER 3 (SEE TABLE V)----------					
31 - 46	----------PRODUCT DEPENDENT (SEE NOTE 1)----------					
47 - 53	--------PRODUCT DEPENDENT PARAMETERS 4 THROUGH 10 (SEE TABLE V, NOTE 3)----------					
54	Version	INT*1	N/A	0 to 255	1	If the message is product data, the upper byte is the version number of the product. The original format of a product will be version 0. (Note 2)
54	Spot Blank	INT*1	N/A	0 to 1	1	If the message is product data, the lower byte is: 1 = Spot Blank ON 0 = Spot Blanking if OFF
55 - 56	Offset to Symbology	INT*4	Halfwords	0 to 400000	1	Number of halfwords from the top of message (message code field in header) to the -1 divider of each block listed. If the offset is zero (0), the block is not part of the product in question
57 - 58	Offset to Graphic	INT*4	Halfwords	0 to 400000	1	Same as above to Graphic Block (NOTE: For Product 62, this will point to the Cell Trend data)
59 - 60	Offset to Tabular	INT*4	Halfwords	0 to 400000	1	Same as above to Tabular Block

Note 1. For the description of the use of the 16 threshold level fields, see Appendix C.

Note 2. For a listing of the current product version numbers see Note 2 after Figure 3-6, Graphic Product Message (Sheet 6) in the RPG Class 1 User ICD.

Note 3**.** For products which are compressed, halfword 51 (P8) denotes the compression method:

halfword 51 contains 0 if no compression is applied
halfword 51 contains 1 if the data are compressed using bzip2 (refer to Appendix D for details)

And halfwords 52 (P9) and 53 (P10) denote the size of the uncompressed product, in bytes, excluding the sizes of the Message Header block and Product Description blocks:

halfword 52 contains size of uncompressed product (MSW), in bytes
halfword 53 contains size of uncompressed product (LSW), in bytes

If the product size less the product header and product description block is less than 1000 bytes, halfword 51 contains 0.

Product Symbology Block

NOTE: WSR-88D products contain 4-byte, 2-byte, and 1-byte data fields. Since the ORPG is supported on little Endian architectures, the data must be written into the product in a very specific manner. Properly accounting for the byte-swapping infrastructure and the alignment of data fields is described in CODE Guide Volume 3, Document 3, Section III - *Writing Product Data Fields*.

The Product Symbology Block always contains the block ID of number 1 and is shown below. If it is available in a product, it will always follow the *Product Description Block*.

Traditional Products

In general, this block contains display data packets which make up the geographic display of the product. These packets may contain vectors, text and special character symbols, map data, radial data, raster data, precipitation data, vector arrow data, wind barb data, and special graphic symbols.

The Symbology Block may, depending upon the product, have multiple "layers" of packets. This occurs in products that have both image type data, mixed with non-image type data. An example of this are the cross section products. The first layer is reflectivity or velocity data in raster packets; the second layer is the vector and text packets that create the grid lines and labels. The layers are started with the (-1) divider.

Guidance for the use of data packets in within the *Symbology Block* and use of multiple layers within the *Symbology Block* is provided in Section II - *Traditional Product Data Packets* of this document.

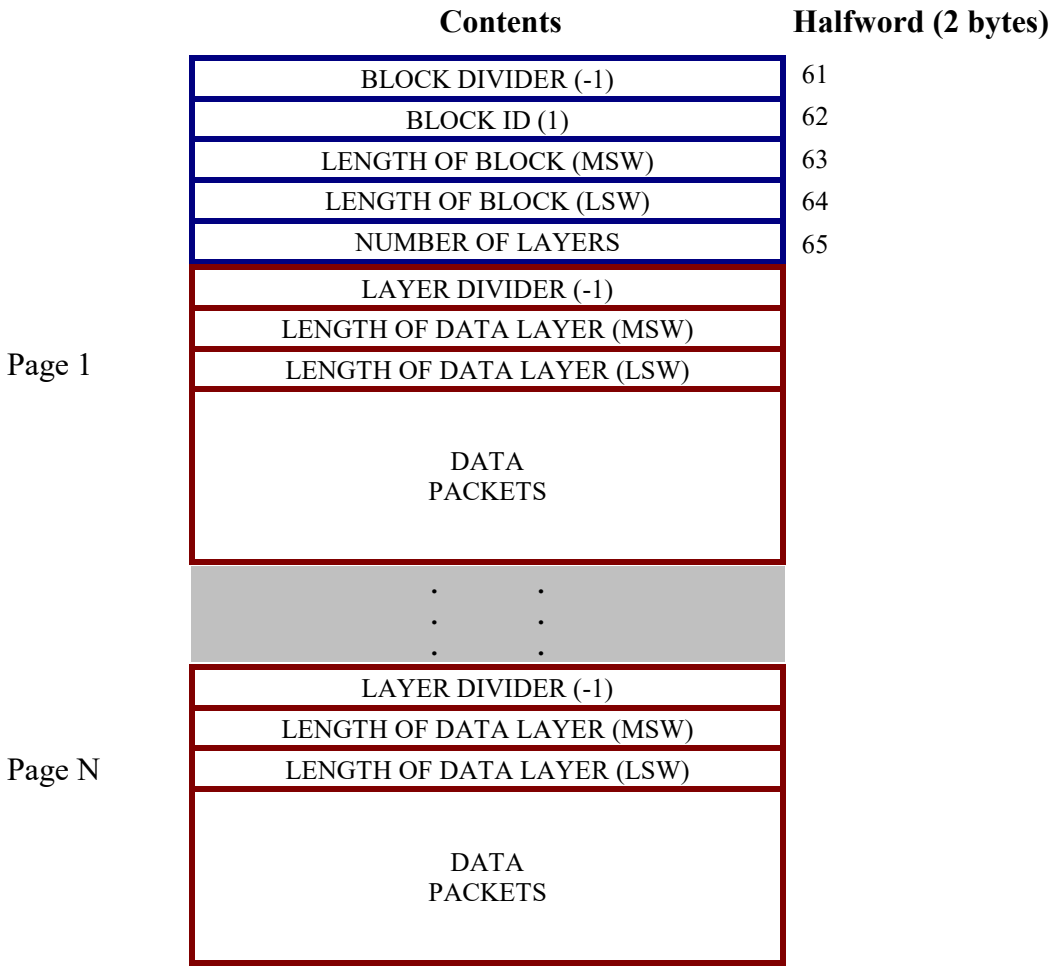
Generic Products

In general this block contains one data packet 28 in layer 1. Though not explicitly prohibited at this time, the product should contain no additional symbology block layers or a GAB or a TAB.

Guidance for the use of data packet 28 within the *Symbology Block* is provided in Section III - *Generic Product Components* of this document.

The following representation of the Product Symbology Block is based upon Figure 3-6 (sheet 3) of the ICD for the RPG to Class 1 User.

The Symbology Block Format



Product Symbology Block Field Descriptions

The field descriptions are documented in Figure 3-6 (sheet 8) of the ICD for the RPG to Class 1 User. Note that the type notations are based on the original FORTRAN documentation. **`INT*4`** represents a 4 byte (32 bit) integer and **`INT*2`** represents a 2 byte (16 bit) integer.

FIELDNAME	TYPE	UNITS	RANGE	PRECISION/ ACCURACY	REMARKS
Block Divider	INT*2	N/A	-1	N/A	Integer value of -1 used to delineate the Product Description from the Product Symbology Block
Block ID	INT*2	N/A	1	N/A	Constant value of 1 which identifies this block
Length of Block	INT*4	Bytes	1 to 400000	1	Length of block in bytes (includes preceding divider and block id)

Number of Layers	INT*2	N/A	1 to 18	1	Number of data layers contained in this block (see Note 2)

Layer Divider	INT*2	N/A	-1	N/A	Integer value of -1 used to delineate one data layer from another
Length of Data Layer	INT*4	N/A	1 to 400000	1	Length of data layer (in bytes) not including layer divider and length field
Display Data Packets	N/A	N/A	N/A	N/A	See Figures 3-7 through 3-15c

Note 2. With traditional products, the various layers are different types of data formats. An example would be the cross section products. The first layer is reflectivity or velocity data in raster packets; the second layer is the vector and text packets that create the grid lines and labels. The length of the layer does not include the divider or the length word. Generic products have 1 layer containing the generic product data packet.

Graphic Alphanumeric Block

The Graphical Alphanumeric Block (GAB) always contains the block ID of number 2 and is shown below. When included in a product, it will always follow the *Product Symbology Block* and precede the *Tabular Alphanumeric Block* (if present). The purpose of this block is to provide data in a tabular format to supplement the graphic product contained in the *Product Symbology Block*. This data is displayed at the top of the graphic product on the PUP display. NEXRAD products having an associated GAB are listed at paragraph 3.2.1.3, Graphic Alphanumeric Block, in the ICD for the RPG to Class 1 User.

The data portion of the GAB is a series of text and vector packets which format the data into 5 lines of text separated by grid lines.

The Graphical Alphanumeric Block Format

	Contents
	BLOCK DIVIDER (-1)
	BLOCK ID (2)
	LENGTH OF BLOCK (MSW)
	LENGTH OF BLOCK (LSW)
	NUMBER OF PAGES
Page 1	PAGE NUMBER
	LENGTH OF PAGE
	TEXT PACKET 1
	. .
	TEXT PACKET N
	. .
Page N	PAGE NUMBER
	LENGTH OF PAGE
	TEXT PACKET 1
	. .
	TEXT PACKET N

Graphic Alphanumeric Block Field Descriptions

The field descriptions are documented in Figure 3-6 (sheet 9) of the ICD for the RPG to Class 1 User. Note that the type notations are based on the original FORTRAN documentation. **`INT*4`** represents a 4 byte (32 bit) integer and **`INT*2`** represents a 2 byte (16 bit) integer.

NOTE: WSR-88D products contain 4-byte, 2-byte, and 1-byte data fields. Since the ORPG is supported on little Endian architectures, the data must be written into the product in a very specific manner. Properly accounting for the byte-swapping infrastructure and the alignment of data fields is described in CODE Guide Volume 3, Document 3, Section III - *Writing Product Data Fields*.

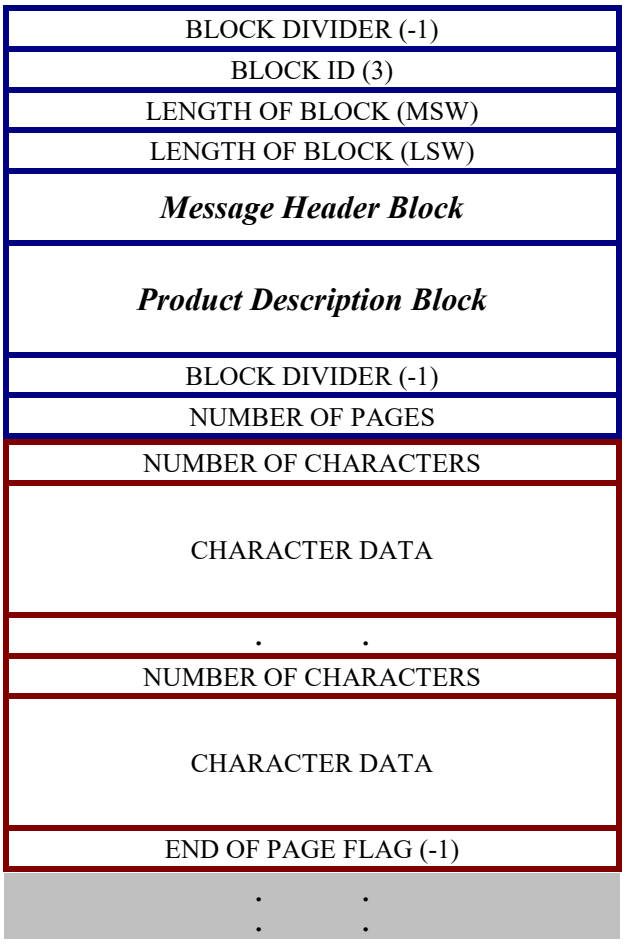
Tabular Alphanumeric Block

The Tabular Alphanumeric Block (TAB) always contains the block ID of number 3 and is shown below. When included in a product, it will always follow the *Product Symbology Block* and *Graphic Alphanumeric Block* (if present). The purpose of this block is to provide a "paired" alphanumeric product accompanying the graphic product contained in the *Product Symbology Block*. NEXRAD products having a paired TAB are listed at paragraph 3.2.1.4, Tabular Alphanumeric Block, in the ICD for the RPG to Class 1 User.

The TAB includes a second *Message Header Block* (MHB) and *Product Description Block* (PDB) as shown in the figure below. The Message Code field in the MHB and the Product Code field in the PDB are changed to a unique value for this "paired" product. The data portion of the TAB is ASCII text formatted into pages of 17 lines and 80 characters.

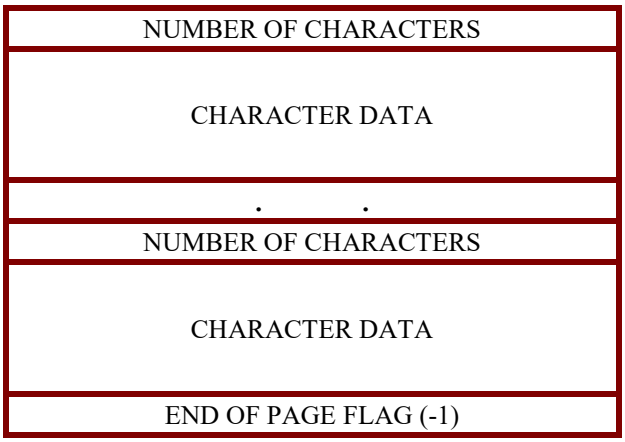
The Tabular Alphanumeric Block Format

Contents



Page 1

Page N



Tabular Alphanumeric Block Field Descriptions

The field descriptions are documented in Figure 3-6 (sheet 10) of the ICD for the RPG to Class 1 User. Note that the type notations are based on the original FORTRAN documentation. **`INT*4`** represents a 4 byte (32 bit) integer and **`INT*2`** represents a 2 byte (16 bit) integer.

NOTE: WSR-88D products contain 4-byte, 2-byte, and 1-byte data fields. Since the ORPG is supported on little Endian architectures, the data must be written into the product in a very specific manner. Properly accounting for the byte-swapping infrastructure and the alignment of data fields is described in CODE Guide Volume 3, Document 3, Section III - *Writing Product Data Fields*.

Data Packet Descriptions

There are approximately 30 types of data packets used in WSR-88D products. These the structure of these packets is documented in Figures 3-7 through 3-15c of the ICD for the RPG to Class 1 User.

Section II of this document introduces the traditional data packets and defines their use in a WSR-88D product. Section III of this document introduces the generic components that can be represented with data packet 28.

Vol 2. Document 3 -
WSR-88D Final Product Format

Section II Traditional Product Data Packets

Part A. Introduction

Currently, approximately 30 data packets are used in various ways to construct weather products. There are several packets which contain text information to be used in a very specific manner in different parts of the product. There are also several data packets for drawing vectors or lines on the graphic display, some for mono color lines and some variable color. Several packets are used to represent special symbols and some of these are nested within other packets. There is a packet used to represent radial data (polar coordinate) intended for display and another to represent raster data (rectangular coordinate) intended for display. There is a data packet used to represent 8-bit radial data that was not originally intended for display.

One common thread in many of these traditional data packets is that the packet data includes display information to some extent. This display content ranges from a specific encoding of display labels, using coordinates that assume a specific display screen resolution, and in some cases contain pixel size / coordinate information used to draw the specific symbols and lines.

Two documents are needed to understand the use of these data packets.

1. The format and field definitions for the data packets are documented in Figures 3-7 through 3-15c of the *Interface Control Document (ICD) for the RPG to Class 1 User, Document number 2620001*. A recent version is provided in the **2620001u_rpg_class1.pdf** file located in the **--/pdf_doc/** directory on the CODE CD.

2. The meaning of the data contained in the data packets and for some data packet the nature of the displayed symbol is contained in the *WSR-88D Product Specification, Document number 2620003*: **2620003t_prod_spec.pdf** (located in the **--/pdf_doc/** directory on the CODE CD).

General Guidance for Structure of Product Symbology Block

The following guidelines are not formal 'rules' contained in the ICDs listed in the previous paragraph. 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.

Many of the data packet listed here are used for specific purposes (some for only a single product). Data packet marked with ** are considered more likely to be useful when creating new products.

Part B. Two-Dimensional Data Array Packets

These data packets provide the colored images that can be used to represent basic radar values like reflectivity, radial velocity, and spectrum width or derived values like rain fall accumulation or cloud tops. Most of the product data arrays are geographic, that is representing two-dimensional ground position with respect to the radar location. A few are non-geographic, one example are the vertical cross section products.

16-Level Radial / Raster Packets (for Display)

A primary design factor for these data packets was size based upon the limiting factors existing in the late 1980's: network and modem bandwidth and computational resources (for example available storage). The data can have a maximum of 16 data levels and products with 16, 8, and 4 data levels are produced). A form of run-length encoding (RLE) is used to reduce the size of the product further. Products using these data packets were intended for display and one feature of the data packets is that the text threshold labels that are displayed next to each color in the legend are encoded in the product itself. Geographic products using these data packets are produced in several horizontal resolutions (250m, 500m, 1000m, and 2000m).

1. Packet AF1F - Radial Data **

   Radial data packets are used in geographic products with the center (radial origin) being the location of the radar antenna. This data packet is often used in elevation type products. In this case the data are derived from a specific elevation cut of the radar scan. The products are not projected to the surface so the range information is the slant range of the conic section of the individual scan.

2. Packet BA07/BA0F - Raster Data **

   Raster data packets are used in both geographic products (the center representing the radar location) and non-geographic products.

When these packets are constructed, to maintain proper structure, there must be an even number of bytes representing the run-length encoded data. The API helper functions will pad the data as required ensuring alignment. The packet header field representing the length of the data must include any padded data.

8-bit Radial Data Arrays

The 8-bit radial data array packet was created to provide a means of encoding higher resolution information that could be used for further computation by external systems. These products are often called 'digital' products because they were originally not intended for display. Up to 256 data levels can be represented. Today products containing this data packet are routinely displayed on AWIPS and other external systems. Geographic products using these data packets are produced in several horizontal resolutions (250m, 500m, 1000m, and 2000m).

1. Packet 16 - Digital Radial Data Array **

Radial data array packets are used in geographic products with the center (radial origin) being the location of the radar antenna. This data packet is often used in elevation type products. In this case the data are derived from a specific elevation cut of the radar scan. The products are not projected to the surface so the range information is the slant range of the conic section of the individual scan.

The real product data should be encoded into the 8-bit integer in a linear fashion described in Appendix B. Unfortunately, existing products are not consistent in the manner real data values are encoded into the array of 8-bit integers. The precise means of encoding this data in more recent products does not always follow the original method.

If the range and precision of the product's data cannot be represented via a linear encoding described in Appendix B, the Generic Radial component (section III of this document) is an alternative. The radial component is being used in a new DPR product.

BZIP2 compression can be used to reduce the product size.

When this packet is constructed, to maintain proper structure, there must be an even number of bytes representing the radial data arrays. The API helper functions will pad the data as required ensuring alignment. The packet header field representing the length of the data must include any padded data

Special Purpose Data Arrays

These data packets are very specific and are only used in the DPA (Digital Precipitation Array) product. The geographic data contained is based upon a particular type of polar limited fine mesh (LFM) projection which is unique to this product.

1. Packet 17 - Digital Precipitation Data Array
2. Packet 18 - Precipitation Rate Data Array

When these packets are constructed, to maintain proper structure, there must be an even number of bytes representing the run-length encoded data. The packet header field representing the length of the data must include any padded data.

Part C. Special Symbols

These data packets provide the various symbols that can be used to represent meteorological features like mesocyclones, tornadoes, and hail. Most of the symbols are used in geographic products, that is their locations represent a two-dimensional ground position with respect to the radar location. Two symbols are non-geographic and are used in specific products.

Geographic Overlay Symbols

For the most part, these data packets have a very specific purpose and are only used in specific type of product. The exception is data packet 20 - Point Feature Data. This is not a Legacy data packet and is designed to be extensible. The products containing these packets are intended to be displayed on top of basic geographic data array products. Each symbol indicates the location of a particular meteorological feature. The data packet location coordinates are in units of 1/4 KM.

1. Packet 3 / 11 - Mesocyclone / Correlated Shear

   Examples of use: M (Mesocyclone) & MD (Mesocyclone Detection)

2. Packet 12 / 26 - TVS / ETVS

   Used in the TVS (Tornado Vortex Signature) product.

3. Packet 13 / 14 -

   [NO LONGER USED]

4. Packet 15 Storm ID

   Examples of use: M (Mesocyclone), MRU (Mesocyclone Rapid Update), HI (Hail Index)

5. Packet 19 - HDA Hail Data

   Used in the HI (Hail Index) product.

6. Packet 23 / 24 - SCIT (Past / Forecast) Position

   Used in the STI (Storm Tracking Information) product and in the MD (Mesocyclone Detection) product. Within these packets, data packet 2 is used to representing the symbols and data packet 6 is used for the connecting lines.

   a. Packet 2 - Symbol (No Value)
      Used in the STI (Storm Tracking Information) product and in the MD (Mesocyclone Detection) product. Nested inside of packets 23 & 24.

b. Packet 6 - Linked Vector (No Value)
Used in the STI (Storm Tracking Information) product and in the MD (Mesocyclone Detection) product. Nested inside of packets 23 & 24.

7. Packet 25 - STI Circle

Used in the STI (Storm Tracking Information) product.

8. Packet 20 - Point Feature Data **

The purpose of this data packet was to create an extensible set of symbols in one packet. Currently there are 11 symbols defined.

Symbol	Current Use	Description
1	MRU (retired)	Segmented Circle, thick line, variable radius (attribute field)
2	MRU (retired)	Segmented Circle, thin line, variable radius (attribute field)
3	MRU (retired)	Solid Circle. thick line, variable radius (attribute field)
4	MRU (retired)	Solid Circle. thin line, variable radius (attribute field)
5	TRU	Triangle on Side, solid color,
6	TRU	Triangle on Side, line only,
7	TRU	Inverted Triangle, solid color,
8	TRU	Inverted Triangle, line only,
9	MDA	Spiked Solid Circle, thick line, variable radius (attribute field)
10	MDA	Solid Circle, thick line, variable radius (attribute field)
11	MDA	Solid Circle, thin line, variable radius (attribute field)

Defining new symbols for a product intended for integration into the operational system is accomplished through design reviews with the Radar Operations Center. For display via CODEview Graphics (CVG), contact the development lead for CVG.

Non-Geographic Symbols

These data packets are each currently only used in a specific product. The data packet location coordinates are in pixel screen coordinates. Though the Class 1 User ICD permits 1/4 KM location coordinates, these packets have never been used in geographic products.

1. Packet 4 - Wind Barb

Used in the VWP (VAD Wind Profile) product.

2. Packet 5 - Vector Arrow

Was used in the Combined Moment product (product discontinued).

Part D. Vector Packets

Linked vector packets are intended to be used when a line is made up of multiple points. Unlinked vector packets are intended to be used for lines having two points, a beginning and an end. Vectors having no value are intended to be drawn with the standard foreground color (typically white). Vectors having uniform value have a value field that could be used to determine color for display. These packets have not always been used in this manner. There are examples where the display device displays no value vectors in more than one color. The data packet location coordinates are in units of 1/4 KM when used in geographic products or in pixel screen coordinates when used in non-geographic products.

Used in GAB

The GAB is a portion of a product that is constructed in a very specific manner. The grid lines in a GAB must be represented by data packet 10 with location in screen coordinates.

1. Packet 10 - Unlinked Vector (Uniform Value)

   Examples of use: M (Mesocyclone), HI (Hail Index), TVS (Tornado Vortex Signature)

General Purpose

These data packets could be used in both geographic and non-geographic products. However, these packets are currently used in only certain situations. For example, data packet 6 is currently only used nested inside of packets 23 & 24 representing tracking information (geographic). Packet 7 is currently only used in cross section products (non-geographic). Packet 9 is currently only used in the VAD product (non-geographic).

1. Packet 6 - Linked Vector (No Value)

   Used in the STI (Storm Tracking Information) product and in the MD (Mesocyclone Detection) product. Nested inside of packets 23 & 24.

2. Packet 7 - Unlinked Vector (No Value)

   Used in Cross Section products.

3. Packet 9 - Linked Vector (Uniform Value)

   Used in the VAD (Velocity Azimuth Display) product.

4. Packet 10 - Unlinked Vector (Uniform Value)

   Used in the VWP (VAD Wind Profile) product.

Special Purpose Vector Packets

The purpose of these data packets is to display data levels via geographic contour lines having different colors. Legacy products using the following data packets were removed from the ORPG in Build 4. However the future Melting Layer product may use these data packets. It is recommended that the Generic Area Component be used instead of these data packets.

These linked and unlinked vector data packets and their associated color level packet should only be used for the purpose of providing geographic contour lines representing 2 dimensional data level. The data packet location coordinates are in units of 1/4 KM. The general purpose vector packets or the Generic Area component should be used for all other vector or line drawing purposes.

1. Packet 0802 - Contour Vector Color

   This packet is located just before a series of contour vector packets to determine an index into the color table.

   Used in the new melting layer (ML) product.

2. Packet 0E03 - Linked Contour Vector

   Used in the new melting layer (ML) product.

3. Packet 3501 - Unlinked Contour Vector

Part E. Text Packets

Text packets having no value are intended to be drawn with the standard foreground color (typically white). Text packets having uniform value have a value field that could be used to determine color for display. These packets have not always been used in this manner. There are examples where the display device displays no value text in more than one color. The data packet location coordinates are in units of 1/4 KM when used in geographic products or in pixel screen coordinates when used in non-geographic products.

When these packets are constructed, to maintain proper structure, there must be an even number of bytes representing the character data. The data should be padded with a 'blank' character if necessary. The packet header field representing the length of the data must include any padded data.

Used in GAB

The GAB is a portion of a product that is constructed in a very specific manner. The text in a GAB must be represented by data packet 8 with location in screen coordinates.

1. Packet 8 - Text (Uniform Value)

   Examples of use: M (Mesocyclone), HI (Hail Index), TVS (Tornado Vortex Signature)

General Purpose

The following text packets are general purpose and have been used in both geographic and non-geographic products. Each text data packet should represent only one line of text. If positioning text in pixel screen coordinates, it is recommended to separate lines of text by at least 12 pixels (difference in J values).

1. Packet 1 - Text (No Value) **

   Examples of use: Cross Section Products.

   NOTE: Several products (**USP**, **DHR**, and **DPA**) use packet 1 in a non-standard fashion and should not be considered examples of use.
   - The coordinates are in pixel screen coordinates rather than 1/4 KM for geographic products.
   - An un-documented aspect of the text data in these products is that the lines can be extremely long and can exceed limits of text display. It is unclear how these are intended to be formatted. CVG insert a line feed every 80 characters when displaying this packet.

2. Packet 8 - Text (Uniform Value) **

Used in the MD (Mesocyclone Detection) (in lieu of packet 15), in VAD (Velocity Azimuth Display) and in VWP (VAD Wind Profile).

Part F. Single Purpose Packets

1. Packet 21 / 22 - Cell Trend Data / Cell Trend Volume Time

   Data packets only used in the SS (Storm Structure) product.

2. Packet 27 - SuperOB

   A unique packet used in the SO (Superob) velocity product (removed in Build 18.0).

Vol 2. Document 3 -
WSR-88D Final Product Format

Section III Generic Product Components

The concept of a generic product is relatively new and ORPG infrastructure support is still evolving.

Note:

- For the area component, noted that even if the following are stated in the product specification ICD, the display attributes for lines, symbols and labels are completely determined by the display device. In the future some effort should be made in defining a set of line and symbol attributes (line thickness, solid / dashed, etc. and label attributes. Standard area component attribute names should also be defined that stipulate the display attributes. For CVG display, the development lead for CVG must be notified. Currently CVG only provides a capability to manually select display attributes from a defined short list. See Part D.

Part A. Introduction

The generic data packet (packet 28) is a collection of generic components each with a different purpose. There are several types of *grid* components that can be used to represent two dimensional binary data in rectangular or polar coordinates. The *radial* component can be used to contain radial data in various formats. The *area* component can be used to represent single geographic points, a geographic line, or an enclosed geographic area. The *table* component is used to represent text information in an organized tabular format. The *text* component is used to represent simple text.

The contents of the generic data packet (packet 28) can be relatively self-descriptive if correctly used. The packet contains no display information in contrast to many of the traditional data packets. However, the flexibility of the generic components can also increase the effort required on the display system to decode and display the product. One example:

The MRU product uses the traditional data packet 20 - *Point Feature Data* to represent different Mesocyclone features (current mesocyclone, extrapolated mesocyclone, current 3D correlated shear, etc.). Packet 20 uses a single field, "point feature type", to distinguish these features which are displayed in a distinctive manner with different symbols. The DMD product (replacing the MRU product) uses the generic *area component* to represent similar features. The nature of the component parameters in the design of the DMD requires a comparison of 4 parameters to make a distinction of which symbol to use with each point (feature).

Another difference between these two products is the method used to convey additional information about each feature. The MRU uses a table in a GAB and TAB. Using a GAB and TAB places limits on the formatting of this additional data but the user system requires no modification to display these portions of the product. The DMD uses component parameters to provide quite a bit of information about the features in the product, which can be displayed in any manner the designers of the display system choose. However, the system will require modification for each new product.

A primary advantage of using generic components is the flexibility of the information that can be contained in a product without defining new structures or packet types. Another advantage is separation of the look and feel of the display from the product content. One disadvantage is that in some cases the display system may require more modification for a new generic product.

1. The format and field definitions for the generic data packet are documented in Figure 3-15c and Appendix E of the *Interface Control Document (ICD) for the RPG to Class 1 User, Document number 2620001*. A recent version is provided in the **2620001u_rpg_class1.pdf** file located in the **--/pdf_doc/** directory on the CODE CD.

   Unlike traditional data packets, the ICD only defines the structure of the header portion of data packet 28 when describing the message structure (in 16-bit integers). The structure of the serialized data portion of the packet is not described in the message structure by the ICD. The serialized data is created with C structures combined in a specified number and order. The C structures are defined in the file **orpg_product.h**. A standard serializing algorithm is used by the ORPG to create that portion of the product message. The user of the product must use a standard deserializing algorithm (available from the ROC) to read the data. The CODE display utilities CVT and CVG are examples of using the deserializing software. Appendix E. of the ICD contains a definition of generic product structure and the individual components that can be used to construct the serialized data portion of the data packet. The actual structure of the components can be better understood by review of the C structures defined in **orpg_product.h**.

   If there is a difference in the type and name of data fields, the C structure should be considered more up-to-date than the contents of Appendix E in the ICD.

2. The meaning of the data contained in the data packets is contained in the *WSR-88D Product Specification, Document number 2620003*: **2620003t_prod_spec.pdf** (located in the **--/pdf_doc/** directory on the CODE CD).

General Guidance for Structure of Product Symbology Block

The following guidelines are not formal 'rules' contained in the ICDs listed in the previous paragraph. 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.

Part B. The Generic Product

The generic product is contained within data packet 28 which is the only packet in layer 1 of the symbology block.

	Contents	Halfword (2 bytes)
	BLOCK DIVIDER (-1)	61
	BLOCK ID (1)	62
	LENGTH OF BLOCK (MSW)	63
	LENGTH OF BLOCK (LSW)	64
	NUMBER OF LAYERS	65
	LAYER DIVIDER (-1)	66
	LENGTH OF DATA LAYER (MSW)	67
	LENGTH OF DATA LAYER (LSW)	68
Packet 28 Header	Packet Code = 28	
	NOT USED (for alignment)	
	Number of Bytes (MSW)	
	Number of Bytes (LSW)	
Packet 28 Data	**Serialized Generic Product Data** `RPGP_product_t`	

The structure **`packet_28_t`** defined in **`packet_28.h`** can be used to access the header but the num_bytes field must be set with the **`RPGC_set_product_int`** function. After writing the **`num_bytes`** field in Big Endian format, it must be read with **`RPGC_get_product_int`**.

The serialized data must be deserialized using standard API functions and the resulting address cast to **`RPGP_product_t *`**. The top level structure of the generic product is represented by the C structure **`RPGP_product_t`**. This structure contains approximately 20 header fields (some of which are redundant with fields in the product description block), a pointer to product parameters, and a pointer to product components. The deserialized data does not need byte swapping.

```
typedef struct {              /* product struct */

    char *name;               /* product name */
    char *description;        /* product description (may contain version
                                 info) */
    int product_id;           /* product id (code) */
    int type;                 /* product type (RPGP_VOLUME... except
                                 RPGP_EXTERNAL) */
    unsigned int gen_time;    /* product generation time */

    char *radar_name;         /* radar name. NULL or empty string indicates
                                 the radar info is not applicable. The radar
                                 info is applicable for products based on
                                 single radar data. The following three
                                 fields are used only if radar_name is
                                 specified. If not used, 0 is assigned. */
```

```
    float radar_lat;        /* radar latitude location (in degrees) */
    float radar_lon;        /* radar longitude location (in degrees) */
    float radar_height;     /* radar height location (in meters) */
    unsigned int volume_time; /* volume scan start time. This and the
                                 following 6 fields are used only for single
                                 radar based products. If not used, 0 is
                                 assigned. */
    unsigned int elevation_time; /* elevation scan start time. Used only
                                    for elevation based products. */
    float elevation_angle;  /* elevation angle in degrees. Used
                               only for elevation based products. */
    int volume_number;      /* volume scan number */
    short operation_mode;   /* operation mode (RPGP_OP_MAINTENANCE...) */
    short vcp;              /* VCP number */
    short elevation_number; /* elevation number within volume scan. Used
                               only for elevation based products. */

    short compress_type;   /* compression type (currently not used and
                              must set to 0) */
    int size_decompressed; /* size after decompressing (currently not used
                              and must set to 0) */

    int numof_prod_params;      /* number of specific product parameters */
    RPGP_parameter_t *prod_params; /* specific product
                                                    parameter list */

    int numof_components;       /* number of components or events */
    void **components;   /* component or event list. See Note 0. */

} RPGP_product_t;
```

Appendix E in the RPG Class 1 User Interface Control Document (ICD) contains a description of the contents of the fields in the generic product structure including the contents of the generic header fields and a detailed description of generic parameters.

Algorithm API support

The algorithm API contains helper functions to fill out the product header fields, fill the contents of the parameter structures, and to serialize / deserialize the product. No support is provided for construction of the components themselves. `RPGP_build_RPGP_product_t()` should be used to insure that correct information is written to the header fields in `RPGP_product_t`. One item to note: the ICD states that the time fields in the generic product structure are all Unix time (seconds since 1/1/1970).

The algorithm API contains some debug print functions that can be used to provide a text output of the contents of the product. The CODE utility CVT provides a more capable text output for analysis.

Generic product and component parameters

Each product parameter and component parameter is represented by the C structure `RPGP_parameter_t`. Generic parameters are used to provide additional data that further describes the product or components. For example, each of the product request parameters that generated a product could be included in a generic product component.

Part C. Two-Dimensional Data Array Components

The following components correspond to the traditional radial, raster, and 8-bit radial data arrays. **These components have not yet been used in a product.** One significant difference is that numerical data can be represented by the correct type rather than having a special encoding technique used in the traditional data packet 16. Arrays of **`8-bit/16-bit/32-bit integers (signed or unsigned)`**, **`float`** (4-byte IEEE), and **`double`** (8-byte IEEE) can be used.

Appendix E in the RPG Class 1 User Interface Control Document (ICD) contains a description of the generic components.

Grid Components

> NOTE: The grid component is not completely defined at this time. Before using this component definitive component parameters must be predefined to represent the step size between the grid rows / columns, the location of the origin, and the coordinate orientation for certain grids.

The grid component is represented by the C structure **`RPGP_grid_t`**. The grid component includes the structure **`RPGP_data_t`** which is used to contain the actual data arrays. Currently there are 4 types of grid components planned:

- A non-geographical array.
- A flat equally spaced grid.
- An equally spaced latitude-longitude grid.
- A rotated pole grid.

> NOTE: The grid component is not completely defined at this time. Before using this component definitive component parameters must be predefined to represent the step size between the grid rows / columns, the location of the origin, and the coordinate orientation for certain grids.

Radial Component

The radial component is represented by the C structure **`RPGP_radial_t`**. The radial component includes the structure **`RPGP_radial_data_t`** which is used to contain one radial of polar coordinate data. The structure **`RPGP_data_t`** contains the actual data arrays which can be one of the following data types: 8-bit / 16-bit / 32-bit signed integers, 8-bit / 16-bit / 32-bit unsigned signed integers, and the real types float and double.

The legacy data packets contain well-used packet type representing radial data in final products. The primary reason to use the radial component in a final product would be to take advantage of representation of real numerical data without having to use a unique or non-standard encoding scheme for representation via an 8-bit integer in data packet 16.

Choice of data array types:

Unsigned Integer Types (**`unsigned char, unsigned short, unsigned int`**). The primary use of unsigned integers is to encode real product data into fewer bits than required by the IEEE floating point. As with the traditional packet 16, encoding should be in a linear fashion. Choose the smallest type that can represent the range and precision of data in a linear fashion. Since the **`float`** consumes the same space as the 32-bit **`int`**, the float should be considered if it reduces processing by the user.

Signed Integer Types (**`char, short, int`**). These types are convenient if the product data are integers. If the data are always positive, one of the unsigned types could be used.

Real Data Types (**`float, double`**). These types may be convenient if they reduce processing. The tradeoff is in increased product size. With the nature of the radar data, the type **`double`** is probably never needed.

The most important use of the radial component may be as a method of representing polar data arrays in intermediate products. This will be demonstrated in a future CODE sample algorithm.

Part D. Two-Dimensional Overlay Components

Appendix E in the RPG Class 1 User Interface Control Document (ICD) contains a description of the generic components.

Area Components

The area component is represented by the C structure **`RPGP_area_t`**. The area component uses one of three structures to represent the location of the point(s) based upon the component type: **`RPGP_location_t`** (latitude/longitude), **`RPGP_xy_location_t`** (rectangular coordinates in KM), and **`RPGP_azran_location_t`** (azimuth in degrees and range in KM). The area component can be used to represent

- A single geographical point
- A geographical polyline
- A closed geographical area

Even though they may be defined in the product specification ICD, currently line attributes (thickness, solid or dashed) and symbols and labels used for points are completely determined by the display device. In the future, it would be nice if a set of lines, symbols, and labels could be defined along with the definition of specific area component attributes to indicate what should be displayed.

For CODEview Graphics (CVG), several line, symbol, and labels options can be manually chosen at display time.

Example of use: DMD (Mesocyclone Detection Data Array Product)

Part E. Components Used for Text Content

Appendix E in the RPG Class 1 User Interface Control Document (ICD) contains a description of the generic components.

Text Component

The text component is represented by the C structure **`RPGP_text_t`**. A standard character string is used to represent the text. Example of use: ASP (Archive III Status Product)

Table Component

The table component is represented by the C structure **`RPGP_table_t`**. The table component uses the structure **`RPGP_string_t`** to represent the row and column labels and the contents of each table cell.

Vol 2. Document 3 -
WSR-88D Final Product Format

Section IV ORPG Application Dependent Parameters

Introduction

Application dependent parameters can be used to provide customizing parameters via the product request message that can be used to change the nature of the product for that specific request. They can also be used to provide additional fields of information in the formatted final product. Except for the requested elevation, only the final product can be customized by the 6 request parameters in the product request message. These parameters are not passed on to tasks upstream producing intermediate products.

Part A. explains the relationship between the product specific parameters contained in the request message and the product dependent parameters in the product. Part B. discusses some rules to maintain consistency in the use of parameters in the product request message, and Part C. provides some guidance for use of the parameters in the product description block portion of the final product message.

Part A. Relationship between the Request Message and the Final Product

New products (product code & buffer number greater than 130) no longer have a hard coded unique relationship between the product parameters in the request message and the product parameters in the product description block. Rather, the 6 parameters contained in the request message map directly to the first 6 product parameters in the product description block.

Product Parameter Number	Product Parameter Index	Halfword in Product Request Message	Halfword in Product Description Block
1	0	20	27
2	1	21	28
3	2	22	30
4	3	23	47
5	4	24	48
6	5	25	49
7	6	---	50
8	7	---	51
9	8	---	52
10	9	---	53

It should be pointed out that this relationship exists only when the use of any of the parameters in the request message for that product has been defined in the **`product_attr_table`** configuration file. Any of the first six parameters not defined for use in the request message can be used in the product description block for other purposes.

Part B. Product Request Message

Product Dependent Parameters (PDP) in the Product Request Message

Product dependent parameters for the request message are configured via the **params** attribute in the **product_attr_table** configuration file. For elevation products, these parameters determine which elevation is being requested. In addition, some products can be customized via passing additional parameters in the product request message.

In the following example, the first two parameters (like most parameters) have a straight forward interpretation. **params 0** (product dependent parameter 1) represents Azimuth from 0 to 359.9 degrees in tenths of a degree (0 degrees is the default). **params 1** (product dependent parameter 2) represents Range from 0 to 124.0 NM in tenths of a NM (0 NM is the default). The third parameter representing elevation (the most common parameter in the system) is a special case.

```
params
        0     0   3599      0      10   "Azimuth"    "Degrees"
        1     0   1240      0      10   "Range"      "Nmiles"
        2   -20   3599      0      10   "Elevation"  "Degrees"
      index  min   max   default  scale   name         units
```

The elevation is scaled in units*10 and can range from -2.0 units (**-20**) to plus 359.9 units (**3599**). The default value is 0 units. The actual interpretation is not that simple. Negative numbers actually represent slices rather than angles and the scale is not applied. This means -4 (unscaled) represents the first 4 elevations in a volume and -20 (unscaled) represents the first 20 elevations in a volume. Scaling is applied to positive numbers. Small positive numbers represent positive elevation angles (34 represents + 3.4 degrees) and a very large positive number represents negative elevations (3595 represents - 0.5 degrees). Note: It is not clear where the transition between representing positive angles and negative angles occurs. According to the ICD for the RPG to Class 1 User, the maximum negative angle is -1.0 degrees and the maximum positive angle is 45.0 degrees. Currently the radar does not scan at negative elevation angles.

Rules for Using PDP in the Product Request Message

These rules are consistent with existing products. Following each rule is an example of the parameters as defined in the **product_attr_table** configuration file.

Notes that apply to the following rules:

Note 1: Represented by a scaled integer.

Note 2: The value -1 is a flag having special meaning. See Table IIa, Product Dependent Halfword Definitions for Product Request Message, in the ICD for the RPG to Class 1 User.

1. All elevation based products will use the third parameter (param index 2) for the elevation.

Parameter		Use / Description	Units	Range	Accuracy / Precision
Number	Index				
3	2	Elevation Angle	Degrees	-1.0 to 45.0	0.1 Note 1

Example of definition in **`product_attr_table`** configuration file:

```
params
                    2 -20 3599 0 10 "Elevation" "Degrees"
```

2. Any product that can be constructed at different levels (altitudes) rather than different elevations (conic sections) should use the third parameter (param index 2) for the level.

Parameter		Use / Description	Units	Range	Accuracy / Precision
Number	Index				
3	2	Altitude	K Feet	0 to 70	1

Example of definition in **`product_attr_table`** configuration file:

```
params
                    2 0 70 2 1 "Altitude" "Kfeet"
```

3. Contour Intervals will be specified in the sixth parameter (param index 5).

Parameter		Use / Description	Units	Range	Accuracy / Precision
Number	Index				
6	5	Contour Interval	Feet	2000 to 30,000	1000

All contour products were retired. However, a future product will use contours.

4. Cross section products will be specified as follows

Parameter		Use / Description	Units	Range	Accuracy / Precision
Number	Index				
1	0	Azimuth of Point 1	Degrees	0 to 359.9	0.1 Note 1
2	1	Range of Point 1	Nautical miles	0 to 124.0	0.1 Note 1
3	2	Azimuth of Point 2	Degrees	0 to 359.9	0.1 Note 1
4	3	Range of Point 2	Nautical miles	0 to 124.0	0.1 Note 1
5	4	**`PARAM_UNUSED`**			

6	5	**PARAM_UNUSED**			

Example of definition in **product_attr_table** configuration file:

```
params
                    0 0 3599 0 10 "Azimuth Point 1" "Degrees"
                    1 0 1240  0 10 "Range Point 1"   "Nmiles"
                    2 0 3599 900 10 "Azimuth Point 2" "Degrees"
                    3 0 1240 1240 10 "Range Point 2"   "Nmiles"
```

5. Window Type Products will be specified as follows

Parameter		Use / Description	Units	Range	Accuracy / Precision
Number	Index				
1	0	Azimuth of Window Center	Degrees	0 to 359.9	0.1 Note 1
2	1	Range of Window Center	Nautical miles	0 to 124.0	0.1 Note 1
3	2	Elevation Angle (if elevation based product)	Degrees	-1.0 to 45.0	0.1 Note 1
4	3	**PARAM_UNUSED**			
5	4	**PARAM_UNUSED**			
6	5	**PARAM_UNUSED**			

Example of definition in **product_attr_table** configuration file:

```
params
                    0   0 3599 0 10 "Azimuth" "Deg"
                    1   0 1240 0 10 "Range" "nm"
                    2 -20  3599 0 10 "Elevation" "Degrees"
```

6. Products requiring a Speed and Direction Input (i.e., Storm Products)

Parameter		Use / Description	Units	Range	Accuracy / Precision
Number	Index				
1	0	Azimuth (If used)	Degrees	0 to 359.9	0.1 Note 1
2	1	Range (If Used)	Nautical miles	0 to 124.0	0.1 Note 1
3	2	Elevation Angle (if elevation based product)	Degrees	-1.0 to 45.0	0.1 Note 1
4	3	Artifact Speed (e.g., storm speed)	Knots	0 to 99.9	0.1 Note 1 & 2
5	4	Artifact Direction (e.g., storm direction)	Degrees	0 to 359.9	0.1 Note 1
6	5	**PARAM_UNUSED**			

Example of definition in **product_attr_table** configuration file:

```
params
                    0   0 3599 0 10 "Azimuth"          "Degrees"
                    1   0 1240 0 10 "Range"            "Nmiles"
                    2 -20  3599 0 10 "Elevation"        "Degrees"
                    3 -10  999 -10 10 "Storm Speed"     "Knots"
                    4 -10 3599 -10 10 "Storm Direction" "Degrees"
```

7. Products specified by a time and duration

Parameter		Use / Description	Units	Range	Accuracy / Precision
Number	Index				
1	0	End Hour	Hours	-1.0 to 23	1 Note 2
2	1	Time Span	Nautical miles	1 to 24	1
3	2	Elevation Angle (if elevation based product)	Degrees	-1.0 to 45.0	0.1 Note 1
4	3	**PARAM_UNUSED**			
5	4	**PARAM_UNUSED**			
6	5	**PARAM_UNUSED**			

Example of definition in **product_attr_table** configuration file:

```
params
                    0 -1 23 12 1 "End Hour" "Hours"
                    1 1  24 24 1 "Time Duration" "Hours"
```

Part C. ICD Final Product Message

1. Product Dependent Parameters (PDP) in the Product Description Block

The product parameters in the product description block portion of the final product serve two purposes. First, they document the parameters contained in the request message. In addition, they can be used to return additional information.

Rules for Using PDP in the Final Product Message

Because of the new relationship between the 6 product parameters in the request message and the 10 product parameters in the product description block, the legacy products (product code & buffer number less than 131) cannot be used to infer rules for new products.

For new products (pcode / buffer numbers 131-1999)

1. Parameters 1 - 6 must correspond to parameters defined for the product request message.
2. Any parameter 1 - 6 that is not defined for use in the product request message can be used for another purpose.
3. Parameters 7 - 10 currently have no restrictions except for product compression.
4. Product Compression
    - Parameter 8 (halfword 51) indicates the compression method (0 - none, 1 - bzip2 compression, 2 - zlib compression). Currently only bzip2 compression is used for final products.
    - Parameters 9 and 10 (halfwords 52 and 53) contain the uncompressed size of that portion of the product following the product description block (PDB). If the portion of the product after the PDB is less than 1000 bytes, then this parameter is 0.

Additional rules remain to be determined. Table V in the ICD for the RPG to Class 1 User documents the use of these parameters in legacy products.

2. Threshold Levels in the Product Description Block

The 16 data level threshold values in the product description block (halfwords 31 - 46) are used to provide the threshold values for the PUP displayable run length encoded products. The legacy documentation refers to this as defining the "color tables".

For 256 level products including the digital data array packet products, these halfwords have been used for other purposes.

Rules for Using Threshold Levels in the Final Product Message

Run length encoded products (maximum of 16 levels)

For run length encoded products (either radial or raster) that are PUP displayable, each halfword represents an encoding of the meaning assigned to this data level. This includes the following:

- The actual numerical value assigned to this data level.
- Above or below threshold.
- Range Folded
- No Data
- Blank

See Note 1 following figure 3-6 (sheet 8) in the ICD for the RPG to Class 1 User for an explanation of the encoding technique. This note is also in Appendix C. of this Volume.

Digital products (encoding real data into unsigned integer arrays)

Non run length encoded products (265 level products using packet code 16 or the unsigned 8-bit and 16-bit arrays in the generic radial component) do not explicitly describe threshold levels directly. Rather, these halfwords partially describe how the real data is encoded into a scaled byte. The following paragraphs provided an overview of how the threshold fields in the product description block describe the encoded data. A detailed description of use of threshold fields and the encoding and decoding of this data is provided in Appendix B of this Volume.

Threshold Level Fields - The Original Parameter Method

The existing use of the available data levels 0 - 255 falls into a pattern but a complete set of rules cannot be inferred from this pattern. Often the first two data levels (0 and 1) are flags representing "below threshold" and "missing" or "below threshold" and "range folded" respectively. Data levels 2 - 255 are typically used to encode numeric values.

The Legacy digital products (and many products added since) had a specific, though incomplete, method of providing information in the Product Description Block to aid in decoding integer values in data packet 16.

Halfword	Field	Use
HW 31	**`Threshold 1`**	contains the minimum value (encoded)
HW 32	**`Threshold 2`**	contains the increment (encoded)
HW 33	**`Threshold 3`**	contains the number of data levels

Threshold Level Fields - The `scale-offset` Parameter Method (Recommended)

The new **`scale-offset`** formula can be used to encode and decode any product having a linear increment between encoded data values. The following threshold fields are being used by future Dual Polarization products to describe the **`scale-offset`** coding.

Halfword	Field	Use
HW 31	**Threshold 1**	the SCALE in IEEE floating point format
HW 32	**Threshold 2**	
HW 33	**Threshold 3**	the OFFSET in IEEE floating point format
HW 34	**Threshold 4**	
HW 36	**Threshold 6**	the **highest data level having meaning, including flag values**
HW 37	**Threshold 7**	the number of leading flag values (can be 0)
HW 38	**Threshold 8**	the number of trailing flag values (can be 0)

Volume 2. ORPG Application Software Development Guide

Document 4. ORPG Internal Data for Algorithm Developers

This document contains helpful technical information concerning ORPG internals and also provides guidance in certain areas. The information presented here is independent of writing algorithm source code but does contain some references to the Application Programming Interface (API). CODE Guide Volume 3 - *WSR-88D Algorithm Programming Guide* contains the tutorial, reference, and sample algorithms for the *WSR-88D Algorithm API* and guidance for the structure of algorithms.

Section I Base Data Format

A reference to the structure of the base data radial message. This message is the format of the WSR-88D radar data provided to the algorithms which differs from the message passed from the RDA to the ORPG. The structure of the base data elevation message is also described.

Section II Algorithm Adaptation Data - Configuration & Use

The WSR-88D uses adaptation data to configure many aspects of the radar system. A portion of this configuration data is used to alter or customize the contents of WSR-88D products. This section contains an overview of algorithm specific adaptation data and procedures for proper configuration.

Section III Other Data Inputs

In addition to base data from the radar and algorithm specific adaptation data, algorithms can use intermediate product data produced by other algorithms, external data obtained from other systems, and miscellaneous configuration data.

Vol 2. Document 4 -
Additional Information & Guidance for WSR-88D Algorithm Developers

Section I Base Data Format

Part A. Volume Scanning Strategies

The volumetric structure of the data consists of multiple elevations of radar data with each elevation consisting of multiple base data radial messages. Each elevation is actually a conic section with data sampled at a particular elevation angle. The data are sampled using one of several predefined scanning strategies, called volume coverage patterns (VCP).

A detailed description of WSR-88D VCPs can be found in the Federal Meteorological Handbook No. 11, Part C, Chapter 5 (***FMH 11, Part C***).

Originally only 4 WSR-88D VCP's were defined. Two of the original VCPs were optimized for severe weather conditions (precipitation mode) and two were optimized for clear air conditions (clear air mode). The following VCPs are currently defined.

As of Build 22.0, WSR-88D volume coverage patterns can be divided into 4 groups.

General Surveillance	VCP 215
Rapidly Evolving Convection	VCP 12 / 212
Hurricanes/Wide-spread Stratiform Precipitation	VCP 112
Clear Air	VCP 31 / 35

Additional Information on Volume Coverage Patterns (VCP)

Detailed knowledge of how the radar samples the atmosphere is not required for basic algorithm development. It is often useful however to know the number of elevations in a VCP and the angle for each. The rest of this section contains additional information if interested. If not, skip ahead to *ORPG Internal Basedata*.

The factors that drive the design of the scanning strategy include the tradeoff between unambiguous range and unambiguous velocity inherent in Doppler radar, the temporal and spatial scales of the meteorological conditions of interest, the desired precision and accuracy of the data obtained, and the characteristics of the specific radar. At elevation angles below approximately 6 degrees, the data are sampled with two (or more) different PRFs in order to assign the Doppler data to the correct surveillance echo (range unfolding). Two methods are used to accomplish this multiple sampling. Below approximately 2.5 degrees, this is accomplished by scanning each elevation more than once (see "split cuts" below) in order to meet accuracy requirements and obtain reflectivity data to 460 km. Generally,

from about 2.5 degrees through 6 degrees this is accomplished in a "batch" mode -- the radar rapidly switching between a lower PRF (surveillance) and a higher PRF (Doppler). Above 6 degrees range unfolding is not required since there are generally no significant weather returns at higher altitudes. This means that no echoes occur at longer slant ranges at higher elevation angles, eliminating the ambiguity. Additional information can be obtained from the Federal Meteorological Handbook No. 11, Part B, Doppler Radar Theory and Meteorology (***FMH 11, Part B***).

The manner in which the data are sampled by the radar is related to but differs from the manner in which the data are presented to the algorithms. Stated in another way, the content of the data messages sent to the ORPG from the radar (from the RDA) is not the same as the base data radial message read by an algorithm. Other than having different structures and contents in the header portion of the message, the data content differs in several significant ways.

1. Data from the RDA "split cuts" have been combined.
2. With the Legacy RDA: Any data sample bins that are "behind" the radar are stripped out of the RDA radial message. For the original radar, typically the first two Doppler bins (both velocity and spectrum width) were discarded. The third RDA Doppler bin became the first RPG Doppler bin.
3. The velocity information provided to the algorithms has been dealiased by the ORPG.
4. Data sampled at the new higher resolution (0.5 deg azimuth and 250 meter reflectivity data bins) may be recombined into the original resolution depending upon the type of internal basedata being read. Only the super resolution types (**`SR_BASEDATA`**, etc.) and the rawdata types (**`RAWDATA`**, etc.) described in Part D have no recombination applied. The increased Doppler range of 300 km at the lower elevation angles is provided to all registration types.
5. The dual polarization fields are pre-processed.

Split Cuts

For all VCPs, the lowest 2 or 3 elevations are scanned twice. The first scan is at a lower PRF and the base data messages transmitted to the ORPG include surveillance data bins (reflectivity data). The second scan at that elevation uses a higher PRF and the data messages transmitted include Doppler data bins (radial velocity and spectrum width data). The RPG inserts reflectivity data from the closest radial (in azimuth) in the first surveillance scan into each radial from the subsequent Doppler scan. Both the surveillance scan and the subsequent Doppler scan of a split cut are present in the rawdata and basedata linear buffers in the RPG. At higher elevations the data are sampled in a single scan.

For VCP 112, the lowest 2 elevations are scanned four times, the 3rd and 4th elevations are scanned three times, and the 6th is scanned twice. The first scan is at a lower PRF and the base data messages transmitted to the ORPG include surveillance data bins (reflectivity data). The subsequent Doppler scans are at varied PRFs and used by the new multi-pulse repetition frequency dealiasing algorithm (MPDA) which help mitigate range folding. All of the subsequent Doppler scans are present in the rawdata linear buffer. The first subsequent Doppler scan includes reflectivity data that has been inserted from the closest surveillance radial from the first surveillance scan. After additional processing (velocity dealiasing) both the surveillance scan and only one Doppler scan from the split cut (includes the inserted reflectivity data) are stored in the basedata linear buffer.

ORPG Internal Basedata

Other than having different structures and contents in the header portion of the message, the internal basedata content differs in several significant ways from the external basedata received from the RDA.

1. Data from the RDA "split cuts" have been combined.
2. With the Legacy RDA: Any data sample bins that are "behind" the radar are stripped out of the RDA radial message. For the original radar, typically the first two Doppler bins (both velocity and spectrum width) were discarded. The third RDA Doppler bin became the first RPG Doppler bin.
3. The velocity information provided to the algorithms has been dealiased by the ORPG.
4. **Data sampled at the new higher resolution (0.5 deg azimuth and the higher resolution 250 meter reflectivity data bins may be recombined into the original resolution depending upon the type of internal basedata being read. The new extended Doppler range of 300 km is not recombined. Only the super resolution types (`SR_BASEDATA`, etc.) and the rawdata types (`RAWDATA`, etc.) described in Part D have no recombination applied.**
5. The dual polarization fields are pre-processed.

> The following discussion has been simplified by referencing only the original base data types. The Super Resolution and Dual Pol types are not included. Whenever **`REFLDATA`** is used, **`SR_REFLDATA`**, and **`DUALPOL_REFLDATA`** also apply. Whenever **`COMBBASE`** is used, **`SR_COMBBASE`**, and **`DUALPOL_COMBBASE`** apply. Whenever **`BASEDATA`** is used, **`SR_BASEDATA`**, and **`DUALPOL_BASEDATA`** apply.

Combining Split Cuts

The ORPG transforms the RDA base data messages into the ORPG base data messages described in this document. Algorithms interested only in reflectivity data register for **`REFLDATA(79)`** input and read ORPG data messages created directly from the first scan of a split cut. Algorithms interested in velocity / spectrum width data (and reflectivity data) register for **`COMBBASE(96)`** input and read ORPG data messages derived from the subsequent Doppler scans of a split cut. The ORPG data messages created from single scan elevation samples are read by all algorithms inputting base data.

Data messages read when registered for **`REFLDATA(79)`** and **`COMBBASE(96)`** contain a single radial of base data. A message containing a collection of radials forming a complete elevation scan can be obtained by registering for **`REFLDATA_ELEV(302)`** and **`COMBBASE_ELEV(303)`**. These elevation messages are used by very few algorithms at this time.

The ORPG base data messages derived from the subsequent Doppler scans of a split cut also contain reflectivity data that has been inserted from first scan of that cut. This process involves selection of the first scan reflectivity radial message with an azimuth closest to the Doppler scan azimuth. Thus, the reflectivity is "velocity mapped"; it no longer includes the original azimuth information. As a result, algorithms that are registered for **`COMBBASE(96)`** or **`COMBBASE_ELEV(303)`** input can process reflectivity data in addition to velocity and spectrum width. With data obtained from a radar before the Build 9

ORDA, the reflectivity data for elevations derived from a split cut (inserted reflectivity data) will have a minor azimuth error.

It is also possible to register for **BASEDATA(55)** and **BASEDATA_ELEV(301)**. This requires additional logic in the algorithm to distinguish between the first surveillance scan and the subsequent Doppler scans in a split-cut elevation. With **BASEDATA(55)** radial messages, the messages from both scans of a split cut will have the same elevation index but a different scan number. With **BASEDATA_ELEV(301)** elevation messages, both messages from a split cut have the same elevation index.

Note that the measured / non-processed Dual Polarization fields (KDP, PHI, and RHO) are only provided in the **SR_BASEDATA(76)**, **SR_REFLDATA(78)**, and **SR_COMBBASE(77**) and that the derived / processed Dual Polarization fields are only available in the **DUALPOL_BASEDATA(305)**, **DUALPOL_REFLDATA(307)**, and **DUALPOL_COMBBASE(306)**.

Velocity Dealiasing

Generally, all processing of the basic moment (R, V, SW) data except velocity dealiasing is accomplished by the RDA. This includes: signal processing, conversion to meteorological units, point target suppression, suppression of data below a set threshold, and range unfolding. Additional information can be obtained from the ***FMH 11, Part B***.

> The following discussion has been simplified by using only the original base data types. The recombined rawdata types are not included. **RECOMBINED_REFL_RAWDATA, RECOMBINED_COMB_RAWDATA, and RECOMBINED_RAWDATA** have been removed in Build 12 and should not be used.

Velocity data that has not been dealiased can be obtained by reading from the **rawdata** linear buffer rather than the **basedata** linear buffer. USING **RAWDATA(54)** IS NOT RECOMMENDED UNLESS THE ORPG VELOCITY DEALIASING ALGORITHM MUST BE BYPASSED.

- When reading from the **rawdata** linear buffer you must register for **REFL_RAWDATA(66)** rather than **REFLDATA(79)** or **COMB_RAWDATA(67)** rather than **COMBBASE**.
- As with any of the '**BASEDATA**' types, registering for **RAWDATA(54)** requires additional logic in the algorithm to distinguish between the first surveillance scan and the subsequent Doppler scans in a split-cut elevation. For VCP 112 this is more complicated than when registering for any of the '**BASEDATA**' types because all of the multiple Doppler scans at the lower levels are present in **RAWDATA(54)**.
- There is no corresponding complete elevation message for reading pre-dealiased data.

Pre-Processing Dual Polarization Data

The RDA provides three Dual Polarization data fields (ZDR, PHI, and RHO). The ORPG processes these three fields (includes a smoothing technique) and generates six additional Dual Polarization data fields (SNR, SMZ, SMV, KDP, SDZ, and SDP).

Part B. Base Data Radial Message

The ORPG provides WSR-88D radar data to algorithms in the message format shown below. Each message represents one radial of polar coordinate base data. The radial message structure and defined offsets are documented in **`basedata.h`**.

> **IMPORTANT NOTE: The base data radial messages defined in this document are internal to the ORPG and used by the ORPG algorithms. This message is not the same as the base data radial message that is passed from the RDA to the ORPG.** The message described here is defined by the C structure **`Base_data_radial`** which contains the structure **`Base_data_header`**.
>
> The other structures defined in **`basedata.h`** (**`RDA_basedata_header`** and **`ORDA_basedata_header`**) should be ignored.

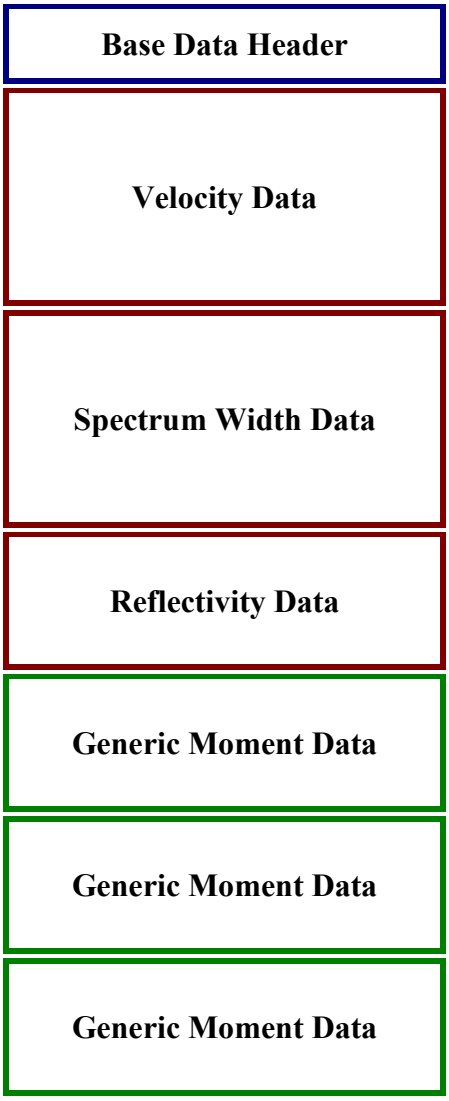
The first part of the message is the *Base Data Header* with a structure documented in Appendix D. Much of this information is not of interest to the algorithm developer but used by the ORPG infrastructure. Certain fields (flagged with "I" in the table) contain data that are placed into the header portions of the final ICD formatted product. The legacy FORTRAN algorithms access the message contents via an offset into the memory block. The ORPG infrastructure and algorithms written in ANSI-C can access the message via a defined structure.

Base Data Radial - Beginning with ORPG Build 10

The actual data follow the header. Beginning with Build 10 these arrays are not in a specified order. The velocity and spectrum width blocks may contain either 920 values (230km range) or 1200 values (300km range) and the size of the reflectivity block may contain either 460 values (460 km range) 1840 values (460 km range in 1/4 km increments).

For all moments, the number of valid data value may be fewer than the block size depending on the range or elevation that data are sampled. Offsets that access the data blocks and header fields that specify their size are described later in this document.

Following the original basic moment data are a series of generic moment structures which will contain the advanced Dual Polarization data fields. This structure is documented in Appendix E.



***The Base Data Header is subject to change. Because of this a standard size should never be assumed. The definition of `BASEDATA_HD_SIZE` is defined as `(sizeof(Base_data_header) / sizeof(short))`.**

Offsets for the Radial Message

If ingesting data from a Build 10 ORDA or later, the basic data moments (reflectivity, velocity, and spectrum width) are not necessarily in any specified order. Therefore the required method of accessing the data blocks is using the offset fields in the base data header: **`ref_offset`**, **`vel_offset`**, and **`spw_offset`**. These offsets are in bytes relative to the beginning of the base data header. NOTE: These offset fields have a different meaning in the elevation base data message.

The position of the first good data value (usually 1) is determined by the fields: **`surv_range`** and **`dop_range`**. The number of valid data values within the array is determined by the value of **`n_dop_bins`** and **`n_surv_bins`**.

The algorithm API provides support for accessing the data arrays. Functions **`RPGC_get_surv_data`**, **`RPGC_get_vel_data`**, and **`RPGC_get_wid_data`** use the offsets to obtain the beginning of the reflectivity, velocity, and spectrum width data. The function **`RPGC_get_radar_data`** is used to access the dual polarization data. See Part G., *Reading Base Data Messages*, in Volume 3, Document 2 Section II.

The Number and Size of Radial Data Arrays

See this topic at the end of Part D - Selecting Desired Base Data Messages.

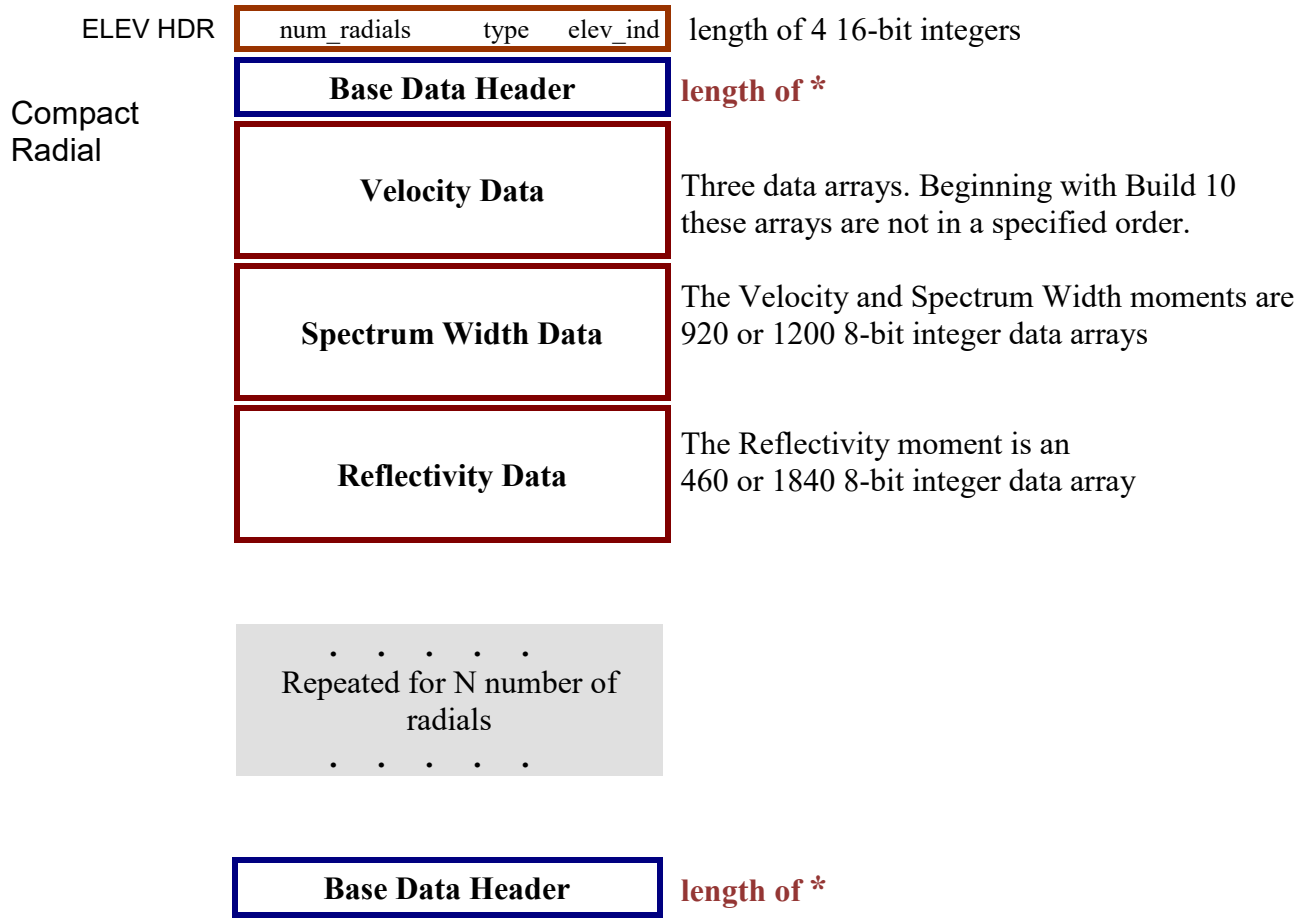
Part C. Base Data Elevation Message

The ORPG provides a second message format to algorithms. This message represents a collection of all radial data comprising a complete elevation scan. The elevation message structure and defined offsets are documented in **`basedata_elev.h`**. 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.

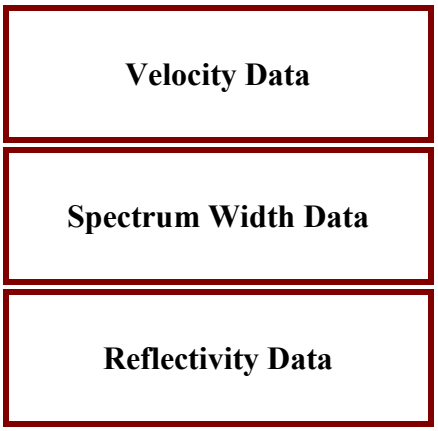
The first part of the message is a short header providing number of radials in the elevation, the type of elevation, and the elevation index. The **`type`** field corresponds to **`msg_type`** in the base data header and the **`elev_ind`** field corresponds to **`rpg_elev_ind`**.

Base Data Elevation - Beginning with ORPG Build 10

The actual radial data follows the elevation header. Beginning with Build 10 these arrays are not in a specified order. As with the *Base Data Radial Message* the velocity and spectrum width blocks may contain either 920 values (230km range) or 1200 values (300km range) and the size of the reflectivity block may contain either 460 values (460 km range) or 1840 values (460 km range in 1/4 km increments). The size of the message has been reduced by using only 1 byte for each data value rather than a 2-byte integer.



Compact Radial



Three data arrays. Beginning with Build 10 these arrays are not in a specified order.

The Velocity and Spectrum Width moments are 920 or 1200 8-bit integer data arrays

The Reflectivity moment is an 460 or 1840 8-bit integer data array

***The Base Data Header is subject to change. Because of this a standard size should never be assumed. The definition of `BASEDATA_HD_SIZE` is defined as `(sizeof(Base_data_header) / sizeof(short))`.**

Offsets for the Elevation Message

If ingesting data from a Build 10 ORDA or later, the basic data moments (reflectivity, velocity, and spectrum width) are not necessarily in any specified order. Therefore the required method of accessing the data blocks is using the offset fields in the base data header: **`ref_offset`**, **`vel_offset`**, and **`spw_offset`**.

The position of the first good data value (usually 1) is determined by the fields: **`surv_range`** and **`dop_range`**. The number of valid data values within the array is determined by the value of **`n_dop_bins`** and **`n_surv_bins`**.

The algorithm API does not provide support for reading the data elements contained in the elevation message. Very few algorithms use the elevation message. See the EPRE algorithm in **`cpc013/tsk003`** (this algorithm is being modified in Build 10 to use the header offsets to access moment data array).

The Number and Size of Radial Data Arrays

See this topic at the end of Part D - Selecting Desired Base Data Messages.

Part D. Selecting Desired Base Data Messages

IMPORTANT NOTE: The base data radial messages defined in this document are internal to the ORPG and used by the ORPG algorithms. This message is not the same as the base data radial message that is passed from the RDA to the ORPG. The message described here is defined by the C structure **`Base_data_radial`** which contains the structure **`Base_data_header`**.

The other structures defined in **`basedata.h`** (**`RDA_basedata_header`** and **`ORDA_basedata_header`**) should be ignored.

Radial Base Data Messages

Most algorithms obtain base data reading individual radial messages rather than the elevation base data messages. There are 3 base data types that are registered within the 3 categories below.

Registration Name	Basic Moments Present			Dual Pol Data Fields Present		Resolution / Size of the Data Arrays		
	R	V	W	Non Processed	Derived / Processed	Reflect Res	Doppler Range	Azimuth Sampling
1. Original Data Registrations Note 5								
`REFLDATA(79)`	Yes					1000 m Note 2	300 km / 230 km Note 2	0.5 Deg / 1.0 Deg Note 1
`COMBBASE(96)`	Yes	Yes	Yes					
`BASEDATA(55)`	Yes	Yes	Yes					
2. Super Resolution Data Registrations (available in Build 10)								
`SR_REFLDATA(78)`	Yes			**`ZDR`** **`PHI`** **`RHO`** Note 3		250 m Note 2	300 km / 230 km Note 2	0.5 Deg / 1.0 Deg Note 1
`SR_COMBBASE(77)`	Yes	Yes	Yes					
`SR_BASEDATA(76)`	Yes	Yes	Yes					
3. Dual Polarization Data Registrations (available in Build 12)								
`DUALPOL_REFLDATA(307)`	Yes				Note 4	250 m	300 km / 230 km Note 2	1.0 Deg
`DUALPOL_COMBBASE(306)`	Yes	Yes	Yes					
`DUALPOL_BASEDATA(305)`	Yes	Yes	Yes					

Note 1 The current VCPs provide 0.5 deg azimuth sampling at the Super Resolution elevations (lower elevations including the split cut elevations) and provide 1.0 deg azimuth sampling at other elevations.

Note 2 **The latest design for Build 12.1 is to provide the higher 250 meter horizontal resolution for reflectivity at all elevations and the extended 300 kilometer range for Doppler data at lower elevations (Contiguous Doppler and Batch).**

Note 3 **The non-processed (measured) Dual Polarization data fields are available in Build 12 with data produced by a Build 12 RDA.**

Note 4 The identity of the Processed and Derived Dual Polarization data fields available in this message are listed below in Part D.

Note 5 **For the original data registrations the ORPG recombines the higher resolution data in order to provide the original azimuth sample interval (1.0 deg) and the original 1 km surveillance interval. The Doppler range (corresponding to the maximum number of bins) remains at 300 km at lower elevations.**
All existing algorithms were modified to only use 230 km Doppler range. However, product 99, Base Velocity Data Array Product (DV), uses the 300 km Doppler range on the elevations provided.

Even though not used by any current algorithm, it is possible to by-pass the ORPG velocity dealiasing accomplished in the ORPG. There are 3 raw data types that can be registered within the categories below.

Registration Name	Basic Moments Present			Dual Pol Data Fields Present		Resolution / Size of the Data Arrays		
	R	V	W	Non Processed	Derived / Processed	Reflect Res	Doppler Range	Azimuth Sampling
Pre - Dealiased Data Registrations (Before Build 10)								
REFL_RAWDATA(66)	Yes					1000 m	230 km	1.0 Deg
COMB_RAWDATA(67)	Yes	Yes	Yes					
RAWDATA(54)	Yes	Yes	Yes					
Pre - Dealiased Data Registrations (After Build 10)								
REFL_RAWDATA(66)	Yes			**ZDR PHI RHO** Note 3		250 m Note 2	300 km / 230 km Note 2	0.5 Deg / 1.0 Deg Note 1
COMB_RAWDATA(67)	Yes	Yes	Yes					
RAWDATA(54)	Yes	Yes	Yes					

1. Original Resolution Data Registrations

   These are the Legacy base data types provided in the original WSR-88D. These data registrations provide

   - The original three base data moments at the original resolution. See the *Summary of Elevation and Radial Characteristics* below.

REFLDATA(79)
This data type is only used when there is no need for radial velocity or spectrum width moments. Reflectivity data is the only basic moment contained in the message.

COMBBASE(96)
This data type is used when either velocity or spectrum width is needed. All moment data (reflectivity, velocity, spectrum width) is contained in the message.

BASEDATA(55)
This data type is rarely used because it requires additional work in the algorithm and additional knowledge of the VCP and split cuts. Like **COMBBASE** all original moments (reflectivity, velocity, spectrum width) are available. Since (beginning with Build 9) radials in all elevation scans are aligned, there is normally no advantage for using **BASEDATA**. For an example of use, see the **recclalg** task in cpc004/tsk006. See *Split Cuts* below.

2. Super Resolution Data Registrations (available in Build 10)

   These data registrations provide

   - The original three base data moments but at the increased resolution as determined by the VCP definitions. See the *Summary of Elevation and Radial Characteristics* below.
   - Beginning with Build 12, the *Non-processed (or Measured)Dual Pol Fields* (listed in Part D below) are included.

SR_REFLDATA(78)
: As with **REFLDATA**, reflectivity is the only basic moment contained in the message.

SR_COMBBASE(77)
: As with **COMBBASE**, all basic moment data (reflectivity, velocity, spectrum width) is contained in the message.

SR_BASEDATA(76)
: As with **BASEDATA**, this type is rarely used because it provides no additional information and requires additional work in the algorithm and additional knowledge of the VCP and split cuts. For an example of use, see the **superes8bit** task in cpc007/tsk015. See *Split Cuts* below. All original moments (reflectivity, velocity, spectrum width) are available.

3. Dual Polarization Data Registrations (available in Build 12)

   These data registrations provide

   - The original three base data moments at the increased reflectivity resolution and the increased Doppler range but with the original 1.0 deg azimuth sampling. See the *Summary of Elevation and Radial Characteristics* below.
   - Beginning with Build 12, the *Processed and Derived Dual Pol Fields* (listed in Part D below) are included.

DUALPOL_REFLDATA(307)
: As with **REFLDATA**, reflectivity is the only basic moment contained in the message.

DUALPOL_COMBBASE(306)
: As with **COMBBASE**, all basic moment data (reflectivity, velocity, spectrum width) is contained in the message.

DUALPOL_BASEDATA(305)
: As with **BASEDATA**, this type is rarely used because it provides no additional information and requires additional work in the algorithm and additional knowledge of the VCP and split cuts. See *Split Cuts* below. All original moments (reflectivity, velocity, spectrum width) are available.

4. Pre-Dealiased Base Data Registrations

   These data registrations are not recommended for algorithm development. There is little reason to bypass velocity dealiasing but pre-dealiased data can be obtained by registering of

REFL_RAWDATA(66) or **COMB_RAWDATA(67)**. All cuts of a split cut can be obtained by registering for **RAWDATA(54)**. See **Velocity Dealiasing** below for more information.

Data is as received from the RDA.

- Prior to Build 10 this only includes the original resolution base data moments.
- Beginning with Build 10 the increased resolution as determined by the VCP definitions are provided.
- Beginning with Build 12 the *Dual Pol Fields* are present on the first cut of a split cut. Early test Dual Pol base data contained the *Dual Pol Fields* on both cuts of a split cut.

Elevation Base Data Messages

Very few algorithms currently read the elevation base data messages. The contents of the messages are the same as their radial counterparts.

1. Original Resolution Data

   REFLDATA_ELEV(302)

   COMBBASE_ELEV(303)

   BASEDATA_ELEV(301)

2. Super Resolution Data (Not Available in the elevation message)
3. Dual Polarization Data (Not Available in the elevation message)

The Number and Size of Radial Data Arrays

This is subject is covered in detail in Volume 3 and is summarized here.

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.

Part E. Dual Polarization Data Fields

The descriptions of these data fields will be expanded in future versions of this guide.

IMPORTANT NOTE: The base data radial messages defined in this document are internal to the ORPG and used by the ORPG algorithms. This message is not the same as the base data radial message that is passed from the RDA to the ORPG. The message described here is defined by the C structure **`Base_data_radial`** which contains the structure **`Base_data_header`**.

The other structures defined in **`basedata.h`** (**`RDA_basedata_header`** and **`ORDA_basedata_header`**) should be ignored.

Non-Processed (or Measured) Dual Pol Fields

The following data fields are received from the ORDA (Build 12) data stream. They are contained in the **`SR_REFLDATA`**, **`SR_COMBBASE`**, and **`SR_BASEDATA`** data types and in the raw data types.

`ZDR` Differential Reflectivity

`PHI` Differential Phase

`RHO` Correlation Coefficient

Processed and Derived Dual Pol Fields

The Build 12 ORPG creates additional data fields called *Processed Dual Pol Fields*.

The following fields along with the 3 basic moments are in **`DUALPOL_REFLDATA`**, **`DUALPOL_COMBBASE`**, and **`DUALPOL_BASEDATA`**. **This is the primary source of Dual Polarization moments for algorithms.**

`DZDR` Differential Reflectivity - processed

`DPHI` Differential Phase - processed (long gate)

`DRHO` Correlation Coefficient - processed

`DSNR` Signal-to-Noise Ratio

`DSMZ` Smoothed Reflectivity

`DSMV` Smoothed Velocity

DKDP Specific Differential Phase

DSDZ Texture (standard deviation) of Reflectivity

DSDP Texture (standard deviation) of Differential Phase

If special quality index fields are desired in addition to the 9 Dual Polarization data fields, the intermediate product **DP_BASE_AND_QUALITY(320)** can be used.

If melting layer data and hydrometer classification data are needed in addition to the 9 Dual Polarization data fields, the intermediate product **HCA(321)** can be used.

Part F. Characteristics of the WSR-88D Volume

The Elevation Characteristics and Radial Characteristics are summarized together in a table below covering WSR-88D Builds 8, 9, and 10. Knowledge of previous builds is provided in the event historical radar data is ingested into the ORPG.

Using Historical Radar Data

When ingesting historical radar data remember:

- With the original RDA (prior to the Build 9 ORDA), the radials were not aligned to a specific azimuth and not aligned from one elevation to the next. The number or radials in an elevation scan often varied slightly.
- The Build 10 and later ORPG makes "recombined" data available to algorithms using the original 1.0 deg sample interval, 1 km surveillance resolution and 230 km Doppler range, regardless of the nature of the input data.
- The higher resolution data types will not be available if ingesting historical data from an RDA prior to Build 10. The Dual Polarization data types will not be available if ingesting historical data prior to Build 12.

Summary of Elevation and Radial Characteristics

WSR-88D Build	Radials Aligned	Number of Radials	Radial Spacing	Beam Width	Surveillance		Doppler	
					Interval	Range	Interval	Range
Legacy RDA	**No**	**~366**	~0.98 deg	(see text)	1 km	460 km	250 m	230 km
Build 9 ORDA	Yes	360	1.0 deg	(see text)	1 km	460 km	250 m	230 km
Build 10 ORDA	Yes	360	1.0 deg	(see text)	250 m*	460 km	250 m	300 km*
Build 10 ORDA (SR Elevations)	Yes	720	0.5 deg	(see text)	250 m*	460 km	250 m	300 km*

* The higher 250 meter horizontal resolution for surveillance data and the extended 300 kilometer range for Doppler data are limited to the Super Resolution Elevations in the Build 10 VCP designs.
The latest design for Build 12.1 is to provide the higher 250 meter horizontal resolution for surveillance data at all elevations and the extended 300 kilometer range for Doppler data at lower elevations (Contiguous Doppler and Batch). Appropriate data fields in the radial message header must be used when reading base data containing the higher resolution data fields to determine the actual array size.
The ORPG recombines the higher resolution data in order to provide the original azimuth sample interval (1.0 deg) and the original 1 km surveillance interval. The Doppler range (corresponding to the maximum number of bins) remains at 300 km. All existing algorithms were modified to only use 230 km Doppler range. However, product 99, Base Velocity Data Array Product (DV), uses the 300 km Doppler range on the elevations provided.

Elevation Characteristics

- Number of Elevations
  Originally, a volume scan strategy could not contain more than 20 elevations. This has been increased to 25.

- Radial Spacing
  - Legacy RDA.
    Elevations normally consist of just over 360 radials. The radial spacing is typically between 0.95 degrees and 1.1 degrees. Under certain rare conditions, an elevation could contain up to 400 radials. Radial spacing is relatively stable for a given radar, however.
  - Build 9 ORDA.
    Elevations normally consist of exactly 360 radials. The radial spacing is typically between 0.9 degrees and 1.1 degrees. Under certain rare conditions, an elevation could contain up to 400 radials.
  - Build 10 / 12 ORDA.
    With the current VCP definitions, the new 0.5 degree sample interval occurs at the lower elevations. **The Build 12.1 design is through the split cut elevations.** **Note 1**
    - 1 Degree Sample Interval: Elevations normally consist of exactly 360 radials. The radial spacing is typically between 0.9 degrees and 1.1 degrees. Under certain rare conditions, an elevation could contain up to 400 radials.
    - 0.5 Degree Sample Interval: Elevations normally consist of exactly 720 radials. The radial spacing is typically 0.5 degrees. Under certain rare conditions, an elevation could contain up to 800 radials.

- Radial Alignment
  - Legacy RDA. Elevations do not begin at the same azimuth and radials from one elevation are not aligned with radials in other elevations. After the radar antenna has repositioned to a new elevation angle, data sampling begins when radar parameters (including antenna positioning) have stabilized within specified limits.
  - Build 9 ORDA & Build 10 ORDA. Even though elevations do not begin at the same azimuth, radials from one elevation are aligned with radials in other elevations (typically within 0.1 degrees). After the radar antenna has repositioned to a new elevation angle, data sampling begins when radar parameters (including antenna positioning) have stabilized within specified limits.

- Antenna Position Accuracy
  The radar pedestal positioning accuracy is ± 0.2 degrees in elevation and azimuth.

Note 1: For all but the Super Resolution base data types, the data are recombined in the ORPG into 1.0 degree radials.

Radial Characteristics

- Beam Width.
  - Legacy RDA.
    The antenna beam width (actually half power beam width) is nominally 0.95 degrees. Radials are spaced approximately every degree.
  - Build 9 ORDA.
    The antenna beam width (actually half power beam width) is nominally 0.95 degrees. Radials are spaced approximately every degree.
  - Build 10 ORDA.
    - 1 Degree Sample Interval: The antenna beam width (actually half power beam width) is nominally 0.95 degrees. However the effective beam width could be as much as 1.5 degrees due to the time windowing sampling function. Radials are spaced every whole degree.
    - 0.5 Degree Sample Interval: The antenna beam width (actually half power beam width) is nominally 0.95 degrees. However the effective beam width could be as much as 1.1 degrees due to the time windowing sampling function. Radials are spaced every half degree.

- Surveillance Range / Interval
  - Legacy RDA & Build 9 ORDA:
    Reflectivity range precision is one data point every 1.0 kilometers from 1 km to 460 km. **Note 1**.
  - Build 10 / 12 ORDA:
    Reflectivity range precision is one data point every 1.0 kilometers from 1 km to 460 km or one data point every 0.25 kilometers from 0.25 km to 460 km. **Note 1**, **Note 3**

- Doppler Range / Interval
  - Legacy RDA & Build 9 ORDA:
    Radial velocity and spectrum width range precision is one data point every 0.25 kilometers from 0.25 km to 230 km. **Note 1**.
  - Build 10 / 12 ORDA:
    Radial velocity and spectrum width range precision is one data point every 0.25 kilometers from 0.25 km to 230 km or 300 km. **Note 1**, **Note 4**.

- Location of first bin.
  For the ORPG internal data used by algorithms:
  - Legacy RDA.
    The center of the first reflectivity bin is normally at 0 kilometers from the radar (placing half of the first bin behind the radar). The center of the first Doppler bin (velocity and spectrum width) is 0.125 kilometers from the radar (placing the leading edge of the bin at the antenna).
  - Build 9 ORDA & Build 10 ORDA.
    The range to the leading edge of the first surveillance bin is 0 km and the range to the leading edge of the first Doppler bin is also 0 km. These correspond to center ranges of 500 m and 125 m, respectively.

Note 1: There is no requirement for data above 70,000 feet MSL. Data values at ranges where altitude exceeds 70,000 feet MSL are not valid.

Note 2: The range scale corresponds to slant range along the beam, not the distance across the earth's surface.

Note 3: The surveillance range interval can be configured as either 1 km or 0.25 km in the Build 10 ORDA VCPs. The Build 10 VCP design uses a 0.25 km range interval at the super resolution elevations and 1 km range at other elevations. **The Build 12.1 VCP definitions provide 0.25 km reflectivity range interval for all elevations.** The data are recombined in the ORPG to obtain the 1 km resolution for use with the original base data types.

Note 4: Before Build 12 the maximum Doppler range can be configured as either 230 km or 300 km in the ORDA VCPs. The Build 10 VCP design used a 300 km Doppler range at the super resolution elevations and the 230 km Doppler range at other elevations. **The Build 12.1 VCP definitions provide 300 km Doppler range for lower elevations (Contiguous Doppler and Batch).**

Part G. Data Characteristics

Encoding details and general characteristics of WSR-88D base data are provided below. Data requirements for coverage area, sensitivity, precision, and accuracy are documented in the WSR-88D System Specification (***SS***), Sections 3.7.1 and 3.7.2.

Data Range and Precision

The actual range of data values that can be measured is based upon many parameters and is beyond the scope of this introduction. The range of values that can be encoded is stated here. Generally, this encoding range exceeds the system's capability to measure / estimate the characteristic (no information is lost due to encoding limitations). ***FMH 11, Part B*** provides a discussion on WSR-88D radar characteristics and data acquisition considerations.

1. The precision of the data is defined as the smallest increment recorded due to the encoding scheme.
2. The range of the data (minimum and maximum values) is the range that can be encoded, not the range of values produced by the radar.

Basic Moment Data Fields:

- **Reflectivity** (equivalent radar reflectivity **Ze)**
  - is provided in increments of **`0.5 dBZ`** from **`-32.0`** to +**`94.5 dBZ`** using 8-bit representation
- **Radial Velocity**
  Generally the RDA is set to provide the best precision of the velocity data (Doppler resolution 1). Under certain conditions the RDA is set to encode a greater range of velocities but with reduced precision (Doppler resolution 2).
  - is provided in increments of **`0.5 meters/second`** from **`-63.5`** to +**`63.0 meters/second`** using 8-bit representation (Doppler resolution 1)
  - is provided in increments of **`1.0 meters/second`** from **`-127`** to +**`126 meters/second`** using 8-bit representation (Doppler resolution 2)
- **Spectrum Width**
  - is provided in increments of **`0.5 meters/second`** from **`-63.5`** to +**`63.0 meters/second`** using 8-bit representation

Dual Polarization Measured Data Fields:

- **`DZDR` - Differential Reflectivity**
  - is provided in increments of **`0.0625 dB`** from **`-7.8750`** to +**`7.9375 dB`** using 8-bit representation
- **`DPHI` - Differential Phase**
  - is provided in increments of **`0.3526 deg`** from **`0.00`** to **`360.00 deg`** using 10-bit representation
- **`DRHO` - Correlation Coefficient**

- is provided in increments of `0.0033` from `0.2067` to `1.0500` (Dimensionless) using 8-bit representation

Dual Polarization Derived Data Fields:

IMPORTANT NOTE: The precision of representation of the data fields `DRHO`, `DPHI`, `DKDP`, and `DSDZ` significantly exceeds the specification of the preprocessing algorithm. Even though the accuracy of the derived / processed fields has not yet officially been determined, the precision of representation of these four data fields also exceed the accuracy of the data. Any algorithm using the data fields `DRHO`, `DPHI`, `DKDP`, and `DSDZ` should not assume that the precision or number of significant digits in the decoded value are representative of the data accuracy.

- `DZDR` - **Differential Reflectivity - processed**
  - is provided in increments of `0.0625 dB` from `-7.8750` to +`7.9375 dB` using 8-bit representation
- `DPHI` - **Differential Phase - processed** ***(Precision exceeds data accuracy)***
  - is provided in increments of `0.016479 deg` from `-0.6722` to `1079.278 deg` using 16-bit representation
- `DRHO` - **Correlation Coefficient - processed** ***(Precision exceeds data accuracy)***
  - is provided in increments of `0.000013` from `0.20003` to `1.05332` (Dimensionless) using 16-bit representation
- `DSNR` - **Signal-to-Noise Radio**
  - is provided in increments of `0.5 dB` from `-12.0` to +`114.5 dB` using 8-bit representation
- `DSMZ` - **Smoothed Reflectivity**
  - is provided in increments of `0.5 dBZ` from `-32.0` to +`94.5 dBZ` using 8-bit representation
- `DSMV` - **Smoothed Velocity**
  - is provided in increments of `0.5 meters/second` from `-63.5` to +`63.0 meters/second` using 8-bit representation (Doppler resolution 1)
  - is provided in increments of `1.0 meters/second` from `-127` to +`126 meters/second` using 8-bit representation (Doppler resolution 2)
- `DKDP` - **Specific Differential Phase** ***(Precision exceeds data accuracy)***
  - is provided in increments of `0.00019 deg/km` from `-2.149` to +`10.650 deg/km` using 16-bit representation
- `DSDZ` - **Texture (standard deviation) for Reflectivity** ***(Precision exceeds data accuracy)***
  - is provided in increments of `0.0012 dBZ` from `0.00` to `30.37 dBZ` using 8-bit representation
- `DSDP` - **Texture (standard deviation) for Differential Phase**
  - is provided in increments of `0.4 deg` from `0.0` to `101.2 deg` using 8-bit representation

Data Accuracy

The accuracy of the data depends upon many factors including PRF, antenna rotation rate, and clutter suppression. ***FMH 11, Part B*** provides a discussion on WSR-88D radar characteristics and data acquisition considerations. Typical values for standard deviation are approximately:

- **Reflectivity:** `1 dBZ`
- **Radial Velocity:** `1 meter/second`
- **Spectrum Width:** `1 meter/second`

Dual Polarization Measured Data Fields:

- `DZDR` - **Differential Reflectivity:** `0.3 dB`
- `DPHI` - **Differential Phase:** `2.0 deg`
- `DRHO` - **Differential Correlation:** `0.005`

Dual Polarization Derived Data Fields:

Currently the accuracy of the processed / derived data fields is not yet officially been determined, though it is likely that the accuracy of `DSMZ`, and `DSMV` are similar to the basic moments Reflectivity and Radial Velocity.

- `DZDR` - **Differential Reflectivity - processed:** `?? dB`
- `DPHI` - **Differential Phase - processed:** `?? deg`
- `DRHO` - **Correlation Coefficient - processed:** `??` (Dimensionless)
- `DSNR` - **Signal-to-Noise Radio:** `?? dB`
- `DSMZ` - **Smoothed Reflectivity:** `?? dBZ`
- `DSMV` - **Smoothed Velocity:** `?? m/s`
- `DKDP` - **Specific Differential Phase:** `?? deg/km`
- `DSDZ` - **Texture (standard deviation) for Reflectivity:** `?? dBZ`
- `DSDP` - **Texture (standard deviation) for Differential Phase:** `?? deg`

Encoding / Decoding Data

The algorithm API includes convenience functions to assist in encoding and decoding radar moment data (reflectivity, radial velocity, and spectrum width) and decoding advanced data fields (Dual Pol data). See CODE Guide Vol 3, Doc 2, Section IV.

The `scale` / `offset` formula

Beginning with Build 10, a new method has been published for specifying the linear encoding and decoding of real data in unsigned integers contained in basedata messages.

These formulas are NOT applied to any value of the scaled integer which represents a flag value (e.g.; "range folded"). They are only meaningful for the encoding and decoding of numerical data values. **Currently all basic moment data and Dual Polarization data fields reserve data levels 0 and 1 for flag values. In other words data levels 0 and 1 are not used for encoded numerical values.**

The maximum range of values in the encoded integer is limited by the type (unsigned char, unsigned short, or unsigned int) minus the integer values used for flag values.

```
encoded_integer = (float_value * SCALE) + OFFSET
```

```
float_value = (encoded_integer - OFFSET) / SCALE
```

Basic Moment Data are encoded in the least significant byte of a 16-bit integer in base data radial messages and encoded into individual bytes in a base data elevation message. Value `00` is a flag for "Data Below Threshold". Value `01` is a flag for "Signal Overlaid" (obscured by range folding).

Most Dual Pol Data Fields are encoded into an 8-bit integer array. PHI, RHO (processed), and KDP use a 16-bit integer array. The values `00` and `01` are not used for encoding data levels.

Data Encoding

The non-flag data values are encoded using the standard formula:

```
encoded_integer = (float_value * SCALE) + OFFSET
```

Basic Moments:

- **Encoded Reflectivity:**
  8-bit `i = (f * 2.0) + 66.0`

- **Encoded Radial Velocity:**
  8-bit `i = (f * 2.0) + 129.0` (Doppler resolution 1)
  8-bit `i = (f * 1.0) + 129.0` (Doppler resolution 2)

- **Encoded Spectrum Width**:
  8-bit `i = (f * 2.0) + 129.0`

Dual Polarization Measured Data Fields:

- **Encoded `DZDR` Differential Reflectivity:**
  8-bit `i = (f * 16.0) + 128.0`

- **Encoded `DPHI` Differential Phase:**
  10-bit `i = (f * 2.8361) + 2.0`

- **Encoded `DRHO` Correlation Coefficient:**
  8-bit `i = (f * 300.0) - 60.0`

Dual Polarization Derived Data Fields:

The Scale and Offset parameters for the derived data fields `DRHO`, `DPHI`, `DKDP`, and `DSDZ` provide an excessive number of significant digits for the decoded values and a precision of representation that significantly exceeds the specification of the preprocessing algorithm and the accuracy of the data (though the accuracy of these fields has not yet been determined). The Scale and Offset

parameters for the other derived data fields provide a precision of representation reflected in the specification of the preprocessing algorithm.

- **Encoded DZDR Differential Reflectivity - processed:**
  8-bit `i = (f * 16.0) + 128.0`

- **Encoded DPHI Differential Phase - processed:**
  16-bit `i = (f * 60.681480) + 42.792198` Precision exceeds accuracy of the data

- **Encoded DRHO Correlation Coefficient - processed:**
  16-bit `i = (f * 76800.00) - 15360.00` Precision exceeds accuracy of the data

- **Encoded DSNR Signal-to-Noise Radio:**
  8-bit `i = (f * 2.0) + 26.0`

- **Encoded DSMZ Processed Reflectivity:**
  8-bit `i = (f * 2.0) + 66.0`

- **Encoded DSMV Smoothed Velocity:**
  8-bit `i = (f * 2.0) + 129.0` (Doppler resolution 1)
  8-bit `i = (f * 1.0) + 129.0` (Doppler resolution 2)

- **Encoded DKDP Specific Differential Phase:**
  16-bit `i = (f * 5120.0) + 11008.0` Precision exceeds accuracy of the data

- **Encoded DSDZ Texture (standard deviation) for Reflectivity:**
  8-bit `i = (f * 8.330) + 2.000` Precision exceeds accuracy of the data

- **Encoded DSDP Texture (standard deviation for Differential Phase:**
  8-bit `i = (f * 2.50) + 2.00`

Data Decoding

The non-flag data values are decoded using the standard formula:

```
float_value = (encoded_integer - OFFSET) / SCALE
```

Basic Moments:

- **Reflectivity (dBZ):**
  8-bit `f = (i - 66.0) / 2.0`

- **Radial Velocity (m sec^{-1}):**

8-bit `f = (i - 129.0) / 2.0` (Doppler resolution 1)
8-bit `f = (i - 129.0) / 1.0` (Doppler resolution 2)

- **Spectrum Width (m sec^{-1}):**
  8-bit `f = (i - 129.0) / 2.0`

Dual Polarization Measured Data Fields:

- **Decoded DZDR Differential Reflectivity (dB):**
  8-bit `f = (i - 128.0) / 16.0`

- **Decoded DPHI Differential Phase (deg):**
  10-bit `f = (i - 2.0) / 2.8361`

- **Decoded DRHO Correlation Coefficient:**
  8-bit `f = (i + 60.0) / 300.0`

Dual Polarization Derived Data Fields:

The Scale and Offset parameters for the derived data fields DRHO, DPHI, DKDP, and DSDZ appear to provide an excessive number of significant digits for the decoded values and a precision of representation that significantly exceeds the specification of the preprocessing algorithm and perhaps the accuracy of the data (though the accuracy of these fields has not yet been published). The Scale and Offset parameters for the other derived data fields provide a precision of representation reflected in the specification of the preprocessing algorithm.

- **Decoded DZDR Differential Reflectivity - processed (dB):**
  8-bit `f = (i - 128.0) / 16.0`

- **Decoded DPHI Differential Phase - processed (deg):**
  16-bit `f = (i - 42.792198) / 60.681480` Precision exceeds accuracy of the data

- **Decoded DRHO Correlation Coefficient - processed:**
  16-bit `f = (i + 15360.00) / 76800.00` Precision exceeds accuracy of the data

- **Decoded DSNR Signal-to-Noise Radio:**
  8-bit `f = (i - 26.0) / 2.0`

- **Decoded DSMZ Processed Reflectivity:**
  8-bit `f = (i - 66.0) / 2.0`

- **Decoded DSMV Smoothed Velocity:**
  8-bit `f = (i - 129.0) / 2.0` (Doppler resolution 1)
  8-bit `f = (i - 129.0) / 1.0` (Doppler resolution 2)

- **Decoded DKDP Specific Differential Phase:**
  16-bit `f = (i - 11008.0) / 5120.0` Precision exceeds accuracy of the data

- **Decoded DSDZ Texture (standard deviation) for Reflectivity:**
  8-bit `f = (i - 2.000) / 8.330` Precision exceeds accuracy of the data

- **Decoded DSDP Texture (standard deviation for Differential Phase :**
  8-bit `f = (i - 2.50) / 2.00`

Vol 2. Document 4 - Additional Information & Guidance for WSR-88D Algorithm Developers

Section II Algorithm Adaptation Data - Configuration & Use

Part A. Introduction

The WSR-88D uses adaptation data to configure many aspects of the radar system. A portion of this configuration data is used to alter or customize the contents of WSR-88D products. This section contains an overview of *algorithm specific* adaptation data and procedures for proper configuration. Site specific adaptation data, covered in Section IV of CODE Guide Vol 2 - Document 2, contains only a few items of information that should be changed to correspond to the site that produced the base data being used as input to the algorithms.

What is Algorithm-Specific Adaptation Data?

It is not sufficient for data to be parameters that determine the functionality of an algorithm in order to be classified as adaptation data. The common include file is often used to define constants, some of which can be a parameterization of logic. Algorithm adaptation data must meet the following criteria.

1. The data must represent a parameterization of algorithm logic / performance or a parameterization of product format.
2. The data must be intended to be operationally field modifiable. This can be either data that can be changed at each site or data that may be unique to a site but require ROC change authority.
3. The data must be fully documented with instructions on what each parameter is used for and what effects changing the parameters will have on algorithm function / performance.

Previous Adaptation Method

The previous method involved the definition of the adaptation data structure in an include file. Meta data (in the form of specially structured comments) in that include file were used to provide attributes such as: name, description, units, minimum value, maximum value, list of valid values, default value, precision, enumeration, etc. Most adaptation data include files still contain remnants of the previous method. With the new DEA mechanism, the meta comments serve only as inline documentation.

Example Include File The following is the contents of the include file for the VIL Echo Tops algorithm.

```
/*
 * RCS info
 * $Author: ccalvert $
 * $Locker:  $
 * $Date: 2007/01/30 22:57:09 $
 * $Id: vil_echo_tops.h,v 1.9 2007/01/30 22:57:09 ccalvert Exp $
 * $Revision: 1.9 $
 * $State: Exp $
 */
/*      This header file defines the structure for the VIL-Echo Tops *
 *      algorithm.  It corresponds to common block VIL_ECHO_TOPS in the      *
 *      legacy code.                                                         */


#ifndef VIL_ECHO_TOPS_H
#define VIL_ECHO_TOPS_H

#include <orpgctype.h>

#define VIL_ECHO_TOPS_DEA_NAME "alg.vil_echo_tops"

/*      VIL/Echo Tops Data     */

typedef struct {

        freal   beam_width;     /*# @name "Beam Width  [BW]"
                                   @desc Angular width of the radar beam
                                         between the half-power points.
                                    @units "degrees" @min 0.5 @max 2.0
                                     @default 0.5  @precision 2
                                     @legacy_name EBMWT
                                */
        freal   min_refl;       /*# @name "Min Ref Threshold  [MRT]"
                                     @desc Minimum reflectivity used in
computing
                                           vertically integrated liquid value.
                                    @units dBZ @min -33 @max 95 @default 0
                                     @precision 2 @legacy_name ENREF
                                */
        fint    max_vil;        /*#  @name "Max VIL Threshold  [MVT]"
                                     @desc Maximum allowable VIL product value.
                                           All computed VIL values above this
                                           threshold will be set to this
                                           threshold for product display.
                                     @units "kg/m**2" @min 1 @max 200 @default
1
                                 */
} vil_echo_tops_t;

#endif
```

Defining adaptation data in a single C structure is convenient but not actually required unless registering the adaptation data reading function as a callback as explained in Part D.

DEA Adaptation Data

The Data Element Attribute (DEA) Method involves the definition of the adaptation data elements in a special meta file called a DEA file. This file also includes initial data values and meta information similar to the previous method including: name, description, value, type, range of values or list of values, accuracy, enumeration and change permission. These data are automatically installed into an internal adaptation data database during an ORPG start.

Reading DEA Adaptation Data within an Algorithm

Within an algorithm, variables must be defined for each data element of the adaptation data. A C structure is a convenient method of aggregating these variables (with Fortran a common block is used). Special functions to read each adaptation data element is provided as part of the algorithm API. These API calls read the current element value from the database which must be assigned to the defined local variable for use within an algorithm. Typically a 'DEA access function' is written to read all applicable data elements using these API calls and is made part of the algorithm. The algorithm updates the local variables by calling the access function as required (beginning of volume, when processing begins, etc.).

Though the 'DEA access function' can be executed by the algorithm to read and update the local data variables, another API call can be used to register this function as a callback. The ORPG infrastructure will then use the 'access function' to update the data element variables at the designated frequency.

The source location of the dea file is the **`~/src/cpc104/lib006`** directory. The run-time location is in the **`~/cfg/dea`** directory. For development activities the current file must be in the run-time location (**`~/cfg/dea`**). When delivering an algorithm to the ROC for integration it must be in the source location.

Summary of Steps

New Algorithm Development

- Determine algorithm parameters to include in adaptation data.
- Create a dea file in the **`~/cfg/dea`** directory.
- Write a 'DEA access function' (as part of the algorithm) using API calls to read the data elements.
- Optionally, register the 'access function' as a callback at the beginning of the algorithm. This provides a slight resource savings since the data are only read from the database if updated (modified).
- Restart the ORPG to install the data.

Modification of an Existing Algorithm

- Review existing algorithm parameters to include in adaptation data.
- If modifying existing adaptation data
    - Locate and modify the dea file in the **`~/cfg/dea`** directory.
    - Modify the existing 'access function' which reads the data elements using API calls. This function could either be
        - part of the algorithm (and optionally registered as a callback function), or

        - part of the adaptation data library (and optionally registered as a callback function)**
- If creating new adaptation data
    - Create a dea file in the **`~/cfg/dea`** directory.
    - Write a 'DEA access function' (as part of the algorithm) using API calls to read the data elements.
    - Optionally, register the access function as a callback at the beginning of the algorithm. This provides a slight resource savings since the data are only read from the database if updated (modified).
- Restart the ORPG to install the data.

** Modification of the adaptation library 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.

Part B. DEA File Description

File Name

The name of the algorithm DEA file must end in '**.alg**'. The first part of the filename is chosen to be descriptive of the algorithm.

File Format

A data element attribute file is an ASCII file that contains attribute values of a set of data elements. Each line in the file specifies one or more attributes for one data element. If the first non-spacing character is "**#**", the line is treated as a comment line. A comment line is not a line of specification and is, thus, ignored.

Leading and trailing "space" and "tab" characters in each line are discarded. If a line is continued by using the "****" character immediately prior to the line feed, then the leading white space on the continued line is ignored as well.

The following are special formatting tokens: equals sign "**=**", colon "**:**", semi-colon "**;**", comma "**,**", backslash "****", brackets "**[]**", braces "**{}**", parentheses "**()**", and "**@**". To avoid ambiguity, if any of these symbols are used for another purpose, it should be quoted with a "****", for example "**\@**" or "****". White space is also insignificant on either side of the certain formatting tokens: "**=**", "**:**", "**;**", "**,**", "**{**", "**}**", "**[**", "**]**". For example, **name= short**, **name =short** and **name=short** are identical.

Data Elements

A data element is a data object that has a defined physical meaning and can be described by a set of attributes. A data element can be one of the primitive types such as integer, floating point number or character string. A data element can have a value or an array of values. A data element can have a set of defined attributes such as **name**, **type**, **value**, **unit** and so on.

Element Identifier

The first token in a line is the data element identifier if it is NOT one of the attribute names followed by "**=**".

In algorithm adaptation data DEA files, the data element identifier is usually a simple identifier name. However, the data element identifier can contain multiple fields separated by "**.**". One example is "**alg.precip_detect.max_elev**". This is the algorithm adaptation data called "**max_elev**" for the for the adaptation data name "**precip_detect**". This is the inverse of the DEA filename **precip_detect.alg**. For algorithm DEA files, the **alg.precip_detect** prefix is not used.

Element Attributes

Following the data element identifier is a number of sections terminated by "**;**". Each section specifies an applicable attribute for the data element. Each section must be in the form of "**attribute_name =**

attribute_description;" where **attribute_name** must a member of a specified list which includes: "**name**", "**type**", "**unit**", "**range**", "**value**", "**default**", "**description**", "**permission**", "**enum**", etc.

- **name**: The name of the data element. An example is "**name = Radar location - latitude;**". The name will appear in the hci editor application.
- **type**: One of the following type names:
  - "**int**", "**short**", "**byte**" (4-byte, 2-byte and 1-byte integer respectively),
  - "**bit**" (1-bit data), "**float**", "**double**" (4-byte and 8-byte IEEE floating point numbers respectively),
  - "**string**" (ASCII character string),
  - "**uint**", "**ushort**" and "**ubyte**" (unsigned versions of int, short and byte).

  An example is "**type = int;**". If type is not specified, "**int**" is assumed.

- unit: The physical unit of the data value. Standard unit names are to be defined. Examples are "**unit = meter;**" and "**unit = percent**".
- **range**: The set of all valid values for the data element. The range can be specified with one of the following three formats:
  - Single interval specification defined by "**[min, max]**" where "min" and "max" are respectively the minimum and maximum values. Examples are: "**range = [1, 2];**" and "**range = [A, Z];**" (character string type). Non-inclusive boundaries using "**(**" and "**)**" are not currently used.
  - A list of valid values: **{ v1, v2, ...}**. Examples are "**range = {1, 2, 3};**" and "**range = {reflectivity, velocity, spectrum width};.**
  - A named method that checks the range. A description of this method is not included with this document.
- **accuracy**: The accuracy of the data. An example for floating point data is "**accuracy = [0.1];**"
- **value**: This attribute can consist of a single value or a list of values (an array). Examples are "**value = 1", "value = 1.0, 2., 3.0;**" and "**value = Yes, No;**". The specifics concerning how arrays are specified and read by an algorithm will be covered in a future version of this document.
- **description**: A text description of the data.
- **enum**: A list of integers as an alternate representation of a set character string values. The **enum** specification must match the **type** and **range** specifications. The **type** must be "**string**". The **range** must be a set of values and the number of the valid values must be the same as the number of integers in **enum** specification. The items in the **enum** specification must be integers. An example is "**enum = 0, 1;**", "**type = string;**" and "**range = {No, Yes};**". In this case, the **enum** values of "**No**" and "**yes**" are respectively 0 and 1.
- **permission**: A list of permission group names. An example is "**permission = [ROC, URC];**" **A permission list including USR means the user (at the operational site) is permitted to edit the values via the appropriate password. A permission list including ROC means that the ROC has authority to set the values. Recent guidance states that if the data is editable by the user in the field, the permission list should only contain the following: "permission = [URC];"** If the **permission** attribute is not present, the data will not appear in the editor.
- **default**: The default value for the data element. If default values are specified, the value attribute is left blank, for example "**value = ;**". If a default value is site dependent, a list of site names terminated with "**:**" proceeds each value ("**:**" is treated as a formatting character in default

specification). A site name is a single token string containing the ICAO 4 letter identifier. The site names in the list are separated by space. The reserved site name of "**Other_sites**" can be used for all other sites that are not specified. "**Other_sites**" must be used after any other site names. Example:
"**default = KTLX KCRI: .9, Other_sites: 1.2;**"

Additional attributes will be described in future versions of this document.

Sample Algorithm DEA File

The following is the contents of **vil_echo_tops.alg**

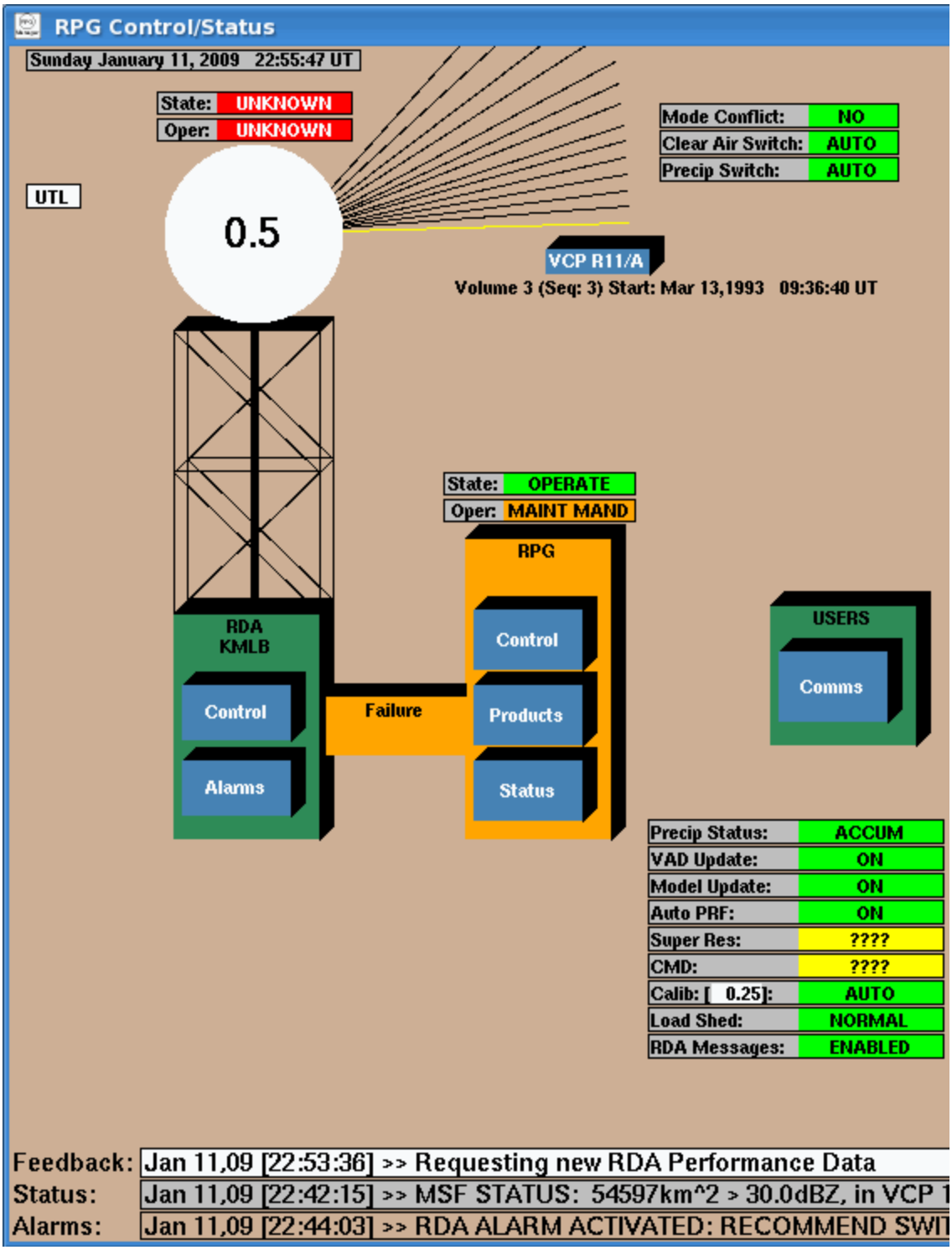
```
# RCS info
# $Author: ryans $
# $Locker:  $
# $Date: 2005/12/06 21:31:39 $
# $Id: vil_echo_tops.alg,v 1.8 2005/12/06 21:31:39 ryans Exp $
# $Revision: 1.8 $
# $State: Exp $

    alg_name            value = Vil/Echo Tops

    beam_width          value = 1.00;
                        name = Beam Width  [BW];
                        type = double;
                        range = [0.50, 2.00];
                        accuracy = [0.01];
                        unit = degrees;
                        description = Angular width of the radar beam between
\
                                      the half-power points.;

    min_refl            value = 18.3;
                        name = Min Ref Threshold  [MRT];
                        type = double;
                        range = [-33.0, 95.0];
                        accuracy = [0.1];
                        unit = dBZ;
                        description = Minimum reflectivity used in computing \
                                      vertically;

    max_vil             value = ;
                        name = Max VIL Threshold  [MVT];
                        type = int;
                        range = [1, 200];
                        unit = kg/m^2;
                        description = Maximum allowable VIL product value.  \
                                      All computed VIL values above this;
                        default = KDDC KILX KSGF: 100, KCLX KINX KRLX KSRX:
120,
                                  KICT: 200, Other_sites: 80;
```

Part C. Changing Algorithm Adaptation Parameters

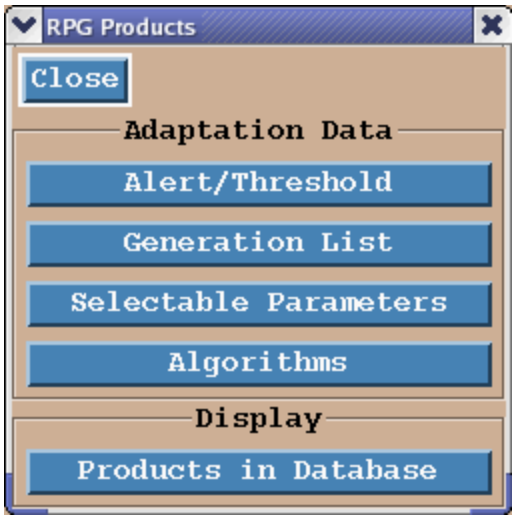
There are two methods for changing adaptation data parameters. One involves modifying the value attribute in the corresponding algorithm DEA file (**`.alg`**) and accomplishing a clean start (i.e. **`mrpg -r startup`**). The other method is modifying the parameters while the ORPG is running using the **`hci`**.

Editing Adaptation Data Parameters at the HCI

To modify the parameters, first launch the control interface by executing **`hci`** on the command line. The ORPG must be running for the **`HCI`** to launch. The main window looks like:

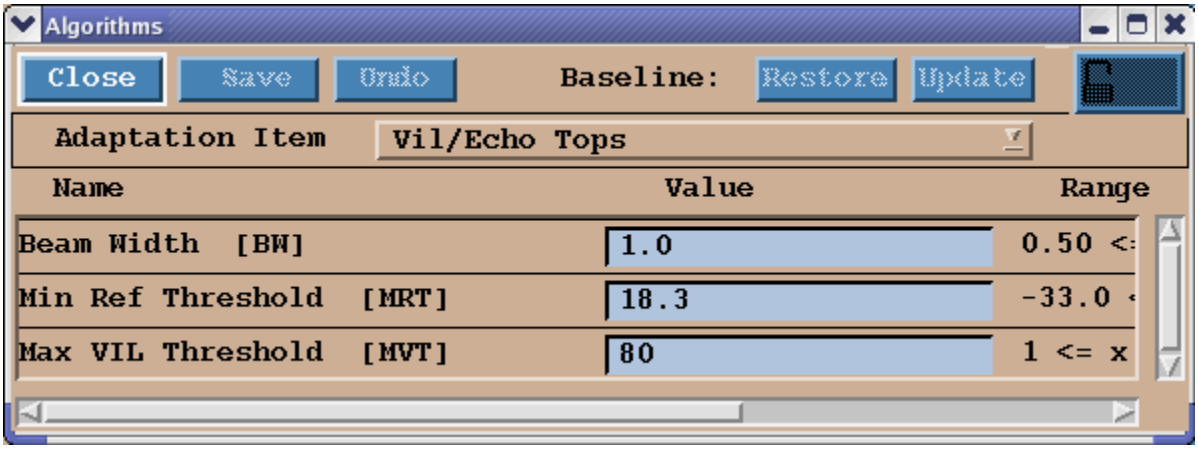


Click on the Products button inside the RPG component box in the center of the screen. The products dialog window is displayed.



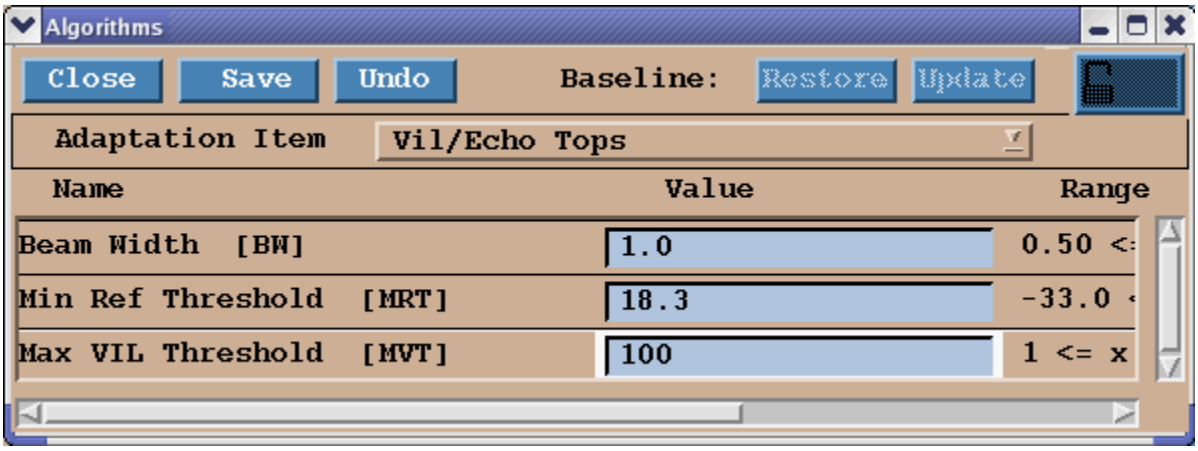
To edit the algorithm specific adaptation data, click on the Algorithms button in the products dialog window.

The Algorithms dialog window is displayed.



Use the Adaptation Item pop-up menu to select which block (or named object) to be displayed.

With the current CODE distribution, the Algorithms dialog window is unlocked at all times. No password is required.



After making desired changes, the changes can be applied by clicking on Save and confirming the change. NOTE: the Save button will be grayed out until at least one change has been made and either the **Return/Enter** key pressed or the cursor moved to another data field.

Depending on the algorithm configuration, the new parameters will be applied immediately, or at the beginning of the next elevation or volume.

Part D. Algorithm API Support

This is a very brief overview of how adaptation data is used by algorithms. See CODE Guide Volume 3 for additional information.

Reading Adaptation Data from the Database

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.

- **`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.

Typically an adaptation data "access function" using these calls is written as part of the algorithm.

Using the DEA Callback Feature

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 adaptation data "DEA access function" so that it is automatically called at a specified interval. This provides a slight resource savings since the data are only read from the database if updated (modified).

Sharing Adaptation Data

Algorithm adaptation data is stored in an ORPG database and is available to any algorithm knowing the full ID of the data element (including the algorithm group name). 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 data by making the access function easily available to all algorithms.

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.

Vol 2. Document 4 - Additional Information & Guidance for WSR-88D Algorithm Developers

Section III Other Data Inputs

Part A. Introduction

In addition to base data from the radar and algorithm specific adaptation data, algorithms can use intermediate product data produced by other algorithms, external data obtained from other systems, and miscellaneous configuration / status data.

Part B. Intermediate Product Data

Intermediate product data is product data related to the radar base data flow, is typically produced every elevation or every volume (configured as elevation data or volume data), and is not distributed to external users. Intermediate products are used to split up algorithm logic into more that one task and to provide a source of intermediate data that can be used by multiple algorithm tasks.

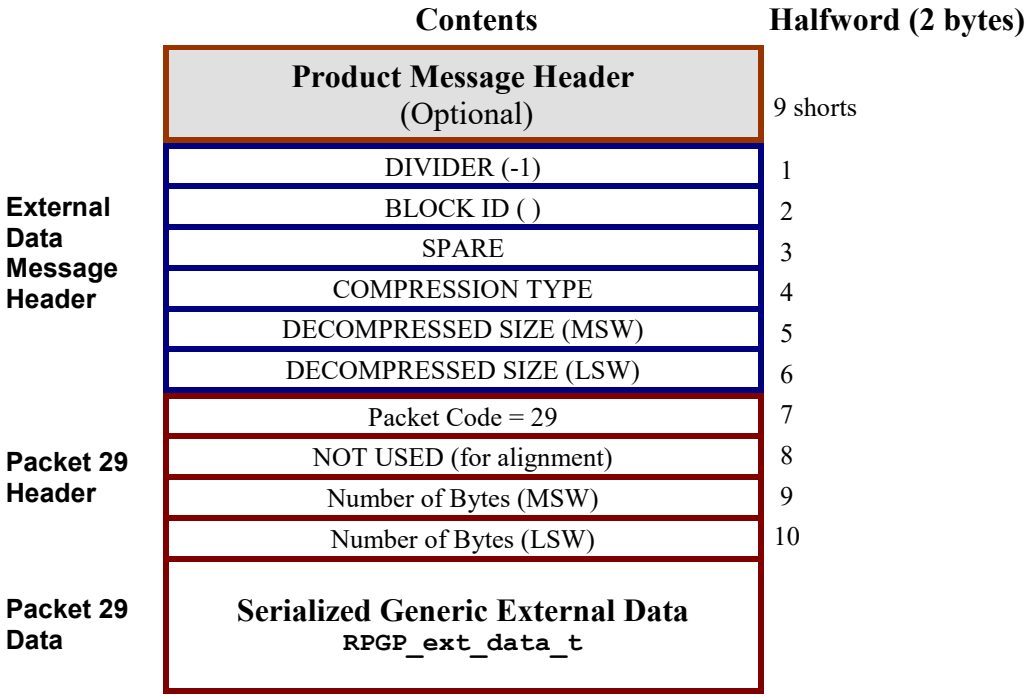
One limitation, except for the requested elevation, only the final product can be customized by the 6 request parameters in the product request message. These parameters are not passed on to tasks upstream producing intermediate products.

No formal guidance exists for the content / structure of intermediate product data. A review of existing intermediate products may turn up similarities or patterns of use, but this review has not been accomplished. For example, some final products that are in rectangular coordinates, "raster products", have the algorithm science accomplished while the data is still in polar coordinates with the results stored in intermediate products.

Part C. External Data

External data is non-product data obtained from external sources. Currently there is only one example of external data in the ORPG. That is environmental data from the RUC model that is received from AWIPS.

The external data message has a defined structure. It contains an optional message header, an external data structure and data packet 29 which includes serialized external data. The header structure of data packet 29 is similar to data packet 28 used in final products.



The structure **`External_data_msg_hdr`** defined in **`product.h`** can be used to access the external data message header. The first field in **`External_data_msg_hdr`** is a **`short divider`** = -1 which can be used to detect the absence of the optional product message header (which is not needed). The structure **`packet_29_t`** defined in **`packet_29.h`** can be used to access the packet 29 header.

As with WSR-88D final products, the external data header and packet 29 header portions of the external data are transmitted in Big Endian (network) format and short data values must be swapped before reading them on a Little Endian platform.

The serialized data must be deserialized using standard API functions and the resulting address cast to **`RPGP_ext_data_t *`**. The top level structure of the generic data is represented by the C structure **`RPGP_ext_data_t`**. This structure contains approximately 15 header fields, a pointer to external data parameters, and a pointer to external data components. The deserialized data does not need byte-swapping.

```
typedef struct {        /* product struct */
```

```
    char *name;              /* product name */
    char *description;       /* product description (may contain version
                                info) */
    int product_id;          /* product id (code) */
    int type;                /* product type (must be RPGP_EXTERNAL) */
    unsigned int gen_time;   /* product generation time */

    short spare[5];   /* fields reserved for future use (must set to 0) */

    short compress_type;   /* compression type (currently not used and
                              must set to 0) */
    int size_decompressed; /* size after decompressing (currently not
                              used and must set to 0) */

    int numof_prod_params;     /* number of specific product parameters */
    RPGP_parameter_t *prod_params; /* specific product parameter list */

    int numof_components;       /* number of components or events */
    void **components;   /* component or event list. See Note 0. */

} RPGP_ext_data_t;
```

Each external data parameter and component parameter is represented by the C structure **RPGP_parameter_t**. A description of generic parameters is beyond the scope of this introduction.

An introduction to the generic components is provided in Volume 2 Document 3 Section III - *Generic Product Components*.

The algorithm API contains some debug print functions that can be used to provide a text output of the contents of the product. The algorithm API also contains helper functions to fill out the product header fields, fill the contents of the parameter structures, and to serialize / deserialize the product. No support is provided for construction of the components themselves.

Part D. Miscellaneous Configuration / Status Data

There are several internal data tables containing information about the current volume scan / radar base data being ingested. Much of the information is available via specific API functions and the base data header. Therefore most algorithms have no need to access these tables. Only those tables that can be accessed via the algorithm API are covered here. For API support see CODE Guide Volume 3, Document 2, Section II.

Scan Summary Information

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, and the spot blanking bitmap. The fields for the last elevation number were added in Build 11 and are the preferred over the contents of the VCP information structure. The **`Scan_summary`** structure is **`defined in orpgsum.h`**.

```
typedef struct {

   int volume_start_date;   /* Modified Julian start date */

   int volume_start_time;   /* From midnight, in seconds */

   int weather_mode;        /* 2 = convective, 1 = clear air */

   int vcp_number;          /* Pattern number:
                               Maintenance/Test: > 255
                               Operational: <= 255
                               Constant Elevation Types: 1 - 99 */

   short rpg_elev_cuts;     /* Number of RPG elevation cuts in VCP */

   short rda_elev_cuts;     /* Number of RDA elevation cuts in VCP */

   int spot_blank_status;   /* Bitmap indicating whether elevation
                               cut has spot blanking enabled */

   unsigned char super_res[ECUT_UNIQ_MAX];
                         /* Bitmap indicating whether elevation
                            cut is Super Resolution.   Each byte
                            corresponds to an RPG elevation cut.
                            Bits are defined in vcp.h */

   unsigned char last_rda_cut;  /* Cut number of last RDA elevation number in VCP.
                                   This could be different from VCP definition if
                                   AVSET is active.  A value of 0xff indicates
                                   undefined. */

   unsigned char last_rpg_cut;  /* Cut number of last RPG elevation number in VCP.
                                   This could be different from VCP definition if
                                   AVSET is active.  A value of 0xff indicates
                                   undefined. */
} Scan_Summary;
```

The field `last_rpg_cut` should be used instead of `rpg_elev_cuts` because in the future RDA the processing of elevations may be terminated before reaching the last elevation defined in a VCP. The function `RPGC_is_buffer_from_last_elev` (which uses the new field `last_rpg_cut`) should be used to determine the current elevation is the last elevation in the volume scan.

Volume Status Message

The volume status message (obtained by `RPGC_read_volume_status`) is the primary method of obtaining the volume number and the volume time if not registered for input product data. Some 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. The `Vol_stat_gsm_t` structure is defined in `gen_stat_msg.h`.

```
typedef struct volume_status {

   unsigned long volume_number;    /* Current volume scan sequence  number.
                                      Monotonically increases. Initial value 0. */

   unsigned long cv_time;          /* Current volume scan time in milliseconds
                                      past midnight. */

   int cv_julian_date;             /* Current volume scan Julian date */

   int initial_vol;                /* Flag, if set, indicates the volume is the
                                      initial volume.  It is assumed for an
                                      initial volume, no radar data-derived
                                      products will be available. */

   int pv_status;                  /* Previous volume scan status:
                                      1 - completed successfully,  0 - aborted. */

   int expected_vol_dur;           /* Expected volume scan duration, in seconds. */

   int volume_scan;                /* The volume scan number [0, 80]. */

   int mode_operation;             /* Mode of operation: 0 - Maintenance Mode
                                      1 - Clear Air Mode  2 - Precipitation Mode */

   int vol_cov_patt;               /* Volume coverage pattern */

   int rpgvcpid;                   /* slot in vcp_table containing VCP data
                                      associated with vol_cov_patt. */

   int num_elev_cuts;              /* Number of elevations in VCP. */

   short elevations[MAX_CUTS];     /* Elevation angles (deg*10). */

   short elev_index[MAX_CUTS];     /* RPG elev index associated with each
                                      elevation angle. */

   int super_res_cuts;             /* Bit map indicating which RPG cuts are
                                      expected to have super res data. */

   Vcp_struct current_vcp_table;   /* The current VCP data. */

} Vol_stat_gsm_t;
```

VCP Information

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. The **Vcp_struct** structure is defined in **vcp.h**.

```
typedef struct {

    short msg_size;     /* number of half words; 23 - 594 depending
                           on type; PFNHW=1 */
    short type;         /* pattern type: PFPATTYP=2
                           Constant elevation cut: 2
                           Horizontal raster scan: 4
                           Vertical raster scan:   8
                           Searchlight:        16 */
    short vcp_num;      /* pattern number: PFPATNUM=3
                           Maintenance/Test: > 255
                           Operational: <= 255
                           Constant Elevation types:   1 -  99
                           Horizontal Raster types:  100 - 149
                           Vertical Raster types:    150 - 199
                           Searchlight types:        200 - 249  */

    /* the following fields are good for Constant elevation cut type */

    short n_ele;        /* number of elevations are scanned in this
                           volume (including repeated elev); 1-25. */

    short clutter_map_num;  /* clutter map group number; 1 - 99. */

    unsigned char vel_resolution; /* velocity resolution;
                                     0.5 meters/second: 2    1.0 meters/second: 4 */

    unsigned char pulse_width;  /* pulse width;   short: 2  long: 4  */

/* Note the following items are actually spare fields. */
    short sample_resolution;    /* sampling range resolution;
                                   250 meters: 0  50 meters:  1  */
    short spare1;
    short spare2;
    short spare3;
    short spare4;

/* Because Ele_attr has a size of 23 shorts, which is not aligned, we use the
   following array to reserve the space for Ele_attr.  When we use this we cast
   to the struct ele_attr = (Ele_attr *)(vcp.vcp_ele[ele_num])       */

    short vcp_ele [VCP_MAXN_CUTS][ELE_ATTR_SIZE]; /* specify the cuts */

} Vcp_struct;
```

Volume 2. ORPG Application Software Development Guide

Appendices

Appendix A. Site Data Listing

	ICAO	Latitude	Longitude	Height	RPG ID		Name
CONUS Operational WSR-88D							
	kabr	45456	-98413	1383	309		ABERDEEN
	kabx	35150	-106824	5951	311		ALBUQUERQUE
	kakq	36984	-77008	209	516		NORFOLK
	kama	35233	-101709	3703	313		AMARILLO
	kamx	25611	-80413	111	728		MIAMI
	kapx	44906	-84720	1561	312		NCL MICHIGAN
	karx	43823	-91191	1357	389		LA CROSSE
	katx	48195	-122494	607	542		SEATTLE
	kbbx	39496	-121632	221	380		BEALE AFB
	kbgm	42201	-75985	1703	319		BINGHAMTON
	kbhx	40499	-124291	2516	359		EUREKA (BUNKER HILL)
	kbis	46711	-100760	1755	321		BISMARCK
	kblx	45854	-108607	3703	318		BILLINGS
	kbmx	33172	-86770	759	320		BIRMINGHAM
	kbox	41956	-71138	231	323		BOSTON
	kbro	25916	-97419	87	324		BROWNSVILLE
	kbuf	42949	-78737	790	325		BUFFALO
	kbyx	24597	-81703	89	386		KEY WEST
	kcae	33949	-81119	344	341		COLUMBIA
	kcbw	46039	-67807	859	329		CARIBOU
	kcbx	43490	-116236	3171	322		BOISE
	kccx	40923	-78004	2486	374		STATE COLLEGE
	kcle	41413	-81860	860	340		CLEVELAND
	kclx	32655	-81042	228	333		CHARLESTON SC
	kcrp	27784	-97511	142	343		CORPUS CHRISTI
	kcxx	44511	-73166	431	326		BURLINGTON
	kcys	41152	-104806	6192	335		CHEYENNE
	kdax	38501	-121677	144	536		SACRAMENTO
	kddc	37761	-99969	2671	350		DODGE CITY
	kdfx	29273	-100280	1196	394		LAUGHLIN AFB
	kdgx	32280	-89984	609	855		BRANDON
	kdix	39947	-74411	230	523		PHILADELPHIA
	kdlh	46837	-92210	1542	352		DULUTH
	kdmx	41731	-93723	1058	348		DES MOINES
	kdox	38826	-75440	163	351		DOVER AFB

	kdtx	42700	-83472	1216	349		DETROIT
	kdvn	41612	-90581	851	530		QUAD CITIES
	kdyx	32538	-99254	1581	353		DYESS AFB
	keax	38810	-94264	1098	385		PLEASANT HILL
	kemx	31894	-110630	5319	556		TUCSON
	kenx	42586	-74064	1907	310		ALBANY
	keox	31460	-85459	537	362		FT RUCKER
	kepz	31873	-106698	4218	357		EL PASO
	kesx	35701	-114891	4948	566		LAS VEGAS
	kevx	30565	-85922	222	307		EGLIN AFB
	kewx	29704	-98029	766	539		AUSTIN/SAN ANTONIO
	keyx	35098	-117561	2873	511		EDWARDS AFB
	kfcx	37024	-80274	2965	534		ROANOKE
	kfdr	34362	-98977	1311	305		ALTUS AFB
	kfdx	34634	-103619	4698	328		CANNON AFB
	kffc	33363	-84566	972	316		ATLANTA
	kfsd	43588	-96729	1495	544		SIOUX FALLS
	kfsx	34574	-111197	7514	361		FLAGSTAFF (RPG)
	kftg	39786	-104546	5610	347		DENVER
	kfws	32573	-97303	764	345		DALLAS/FT WORTH
	kggw	48206	-106625	2384	365		GLASGOW
	kgjx	39062	-108214	10100	368		GRAND JUNCTION (RPG)
	kgld	39367	-101700	3717	366		GOODLAND
	kgrb	44499	-88111	806	371		GREEN BAY
	kgrk	30722	-97383	602	332		FT HOOD
	kgrr	42894	-85545	875	369		GRAND RAPIDS
	kgsp	34883	-82220	1068	555		GREER
	kgwx	33897	-88329	589	342		COLUMBUS AFB
	kgyx	43891	-70257	473	528		PORTLAND ME
	khdx	33077	-106120	4270	376		HOLLOMAN AFB
	khgx	29472	-95079	115	378		HOUSTON
	khnx	36314	-119631	340	363		SAN JOAQUIN VALY
	khpx	36737	-87285	624	364		FT CAMPBELL
	khtx	34931	-86084	1859	826		NORTHEAST ALABAMA
	kict	37654	-97443	1403	562		WICHITA
	kicx	37591	-112862	10756	330		CEDAR CITY (RPG)
	kiln	39420	-83822	1170	338		CINCINNATI
	kilx	40150	-89337	730	549		LINCOLN
	kind	39708	-86280	887	381		INDIANAPOLIS
	kinx	36175	-95564	749	557		TULSA
	kiwa	33289	-111670	1426	524		PHOENIX
	kiwx	41359	-85700	1055	827		NORTHERN INDIANA
	kjan	32318	-90080	322	382		JACKSON MS
	kjax	30485	-81702	159	383		JACKSONVILLE
	kjgx	32675	-83351	618	535		ROBINS AFB

	kjkl	37591	-83313	1461	373		JACKSON KY
	klbb	33654	-101814	3340	398		LUBBOCK
	klch	30125	-93216	136	391		LAKE CHARLES
	klix	30337	-89825	179	545		SLIDELL
	klnx	41958	-100576	3067	517		NORTH PLATTE
	klot	41604	-88085	760	337		CHICAGO
	klrx	40740	-116803	6895	564		ELKO (RPG)
	klsx	38699	-90683	721	308		ST LOUIS
	kltx	33989	-78429	145	563		WILMINGTON
	klvx	37975	-85944	833	397		LOUISVILLE
	klwx	38975	-77478	369	303		STERLING
	klzk	34836	-92262	649	395		LITTLE ROCK
	kmaf	31943	-102189	2961	518		MIDLAND/ODESSA
	kmax	42081	-122716	7553	500		MEDFORD (RPG)
	kmbx	48393	-100864	1590	507		MINOT AFB
	kmhx	34776	-76876	144	375		MOREHEAD CITY
	kmkx	42968	-88551	1022	504		MILWAUKEE
	kmlb	28113	-80654	116	302		MELBOURNE
	kmob	30679	-88240	289	509		MOBILE
	kmpx	44849	-93565	1101	506		MINNEAPOLIS
	kmqt	46531	-87548	1525	399		MARQUETTE
	kmrx	36168	-83402	1434	387		KNOXVILLE
	kmsx	47041	-113986	7978	508		MISSOULA (RPG)
	kmtx	41263	-112448	6593	537		SALT LAKE CITY (RPG)
	kmux	37155	-121897	3550	541		SAN FRANCISCO
	kmvx	47528	-97325	1080	360		FARGO/GRAND FORKS
	kmxx	32537	-85790	560	354		MAXWELL AFB
	knkx	32919	-117041	1052	540		SAN DIEGO
	knqa	35345	-89873	435	501		MEMPHIS
	koax	41320	-96366	1260	519		OMAHA
	kohx	36247	-86563	676	512		NASHVILLE
	kokx	40866	-72864	198	515		BROOKHAVEN
	kotx	47681	-117626	2449	547		SPOKANE
	kpah	37068	-88772	505	521		PADUCAH
	kpbz	40531	-80218	1266	526		PITTSBURGH
	kpdt	45691	-118852	1580	522		PENDLETON
	kpoe	31155	-92976	472	339		FT POLK
	kpux	38460	-104181	5363	529		PUEBLO
	krax	35665	-78490	461	531		RALEIGH/DURHAM
	krgx	39754	-119461	8396	533		RENO (RPG)
	kriw	43066	-108477	5633	392		RIVERTON/LANDER
	krlx	38311	-81723	1212	334		CHARLESTON WV
	krtx	45715	-122964	1686	527		PORTLAND OR
	ksfx	43106	-112686	4539	546		POCATELLO
	ksgf	37235	-93400	1375	548		SPRINGFIELD

	kshv	32451	-93841	386	543		SHREVEPORT
	ksjt	31371	-100492	2004	538		SAN ANGELO
	ksox	33818	-117636	3106	574		SANTA ANA MTS
	ksrx	35291	-94362	721	825		WESTERN ARKANSAS
	ktbw	27705	-82402	122	552		TAMPA
	ktfx	47460	-111385	3804	370		GREAT FALLS
	ktlh	30398	-84329	176	551		TALLAHASSEE
	ktlx	35333	-97278	1277	1		NORMAN
	ktwx	38997	-96232	1415	554		TOPEKA
	ktyx	43756	-75680	1960	850		FT DRUM
	kudx	44125	-102830	3194	532		RAPID CITY
	kuex	40321	-98442	2057	367		GRAND ISLAND
	kvax	30890	-83002	330	510		MOODY AFB
	kvbx	34839	-120398	1354	559		VANDENBERG AFB
	kvnx	36741	-98128	1258	558		VANCE AFB
	kvtx	34412	-119179	2807	396		LOS ANGELES
	kyux	32495	-114656	239	393		YUMA (RPG)
Overseas Operational WSR-88D							
	pgua	13456	144811	386	314		ANDERSEN AFB
	tjua	18116	-66078	2958	502		SAN JUAN FAA (RPG 1)
	lpla	38730	-27321	3415	390		LAJES AB
	pabc	60793	-161874	304	304		BETHEL FAA (RPG 1)
	pacg	56853	-135528	272	553		SITKA FAA (RPG 1)
	paec	64512	-165293	90	346		NOME FAA (RPG 1)
	pahg	60726	-151349	356	344		ANCHORAGE FAA (RPG 1)
	paih	59462	-146301	132	505		MIDDLETON ISLAND (RPG 1)
	pakc	58680	-156627	144	568		KING SALMON FAA (RPG 1)
	papd	65036	-147499	2707	525		FAIRBANKS FAA (RPG 1)
	phki	21894	-159552	340	550		SOUTH KAUAI FAA (RPG 1)
	phkm	20125	-155778	3965	377		KAMUELA/KOHALA APT(RPG 1)
	phmo	21133	-157180	1444	336		MOLOKAI FAA (RPG 1)
	phwa	19095	-155569	1461	570		SOUTH SHORE FAA (RPG 1)
	rkjk	35921	126625	191	388		KUNSAN AB
	rksg	36956	127021	133	327		CAMP HUMPHREYS
	rodn	26302	127910	332	384		KADENA AB
Development and Testing Sites							
	kbix	30524	-88985	62	572		KEESLER AFB OPS TRNG
	dkm1	30524	-88985	62	575		KEESLER AFB MNTC TRNG A
	dkm2	30524	-88985	62	573		KEESLER AFB MNTC TRNG B
	ntc1	38810	-94264	1295	571		TRAINING CENTER #1 NWSTC
	ntc2	38810	-94264	1295	571		TRAINING CENTER #2 NWSTC
	nrc3	38810	-94264	1295	0		NRC #3
	nrc1	38810	-94264	1295	315		NRC #1

	nrc2	38810	-94264	1295	578		NRC #2
	nhq1	38975	-77478	1295	0		NWSHQ TESTBED(RPG)
	npc1	38975	-77478	1295	0		PRC (RDASIM/RPG)
	drx1	35238	-97460	1314	520		OPEN SYSTEMS (KREX)
	fop1	35238	-97460	1314	520		OSF REDUNDANT (RPG 1)
	nop3	35238	-97460	1314	520		open rpg 3
	nop4	35238	-97460	1314	520		roc4 nws
	nop2	35238	-97460	1314	520		roc nws
	rop2	35238	-97460	1314	520		roc nws redundant
	dop1	35238	-97460	1314	520		roc dod
	rop3	35238	-97460	1314	520		roc3 nws redundant
	rop4	35238	-97460	1314	520		roc4 nws redundant
	nwsg	35238	-97460	1314	520		nws generic
	dodg	35238	-97460	1314	520		dod generic
	faag	35238	-97460	1314	520		faa generic
	faab	35238	-97460	1314	520		faa generic with bdds
	dodb	35238	-97460	1314	520		dod generic with bdds
Other							
	ncdc	35238	-97460	1295	852		NATIONAL CLIMATIC DATA CENTER
	kvwx	38260	-87724	614	851		EVANSVILLE
	rcwf	25073	121773	2601	301		TAIWAIN

Volume 2. Appendices

Appendix B. Encoding Data into Unsigned Integers

The `Scale-Offset` formula

Beginning with Build 10, a new method was published for specifying the linear encoding and decoding of real data in unsigned integers contained in basedata messages. These formulas are NOT applied to any value of the scaled integer which represents a flag value (e.g.; "range folded"). They are only meaningful for the encoding and decoding of numerical data values.

```
encoded_integer = (float_value * SCALE) + OFFSET

float_value = (encoded_integer - OFFSET) / SCALE
```

The Scale-Offset formulas can be applied for decoding base data. Currently all basic moment data and Dual Polarization data fields reserve data levels 0 and 1 for flag values. In other words data levels 0 and 1 are not used for encoded numerical values. The API provides a function for decoding data.

The `Scale-Offset` formulas can be applied to product data which use unsigned integers and has data encoded in a manner described in this appendix. This would include traditional data packet 16 as well as data elements within the generic radial component. Data packet 16 contains an array of 8-byte integers (unsigned char). Beginning with Build 10, the generic radial component can support 8, 16, and 32 byte integer types. The maximum range of values in the encoded integer is limited by the type (unsigned char, unsigned short, or unsigned int) minus the integer values used for flag values.

These formulas do not include knowledge of the number of leading / trailing flag values. So

- When decoding product data values using the second formula, the user (or decoding system) must only apply the formula to the non-flag integer data values.
- When writing product data values, the product algorithm must set the flag values as needed and ensure that only valid meteorological values are encoded into the integer values contained in the product using the first formula.

Recommended Encoding Method for Unsigned Integer Arrays

Data packet 16 contains an array of 8-bit integers (unsigned char). Beginning with Build 10, the generic radial component can support 8, 16, and 32 bit integer types. Interpreting digital products depends upon a consistent method of encoding data into unsigned integers. Many of the existing products using data packet 16 use an encoding method that is consistent with the encoding of the basic base data moments: reflectivity, velocity, and spectrum width.

Though not required by the *ICD for RPG to Class 1 User*, the recommended method of representing real values in unsigned integer arrays (including data packet 16 and the generic radial component) is as follows:

1. The encoding method for products of the same type should be the same.

2. Whether the numerical real values are negative or positive, they are encoded into the integer values where the real value increases with the integer value. This results in the **`Scale`** parameter having a non-zero positive value.

3. The first (or lowest) integer value used is always 0.

4. The interval between the adjacent encoded numerical values is a constant. That means the encoded real numerical values increase linearly as the integer value increases.

5. Flag values are special purpose data values that are not numerically encoded. If the data contain flag values, they are not intermixed with encoded numerical values.

    a. Leading Flags - (if any) are represented by the beginning integer values (0, 1, ...). Many existing products have one or two leading flags. The first numerical value is first integer value after the last leading flag.

    b. Trailing Flags - (if any) are represented by the integer values immediately after the highest numerical value. Most current products use all available integer data values and have no trailing flag values. A few products use all of the available integer data values (in an 8-bit integer) and have one trailing flag represented with the integer value 255.

Not all products use this method of encoding data into an 8-bit integer. Product ID 134, DVL, uses a linear method for part of the data range and a non-linear method for the remainder. In addition the **`Threshold`** fields are used in a unique manner (see 'Providing Decoding Parameters in the Product' below).

8-bit Example

One example of using the recommended method is product ID 87, DBV. Binary value **0** represents a "Below Threshold" flag and binary value **1** represents "Range Folding". Binary **2** represents the minimum value of -63.5 meters/second. Binary **255** represents the maximum value of 63 meters/second. The real increment between numerical values is 0.5 meters/second.

Providing Decoding Parameters in the Product

The Product Description Block in the WSR-88D final product contains 16 Threshold Level data fields. These threshold fields were originally intended to explicitly define the threshold labels to be displayed for the 4-bit run length encoded products which were intended for display on the original display device. These 16 threshold fields (halfwords 31 - 46 in the final product message) are also used to provide decoding parameters for "digital products" which have real data values encoded into integer data levels.

Threshold Level Fields - The **Scale-Offset** Parameter Method **(Recommended)**

The new **Scale-Offset** formula can be used to encode and decode any product having a linear increment between encoded data values. The following threshold fields are being used by future Dual Polarization products 159 (DZD), 161 (DCC), and 163 (DKD) to describe the **Scale-Offset** coding.

Halfword	Field	Use
HW 31	**Threshold 1**	the SCALE in IEEE floating point format
HW 32	**Threshold 2**	
HW 33	**Threshold 3**	the OFFSET in IEEE floating point format
HW 34	**Threshold 4**	
HW 36	**Threshold 6**	the **highest data level having meaning, including flag values**
HW 37	**Threshold 7**	the number of leading flag values (can be 0)
HW 38	**Threshold 8**	the number of trailing flag values (can be 0)

Unlike the formula provided for the Original Method, the **Scale-Offset** formulas do not include knowledge of the number of leading / trailing flag values because the flag values do not affect the numerical coding directly. So

- When decoding product data values using the second formula, the user (or decoding system) must only apply the formula to the non-flag integer data values.
- When writing product data values, the product algorithm must set the flag values as needed and ensure that only valid meteorological values are encoded into the integer values contained in the product using the first formula.

The following formulas can be useful in encoding / decoding and display of a product.

The number of numerical data levels is

```
(HW_36_value + 1) - HW_37_value - HW_38_value
```

The lowest numerical data level is encoded into integer value:

```
0 + HW_37_value
```

The highest numerical data level is encoded into integer value:

```
HW_36_value - HW_38_value
```

The first trailing flag (if it exists) is represented by integer value:

```
HW_36_value - HW_38_value + 1
```

The encoding formula is:

```
integer_data_level = (real_value * HW_31_32_value) + HW_33_34_value
```

The decoding formula is:

```
real_value = (integer_data_level - HW_33_34 _value) / HW_31_32_value
```

8-bit Example

Using the same product as in the 8-bit example, product ID 87 (DBV) data fields would be encoded and decoded as follows.

To encode non-flag data values (that is integer values 2 - 255):

```
encoded_integer = (float_value * 2.0) + 129.0
```

To decode the (non-flag) integer values in the product:

```
float_value = (encoded_integer - 129.0) / 2.0
```

where
SCALE = 2.0
OFFSET = 129.0

Threshold Level Fields - The Original Parameter Method **(Not recommended for new development)**

The Legacy digital products (and many products added since) had a specific, though incomplete, method of providing information in the Product Description Block to aid in decoding integer values in data packet 16.

Halfword	Field	Use
HW 31	**Threshold 1**	contains the minimum value (encoded)
HW 32	**Threshold 2**	contains the increment (encoded)
HW 33	**Threshold 3**	contains the number of data levels

NOTE: The number of levels field (HW 33) is not used in a consistent fashion. Sometimes it is the maximum data level and sometimes the number of levels.

The information provided in the three parameters is not sufficient to decode the product.

a. Since the minimum value and the increment are stored on a 2-byte integer, an unstated scaling factor has to be applied to shift the decimal point and convert to real numbers. Some products had different scaling factors for the minimum value and increment. **NOTE:** This 'scaling factor' is not the same as the SCALE in the **Scale-Offset** formula.

b. There is no parameter specified for how many data levels are flag values. The decoding formula using minimum value, increment, and scale factors is not applied to flag values.

The Class 1 ICD does not provide an encoding or decoding function for the original parameter method. For product following the four encoding guidelines stated above:

To obtain the encoded integer value from the real floating point value:

```
data_level = num_flags +
    [real_value - (HW_31_value/min_val_scale)] * (incr_scale/HW_32_value)
```

To obtain the decoded floating point value from the encoded integer value:

```
real_value = (HW_31_value/min_val_scale) +
                [(data_level - num_flags) * (HW_32_value/incr_scale)]
```

where:
HW_31_value is the contents of HW 31 (encoded minimum value)
HW_32_value is the contents of HW 32 (encoded increment)
min_val_scale is the scaling factor used to convert the min value into a real number
incr_scale is the scaling factor used to convert the increment into a real number
num_flags is the number of leading value flags (beginning at data level 0)

The scaling factors are typically 10, 100, or 1000 which move the decimal point to the left to convert the integer data value into a real value. **NOTE:** This 'scaling factor' is not the same as the SCALE in the **scale-offset** formula.

One aspect of this coding may not be universally understood or consistently applied. With leading flag values, which integer data level does the encoded minimum value in HW 31 correspond to? For base data arrays in the radial base data messages, in the packet 16 arrays in the 8-bit base data products, and in the ITWSDBV product, the minimum value corresponds to the first non-flag data level. It should be noted that, for some products at least, AWIPS incorrectly applies the minimum value to data level 0. This has gone unnoticed because of the small size of the error.

8-bit Example

For product ID 87 (DBV), or ITWSDBV product,

- **Threshold 1** contains **-635**, for the minimum value (**HW_31_value**)
- **Threshold 2** contains **+5**, for the interval (**HW_32_value**)
- **Threshold 3** contains **256**, for the number of levels (**HW_33_value**)

Additional information is required to decode this product.

- **min_val_scale = 10** (which converts the minimum value to **-63.5**)
- **incr_scale = 10** (which converts the increment value to **0.5)**
- **num_flags = 2** (which associates the minimum value with data level **2**)

Using the decoding formula, integer data level 2 decodes to -63.5 and data level 3 decodes to -63.0.

Issue 1

The biggest issue is that even though the *ICD for RPG to Class 1 User* describes how existing products use the **Threshold** fields in the *Product Description Block*, no formal guidance for future use has been developed.

Issue 2

The *ICD for RPG to Class 1 User* does not explicitly state how the minimum value in **Threshold 1** is applied.

Issue 3

The description for the use of **`Threshold 3`** in the *ICD for RPG to Class 1 User* is inconsistent, or at least not clear. The ICD states this value should have a range of 0-255 but that the meaning of the value is 'the number of data levels', which could be 256. This may be related to existing products not using this header field in the same manner.

Issue 4

Not all products use this method of encoding data into an 8-bit integer. For example, product ID 134, DVL, uses a linear method for part of the data range and a non-linear method for the remainder. In addition the **`Threshold`** fields are used in a unique manner.

Because the information in `Threshold 1` - `Threshold 3` is incomplete, the CODE product display tool CODEview Graphics (CVG) does not currently use the `Threshold` fields in the product header. CVG configuration files for each digital product are used to provide all information required.

16-bit Example

TBD IN A FUTURE EDITION OF CODE

As with the 8-bit data scaled integer arrays, the CODE product display tool CODEview Graphics (CVG) does not actually use the **`Threshold`** fields in the product header. CVG configuration files for each digital product are used to provide all information required.

Threshold Level Fields - Additional Parameter Methods

In addition to product 134 (DVL) which uses a unique partially non-linear method, other products do not follow either the Original Method or the Scale-Offset method. Most of the proposed Dual Pol products are being modified to use the **`scale-offset`** parameter method. .Several that do not follow either the Original Method or the Scale-Offset method are:

Products 165 (DHC) and 163 (HHC)

These products use the concept of an enumerated type with a table defining the enumeration.

CVG use of Threshold Level Fields

For products using the Legacy 'original method', the information in **`Threshold 1 - Threshold 3`** is incomplete, the CODE product display tool CODEview Graphics (CVG) does not actually use the **`Threshold`** fields in the product header. CVG configuration files for each digital product are used to provide all information required. Currently CVG can use the 7 factors in the 'original method' (contained in CVG product display configuration files) for decoding the data values into the legend threshold labels in for 'Method 1' of legend configuration.

Beginning with Build 12, **for those products using Method 5 for display configuration, and having the `Scale-Offset` parameters in the product description block, CVG can use the contents of the threshold fields to decode the data levels for information display in for calculating legend threshold levels**. This permits the legend labels to be calculated dynamically as the increment in certain products changes (e.g., DSP).

For those products not using the **`Scale-Offset`** parameters in the product description block, parameters in legend configuration files will be used to decode the data levels and explicitly state static threshold labels.

NOTE: The dynamic change in legend threshold labels will NOT be provided for products like DSP if they are not modified to provide the Scale Offset parameters in the product description block.

Decoding will not be provided for any product that cannot be described with Scale Offset parameters. This includes products that do not have a linear encoding like DVIL. Both the recommended Method 5 and the original Method 2 will provide the capability to explicitly state threshold labels for products like DVIL that do not have a linear encoding.

.

CVT use of Threshold Level Fields

Beginning with Build 12,

- **for those products having the `Scale-Offset` parameters in the product description block, CODEview Text (CVT) can use the contents of the threshold fields to decode the data levels for information display.** This permits the legend labels to be calculated dynamically as the increment in certain products changes (e.g., DSP).
- **CVT can also decode any scaled offset-product whose encoding can be described using `Scale-Offset` parameters in a configuration file.**

Decoding will not be provided for any product that cannot be described with Scale Offset parameters. This includes products that do not have a linear encoding like DVIL. Both the recommended Method 5 and the original Method 2 will provide the capability to explicitly state threshold labels for products like DVIL that do not have a linear encoding.

Volume 2. Appendices

Appendix C. Data Level Threshold Values in Existing Products

==========

The use of the data level thresholds, which are contained in halfwords 31 - 46 in the Product Description Block (PDB), are documented in the *Interface Control Document (ICD) for the RPG to Class 1 User, Document Number 2620001*. The following text is taken from Note 1 following Figure 3-6 Graphic Product Message (Sheet 6).

Figure 3-6. Graphic Product Message (Sheet 6)

Note 1. The Data Level threshold values used to define the color table of products, described in Table III, consist of up to 16 Data Levels. The exceptions to this are products 32, 81, 93, 94, 99, 138, 153, 154, and 155 that may have up to a maximum of 255 equally spaced data levels. Additionally, product 134 (High Resolution VIL) can provide 255 data levels not necessarily with equal spacing. Also, product 135 (High Resolution Enhanced Echo Tops) can provide up to 199 data levels due to using the most significant bit as a "topped" flag.

For products 32, 94, and 153, data level codes 0 and 1 correspond to "Below Threshold" and "Missing", respectively. Data level codes 2 through 255 denote data values starting from the minimum data value in even data increments. The threshold level fields are used to describe the 256 levels as follows:

halfword 31 contains the minimum data value in dBZ * 10
halfword 32 contains the increment in dBZ * 10.
halfword 33 contains the number of levels (0 - 255)

For product 81, data level codes 0 will correspond to no accumulation and data level code 255 will represent data outside the coverage area. Data level codes 1 through 254 denote data values starting from the minimum data value in even data increments. The threshold level fields are used to describe the 256 levels for product 81 as follows:

halfword 31 contains the minimum data value in dBA*10
halfword 32 contains the increment in dBA * 1000.
halfword 33 contains the number of levels (0 - 255)

For products 93, 99, 154, and 155, data level codes 0 and 1 correspond to "Below Threshold" and "Range Folded", respectively. For products 93, 99, and 154, data levels 2 through 255 denote data values starting from the minimum data value in even data increments. For product 155, data levels 129 through 149 denote data values starting from the minimum data value in even data increments. The threshold level fields are used to describe (up to) 256 levels as follows:

halfword 31 contains the minimum data value in m/s*10
halfword 32 contains the increment in m/s*10
halfword 33 contains the number of levels (0 - 255)

For product 134, data level codes 0 and 1 correspond to "Below threshold" and "flagged data", respectively. Data level 255 is reserved for future use. Data levels 2 through 254 relate to VIL in physical units (kg m-2) via either a linear or log relationship. Any value of VIL above 80 kg m-2 is set to a data value of 254. The

coefficients used in the equations to relate the data values to VIL are float values. The IEEE standard for 32-bit floating point arithmetic (ANSI/IEEE Standard 7541985) has been adopted and modified to utilize the 16-bit (2 byte short) half words available here to describe the coefficients. Half words 31, 32, 33, 34, and 35 are used for this purpose as follows:

halfword 31 contains the linear scale encoded hex value of 0x5BB4 (short int 23476)
halfword 32 contains the linear offset encoded hex value of 0xC82A (short int -14294)
halfword 33 contains the digital log start value of 20
halfword 34 contains the log scale encoded hex value of 0x54DC (short int 21724)
halfword 35 contains the log offset encoded hex value of 0x593E (short int 22846)

For Build 9 and beyond, the linear scaling for HRVIL has been modified to provide improved depiction for weak weather signatures. Thus, halfwords 31 and 32 are redefined as follows:

halfword 31 contains the linear scale encoded hex value of 0x59AB (short int 22955)
halfword 32 contains the linear offset encoded hex value of 0x4400 (short int 17408)

The halfword hex values must be decoded to use the equations to convert a digital data value to VIL.
For digital values below the value of halfword 33, the linear equation is used:

Digital data value = decoded halfword 31*VIL + decoded halfword 32

For digital data values equal to or greater than the value of halfword 33, the log equation is used:

Digital data value = decoded halfword 34*LN(VIL) + decoded halfword 35

To decode the hex values, a two stage process based on the following methodology is used.
The 32-bit IEEE standard for floating point arithmetic has been modified for a 16 bit short as:

S	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F
0	1				5	6									15

The top row of the above table describes the designation as S for the one sign bit, E for the 5 exponent bits, and F for the ten fraction bits. The middle row notes the bit number starting with the MSB of 0. The bottom row relates 4 bit sequences to half byte sections.

First, convert the halfword hex value to its binary equivalent. Then, using the S, E, and F bit designations in the above table, build the decimal coefficient values using the guide below:

For E = 0, coefficient value = $(-1)^S * 2 * (0 + (F/2^{10}))$,
and for $0 < E < 255$;
coefficient value = $(-1)^S * 2^{E-16} * (1 + (F/2^{10}))$

For example, a coefficient value of (Hex) 5BB4, (bit sequence 0101 1011 1011 0100) is interpreted as:
$(-1)^0 * 2^{22-16} * (1 + (948/2^{10}))$ which resolves to a float value of 123.25.

For product 135, data level codes 0 and 1 correspond to “Below threshold” and “bad data”, respectively. Each echo top byte contains two pieces of information: the echo top in kft and an indication of if it were “topped”. The echo top data, thus, are grouped into two sets: 2-71 and 130-199. The second set is the same echo tops set as the first except that the most significant bit is set to1 to indicate a “topped” value. Each increment represents an increase of 1 kft. Any value of Echo Tops above 70 kft is set to a data value of 1. Half words 31, 32, 33, and 34 are provided to use for extracting the echo top value and “topped” flag:

halfword 31 contains the DATA_MASK 127 or 0x7f (hex) identifying the data bits
halfword 32 contains the DATA_SCALE 1
halfword 33 contains the DATA_OFFSET 2
halfword 34 contains the TOPPED_MASK 128 or 0x80 (hex)

The following relations are used when HREET data are decoded,

Value : Integer HREET altitude, expressing thousands of feet.
Topped: Boolean describing HREET "topped" condition.
Data : Packed integer HR-EET value.
== : Equality evaluation.
!= : Inequality evaluation.
& : Binary 'AND' operator.
| : Binary 'OR' operator.
? : Conditional expression:

(A ? B : C) returns B if A is true, returns C if A is false.

Use the following when decoding HREET data elements from NEXRAD product messages,

if (Data == 0)

Value is declared below threshold.
Topped is declared false.

else if (Data == 1)

Value is declared bad.
Topped is declared false.

else

Value = ((Data & DATA_MASK) / DATA_SCALE) - DATA_OFFSET
Topped = (Data & TOPPED_MASK) != 0

==========

Encoding for Run Length Encoded (4-bit) Products

Except for Products 32, 81, 93, 94, 99, 134, 135, 138, 153, 154, and 155 the Data Level Threshold halfwords are coded as follows:

If bit 0 (most significant bit) is set to one (1), then the least significant byte (bits 8-- 15) is interpreted as a code for:

0 = "BLANK"
1 = TH
2 = ND
3 = RF

If bits 1, 2, 3, 4, 5, 6 or 7 of the most significant byte are set to 1, then they are interpreted as a code for:

Bit 1 - If set the data field in the least significant byte is scaled by 100, to allow two decimal places of accuracy in some of the Threshold tables.

Bit 2 - If set the data field in the least significant byte is scaled by 20, to allow two decimal places of accuracy in some of the Threshold tables.

Bit 3 - If set the data field in the least significant byte is scaled by 10, to allow for one decimal place of accuracy in some of the threshold tables.

Bit 4 = ">"
Bit 5 = "<"
Bit 6 = "+"
Bit 7 = "-"

If bit 0 (most significant bit) is zero (0), then the low-order byte (bits 8 - 15) is a numeric value.

Example: A data level value of (Hex) 8401, (bit sequence 1000 0100 0000 0001) is interpreted as: < TH

==========

For product 138, data level code 0 corresponds to no accumulation and data level codes 1 through 255 denote accumulation values in units of hundredths-of-inches (.01"), in even data increments, with data level code 1 being the first non-zero accumulation value. The threshold level fields are used to describe the 256 levels for product code 138 as follows:

Halfword 31 contains the minimum data value (i.e., 0)
Halfword 32 contains the increment in .01" units
Halfword 33 contains the number of levels (0 - 255)

The Data Level threshold values used to define the color table of products, described in Table III, consist of up to 16 Data Levels. The exceptions to this are products 32, 81, 93, 94, 99, 156 and 157that may have up to a maximum of 255 equally spaced data levels.

For product 156, halfwords 31, 32, 33 and 34 contain parameters for decoding the digital (encoded) EDR, “DEDR”, to EDR via a linear scale as follows:

halfword 31 contains the linear scale factor (increment) in units of $m^{2/3}$ s^{-1} * 1000
halfword 32 contains the linear offset in units of $m^{2/3}$ s^{-1} * 1000
halfword 33 contains the total number of data levels (currently 64)
halfword 34 contains the number of leading data flags (currently 1)

Thus, DEDR = 0 represents flagged data and EDR = (halfword 31 / 1000) * DEDR + (halfword 32 / 1000) for DEDR values from 1 to 63.

For product 157, halfwords 31, 32, 33 and 34 contain parameters for decoding the digital (encoded) EDC, “DEDC”, to EDC via a linear scale as follows:

halfword 31 contains the linear scale factor (increment)
halfword 32 contains the linear offset
halfword 33 contains the number of data levels (currently 8)
halfword 34 contains the number of leading data flags (currently 0)

Thus, EDC = (halfword 31 / 1000) * DEDC + (halfword 32 / 1000) for DEDC values from 0 to 7.

Appendix D. Base Data Header Field Definitions

Base Data Header - Beginning with Build 12			
ANSI-C struct Base_data_header			
Data Type	**Name of Component**	**Description**	**Code**
unsigned short	msg_len	in the ORPG, the size of this message in shorts (2-bytes)	
short	msg_type	This field contains bit flags that are used to describe the message type and the enabled moments. See the table *Bit Flag Definitions for the msg_type Field* for a description.	
short	version	Version number for the radial format. Currently 0.	**
char	radar_name[6]	Radar name string consisting of 4 characters plus NULL terminator.	**
int	time	Collection time for this radial in milliseconds past midnight (GMT).	R
int	begin_vol_time	volume start time of in MS past midnight	RSI
unsigned short	date	Radial date, Modified Julian date (from 1/1/70)	R
unsigned short	begin_vol_date	Beginning of Volume. Modified Julian date (starting from 1/1/70)	RSI
float	latitude	Latitude of the RDA. Build 9 - from site adaptation data. Build 10 - from RDA message.	**
float	longitude	Longitude of the RDA. Build 9 - from site adaptation data. Build 10 - from RDA message.	**
unsigned short	height	Height of the radar in meters MSL. Build 9 - from site adaptation data and is the same as feedhorn height. Build 10 - from RDA message.	**
unsigned short	feedhorn height	Height of the feedhorn in meters MSL. Build 9 - from site adaptation data. Build 10 - from RDA message.	**
short	weather_mode	Set to 1 (clear air) or 2 (convective)	SI

	short	vcp_num	Volume coverage pattern. For example: 11 = (16 elev scans / 5 mins); 21 = (11 elev scans / 6 mins); 31 = (8 elev scans / 10 mins); 32 = (7 elev scans / 10 mins)	RSI
	short	volume_scan_num	Volume scan number (1 - 80). Recycles to 1 after 80. NOTE: Very first volume is 0.	SI
	short	vol_num_quotient	Quotient for dividing volume sequence number by MAX_VSCAN.	
	float	azimuth	Radial azimuth angle in degrees	r
	float	elevation	Elevation angle in degrees	r
	short	azi_num	Radial number within elevation scan (1, 2, ...)	R
	short	elev_num	RDA elevation number within a volume (1, 2, ...) scan. This is the ordinal of the scan. An elevation produced via a split cut will be made up of data from two scans.	R
	short	rpg_elev_ind	The RPG elevation index within a volume (1, 2, ...). This is the ordinal of the elevation.	I
	short	target_elev	Target elevation in .1 degrees (also found in VCP tables). This is the elevation at which the RDA is attempting to sample the data.	
	short	last_ele_flag	Set to 1 if this is the last cut, set to 0 otherwise.	
	short	start_angle	Calculated radial start angle in .1 degrees.	I
	short	delta_angle	Calculated radial width (angle between start angles) in .1 degrees.	I
	unsigned char	azm_index	Azimuth Index value (deg*100) = 100 if azimuth is aligned on even degrees. = 50 if aligned on 0.5 deg, = 0 if not aligned	
	unsigned char	azm_reso	Azimuth resolution. 1 = BASEDATA_HALF_DEGREE; 2 = BASEDATA_ONE_DEGREE.	****
	float	sin_azi	Sine of the azimuth angle.	
	float	cos_azi	Cosine of the azimuth angle.	
	float	sin_ele	Sine of the elevation angle.	
	float	cos_ele	Cosine of the elevation angle.	

	short	status	Radial status: 0x00 = beginning of elevation; 0x01 = intermediate radial; 0x02 = end of elevation; 0x03 = beginning of volume; 0x04 = end of volume; 0x08 = pseudo end of elevation; 0x09 = pseudo end of volume;	R
	char	pbd_alg_control	bits 0-2, processing control flag: = PBD_ABORT_FOR_NEW_EE; PBD_ABORT_FOR_NEW_EV; PBD_ABORT_FOR_NEW_VV bits 3-7, processing control abort reason: **see basedata.h**	
	char	pbd_aborted_volume	Set in conjunction with pbd_alg_control, this is the volume scan number to abort.	
	short	atmos_atten	[Elev attribute] Atmospheric attenuation factor; range -2 to -20; (scaled: val/1000 = dB/KM)	R
	short	spot_blank_flag	0 - none; 1- SPOT_BLANK_RADIAL; 2 - SPOT_BLANK_ELEVATION; 4 - SPOT_BLANK_VOLUME	RS
	float	horiz_noise	[Radial Attribute] Horizontal Noise, dBm	
	float	vert_noise	[Dual Pol] [Radial Attribute] Vertical Noise, dBm	
	float	calib_const	System gain calibration constant (-50. to +50.) (dB biased).	R
	float	horiz_shv_tx_power	Horizontal channel power (KW)	
	float	vert_shv_tx_power	[Dual Pol] Vertical channel power (KW)	
	float	sys_diff_refl	[Dual Pol] Calibration of system ZDR	
	float	sys_diff_phase	[Dual Pol] Differential phase (deg*182.049882)	
	short	sector_num	PRF Sector number within the cut (1, 2, 3)	R
	short	vel_offset	Byte offset to start of velocity data	***
	short	n_dop_bins	Number of Doppler bins in the msg. Does not include data above 70,000 ft. MSL.	r
	short	dop_bin_size	Bin size in meters.	r
	short	dop_range	Range in number of bins to first good Doppler bin (first bin is 1).	r
	short	range_beg_dop	Range to beginning of first Doppler bin in meters.	r
	short	dop_resolution	Set to 1 if RDA message vel_resolution = 2 (0.5 m/s). Set to 2 if RDA message vel_resolution = 4 (1.0 m/s).	r
	short	unamb_range	[Radial Attribute] Unambiguous range (scaled: val/10 = KM)	R

	short	nyquist_vel	[Radial Attribute] Nyquist velocity (scaled: val/100 = m/s). Set to 0 if the Doppler data is missing.	R
	short	vel_snr_thresh	SNR threshold (dB*8)	**
	short	vel_tover	Minimum difference in echo power for two signals to not be labeled as overlaid (dB*10)	**
	short	ref_offset	Byte offset to start of reflectivity data	***
	short	n_surv_bins	Number of surveillance bins in the msg. Does not include data above 70,000 ft. MSL.	r
	short	surv_bin_size	Bin size in meters.	****
	short	surv_range	Range in number of bins to first good surveillance bin (first bin is 1).	r
	short	range_beg_surv	Range to beginning of first surveillance bin in meters.	r
	short	sc_azi_num	Split cut azimuth number of reflectivity radial	
	short	surv_snr_thresh	SNR threshold (dB*8)	**
	short	spw_offset	Byte offset to start of spectrum width data	***
	short	spw_snr_thresh	SNR threshold (dB*8)	**
	short	spw_tover	Minimum difference in echo power for two signals to not be labeled as overlaid (dB*10)	**
	short	spare3	Unused	
	short	spare4	Unused	
	short	no_moments	[Dual Pol] Number of additional data field arrays.	
	unsigned int	offsets[20]	[Dual Pol] Byte offset from the beginning of the basedata header to the additional data field arrays. Note: it is possible to have an offset of value 0, which means no data.	

Code	Definition
**	Data to be included in RDA message beginning with Build 10. Currently data supplied from ORPG internal data (site data, VCP info, scan summary table).
***	In the basedata radial messages these offsets are relative to the beginning of the basedata header. In the basedata elevation message: Prior to Build 10 these offsets are relative to the beginning of the basedata header. After Build 10 these offsets are relative to the end of the basedata header (the beginning of the first base moment array).
****	These data fields are used to determine the resolution of the data when not registered for the original data types. **`azm_reso`** provides the radial spacing (1 degree or half degree) and **`surv_bin_size`** provides the reflectivity range sample size (1000 meters or 250 meters).
R	Data produced by the RDA and passed on to the ORPG Base Data Radial Message.
r	Data produced by the RDA but either slightly modified or encoded differently by the ORPG. Includes data that have been derived from other RDA data.

S Data, from the first radial of the first elevation in a volume, is stored in Scan Summary table. The **`time`** is rescaled to seconds. The **`weather_mode`** is encoded differently.

I Data used in an ICD graphic product header fields.

The bit values in the **`msg_type`** field are used by the infrastructure to determine the type of base data message. There are some situations where these bit fields are also useful in algorithms.

Bit Flag Definitions for the **`msg_type`** Field			
Bit Set	**Defined Bit Masks**	**Integer Value**	**Use**
Bit 0	REF_INSERT_BIT	1	Identifies data from second cut of a split cut. Used when registered for BASEDATA and determining which cut the data are from.
Bit 1	VEL_DEALIASED_BIT	2	
Bit 2	REF_ENABLED_BIT	4	Indicates that the basic reflectivity moment is enabled in the RDA.
Bit 3	VEL_ENABLED_BIT	8	Indicates that the basic velocity moment is enabled in the RDA.
Bit 4	WID_ENABLED_BIT	16	Indicates that the basic spectrum width moment is enabled in the RDA.
The following bit masks are cut types - Note 1			
Bit 5	REFLDATA_TYPE	32	This message was derived from the first cut of a split cut. This is a Continuous Surveillance wave type cut.
Bit 6	COMBBASE_TYPE	64	This message was derived from the second cut of a split cut. This is a Continuous Doppler wave type cut.
Bit 7	BASEDATA_TYPE	128	This message was derived from a Batch Cut:. - Batch wave type - Continuous Doppler batch wave type - Staggered Pulse wave type
The following bit masks are radial types - Note 2			
Bit 8	SUPERRES_TYPE	256	**As of Build 12.1, this means the data are in increase resoltion of 0.5 deg azimuth sampling. The 250 m surveillance range and 300 km Doppler range are defined independently.**
Bit 9	DUALPOL_TYPE	512	(Build 12) Data contain the measured Dual Pol data fields.
Bit 10	RECOMBINED_TYPE	1024	(Build 10) Data in 0.5 deg resolution has been recombined to 1.0 resolution; reflectivity in 250m resolution has been recombined to 1 km resolution.
Bit 11	PREPROCESSED_DUALPOL_TYPE	2048	(Build 12) The Dual Polarization data fields have been processed by the RPG.

Bit 12	**HIGHRES_REFL_TYPE**	**4096**	**Added in Build 12. Reflects horizontal resolution of surveillance bins 250 m or 1000 m.**

Additional masks (multiple bits)			
Bits 5, 6, 7	CUT_TYPE_MASK	0x00e0	Mask used to only pass the cut type bits.
8, 9, 10, 11	RADIAL_TYPE_MASK	0x0f00	Mask used to only pass the radial type bits.
5, 6, 7, 8, 9, 10, 11	BASEDATA_TYPE_MASK	0x0fe0	Mask used to pass the cut type and radial type bits.
5, 6, 7, 8, 9, 10, 11, 12	ALL_TYPES	0x1fe0	Mask used to pass the cut type and radial type bits.

Note 1 The bits representing the type of elevation cut (bits 5, 6, 7) can be used to distinguish between the first / second cut of a split cut and batch cuts. Making this distinction is not required unless registered for **BASEDATA** instead of **REFLDATA** or **COMBBASE**, or registered for **BASEDATA_ELEV** instead of **REFLDATA_ELEV** or **COMBBASE_ELEV**, or registered for **RAWDATA** rather than the **REFL_RAWDATA** or **COMB_RAWDATA,** or registered for **SR_BASEDATA** instead of **SR_REFLDATA** or **SR_COMBBASE**, or registered for **DUALPOL_BASEDATA** instead of **DUALPOL_REFLDATA** or **DUALPOL_COMBBASE**.

Note 2 Currently the values of the bits representing the type of radial message (bits 8, 9, 10, 11, 12) are not actually needed for algorithms.

- Instead of the **SUPERRES_TYPE** bit (azimuth resolution), the basedata header field: **azm_reso** can be used to determine the azimuth resolution.
- Instead of **HIGHRES_REFL_TYPE** (surveillance range resolution), the basedata header field: **surv_bin_size** can be used to determine range resolution.
- The other radial type bits simply correspond to the type of base data registered for.

Data Type Definitions				
char	8 bits		int	32 bits
short	16 bits		float	32 bit, IEEE Std. 754-1985

ANSI-C struct Base_data_header			
Data Type	**Name of Component**	**Description**	**Code**
short	msg_len	in the ORPG, the size of this message in shorts (2-bytes)	
short	msg_type	This field contains bit flags that are used to describe the message type and the enabled moments. See the table *Bit Flag Definitions for the msg_type Field* for a description.	
short	version	Version number for the radial format. Currently 0.	**
char	radar_name[6]	Radar name string consisting of 4 characters plus NULL terminator.	**
int	time	Collection time for this radial in milliseconds past midnight (GMT).	R
int	begin_vol_time	volume start time of in MS past midnight	RSI
unsigned short	date	Radial date, Modified Julian date (from 1/1/70)	R
unsigned short	begin_vol_date	Beginning of Volume. Modified Julian date (starting from 1/1/70)	RSI
float	latitude	Latitude of the RDA. Build 9 - from site adaptation data. Build 10 - from RDA message.	**
float	longitude	Longitude of the RDA. Build 9 - from site adaptation data. Build 10 - from RDA message.	**
unsigned short	height	Height of the radar in meters MSL. Build 9 - from site adaptation data and is the same as feedhorn height. Build 10 - from RDA message.	**
unsigned short	feedhorn height	Height of the feedhorn in meters MSL. Build 9 - from site adaptation data. Build 10 - from RDA message.	**
short	weather_mode	Set to 1 (clear air) or 2 (convective)	SI
short	vcp_num	Volume coverage pattern. For example: 11 = (16 elev scans / 5 mins); 21 = (11 elev scans / 6 mins); 31 = (8 elev scans / 10 mins); 32 = (7 elev scans / 10 mins)	RSI
short	volume_scan_num	Volume scan number (1 - 80). Recycles to 1 after 80. NOTE: Very first volume is 0.	SI
short	vol_num_quotient	Quotient for dividing volume sequence number by MAX_VSCAN.	
float	azimuth	Radial azimuth angle in degrees	r
float	elevation	Elevation angle in degrees	r

short	azi_num	Radial number within elevation scan (1, 2, ...)	R
short	elev_num	RDA elevation number within a volume (1, 2, ...) scan. This is the ordinal of the scan. An elevation produced via a split cut will be made up of data from two scans.	R
short	rpg_elev_ind	The RPG elevation index within a volume (1, 2, ...). This is the ordinal of the elevation.	I
short	target_elev	Target elevation in .1 degrees (also found in VCP tables). This is the elevation at which the RDA is attempting to sample the data.	
short	last_ele_flag	Set to 1 if this is the last cut, set to 0 otherwise.	
short	start_angle	Calculated radial start angle in .1 degrees.	I
short	delta_angle	Calculated radial width (angle between start angles) in .1 degrees.	I
unsigned char	azm_index	Azimuth Index value (deg*100) = 100 if azimuth is aligned on even degrees. = 50 if aligned on 0.5 deg, = 0 if not aligned	
unsigned char	azm_reso	Azimuth resolution. 1 = BASEDATA_HALF_DEGREE; 2 = BASEDATA_ONE_DEGREE.	****
float	sin_azi	Sine of the azimuth angle.	
float	cos_azi	Cosine of the azimuth angle.	
float	sin_ele	Sine of the elevation angle.	
float	cos_ele	Cosine of the elevation angle.	
short	status	Radial status: 0x00 = beginning of elevation; 0x01 = intermediate radial; 0x02 = end of elevation; 0x03 = beginning of volume; 0x04 = end of volume; 0x08 = pseudo end of elevation; 0x09 = pseudo end of volume;	R
char	pbd_alg_control	bits 0-2, processing control flag: = PBD_ABORT_FOR_NEW_EE; PBD_ABORT_FOR_NEW_EV; PBD_ABORT_FOR_NEW_VV bits 3-7, processing control abort reason: **see basedata.h**	
char	pbd_aborted_volume	Set in conjunction with pbd_alg_control, this is the volume scan number to abort.	
short	atmos_atten	[Elev attribute] Atmospheric attenuation factor; range -2 to -20; (scaled: val/1000 = dB/KM)	R
short	spot_blank_flag	0 - none; 1- SPOT_BLANK_RADIAL; 2 - SPOT_BLANK_ELEVATION; 4 - SPOT_BLANK_VOLUME	RS

	float	horiz_noise	[Radial Attribute] Horizontal Noise, dBm	
	float	vert_noise	[Dual Pol] [Radial Attribute] Vertical Noise, dBm	
	float	calib_const	System gain calibration constant (-50. to +50.) (dB biased).	R
	float	horiz_shv_tx_power	Horizontal channel power (KW)	
	float	vert_shv_tx_power	[Dual Pol] Vertical channel power (KW)	
	float	sys_diff_refl	[Dual Pol] Calibration of system ZDR	
	float	sys_diff_phase	[Dual Pol] Differential phase (deg*182.049882)	
	short	sector_num	PRF Sector number within the cut (1, 2, 3)	R
	short	vel_offset	Byte offset to start of velocity data	***
	short	n_dop_bins	Number of Doppler bins in the msg. Does not include data above 70,000 ft. MSL.	r
	short	dop_bin_size	Bin size in meters.	r
	short	dop_range	Range in number of bins to first good Doppler bin (first bin is 1).	r
	short	range_beg_dop	Range to beginning of first Doppler bin in meters.	r
	short	dop_resolution	Set to 1 if RDA message vel_resolution = 2 (0.5 m/s). Set to 2 if RDA message vel_resolution = 4 (1.0 m/s).	r
	short	unamb_range	[Radial Attribute] Unambiguous range (scaled: val/10 = KM)	R
	short	nyquist_vel	[Radial Attribute] Nyquist velocity (scaled: val/100 = m/s). Set to 0 if the Doppler data is missing.	R
	short	vel_snr_thresh	SNR threshold (dB*8)	**
	short	vel_tover	Minimum difference in echo power for two signals to not be labeled as overlaid (dB*10)	**
	short	ref_offset	Byte offset to start of reflectivity data	***
	short	n_surv_bins	Number of surveillance bins in the msg. Does not include data above 70,000 ft. MSL.	r
	short	surv_bin_size	Bin size in meters.	****
	short	surv_range	Range in number of bins to first good surveillance bin (first bin is 1).	r
	short	range_beg_surv	Range to beginning of first surveillance bin in meters.	r

	short	sc_azi_num	Split cut azimuth number of reflectivity radial	
	short	surv_snr_thresh	SNR threshold (dB*8)	**
	short	spw_offset	Byte offset to start of spectrum width data	***
	short	spw_snr_thresh	SNR threshold (dB*8)	**
	short	spw_tover	Minimum difference in echo power for two signals to not be labeled as overlaid (dB*10)	**
	short	spare3	Unused	
	short	no_moments	[Dual Pol] Number of additional data field arrays.	
	unsigned short	offsets[17]	[Dual Pol] Byte offset from the beginning of the basedata header to the additional data field arrays. Note: it is possible to have an offset of value 0, which means no data.	

Appendix E. The Generic Moment Structure

Generic Moment - Implemented in ORPG Build 12			
ANSI-C struct Generic_moment_t		**THE DEFINITION OF THIS STRUCTURE IS STILL UNDER DEVELOPMENT.**	
Data Type	**Name of Component**	**Description**	
`char`	`name[4]`	Name of this moment. See Note 1 for possible values.	
`unsigned int`	`info`	Offset to the moment specific information for this moment. This is currently not used, value set to 0.	
`unsigned short`	`no_of_gates`	number of gates (size of the data array) for this moment	
`short`	`first_gate_range`	Range to the center of the first gate, in `meters`	
`short`	`bin_size`	Size of each gate for this moment, in `meters`	
`short`	`tover`	The minimum difference in echo power between two resolution gates for them not to be labeled "overlayed", in `dB*10`	
`short`	`SNR_threshold`	Signal to Noise Ratio for valid data, in `dB*10` (error in comment in generic_basedata.h)	
`unsigned char`	`control_flag`	0 - None; 1 - recombined azimuth; 2 - recombined range gates; 3 - recombined azimuth and range gates (Legacy res)	
`unsigned char`	`data_word_size`	Number of bits used for each gate of data. 8, 12, 16, or 32	
`float`	`scale`	Scale factor used to quantify the data (encode floating point data into an integer). A value of 0.0 is used to distinguish 32-bit floating point data from 32 bit integer data. See Note 2.	
`float`	`offset`	The shift factor used to quantify the data (encode floating point data into an integer). See Note 2.	
`union {`		A variable length data array. `no_of_gates` indicates the number of elements in the array and `data_word_size` indicates the type and size of the data. One of the following types:	
		`unsigned char b[0]`	`data_word_size` 8, 12
		`unsigned short u_s[0]`	`data_word_size` 16
		`unsigned int u_i[0]`	`data_word_size` 32 and `scale` not 0.0
		`float f[0]`	`data_word_size` 32 and `scale` 0.0
	`} gate`		

Note 1 Name used to identify this moment type. Note space padded on the right. **The initial Dual Pol Implementation is targeted for Build 12.1.**

Measured Data Fields (from **SR_BASEDATA**, etc.)
- "**DZDR**" Differential Reflectivity - non-processed
- "**DPHI**" Differential Phase - non-processed
- "**DRHO**" Correlation Coefficient - non-processed

Derived / Processed Data Fields (from **DUALPOL_BASEDATA**, etc.)
- "**DZDR**" Differential Reflectivity - processed
- "**DPHI**" Differential Phase - processed
- "**DRHO**" Correlation Coefficient - processed
- "**DSNR**" Signal-to-Noise Ratio
- "**DSMZ**" Processed Reflectivity
- "**DSMV**" Smoothed Velocity
- "**DKDP**" Specific Differential Phase
- "**DSDZ**" Texture (standard deviation) for Reflectivity
- "**DSDP**" Texture (standard deviation) for Differential Phase

The Base Data Moments ("**DREF**" Reflectivity, "**DVEL**" Velocity, and "**DSW** " Spectrum Width) in the external RDA message (message 31) are not in the internal basedata generic structures, they are located in the basic arrays of the internal OPRG basedata message.

Note 2 Formulas for encoding and decoding the moment data.

To convert floating point moment to integers (encoding):

```
i = (f * scale) + offset
```

To convert integer moment data to floating point (decoding):

```
f = (i - offset) / scale
```

Volume 2. Appendices

Appendix F. Software Removed for the Public Edition

Differences between the U.S. Government and Public Editions of CODE

The significant difference between the U.S. Government Edition and the Public Edition of CODE is the removal of certain proprietary software components in the Public release. The source code archive provided with the Public Edition has been modified to eliminate this software and the filename changed to include the term "pub" for public (e.g., **`rpg_b##_#r#_##_pub_src.tgz`**) in order to identify the correct archive.

Currently, 8 operational tasks have been removed from the NWS Edition. A summary of the software removed is contained in the following table.

Operational Processes Removed for the Public Edition					
Source Code Directory	Executable Task Name	Product Name	ID	Product Description	Source
`cpc010`	**`nexradMigfa`**	**`MIGFA`**	**`140`**	GFM Gust Front MIGFA	MIT/LL
`cpc022/tsk001`	**`ntda_alg`**	**`NTDA_EDR_IP`** **`NTDA_CONF_IP`**	**`315`** **`316`**	NTDA EDR Intermediate Prod NTDA CONF Intermediate Prod	NCAR
`cpc022/tsk002`	**`ntda_fp`**	**`NTDA_EDR`** **`NTDA_CONF`**	**`156`** **`157`**	NTDA EDR Final Product NTDA CONF Final Product	NCAR
`cpc022/tsk003`	**`data_qual`**	**`DQA`**	**`297`**	Edited Reflectivity Data	MIT/LL
`cpc022/tsk004`	**`hiresvil`**	**`HRVIL`**	**`134`**	High Resolution Digital VIL	MIT/LL
`cpc022/tsk005`	**`hireseet`**	**`HREET`**	**`135`**	Enhanced Echo Tops	MIT/LL
`cpc022/tsk007`	**`icing_hazard`**	**`IHL`**	**`178`**	IHL Icing Hazard Level	MIT/LL
`cpc022/tsk008`	**`hail_hazard`**	**`HHL`**	**`179`**	HHL Hail Hazard Layer	MIT/LL
`cpc023/tsk004`	**`aca`**	**`AC`**	**`1965`**	Aviation Classification	MIT/LL

Volume 2. Appendices

Appendix G. Quick Reference for Starting the ORPG

This is a quick reference to running the ORPG. Complete procedures for starting and stopping the ORPG along with troubleshooting hints are included with CODE Guide Volume 1 Document 1 Section IV.

If you have problems starting the ORPG software, troubleshooting hints are contained in Volume 1 Appendix H - ORPG Launch Problems.

TO START ORPG TASKS:

- Log in as an appropriate user, that is the account into which the ORPG is installed.
- Type: **`mrpg -p -v startup`**

  The **-v** option provides a verbose output.
  The **-p** option cleans up all data stores before starting up.

- Wait for the command prompt to return. Startup normally requires less than one minute.

  A sample output of this command is provided in Volume 1 Appendix I.

To Check Status of Running Programs:

- Type: **`rpg_ps`**

  A sample output of ***rpg_ps*** is provided in Volume 1 Appendix J.

Note: The ***rpg_ps*** command does not work unless certain ORPG tasks are running (it will not work after executing **`mrpg cleanup`**). In this case, the status of running tasks can be checked with the standard **`ps -ef`** command.

To Launch the ORPG User Interface Program:

Type: **`hci`**

Note: ORPG algorithm tasks will run without launching the *hci*. Documentation of the *hci* is not included with this package.

Ingest a Source of Base Data

The ORPG utility "**`play_a2`**" is used for disk file and tape playback. In addition to the command line mode for Archive II disk files, **`play_a2`** includes an interactive mode for both Archive II disk files and 8mm Archive II tapes. If current weather data is desired the tool "**`hci_read_l2″`** is provided to ingest live level 2 (lower-case L2) data over the Internet.

Using Archive II data disk files:

The ORPG utility "**`play_a2`**" reads individual files each containing a volume of Archive II data and ingests the data into the ORPG. In order to provide a quick test of the ORPG, three files are included with the CODE ORPG configuration files and have been installed in **`$HOME/ar2data`**. The CODE CD contains additional Archive II disk files.

Execute the following command to ingest these files.

- Type: **`play_a2 -d $HOME/ar2data`**
- If you have launched the ***hci***, observe the RDA radome indicate scanning in progress on the GUI window

If the variable **`AR2_DIR`** has been set to the **`$HOME/ar2data`** directory, executing '**`play_a2`**' will suffice.

See the CODE Utility documentation contained in CODE Guide Volume 4 for additional information the command line mode of **`play_a2`**.

For Graphic Display of Final Products:

NOTE: The environmental variable **`CVG_DEF_PREF_DIR`** must be defined as the path of the location of the default preferences files (normally **`$HOME/tools`**) for **`CVG`** to function properly.

- Type: **`cvg`** to launch the CODEview Graphics Display tool

Once the utility is launched,

A product must be selected from the product database using the product list on the main CVG window.

After the product is selected, the desired data packets for display are chosen from the Packet Selection popup-screen.

See the CODE Utility documentation contained in CODE Guide Volume 4 for additional information.

TO STOP ORPG TASKS:

- Type: **`mrpg shutdown`**
- Type: **`mrpg cleanup`**

IMPORTANT: Even though **`mrpg cleanup`** command is optional, it **should always be executed in a development environment when stopping the ORPG**. It is important to execute **`mrpg cleanup`** if the ORPG is installed in more than one account. If not, an ORPG that is installed in another account will not launch unless the value of **`RMTPORT`** has been modified.

To Stop Ingest of Base Data:

Archive II disk files:

- Type **ctrl-c** in the terminal that started the 'play_a2' utility